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

To rigidly test different models for regression and classification against eachother, we developed a flexible Feed Forward Neural Network that can be specified for various architectures. We used this network to test different schedulers, activation functions and model complexities in order to achieve a balance of satisfying results and low computational cost accross different datasets. Pitting the schedulers against eachother to fit a polynomial approximation of the Franke Function using linear regression, Adam and Adagrad came out on top, achieving MSE's that were 33% lower than the other schedulers after 100 epochs[1]. For neural networks problems, Adam proved to be the superior scheduler for both classification and regression. Using this scheduler, we attained a test accuracy of 98.59% on the Wisconsin Breast Cancer Dataset and a test accuracy of 99.07% on the MNIST 8x8 dataset. However, we also found that simple logistic regression performed similarly in terms of accuracy, achieving a test accuracy of 98.76% and 99.54% on the cancer and MNIST dataset, respectively. Moreover, these results came at a much lower computational cost. For regression applied to the Franke Function, we found that our neural net performed worse than a simple polynomial fit using linear regression, attaining a MSE of 0.004781, which is 24% higher than the MSE gained by doing simple polynomial fit using analytical linear regression, which was 0.0038302. We conclude that allthough neural networks are powerful tools which can be used in many contexts, there are many cases in which simpler and less computationally intensive methods can be used to gain similar or even superior performance.

2 Introduction

Many problems in machine learning cannot be solved analytically. Thus, in our search for the optimal solution, we turn to numerical methods. This transition, however, implicitly comes with computationally heavy caveats due to numerical optimization often requiring calcluations of large matrices over many iterations. Finding a solution becomes a waiting game, and finding the many solutions of varying hyperparameters in order to make an informed model selection becomes virtually impossible. This is a problem that lies at the heart of machine learning, as the optimal model selection allows us to make predictions with the greatest possible accuracy. Finding ways around the limitations of numerical methods becomes a must for proper usage of these powerful machine learning tools.

In this paper we will investigate the numerical optimization technique of gradient descent and various techniques used to

[1] Adam, Adagrad and Adagrad with momentum had an average test MSE of 0.0166, while RMS prop, Constant and Momentum had an average test MSE of 0.0250

boost its performance. Thus, we will compare full batch versus stochastic gradient descent, different schedulers such as constant, adaptive and momentum-encanced schedulers, with the aim of decreasing the computational cost of our calculations. We will then study different supervised learning models which utilize gradient descent such as linear regression, logistic regression and a feed forward neural network (FFNN) in order to gain an understanding of model selection across regression, binary- and multi-class classification problems. Furthermore, we will expand our research to include comparisons between different activation functions and model architectures, gaining insights of the different trade-offs each decision entails. Through this, we will attempt to answer the following questions: How complex does our model have to be to achieve satisfying results? Which tradeoffs have to be made to reach these results while balancing the computational effort required to achieve them?

First, we will give an explanation of the methods and theory used in this paper, followed up by a presentation and critical discussion of our results. Finally, we will provide a conclusion summarizing the core results and lessons.

3 Method

3.1 Theory

This sections covers the theoretic background which underlies the methods used in this paper.

3.1.1 Linear Regression

Linear regression aims to find a relationship between a set of dependent and explanatory variables. Assume we have a dataset, which contains a series of inputs $\{x_i\}$, and corresponding target values $\{y_i\}$. Our assumption is that there is some relationship $y_i = f(x_i) + \varepsilon_i$ where $\varepsilon_i \sim N(0, \sigma^2)$, and we wish to create a model which approximates this relationship. More specifically, we wish to find a vector β which gives us the "best" approximation $\tilde{\boldsymbol{y}}$ of $\boldsymbol{y}$, by using the parameters in the following equation:

$$X\boldsymbol{\beta} = \tilde{\boldsymbol{y}} \tag{1}$$

Here, X is our design matrix, containing our explanatory variables, our features, while $\tilde{\boldsymbol{y}}$ is our predicted vector. By defining a cost function, such as Mean Squared Error, we are able to find the parameter β which minimizes the difference between our prediction and the target values. By using the normal equations, we can solve this problem analytically.

3.1.2 Logistic Regression

Logistic regression 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). However, this method is not limited to binary classification, as the substitution of the activation function for SoftMax can allow for multiclass classification.

3.1.3 Gradient Descent

In linear regression problems, the optimal solution can be found analytically. 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 rely on 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 function representing our cost until a local optimum is reached. The gradient itself is nothing more than a first order derivative of a given function with respect to all its parameters, which points in the direction of the greatest change.

The application of gradient descent to our problems as a means of finding the ideal solution implies that the parameters used in our prediction be initialized randomly. The most common approach is to set the parameters according 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:

$$\frac{\partial C}{\partial w_{jk}} = \frac{\partial C}{\partial a_j}\frac{\partial a_j}{\partial w_{jk}}. \tag{2}$$

Although gradient descent 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 the standard gradient descent 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, SGD has proven to give a large computational advantage over full batch gradient descent. Furthermore, the stochastic nature of this method can allow SGD to avoid getting stuck in local minima [Raschka and Mirjalili, 2019, 46].

3.1.4 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, meaning it keeps a sort of memory of the previous iterations. This hyperparameter 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 [Hjorth-Jensen, 2022a]. Examples of schedulers using the second moment are RMS-prop, Adam and Adagrad. The mathematical formulas for these scheduling algorithms can be found in Hjorth-Jensens lecture [Hjorth-Jensen, 2022a].

3.1.5 Feedforward Neural Networks

For some problems, 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, or 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.

$$a_i^l = f^l(z_i^l) = f^l\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l a_j^{l-1} + b_i^l\right). \tag{3}$$

3.1.6 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:

$$y = \frac{1}{1+exp(x)}. \tag{4}$$

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:

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

$$LRELU(x) = \begin{cases} -\alpha \cdot x & if \ x < 0, \\ x & if \ x \geq 0. \end{cases} \tag{6}$$

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].

3.1.7 Backpropagation

As in the case of 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 [Hjorth-Jensen, 2022b]:

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

$$y_i^l = f^l(u_i^l) = f^l\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l y_j^{l-1} + b_i^l\right). \tag{7}$$

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

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l). \tag{8}$$

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

$$w^l_{jk} \leftarrow= w^l_{jk} - \eta \delta^l_j a^{l-1}_k \tag{9}$$

$$b^l_j \leftarrow b^l_j - \eta \frac{\partial C}{\partial b^l_j} = b^l_j - \eta \delta^l_j. \tag{10}$$

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

3.2 Implementation

Our implementation can be found at `github.com/EricEReber/FFNN_optimization.git`.

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`. This unity in implementation ensures that we can directly compare metrics obtained from the different models we are comparing, e.g neural network versus logistic regression. 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 Hjorth-Jensens lectures [Hjorth-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 average all the gradients for each weight matrix together and feed this into our scheduling calculation to determine how much our weights should change.

Training of 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 Hjorth-Jensens lectures [Hjorth-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.

For verification that our implementation is working correctly, we train on a simple second degree polynomial, and see that our predicted y value is very close to the actual y value[2]. See Fig. 1.

3.3 Datasets

Below is a short description of each dataset used in this report, along with a table (Table. 1) listing their characteristics.

3.3.1 Franke Function

The Franke Function is a simple weighted sum of exponentials, taking two inputs and returning a real value. It is commonly used to test machine learning algorithms for regression problems. For our linear regression problems, our training data will be based on a 9th degree polynomial fit of the Franke Function, with 56 features. For the feed forward neural network, we will use the x and y values directly. The Franke Function is defined as follows.

$$f(x,y) = \frac{3}{4}e^{\left(-\frac{(9x-2)^2}{4}-\frac{(9y-2)^2}{4}\right)} + \frac{3}{4}e^{\left(-\frac{(9x+1)^2}{49}-\frac{(9y+1)}{10}\right)} + \frac{1}{2}e^{\left(-\frac{(9x-7)^2}{4}-\frac{(9y-3)^2}{4}\right)} - \frac{1}{5}e^{\left(-(9x-4)^2-(9y-7)^2\right)}. \tag{11}$$

[2] see `src/verification.py`

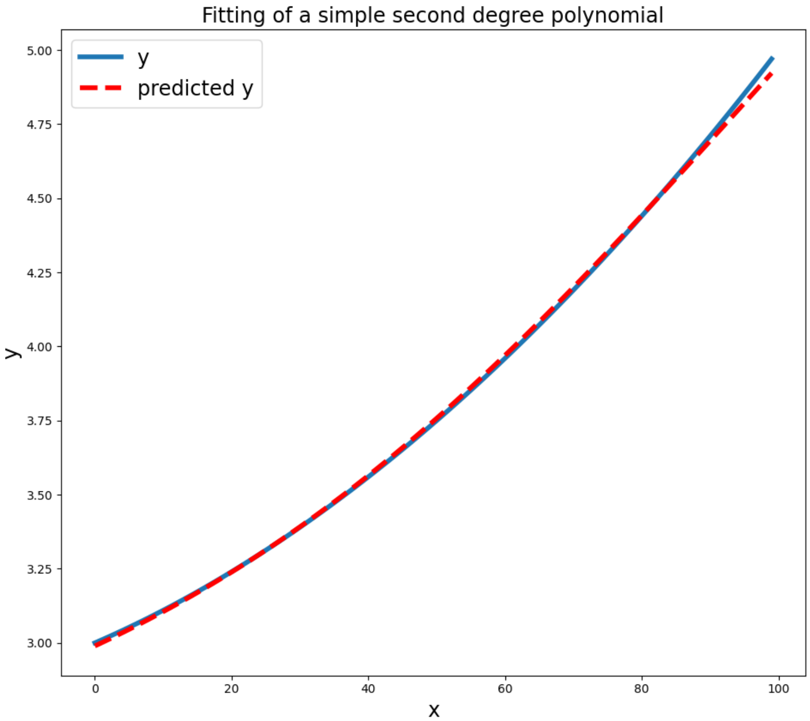

Fig. 1. Simple verification case

In practice, to better test our models ability to generalize and to mimic real terrain data, we add a normally distributed noise $N(0, 0.05)$ to this function.

3.3.2 Wisconsin Breast Cancer Dataset

This dataset contains 30 features describing important characteristics of tumors, where each feature is a real numbers. Each instance has an associated class, indicating whether the tumor was benign or not. This leads to a binary classification problem.

3.3.3 MNIST 8x8

This dataset contains a preprocessed set of black and white images of hand drawn digits. The images are 8 by 8 pixels, leading to 64 features, where each feature is an integer in the range $[0-16]$ describing the intensity of the pixel at that position[3]. Each image has an associated class, which is simply the digit depicted. This leads to a multiclass classification problem with 10 classes.

Table 1. Characteristics of each dataset used in the report

Dataset	Number of instances	Features	Type
Franke Function (raw data)	400	2	regression
Franke Function (polynomial features)	400	56	regression
Wisconsin Breast Cancer	569	30	binary classification
MNIST 8x8	1797	64	multiclass classification (10 classes)

[3] in reality, this describes the number of ”on” pixels in a 4x4 region in the original image

4 Results

4.1 Different schedulers used with linear regression

In order to study the attributes of the different scheduler methods, we performed numerical linear regression using gradient descent on the Franke Function with 56 polynomial features. This simple problem will allow us to gain an intuition of how the different schedulers compare against eachother. From Fig. 2, we can observe that adding momentum improves upon the solutions found by the constant scheduler considerably, reaching an MSE of 0.09 for the test data after 200 epochs. By adding a memory of the previous gradients, we gain speed in the direction where most gradients point, speeding up our descent. This allows us to obtain a lower MSE scores in fever epochs, when compared to the constant scheduler's MSE of 0.21.

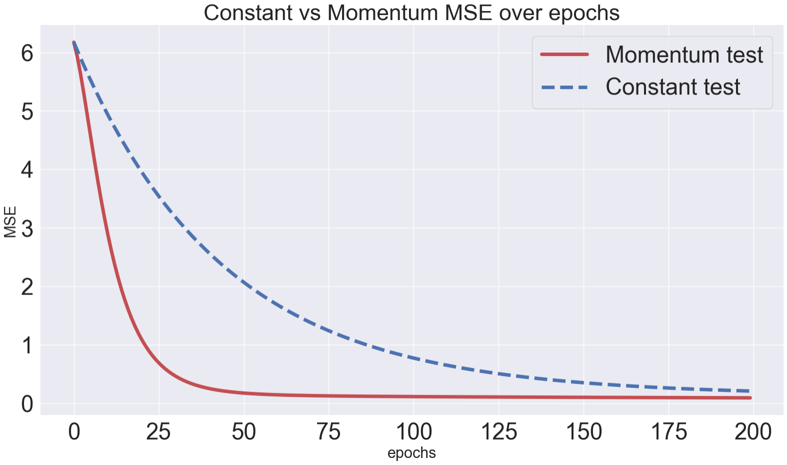

Fig. 2. Constant vs momentum MSE for test data plotted for 200 epochs using full batch gradient descent. The final MSE for Constant was 0.21, considerably higher than Momentum's 0.09.

By introducing mini-batch gradient descent, we can observe how decreasing the batch size improves our performance further (Fig. 3). Mini-batch gradient descent clearly yields a lower MSE for the test data than full batch gradient descent for the same amount of epochs. The best result for both constant and momentum were reached for a batch size equal to 2, close to a fully stochastic gradient descent. The test data MSE reached 0.044, an impressive improvement over constant's previous result of 0.198 MSE when using full batch gradient descent. Because we first gridsearched the optimal values of eta and lambda with a predefined base batch size, we must acknowledge that these results will have some bias towards the batch size used when searching for the optimal hyperparameters. However, given that we have gridsearched with different base batch sizes accross our testing, we have found the results to be similar accross different base batch sizes, and that this bias which we induced to save computational time was insignificant. A full comparison of MSE over batch sizes for all the different schedulers can be viewed in Figure 13 in the appendix, where we can see that all schedulers yield increasingly better results as the batch size decreases.

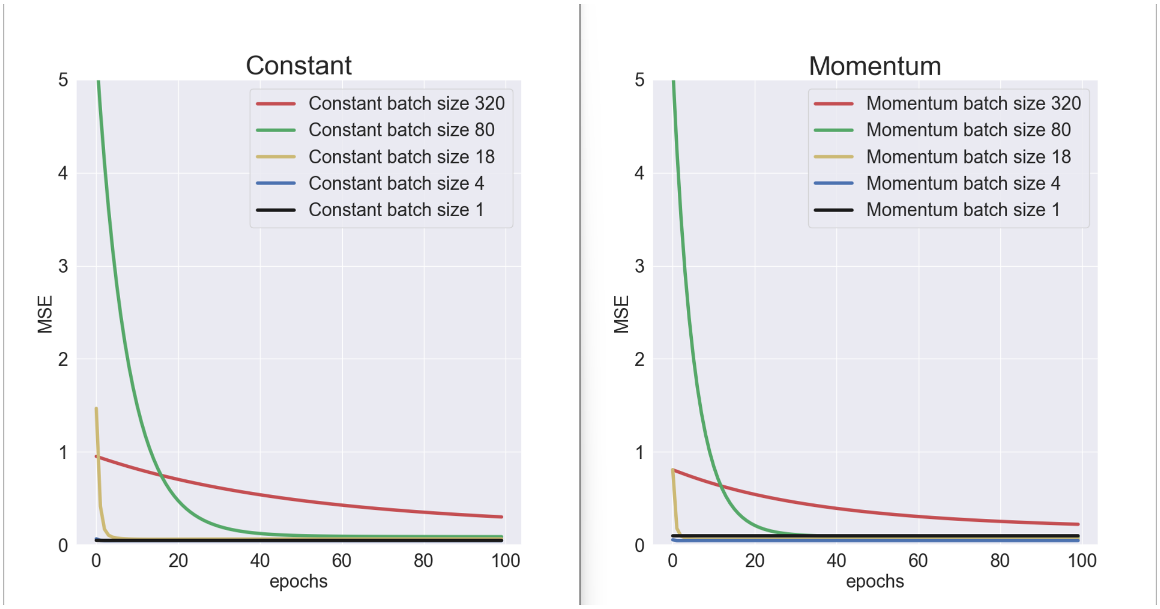

Fig. 3. Constant vs momentum test MSE for different batch sizes, plotted for 200 epochs using full batch gradient descent. A batch size of 320 is full batch GD, whilst a batch size of 1 is fully stochastic GD.

Furthermore, with the introduction of various adaptive scheduler methods such as RMS prop, Adagrad (with and without momentum) and Adam, we can observe from Fig. 4 that we can reach even greater results in fewer epochs. In order to get a better understanding of the computational cost we have also compared the time it took to fit the data to an MSE of 0.03. Looking at Table. 2, RMS prop took the least time, reaching the target MSE over 5 times faster than our constant scheduler. If we take a look at figure 4, we can see that RMS_prop has a fast descent in the first iterations, however, it struggles in the later parts, reaching MSE values that are much higher than Adam and Adagrad.

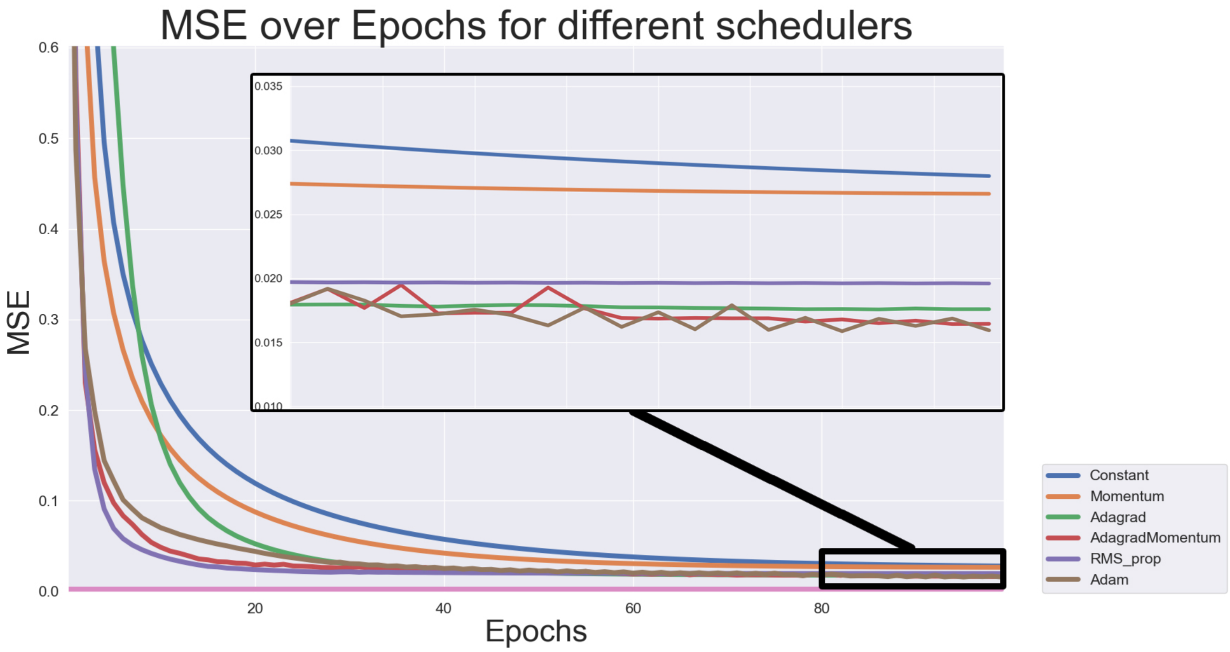

Fig. 4. Test MSE over epochs for all schedulers

Table 2. Table over time take for each scheduler

Scheduler	Time to reach 0.2 test MSE
Constant	0.489
Momentum	0.379
RMS_prop	0.089
AdaGrad	0.205
AdaGrad with Momentum	0.135
Adam	0.209

Moving over to Fig. 5, we gain interesting insight into the behavior of our schedulers, with all of them preferring rather high learning rates and mid to low values of regularization. This plot was generated for 100 epochs due to the computational complexity of gridsearching for these parameters for all six schedulers. The observable bias towards higher values of learning rates is certainly a side effect of the limiting the number of epochs, as this short interval favours parameters which allow the schedulers to reach low MSEs fast, even if it might lead to instabilities in later iterations. These high learning rates coupled with values of lambda higher than zero, indicate that Ridge regression systematically outperforms ordinary least squares. We can see that the addition of a momentum term makes a small difference for both the constant scheduler and Adagrad. On the other hand, we see that adaptively changing the learning rate, like Adagrad, RMS prop and Adam does, has a much bigger impact on the final MSE than the momentum term.

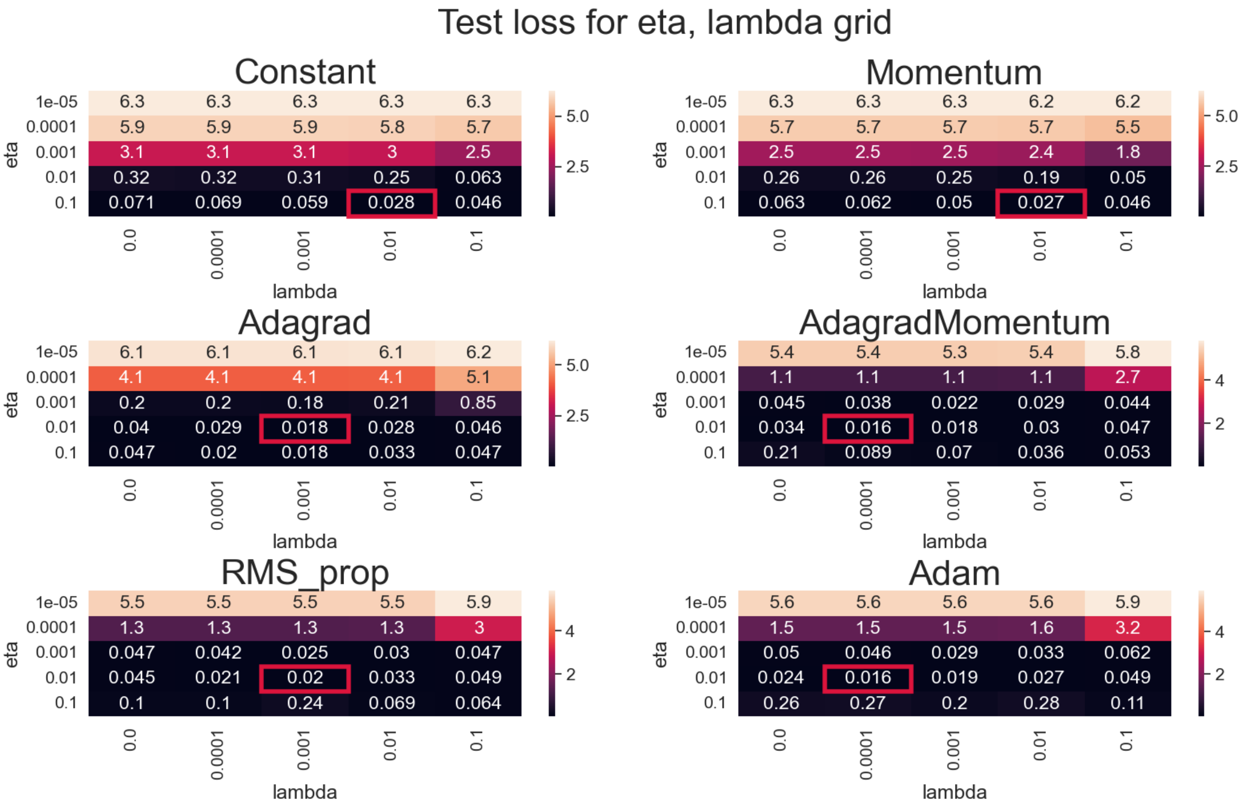

Fig. 5. Heatmap for each scheduler, showing the optimal lambda and eta for each method, as well as the best MSE reached (highlighted in red)

Based on these findings, we decided to push Adagrad and Adam further, allowing them to run for much longer. After 5000 epochs, we reached an MSE of 0.012 for Adam and 0.011 for Adagrad. The plot for the MSE over epochs can be found in the appendix (Fig. 14). Our models have also been compared to SciKit's implementations (Fig. 15).

4.2 Neural Network for regression

Having gained an understanding of how the different schedulers function, and having created the necessary infrastructure to study our models' performance based on a number of different factors, we use these methods to optimize our neural network for linear regression. More specifically, we attempt to fit the Franke function. First, we perform a preliminary test on MSE over model complexity. For comparisons sake, we use the same scheduler for each architecture, and fix the regularization parameter at 0. By fixing the regularization parameter at 0, we are of course favouring less complex models, but this a good way to get a general sense of when the model starts to overfit. Using the sigmoid activation function, we can study the MSE as a function of the total hidden node count (Fig. 6). Here we see, somewhat surprisingly, that our train and test MSE are not deviating particularly much from eachother, even with very complex networks. There are many reasons that this may be the case. As we have stated before, once we start using numerical methods, computational effort becomes a large factor in any test, and many decisions have to made regarding how much time can be spent on each test. Due to these computational constraints, we have not been able to grid search for the best parameters over each model complexity, nor have we been able to train for more than 200 epochs for each model. These shortcomings greatly affect how our models perform, as it is unlikely that each model uses the same learning rate and scheduler, or that more complex models use the same amount of epochs to train their weights. Based on this, we assume that the higher complexity models have not been trained enough, and that this explains why both their train and test MSE are higher than the less complex models. Regardless, it seems that for up to about 200 total nodes, three hidden layers does the best, and as such we chose an architecture of three hidden layers with 66 nodes each (which is where we achieved our lowest MSE).

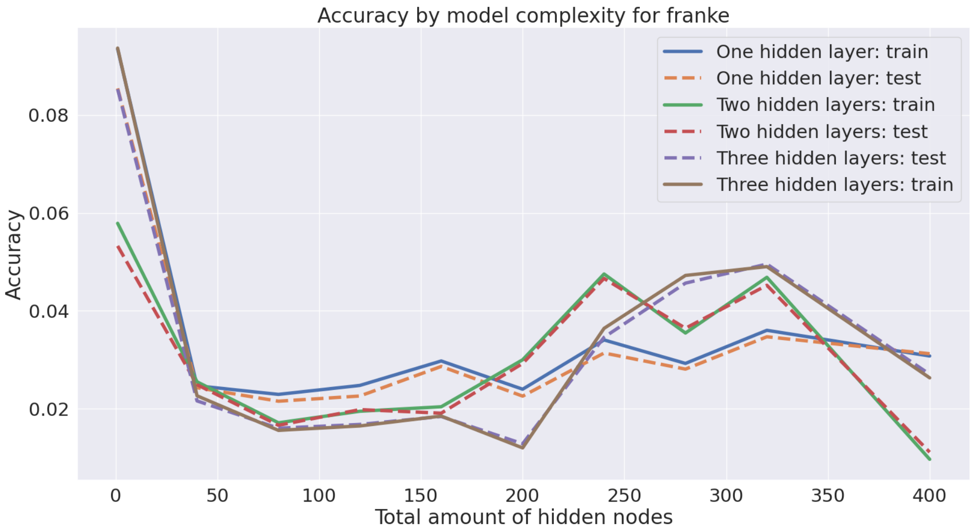

Fig. 6. Comparison between a neural net with one hidden layer, and one with two hidden layers, for different totals of hidden nodes. For the network with two hidden layers, the total is evenly divided between the layers, such that at $x = 60$ we have a network with two hidden layers of 30 nodes each. In this case, the hidden layers used the sigmoid activation function.

Using this architecture, we performed a gridsearch. Again, due to computational constraints, we have felt it necessary to limit our gridsearch to only two schedulers. Based on our testing of linear regression with gradient descent, we have seen that Adam and Adagrad seems to be the best performers in terms of MSE[4]. Following Goodfellows recommendations, we started with the RELU activation function [Goodfellow et al., 2016, 188]. With this, we found Adam to be the best scheduler, with an eta of 0.01 and no regularization. However, when using these parameters, we encountered the so called ”dying RELU problem”, where too many of our RELU nodes output zero, leading to our gradient descent slowing down and eventually stopping completely. A better weight initialization, for example using Xavier Initilization, could solve this problem, but another way to circumvent this issue is to use Leaky RELU. This activation function has a small but non-zero gradient for negative values, instead of zero as with RELU. Gridsearching with Leaky RELU instead gave the exact same parameters.

Using these parameters along with Adam and LRELU, we acheived an MSE of 0.004781, which is better than the results gained using gradient descent on our polynomial fit. We can see that by not forcing our model to try and find a polynomial

[4] Adagrad with momentum was discarded even though it performed well, due to the gridsearch for this method involving the momentum parameter as well, making the gridsearch take much longer than the other methods

fit, we are achieving better results. Unfortunately, we are still unable to beat the analytical solution to the best polynomial fit at polynomial degree 9, which is somewhat disappointing. A possible explanation for this is that the Franke function can be represented as a weighted sum of polynomials, and as such a polynomial fit is especially suited to approximate this function. Our neural net must learn the relation between the inputs and outputs on its own, a process which is computationally intensive and difficult. Looking at our predicted terrain (Fig. 7), we can see that our prediction is somewhat reasonable, but not entirely convincing. Furthermore, we can see that the polynomial fit seems to match the actual terrain more closely, which it achieves in a fraction of the runtime.

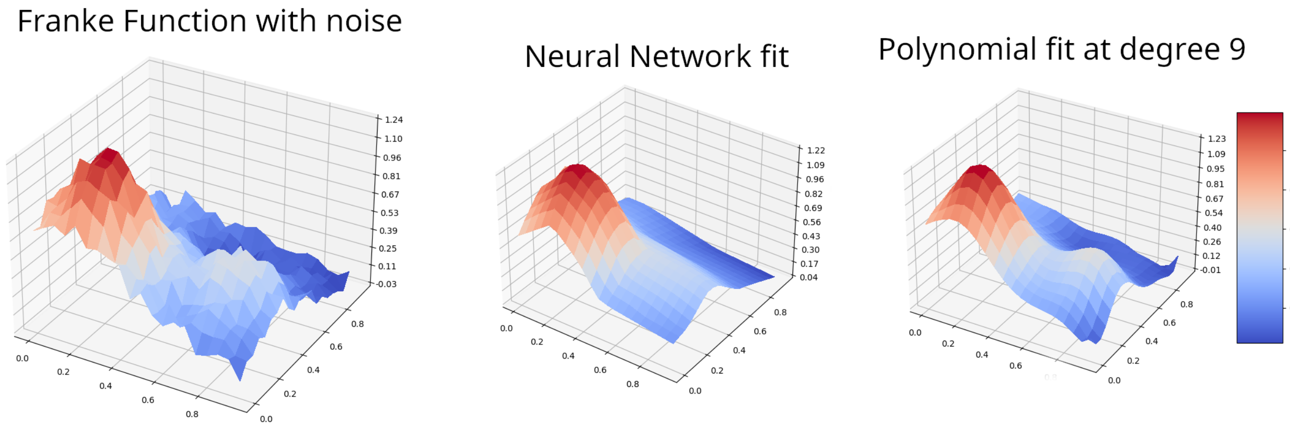

Fig. 7. Surface plot of the actual terrain created by the Franke function, the best fit achieved by our Neural Network, and the best polynomial fit for polynomial degree 9

4.3 Neural Network for binary and multiclass classification

With mediocre results on regression, we turn to classification. To test our neural networks ability to solve classification problems, we first applied it to the Wisconsin Breast Cancer dataset. As with regression, in order to achieve satisfying results, the hyperparameters of our model need to be tuned, which requires us to consider the tradeoff between performance and computational complexity. We start by trying to find a good model complexity, i.e the best number of hidden layers, and the amount of nodes in them. To do this, we plotted the accuracy over different model architectures. Like we did during regression, we fix all the other parameters and study the architecture by itself. Using the sigmoid activation function (Fig. 8), our results are somewhat surprising. We see that even at 400 total hidden nodes, we are not overfitting. This implies that the dataset is quite homogenous, and that the test set is very similar to our training set. With such a dataset, it is possible to gain a very good fit using more and more complex models, although the computational effort obviously increases as well. Moreover, it is not given that instances outside this dataset are as homogenous as the relatively small (569) sample we have here, meaning one should be careful to use an overcomplex model even if it does not hurt the test accuracy for this particular dataset. Using the RELU activation function, the results are quite different, as can be seen in Fig. 9. Here we can observe that a neural net with two or three hidden layers starts performing very inconsistently when the number of nodes increases. Looking more closely at the training process, we can see that training stops completely in some of our crossvalidations. Clearly, we are again running into the ”dying RELU problem”. Here we can clearly see some of the benefits of the sigmoid, as even when the gradients become very small, they will never be zero, as with RELU. Regardless, we see similar performance for RELU using only one hidden layer. Based on these findings, we decided to use a model with a single hidden layer of 100 nodes, using the RELU activation function.

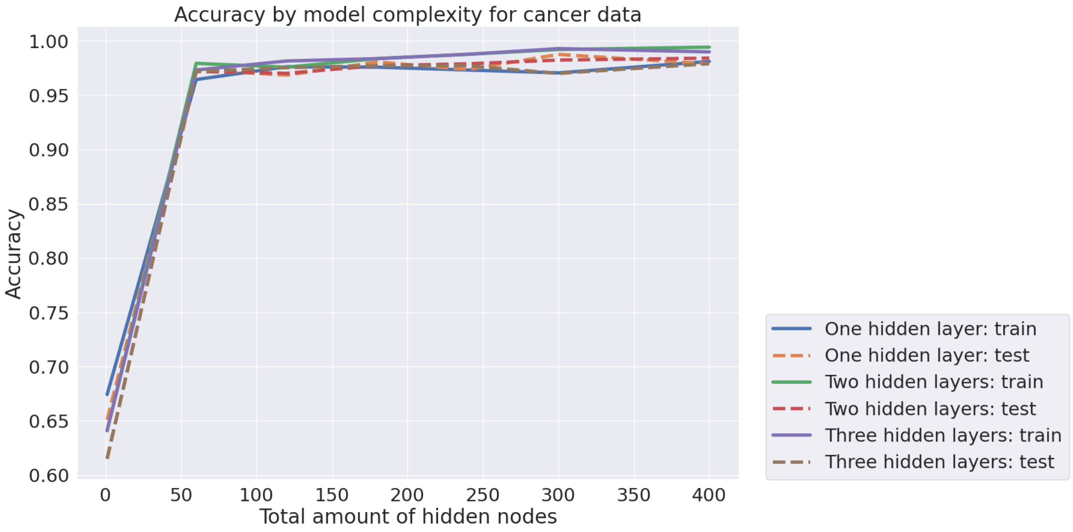

Fig. 8. Comparison between a neural net with one hidden layer, and one with two hidden layers, for different totals of hidden nodes. For the network with two hidden layers, the total is evenly divided between the layers, such that at $x = 60$ we have a network with two hidden layers of 30 nodes each. In this case, the hidden layers used the sigmoid activation function.

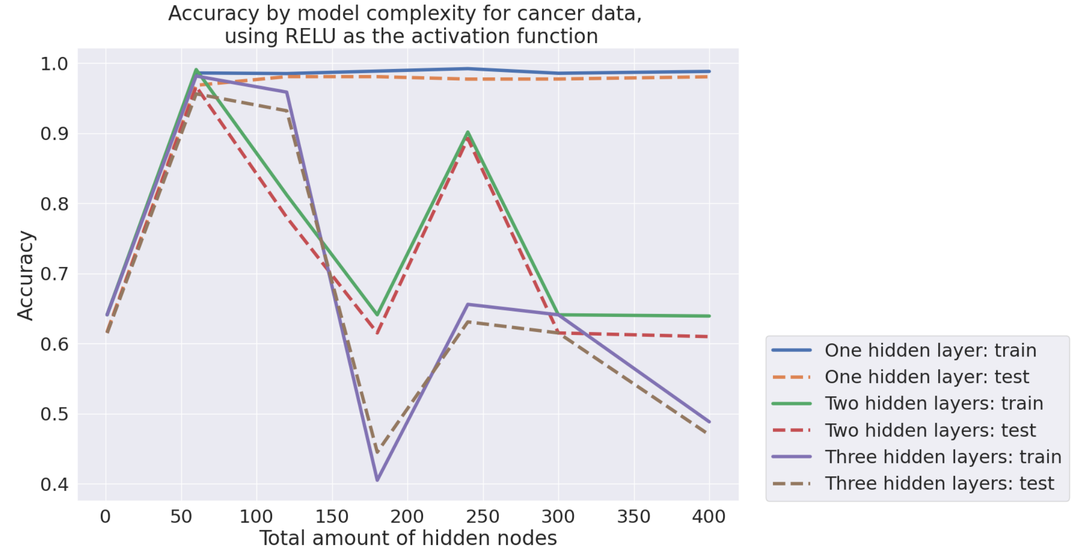

Fig. 9. Comparison between a neural net with one hidden layer, and one with two hidden layers, for different totals of hidden nodes. For the network with two hidden layers, the total is evenly divided between the layers, such that at $x = 60$ we have a network with two hidden layers of 30 nodes each. In this case, the hidden layers used the RELU activation function.

Using our gridsearch method on this model, we found that the best scheduler was Adam, with an eta of 0.001 and a lambda of 0.0001. With this combination of architecture and scheduler, we were able to achieve a crossvalidated test accuracy of 98.59%, which is a great result. This outperforms SciKit's MLP classifier, which achieved a crossvalidated test accuracy of around 95%. Looking at the confusion matrix (Fig. 10), we see that we are achieving good predictions for both true positives and true negatives.

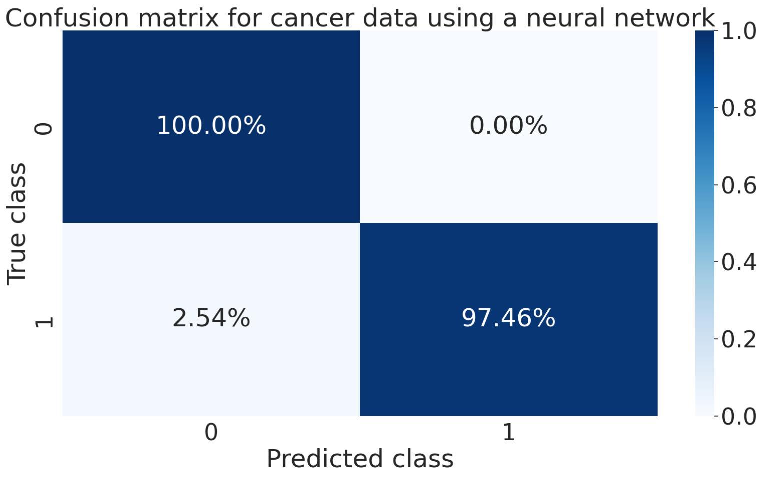

Fig. 10. Confusion matrix for the cancer dataset using a neural net with a single hidden layer of 60 nodes

Confident in our networks's ability to do binary classification, we turned to the MNIST dataset. Here, we perform an educated guess on the different parameters, based on our previous testing. We decided upon an architecture of a single layer with 100 nodes using the RELU activation function. For our scheduler, we used Adam with an eta of 0.001 and no regularization. Being a multiclass problem with ten different classes, one might expect our neural network to struggle more with this dataset. However, this was not the case, as we achieved a crossvalidated test accuracy of 99.07%. Based on these results, we conclude that the initial guess for our model was a good one, and that further optimization has little room for improvement. From these two datasets, we can see that a neural network is adept at solving classification problems. On the other hand, these good results, and the ease with which our network attained them, could imply that the problems we have dealt with are not as complicated as they may seem.

4.4 Logistic Regression

Based on our good results with the neural net, we attempt to solve the same problems using simple logistic regression. Using logistic regression on the cancer dataset, we found through our gridsearch that Adam with an eta 0.1 gave the best results. Here we found that logistic regression performed essentially equivalent to our neural net, achieving a test accuracy of 98.76%. Looking at Fig. 11, we see that logistic regression keeps up and even surpasses our neural network in accuracy over epochs. The value of logistic regression becomes even greater when we consider the computational cost of each method. In this case, the logistic regression was about 65% faster[5], with results surpassing our neural network (see Fig. 12). In the MNIST dataset, logistic regression also performed well, achieving an accuracy of 99.54%. This implies that our classification datasets were perhaps too simple, and that using Neural Networks was not needed to achieve good results.

[5] The time taken to complete a crossvalidation with 5 folds up to 200 epochs was about 52 seconds for our neural network, but only 18 seconds for logistic regression

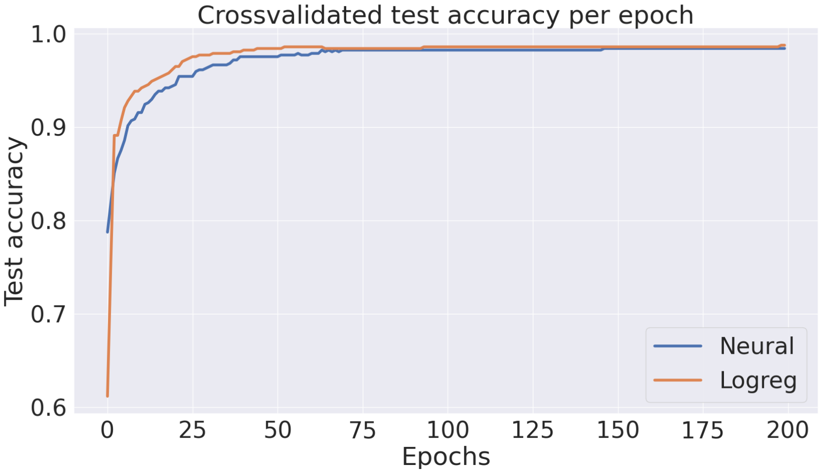

Fig. 11. Comparison between the test accuracy of logistic regression and a neural network on classifying the cancer dataset

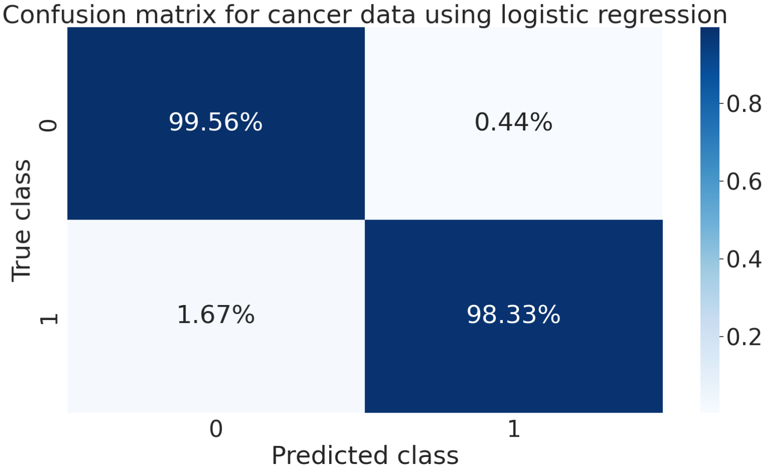

Fig. 12. Confusion matrix for the cancer dataset using logistic regression

5 Conclusion

In this paper we have studied supervised learning algorithms used for classification and regression problems, using the numerical optimization technique of gradient descent as a means of finding an optimal solution. We have observed how different schedulers improve the performance of our gradient descent, and found that amongst all the schedulers we tested, Adam and AdaGrad exhibited the best performance overall, achieving MSE's that were 32% lower than the other schedulers in our testing. Using these schedulers, we solved regression and classification problems with our neural network implementation, but found that it underperformed compared to simpler models. For regression, our network achieved an MSE of 0.004781, 24% higher than a regular polynomial fit using the normal equations. In classification, our network achieved accuracies above 98% for both the Wisconsin Breast Cancer and MNIST datasets, but regular logistic regression also achieved accuracies above 98%, in a much shorter time. To build upon the resarch conducted in this paper, one avenue to explore is to use larger and more complicated datasets, or to use different forms of neural networks such as convolutional neural networks.

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.

[Chandra, 2019] Chandra, A. L. (2019). Learning parameters part 5. URL `https://towardsdatascience.com/learning-parameters-part-5-65a2f3583f7d`.

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

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

[Hjorth-Jensen, 2022b] Hjorth-Jensen, M. (2022b). Week 40: Neural networks. 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.

Table 3. This table shows the file which created every plot in the report

Figure	Shell command
1	`python3 verification.py`
2	`python3 constant_v_momentum.py`
3	`python3 constant_v_momentum.py`
5	`python3 task_a.py`
6	`python3 bycomplexityfranke.py`
7	`python3 task_b.py`
8	`python3 bycomplexitycancer.py`
9	`python3 bycomplexitycancer.py`
10	`python3 task_d.py`
11	`python3 task_d.py`
12	`python3 task_d.py`
13	`python3 task_a.py`
14	`python3 linreg_franke.py`
15	`python3 scikit_compare.py`

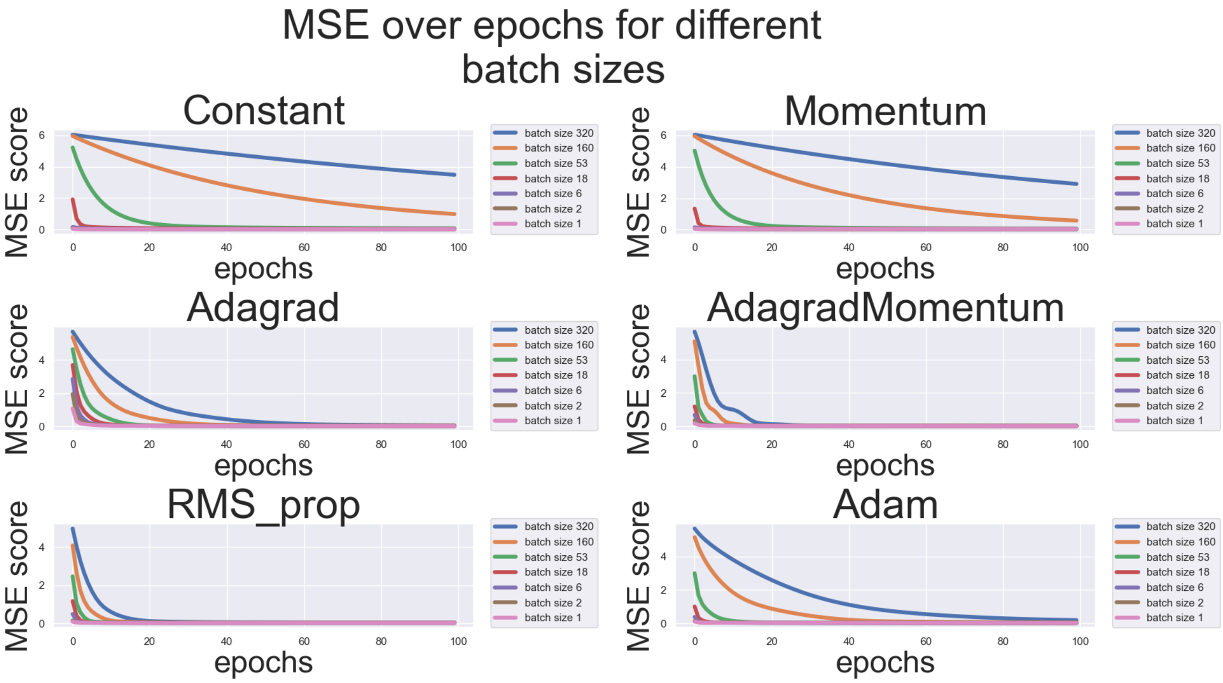

Fig. 13. MSE over epochs for different batch sizes, for all different schedulers

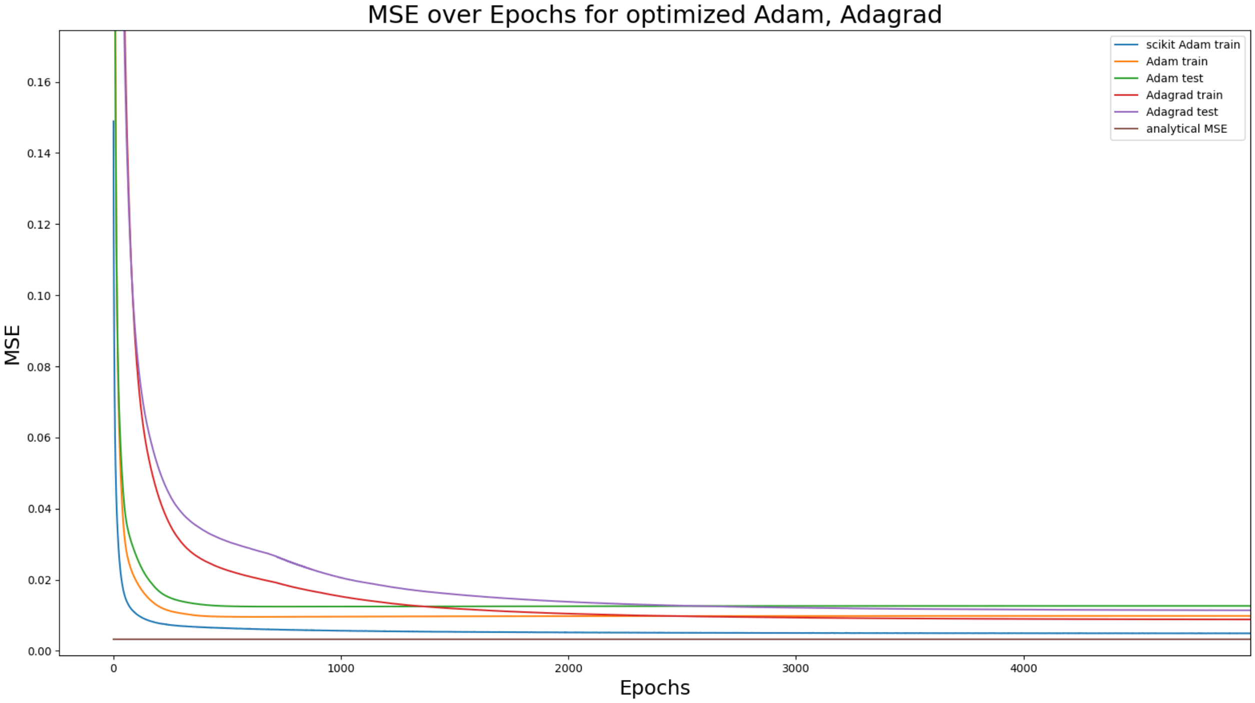

Fig. 14. MSE test error over 5000 epochs for Adam and Adagrad schedulers

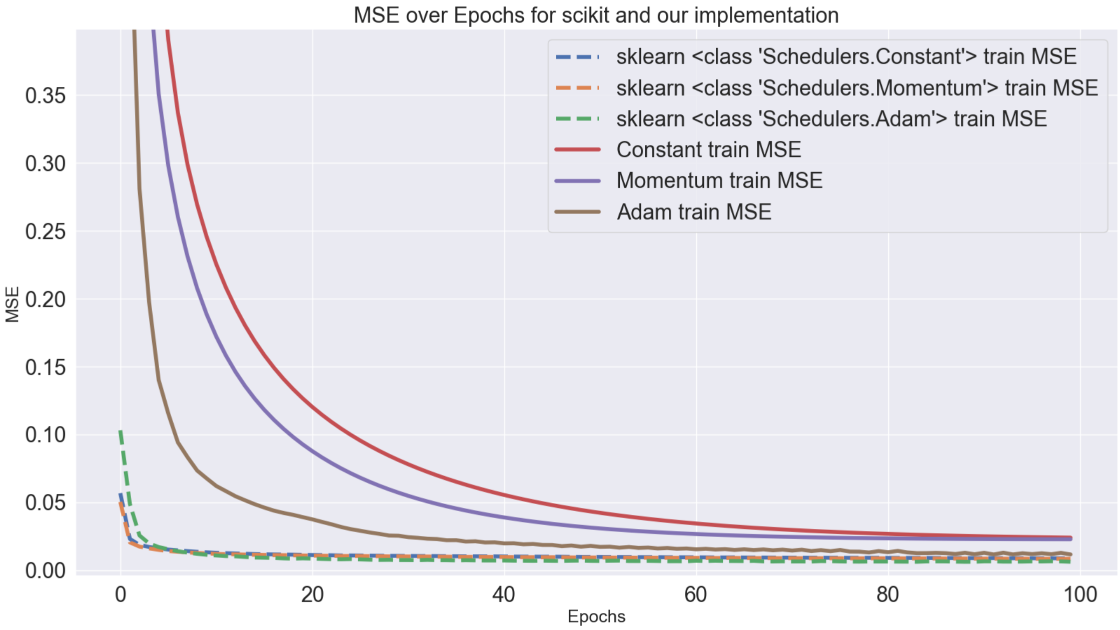

Fig. 15. Comparison between our own implementation of Adam and Adagrad with SciKit's implementation