Neural networks for solving regression and classification problems

Jonatan H. Hanssen
Bachelor Student, Robotics and Intelligent Systems
Department of Informatics
The faculty of Mathematics and Natural Sciences
Email: jonatahh@ifi.uio.no

Eric E. Reber
Bachelor Student, Robotics and Intelligent Systems
Department of Informatics
The faculty of Mathematics and Natural Sciences
Email: ericer@ifi.uio.no

Gregor Kajda
Bachelor Student, Robotics and Intelligent Systems
Department of Informatics
The faculty of Mathematics and Natural Sciences
Email: grzegork@ifi.uio.no

1 Abstract
We developed a Feed Forward Neural Network (FFNN) and applied it to many common datasets used for benchmarking machine learning algorithms. Furthermore, we compared our results to those attained by linear and logistic regression. To optimize our network, we explored the use of different schedulers in our gradient descent method, and compared their performance against eachother. Overall, we found that our FFNN performed well in all classification tasks we tested, attaining test accuracies between 95 and 100 percent for all classification problem. However, we also found that simple logistic regression performed similarly, with a much lower computational cost. For regression on terrain data, we found our neural net lacking, not being able to beat our polynomial fit from our previous project, and only attaining an MSE of 1 BILLION, which is 2000 BILLION lower than our best polynomial fit of the Franke Function. In our gradient descent, we found that SCHEDULER performed best, being 1 BILLION percent better than the average of all other methods.

2 Introduction
Neural Networks are all the rage these days, everyone wants to have a neural network in their home. Ive even seen people install neural networks in their dogs.

3 Method
3.1 Theory
This sections covers the theoretic background which underlies the methods used in this paper.

3.1.1 Logistic Regression
As part of the research conducted in this project, we introduce a new type of regression model, known as Logistic regression. This is a powerful, yet simple algorithm used in machine learning for the purpose of approximating a probability associated with the occurrence of a specific event given measured data. The fundamental idea behind logistic regression is based on the exact approach used in Linear Regression, meaning that we wish to describe a relationship between dependent and explanatory variables in terms of a set of estimated parameters. However, in this case we are not using the weighted sum of our basis functions directly, but feeding them into an activation function. This function, often the sigmoid function, maps the reals onto a range $y \in (0, 1)$. This in turn allows for the computation of probability related to an event taking place. Rounding

of said probability leads to labeling of each input as belonging to one of two classes, meaning that each object fed into the model will be classified as being of class 0 or 1 (in the case of binary classification).

3.1.2 Gradient Descent

In our previous project, we have primarily dealt with problems that have an analytical solution. This means that once a model has been decided on (polynomial degree and value for lambda for example), finding the optimal solution is a deterministic process which does not require further tuning. However, for the methods we use in this project - such as Logistic Regression - no analytical solutions exist, thus we will have to turn to numerical techniques to find the 'ideal' solution. Our method of choice in this case is gradient descent, an optimization technique which reduces the residuals by utilizing the idea of moving downhill a convex function until a globally minimal scalar value is reached. The gradient itself is nothing more than a first order derivative of a given function, which points in the direction of the greatest change.

The application of gradient descent to our problems as means of finding the ideal solution implies that the parameters be initialized randomly. The most common approach is to set each parameter equal to a normal distribution with mean zero and variance equal to one, and iteratively update the parameters by calculating the gradient at each given point in the residual space [Brunton and Kutz, 2018, preface p. x]. For any one-layer model, the gradient function used to tune the weights will take the form of the product of the partial derivative of the cost function with respect to the activation function, and the partial derivative of the activation function with respect to the weights:

$$\frac{\partial \mathcal{C}}{\partial w_{jk}} = \frac{\partial C}{\partial a_j} \frac{\partial a_j}{\partial w_{jk}} \tag{1}$$

Although gradient decent is an excellent algorithm for finding optimal solutions, it can also become quite computationally expensive for problems where large numbers of datapoints are available. However, this issue may be effectively countered by applying a variant of standard gradient approach known as Stochastic Gradient Decent or simply SGD. The SGD is based on randomly choosing one or a small number of instances from the dataset to update the parameters, hence the name stochastic. While it may seem like a poor idea to approximate the gradient instead of using the true gradient, the SGD has proven to give a large computational advantage and will often allow the model to converge into a global minima faster than GD. Furthermore, the stochastic nature of this method can allow SGD to avoid getting stuck in local minima [Raschka and Mirjalili, 2019, 46].

3.1.3 Schedulers

Whilst SGD itself gives a significant boost to the learning pace of machine learning models, it is possible to improve the algorithm further by performing what is known as scheduling. These methods work by adaptively adjusting the learning rate, thus allowing us to avoid local minima and converge even faster. One of the simpler methods involves adding a fraction of the previous change to the current value used to update the weights, thus keeping a sort of memory of the previous iterations. This is called momentum, due to the physical analogue of a moving particle having momentum. With this addition, we will keep moving in parameter space even if our current gradient is small, because we have gained momentum in a certain direction based on our previous iterations. Furthermore, we will keep moving in the direction where most gradients point, even if outliers point in other directions. Other, more complex methods, also keep track of approximations of the second moment of the cost function. These methods aim to adaptively change the learning rate for different directions in parameter space, so that small steps are taking in steep, narrow directions and large steps taken in shallow, flat directions [Hjort-Jensen, 2022a]. Examples of schedulers using the second moment are RMS-prop, Adam and Adagrad.

3.1.4 Feedforward Neural Networks

For some problems, like approximating a simple function like the Franke Function in project 1, it is sufficient to choose a set of basis functions and find their optimal linear combination. Given that our problem can be reasonably approximated by this basis, we can achieve good results. However, by limiting ourselves to linear functions, we are possibly limiting our model, as our dataset may well be better approximated by a different basis, our perhaps not even by a linear combination of functions at all. Because of this, linear regression is not able to understand the interaction between two arbitrary input variables [Goodfellow et al., 2016, 165]. A better approach would be to simply feed our model our features directly, and hope that it may somehow learn the relationship organically.

To learn any arbitrary relationship between our features may at first seem like an intractable problem, but we are actually able to achieve this with relatively small changes to our linear regression methods. Instead of having our prediction vector

be a linear combination of our input vector, we use the linear combinations of the inputs to create an intermediate vector, and apply an activation function to this vector. The resulting values from this function are either passed to the output as a linear combination again, or sent to another intermediate vector. By having the activation function be a nonlinear function, such as the sigmoid or RELU function, we are able to approximate any continous function to arbitrary accuracy [Bishop, 2006, 230]. By introducing these intermediate vectors, we have created a layered structure, where information is fed forward from layer to layer. Because of this, we call this model a Multilayer Perceptron or a Feedforward Neural Network.

Mathematically, each node on a layer is a linear combination of all the nodes of the previous layer, fed into the activation function for the current layer.

$$a_i^l = f^l(z_i^l) = f^l\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l a_j^{l-1} + b_i^l\right). \tag{2}$$

3.1.5 Activation functions

As we have seen, the behaviour of our neural network is defined by the amount of hidden layers and the amount of nodes in each one (our architecture), but also by our choice of activation functions. For the output layer, the activation is usually easily decided based on the problem (identity for regression problems, sigmoid for binary classification, SoftMax for multiclass). For the hidden layers, the choice of activation function is often decided through a process of trial and error, as this is an area without definitive guiding theoretical principles [Goodfellow et al., 2016, 188]. Below are the definitions of some of the most common activation functions.

The sigmoid is defined as follows:

$$y = \frac{1}{1 + exp(x)} \tag{3}$$

As we can see from the equation, this function has a range of $y \in (0,1)$, and for high values of x, the rate of change is very small. This saturation is a problem for gradient descent methods, and especially for backpropagation, as the repeated multiplication of very small gradients can lead to weights barely being updated and training slowing down (or even stopping completely). Because of this saturation, the sigmoid is generally not used as an activation function for the hidden layers [Goodfellow et al., 2016, 191].

Another activation function is the Rectified Linear Unit, and its cousin, the Leaky Rectified Linear Unit:

$$RELU(x) = \begin{cases} 0 & if \ x < 0, \\ x & if \ x \geq 0. \end{cases} \tag{4}$$

$$LRELU(x) = \begin{cases} -\alpha \cdot x & if \ x < 0, \\ x & if \ x \geq 0. \end{cases} \tag{5}$$

These do not suffer the same saturation problems displayed by the sigmoid function, and as such are often the default choice of activation for the hidden layers [Goodfellow et al., 2016, 188].

3.1.6 Backpropagation

Like logistic regression, the optimal weights for each layer in an FFNN can not be found analytically, and we have to turn to gradient descent. By repeated use of the chain rule, we are able to calculate the gradient for each layer iteratively, using the gradient of the layer after it and starting at the output. This is simply the reverse mode of automatic differentiation, and is

computationally preferred compared to forward mode, where we start at the input layer [Raschka and Mirjalili, 2019, 416]. The equations for calculating the gradients in an FFNN, which we will not derive here, are as follows [Hjort-Jensen, 2022b]:

First we calculate the so called delta terms for all layers, starting with the output layer:

$$y_i^l = f^l(u_i^l) = f^l\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l y_j^{l-1} + b_i^l\right). \tag{6}$$

We use this term in an iterative process, working backwards through the network with the following equation:

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l). \tag{7}$$

Using these delta terms, we can calculate the gradients for the weights and biases in each layer:

$$w_{jk}^l \leftarrow= w_{jk}^l - \eta \delta_j^l a_k^{l-1} \tag{8}$$

$$b_j^l \leftarrow b_j^l - \eta \frac{\partial \mathcal{C}}{\partial b_j^l} = b_j^l - \eta \delta_j^l \tag{9}$$

With the theory explained, we can go on to describe how we have applied it to our problem.

3.2 Implementation

We implement the FFNN as a class, using NumPy's many matrix functions. In the instantiation of the class, one is able to specify an arbitrary architecture, and choose activation functions for the output and the hidden layer. The cost function can also be specified, and this, along with the choice of output function, allows the network to be used for regression, binary classification and multiclass classification. By specifying no hidden layers in the architecture, the model can be used for linear and logistic regression as well. Because of this, no explicit implementation of linear and logistic regression has been made, which means that almost all relevant code is contained in `src/FFNN.py` and `src/Schedulers.py`. When instatiating the model, we set the weights to normal distributions with mean zero and variance one. We set the bias to 0.1 as recommended in Goodfellow et Al. [Goodfellow et al., 2016, 189].

The feedforward part is implemented as a simple matrix vector multiplication, followed by an elementwise application of the chosen activation function. For processing batches of inputs, the same multiplication code is used, but this time the input is the design matrix instead of a single row. Each hidden layer will now contain matrices themselves, consisting of a number of rows equivalent to the batch size and a number of columns equal to the number of nodes in this layer.

The backpropagation is implemented by differentiating all the functions declared when the class is instatiated, and following the algorithm laid out in Hjort-Jensens lectures [Hjort-Jensen, 2022b]. For batch backpropagation, we accumulate the gradient. In practice this means that the gradient for the weights, normally a matrix the same size as the relevant weight matrix, is now one matrix per element in our batch (in practice implemented as a NumPy array of three dimensions). At the end of the batch, we sum all the gradients for each weight matrix together and subtract this from the corresponding weight matrix.

Training a model is implemented as repeated calls to feedforward and backpropagation, once per batch per epoch. During training, one is able to specify which scheduler to use. These are implemented as a series of classes all inheriting from a base class, and are all implemented following Hjort-Jensens lectures [Hjort-Jensen, 2022a]. For finding the optimal parameters, we implement our own gridsearch which finds the lowest error among combinations of lambda, eta, and parameters used for the scheduling.

Dataset	Std of noise	Best MSE	Best degree
Franke	0	0.0005109	9
Franke	0.05	0.0038302	7
Franke	0.10	0.0139833	7
Terrain 1	-	0.0142683	19

4 Results

4.1 Different schedulers used with linear regression

4.2 Neural Network for regression

For the organic terrain data used in our previous project, we found that our neural net performed slightly better, gaining an MSE of NUMBER, which was about PERCENTAGE better than our best polynomial fit. However, looking at our prediction visually, and also comparing the MSE to the one gained on the Franke Function, we see that we are still lacking. For terrain data like this, which contain a lot of spatial information, we believe that Convolutional Neural Networks could be a better fit.

4.3 Neural Network for binary and multiclass classification

To test our networks ability to classify data, we trained it on many common datasets used in the development of neural networks. The datasets were the Wisconsin Breast Cancer dataset and the 8x8 MNIST dataset, as well as other datasets[1]. We found good results on all datasets achieving a test accuracy of over 97% on all

4.4 Logistic Regression

Although our neural network performed well, we also found very similar results with logistic regression, at a much lower computational cost. This implies that our classification datasets were perhaps too simple, and that using Neural Networks was perhaps not needed to achieve good results.

4.4.1 Ordinary Least Squares

4.4.2 Ridge

4.4.3 Ridge Regression

As our model is not overfitting, we do not expect Ridge to offer much improvement. This proves to be the case, as we see from Fig. 15 in the appendix. Here we see that different values of lambda have very little impact, as there are no parameters that need regularization, and no overfitting to supress. For very large values, we see that we are shrinking the parameters too much, and we see a slight increase in MSE, but overall we do not see much of an effect. We could expect to see that Ridge shows improvement at higher polynomial degrees, when OLS starts to overfit, but due to computational constraints we have not been able to push our grid search much further than 9 degrees. Using scikit's GridSearch, we find that the optimal lambda is 0, which is exactly what we would expect on a model that does not overfit, as Ridge regression with a lambda equal to 0 is equivalent with OLS. See Table 1.

4.4.4 Lasso Regression

With lasso regression, we see similar results as with Ridge. Fig. 16 in the appendix shows little change, except for when our λ is too high and we force all parameters to 0. Comparing lasso to the two other methods for the real dataset has proven to be difficult, as the numerical methods for finding the best parameters are very time consuming. We have not been able to gridsearch up to polynomial degree 19, but up to degree 9 we found the best lambda to be 2.154^{-7} for polynomial degree 9, giving an MSE of 0.01685.

Comparing all three methods up to degree 9 for different lambdas (using the same lambda for Lasso and Ridge, just to get an overview) in Fig. 17 in the appendix, we see the same trend as we have been describing. Regularization offers little reward, which is to be expected when we are not overfitting.

5 Conclusion

In this paper we have compared Ordinary Least Squares, Ridge and Lasso regression for both synthetic and real data. We have seen the value of using resampling methods such as bootstrap and cross validation to gain valuable insights about the quality of our predictions. Using resampling methods, we have found that linear regression was well suited for approximating

[1] Sonar dataset, House dataset

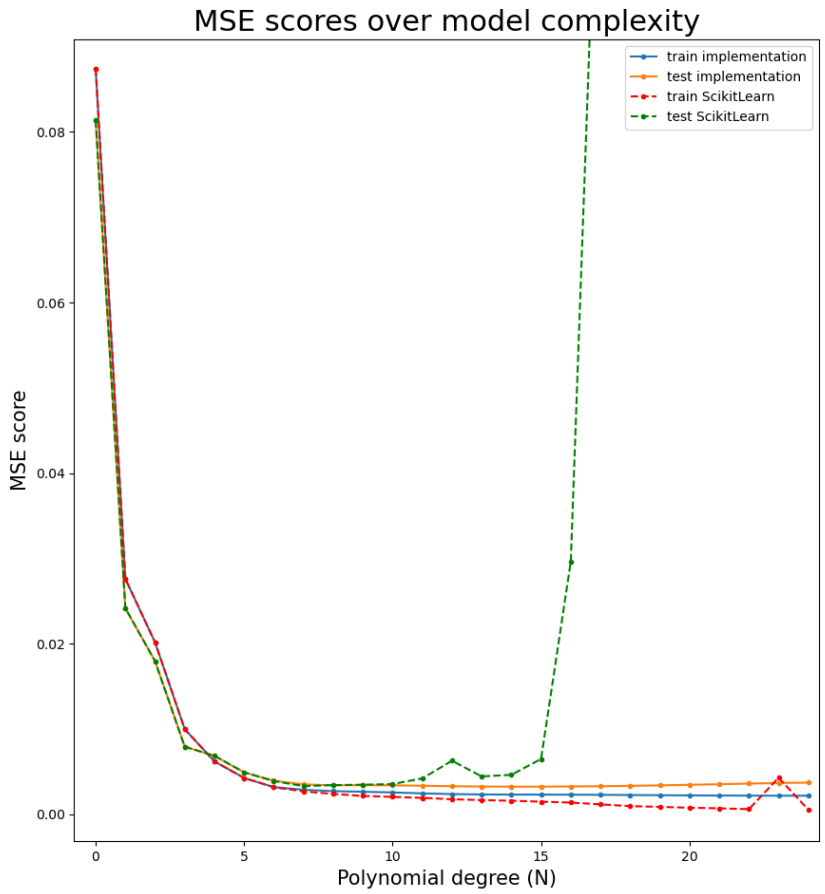

Fig. 1. Test and train MSE for ridge

Table 1. Best MSE for OLS, Ridge and Lasso regression on real elevation data. We only fit up to degree 9 due to computational constraints

Regression	MSE	Degree	λ
OLS	0.01538	9	-
Ridge	0.01538	9	0
Lasso	0.01685	9	$2.154 \cdot 10^{-7}$

the synthetic Franke function, but less suited for the real elevation data. We have seen that a correctly chosen regularization parameter can allow us to push our model complexity higher than we can with OLS, and thus give us a better MSE than OLS. Looking at the real data, we have seen that regularization does not offer much value when we are not overfitting our dataset. Thus all three linear regression methods perform similarly on this dataset.

References

[Bishop, 2006] Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer, New York, NY.

[Brunton and Kutz, 2018] Brunton, S. L. and Kutz, J. N. (2018). *Data Driven Science & Engineering: Machine Learning, Dynamical Systems, and Control*. Brunton & Kutz, Seattle, WA.

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

Table 2. This table shows the command to execute to reproduce every figure in the report. (More info about the scripts and their parameters can be found in the README)

Figure	Shell command (leading `python3` omitted)
??	`task_b.py --noise 0 -n 25`
1	`task_b.py --noise 0 -n 25` (*Model has been changed to ridge with lambda*=$1.43 \cdot 10^{-7}$)
??	`task_b.py --noise 0 -n 25` (*Model has been changed to ridge with lambda*=$1.43 \cdot 10^{-7}$)
9	`task_b.py --file ../data/small_SRTM_data_Norway_1.tif -n 19 --noscale`
10	`task_b.py --file ../data/small_SRTM_data_Norway_1.tif -n 19 --noscale`
??	`task_c.py --noise 0 -n 24`
??	`task_c.py -n 24`
??	`task_c.py -n 24 --step 0.1`
6	`task_e_biasvariance.py -n 24 --step 0.1`
8	`task_f_biasvariance.py -n 9 --step 0.15`
15	`task_e_biasvariance.py --file ../data/small_SRTM_data_Norway_1.tif -n 24`
7	`task_e_betas.py -n 24`
5	`task_c.py -n 24`
4	`noise_plot.py`
3	`task_c.py --noise 0 -n 30 --step 0.01`
11	`task_b.py --file ../data/small_SRTM_data_Norway_1.tif -n 19`
14	`task_c.py --file ../data/tiny_SRTM_data_Norway_1.tif -n 19`
??	`task_c.py --file ../data/small_SRTM_data_Norway_1.tif -n 19`
??	`task_c.py --file ../data/small_SRTM_data_Norway_1.tif -n 19`
12	`task_b.py --file ../data/small_SRTM_data_Norway_1.tif -n 40`
13	`task_b.py --file ../data/small_SRTM_data_Norway_1.tif -n 40`
??	`task_d.py -n 15 --step 0.02`
17	`task_f_msecomparison.py --file ../data/small_SRTM_data_Norway_1.tif -n 9`

[Hjort-Jensen, 2022a] Hjort-Jensen, M. (2022a). Week 39: Optimization and gradient methods. On the WWW. URL `https://compphysics.github.io/MachineLearning/doc/pub/week39/html/week39.html`.

[Hjort-Jensen, 2022b] Hjort-Jensen, M. (2022b). Week 40: Neural networks. On the WWW. URL `https://compphysics.github.io/MachineLearning/doc/pub/week40/html/week40.html`.

[Raschka and Mirjalili, 2019] Raschka, S. and Mirjalili, V. (2019). *Python Machine Learning*. Packt, Birmingham.

Appendix A: Plots and tables

The next pages contain plots and tables that are of interest.

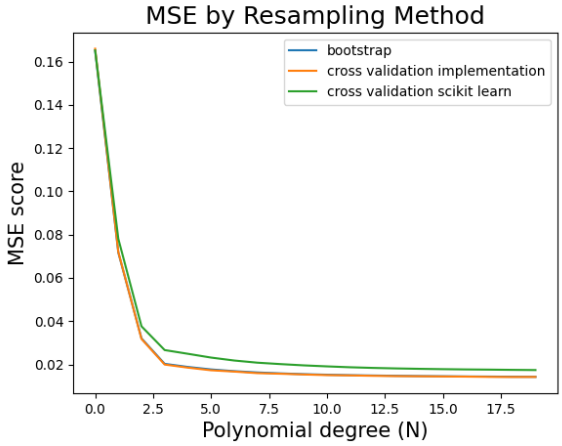

Fig. 2. Comparison between approximated MSE for different resampling methods for the real terrain data

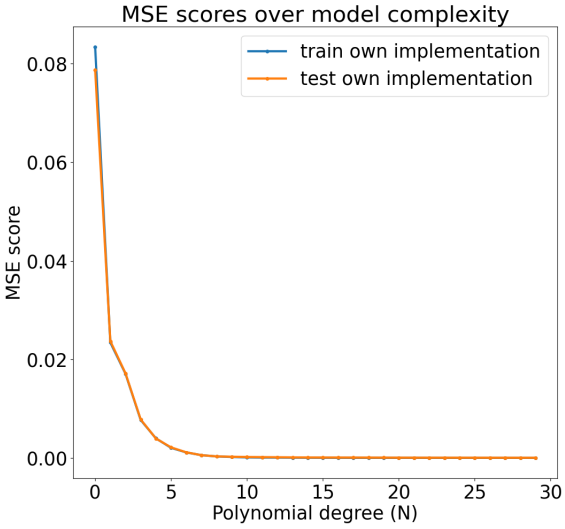

Fig. 3. Franke function fitted up to polynomial degree 30, with 10000 datapoints instead of 400. As we can see, we do not overfit even at this polynomial degree when the availiable data is plentiful. The highest MSE was reached at $N = 29$, and was 0.00003511, one tenth of our best prediction with only 400 datapoints

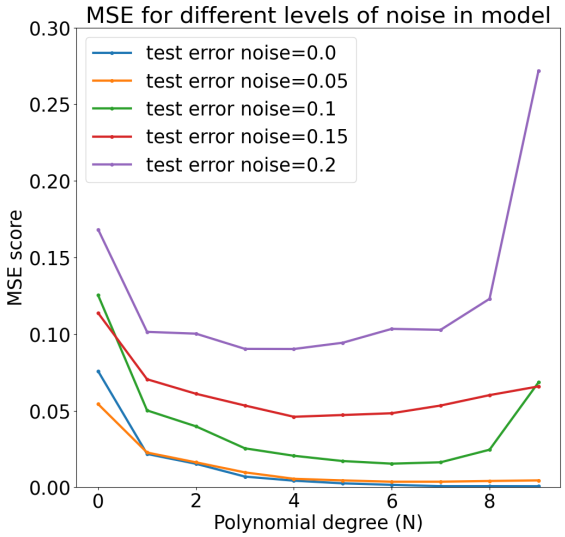

Fig. 4. Test and train MSE for the OLS prediction of Franke function with an increasing amount of noise

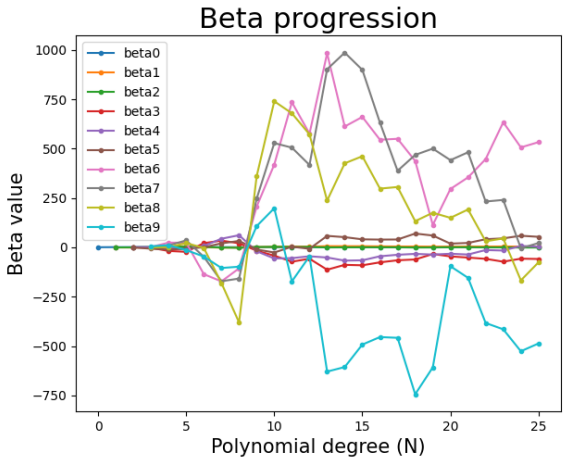

Fig. 5. Beta progression for OLS

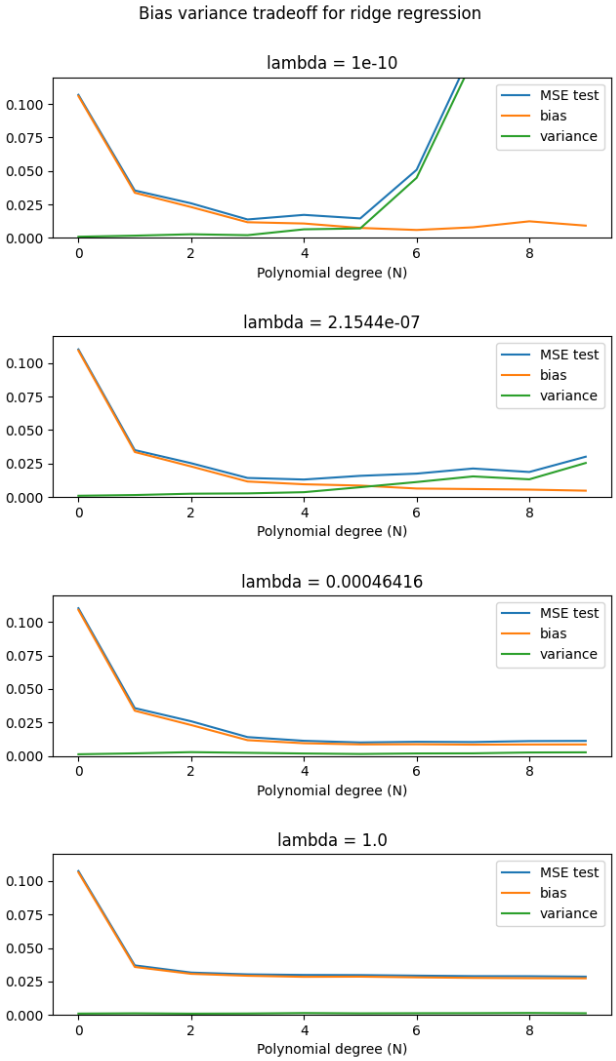

Fig. 6. Bias, variance and test MSE plotted over polynomial degree for our Ridge prediction on the Franke function with an added stochastic noise $\varepsilon \sim N(0, 0.05)$, using lambdas of increasing size. Here we have reduced the number of datapoints to only 100 to force overfitting earlier.

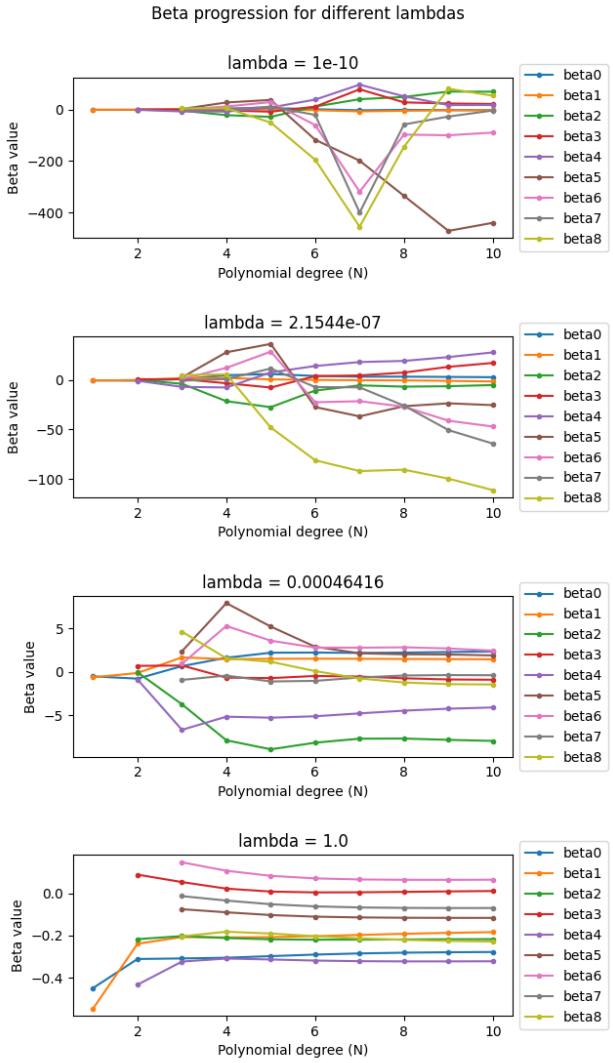

Fig. 7. Beta progression for different values of lambda in Ridge regression. Note the different scales on each of the plots, showing how increasing the regularization parameter shrinks the betas

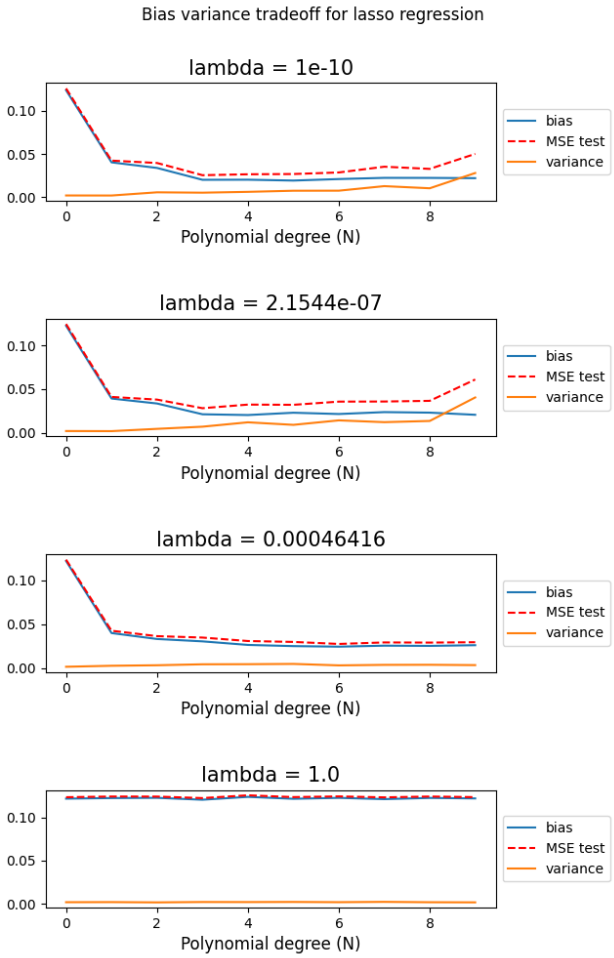

Fig. 8. Bias, variance and test MSE plotted over polynomial degree for our Lasso prediction on the Franke function with an added stochastic noise $\varepsilon \sim N(0, 0.05)$, using lambdas of increasing size. Here we have reduced the number of datapoints to less than 100 to force overfitting earlier.

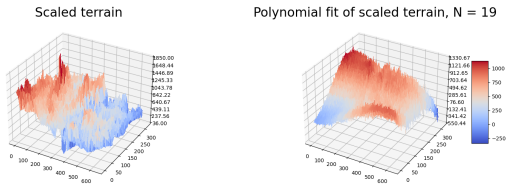

Fig. 9. Our prediction for the real terrain using unscaled data, showing that we reach a very poor result

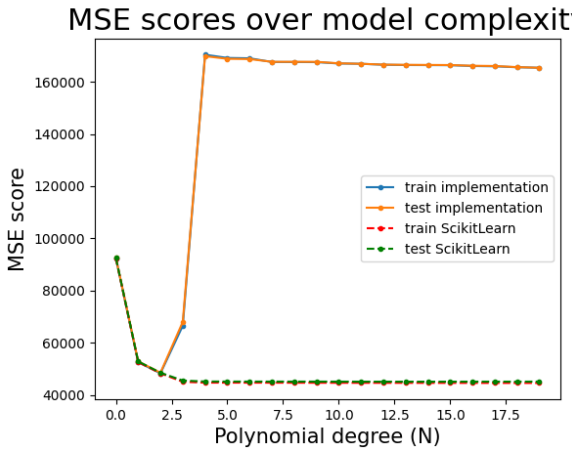

Fig. 10. Test and train MSE for our prediction for the real terrain using unscaled data, showing that we reach a very poor result

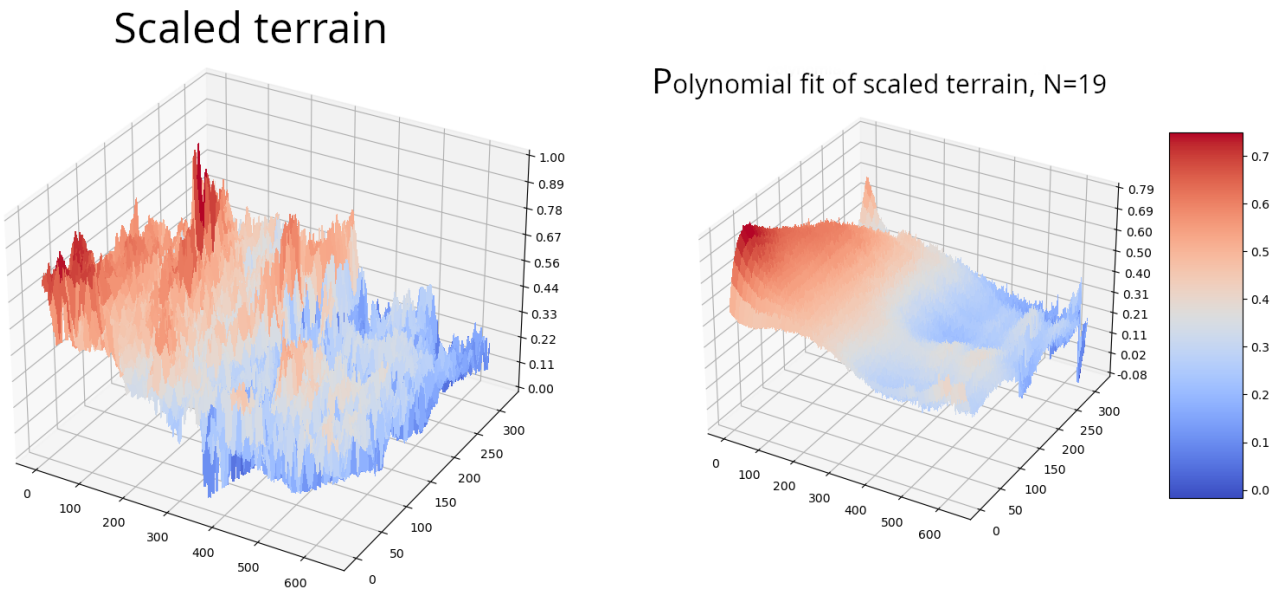

Fig. 11. A comparison of our best OLS prediction and actual terrain data (terrain 1), at polynomial degree 19

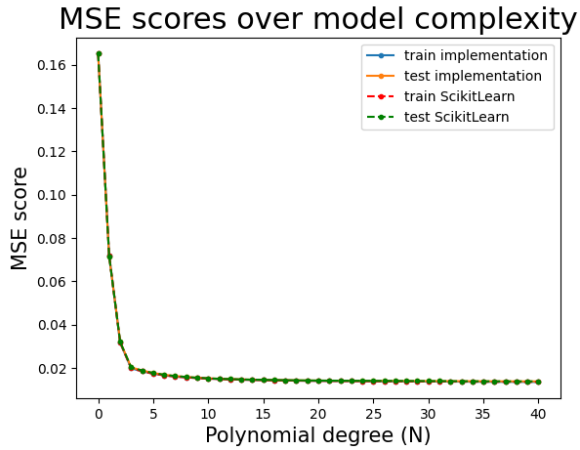

Fig. 12. Test and train MSE on our terrain, this time up to degree 40. We see only a slight decrease in MSE, down to 0.01381, even though we double the amount of degrees. This has not been bootstrapped due to computational limitations

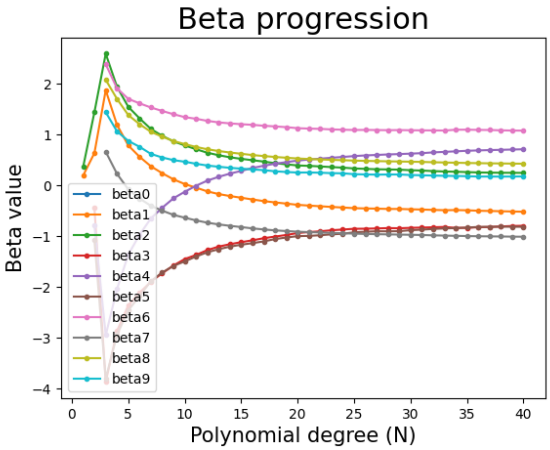

Fig. 13. Beta progression up to polynomial degree 40 for real terrain data, showing that the are relatively constant, implying that we are not overfitting

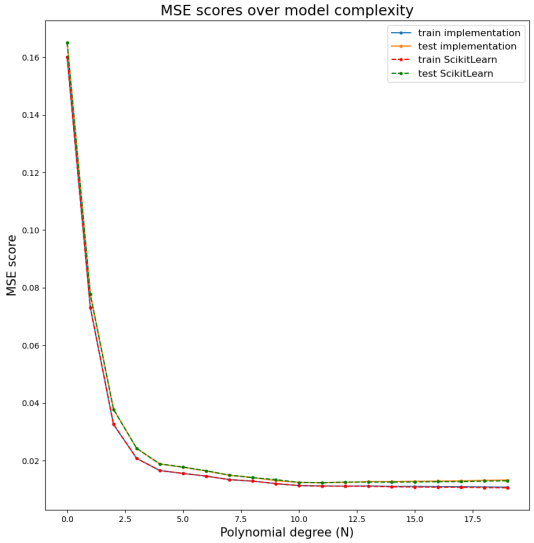

Fig. 14. Test and train MSE for the OLS prediction of terrain 1, this time downscaled to only $50x100$. Here the optimal degree was 11, showing that with lower data we start to overfit more rapidly.

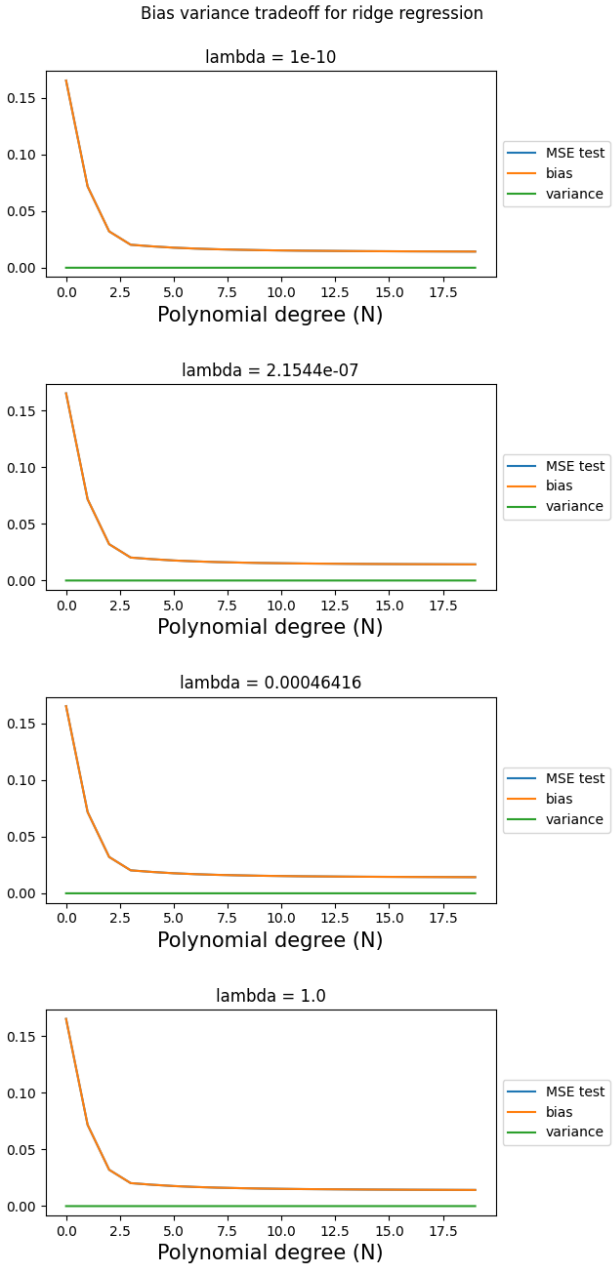

Fig. 15. Bias variance tradeoff for ridge regression with different lambdas, for the real dataset

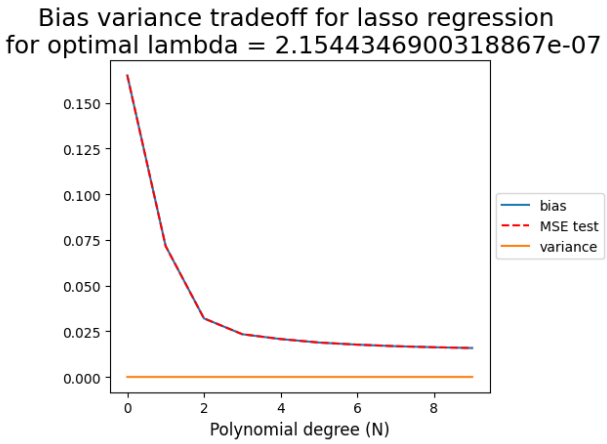

Fig. 16. Bias variance tradeoff for lasso regression with different lambdas, for the real dataset

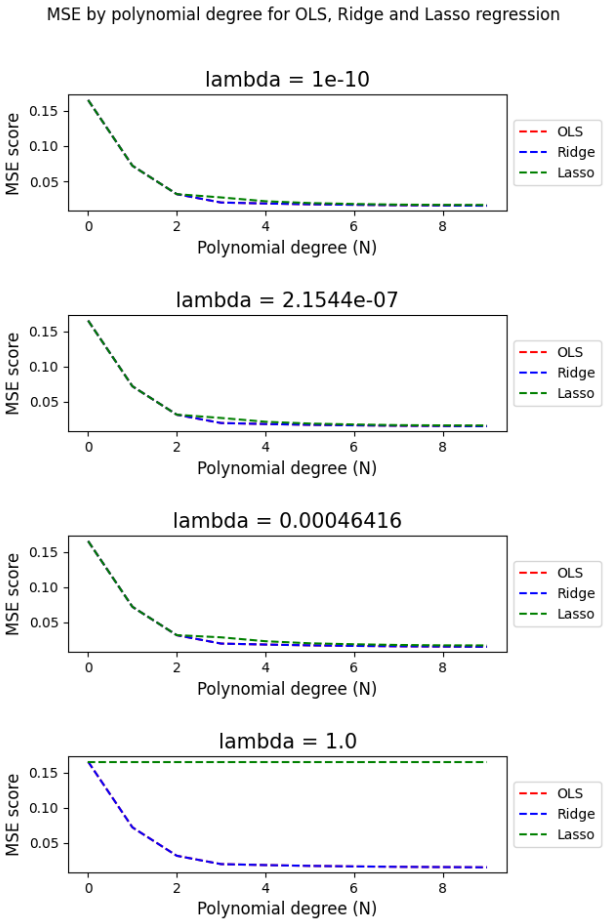

Fig. 17. Comparison between OLS, Ridge and Lasso for different values of lambda, for real terrain data