Project 2

Edvard B. Rørnes* and Isak O. Rukan†
Institute of Physics, University of Oslo,
0371 Oslo, Norway
(Dated: November 4, 2024)

In recent years, neural networks have become increasingly important for advancements in various scientific fields. In this report, we develop a program[a] that consists of both linear and logistic regression methods and a feed-forward neural network. In particular, the program implements gradient descent, stochastic gradient descent, and a simple feed-forward neural network with backpropagation. The neural network is evaluated against linear regression on the Franke function and against logistic regression using sklearn's breast cancer data, where it identifies malignant tumors. We explore three activation functions: Sigmoid, ReLU, and LeakyReLU. The analysis involves tuning the number of epochs, hidden nodes, and the hyperparameters λ (regularization) and η (learning rate) to optimal values. On the Franke function, our own implementation of the neural network achieved a peak R^2-score of x and y respectively. For the cancer data, the highest accuracies achieved with ReLU, , and Sigmoid activation functions were $97.9\%, 98.6\%$, and 97.9%, respectively, compared to $z\%$ for logistic regression. Notably, both the Sigmoid and LeakyReLU activation functions exhibited minimal sensitivity to variations in the hyperparameter λ at optimal learning rates, whereas ReLU showed significant responsiveness. Overall, while LeakyReLU yielded the best performance across all tests, its improvement over the other activation functions was marginal when applied to the cancer dataset, but considerate on the Franke function. Additionally, the Sigmoid activation function was found to be appreciably slower than the ReLU variants.

1. INTRODUCTION

Over the last few years, machine learning and neural networks have become an increasingly important part of data analysis with an enormous range of applications. From image recognition to predictive analytics and scientific simulations, these techniques are reshaping the way the scientific community tackles complicated problems. Linear and logistic regression play a fundamental role in machine learning, providing robust ways for modelling linear and logistic relationships in data. They also serve as important building blocks in understanding more advanced techniques, such as neural networks.

Neural networks excel at handling complex, nonlinear relationships in data. Their flexibility in approximating intricate patterns has made them indispensable across diverse fields, including biology, engineering, finance, and physics. For just a few examples, see [1–3].

The main goal of this project is to gain a deeper understanding of neural networks by first exploring the fundamentals of linear and logistic regression. To achieve this, we implement and experiment with various optimization techniques, comparing the performance of linear and logistic regression to that of neural networks. We use data generated from the Franke function to evaluate linear regression, and the binary breast cancer dataset from `sklearn`'s datasets [4] to assess logistic regression. The project first introduces the theory behind linear/logistic regression and neural networks, before proceeding to explain how this is implemented. Results are then presented, and the performance of linear/logistic regression is compared to the performance of the implemented neural network, and the known neural network of `sklearn`'s `keras`.

2. METHODS

In this section we present the various methods used in this report. For reference, the Franke function which we will be using to test our linear regression methods is:

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

2.1. Linear Regression

As discussed in a previous project [5], linear regression is the simplest method for fitting a continuous function to a given data set. The data set is approximated by $\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta}$ and the β coefficients are found by minimizing the cost function. For this project we consider the two

* e.b.rornes@fys.uio.no
† icrukan@uio.no
[a] Github

regression methods:

$$C_{\text{OLS}}(\boldsymbol{\beta}) = \frac{2}{n}(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^2, \tag{2}$$

$$C_{\text{Ridge}}(\boldsymbol{\beta}) = C_{\text{OLS}}(\boldsymbol{\beta}) + \lambda||\boldsymbol{\beta}||_2^2. \tag{3}$$

We then insist that the derivative of these w.r.t. $\boldsymbol{\beta}$ is 0, and choose the resulting β coefficients as out model. Doing this we arrive at:

$$\boldsymbol{\beta}_{\text{OLS}} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}, \tag{4}$$

$$\boldsymbol{\beta}_{\text{Ridge}} = (\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}^T\boldsymbol{y}. \tag{5}$$

2.2. Regularization Terms

Regularization is a technique to prevent overfitting by adding a penalty to the cost function that discourages complex models. Overfitting occurs when a model learns the noise in the training data rather than the underlying patterns, leading to poor generalization on unseen data. In the context of neural networks, regularization plays a critical role, especially when working with architectures that have many parameters.

The two common regularization methods that we inspected previously are Ridge and Lasso regularization. In this project, we will only be considering Ridge regularization, where the cost function is given by

$$C_{\text{Ridge}}(\boldsymbol{\beta}) = C_{\text{OLS}}(\boldsymbol{\beta}) + \lambda\boldsymbol{\beta}_2^2, \tag{6}$$

where $C_{\text{OLS}}(\boldsymbol{\beta})$ is the ordinary least squares cost function, $\boldsymbol{\beta}$ represents the model parameters, and the hyperparameter λ controls the magnitude of the penalty to large coefficients. For more details on regularization in linear regression, see [5].

In the context of neural networks, regularization techniques are crucial for controlling model complexity and improving generalization. Ridge regularization can be implemented in neural networks by applying ℓ^2 regularization to the weights of the network. This approach adds the penalty term $\lambda\|\mathbf{W}\|_2^2$ to the loss function, where $\mathbf{W}$ is the weight matrix of the neural network. The modified cost function for a neural network with ℓ^2 regularization becomes

$$C_{\text{NN}} = C_{\text{loss}} + \lambda\|\mathbf{W}\|_2^2, \tag{7}$$

where C_{loss} will be the MSE for regression tasks and cross-entropy loss for classification tasks.

Incorporating regularization in neural networks offers several important benefits. Firstly, ℓ^2 regularization effectively applies a weight decay, which shrinks the weights during training and helps prevent the model from becoming overly complex. This leads to smoother decision boundaries and reduces the risk of overfitting, allowing the model to generalize better to unseen data. Secondly, regularized models tend to exhibit increased robustness to variations in the data, as the penalty encourages the learning of simpler models that do not fit the noise in the training set. Lastly, by tuning the hyperparameter λ, practitioners can effectively control the model's complexity, striking a balance between bias and variance and tailoring the model to the specific problem at hand. These advantages highlight the critical role that regularization plays in developing neural networks that perform well in real-world applications.

Overall, incorporating regularization terms such as Ridge regularization in the training of neural networks is essential for developing models that generalize well to new, unseen data.

2.3. Logistic Regression

Whilst linear regression is quite successful in fitting continuous data, when the output is supposed to be discrete it fails. Linear regression predicts values across a continuous spectrum, resulting in predictions outside the range of valid class labels, such as giving negative probabilities. Logistic regression on the other hand is specifically designed for binary classification problems, and is thus ideal when dealing with discrete outcomes.

Logistic regression models the probability that an input belongs to a particular class by mapping real-valued inputs to a range between 0 and 1 using a 'transformation function'. Typically, the sigmoid function σ is used, converting the linear prediction into a probability in a smooth manner. Given an input vector X and a set of weights β, the predicted probability that the class label y equals 1 is expressed as:

$$P(y = 1|\boldsymbol{X}) = \sigma(\boldsymbol{X}\boldsymbol{\beta}) = \frac{1}{1 + e^{-\boldsymbol{X}\boldsymbol{\beta}}}. \tag{8}$$

For each sample i, we refer to this probability as $\hat{y}_i$. To optimize the weights, logistic regression minimizes the cross-entropy loss function:

$$C(\boldsymbol{\beta}) = -\frac{1}{n}\sum_{i=1}^{n}\left(y_i \ln(\hat{y}_i) + (1 - y_i)\ln(1 - \hat{y}_i)\right) \tag{9}$$

where y_i is the class label.

Furthermore, to penalize overfitting, a regularization term may be added to (9). This terms adds a penalty for large weights, trying to keep the weights (relatively) small. Adding the ℓ^2 regularization term results in

$$C(\boldsymbol{\beta}) = -\frac{1}{n}\sum_{i=1}^{n}\left(y_i \ln(\hat{y}_i) + (1 - y_i)\ln(1 - \hat{y}_i)\right) + \lambda\sum_{i=1}^{n} w_j^2. \tag{10}$$

2.4. Resampling Methods

Resampling methods are used to estimate the accuracy of predictive models by splitting the data into training and testing sets or by generating multiple datasets.

Common techniques include cross-validation and bootstrapping. In cross-validation, the data is split into k folds, and the model is trained on $k-1$ folds and tested on the remaining fold. This process is repeated k times, and the average accuracy is computed. Bootstrapping involves sampling with replacement from the dataset to create multiple training sets. These methods help assess model stability and generalizability on unseen data. For more details see [5].

2.5. Gradient Descent

Gradient descent (GD) is an essential optimization algorithm in machine learning, commonly used to minimize cost functions by adjusting model parameters iteratively. Given model parameters θ and a cost function $C(\theta)$, the GD update rule adjusts parameters in the opposite direction of the gradient:

$$\theta_i^{(j+1)} = \theta_i^{(j)} - \eta \frac{\partial C}{\partial \theta_i}, \tag{11}$$

where η is the learning rate. Batch gradient descent (BGD) calculates the gradient over the entire dataset:

$$\theta^{(j+1)} = \theta^{(j)} - \eta \nabla_\theta C. \tag{12}$$

BGD is computationally expensive for large datasets but provides smooth convergence toward the minimum.

The learning rate η does not necessarily need to be constant, and can change with each iteration. There are several ways to implement a varying learning rate. In this report, we either use a constant learning rate or a learning rate on the form

$$\eta(e, i; N; b; t_0; t_1) = \frac{t_0}{e \cdot N/b + i t_1}, \tag{13}$$

where e is the current epoch, N the data-size, b the batch size (see sec. 2.6) and i the current batch-iteration. The parameter t_0 is related to the initial magnitude of the learning rate, allowing larger learning rates at the beginning of the algorithm. Keeping this parameter fairly large can be beneficial in scenarios where multiple local minima are present, as the learning rate will 'wait' a bit before it starts converging on a solution. The parameter t_1 on the other hand influences how quickly the learning rate decreases over 'time'. For example, in scenarios where the data is sensitive to small changes, keeping t_1 small can help increase the accuracy of the model near the end of the training.

2.6. Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is a variation of gradient descent where each parameter update is performed on a single data point or a small batch. The update rule for SGD is:

$$\theta_i^{(j+1)} = \theta_i^{(j)} - \eta \frac{\partial C^{(i)}}{\partial \theta_i}$$

where $C^{(i)}$ is the cost function evaluated at a single data point i. While SGD introduces noise in the updates, it often converges faster for large datasets, and helps escape local minima, making it ideal for training neural networks.

'Plain' gradient descent or stochastic gradient descent may also keep track of a so-called 'momentum' parameter m, which is supposed to push the descent algorithm in the correct direction. This parameter helps build up speed towards a solution, and can be helpful to overcome local minima. It is implemented in the following way:

$$m_i = \beta m_i + (1-\beta)(\nabla_\theta C)_i, \tag{14}$$

and modifies the new θ_{i+1} like

$$\theta_{i+1} = \theta_i - \eta m_i. \tag{15}$$

2.7. Optimization Algorithms

To reach the global minima of the cost function, there exists several optimization algorithms which can help speed up the process and becoming trapped in local minima. These algorithms essentially modify the learning rate by analyzing the magnitude and behavior of the gradients. This section gives a brief summary of three optimization algorithms; the adaptive gradient (AdaGrad) algorithm, the root mean squared propagation (RMSprop) algorithm and the adaptive moment estimation (Adam) algorithm.

2.7.1. AdaGrad

The AdaGrad algorithm modifies the learning rate by keeping track of how large contributions from the gradients build up over time;

$$\eta_i \to \eta_i \frac{1}{\epsilon + \sqrt{\sum_{j=1}^{i} (\nabla_\theta C)_j^2}}, \tag{16}$$

where ϵ is a small parameter to avoid division by zero.

2.7.2. RMSprop

The RMSprop optimization algorithm has a similar goal as AdaGrad, minimizing the negative effects of large gradients. However, RMSprop does this a bit differently, by calculating a 'decaying average of the squared gradients'. Specifically, it keeps track of a parameter G_i which represents how the average of the squared gradients change, and uses a parameter r which controls the

'rate of decay'. Typically, r is set very close to 1, and we have used $r = 0.99$ throughout. The RMSprop algorithm modifies the learning rate as such:

$$G_i = rG_{i-1} + (1-r) \cdot (\nabla_\theta C)_i^2, \tag{17}$$

$$\eta_i \to \frac{\epsilon + \eta_i}{\sqrt{G_i}}, \tag{18}$$

where ϵ is again introduced to avoid zero-division.

2.7.3. Adam

The Adam algorithm is perhaps the most advanced optimization algorithm of the ones we present here. It works by essentially combining RMSprop and momentum. It adjusts the learning rate by computing estimates of the mean ('first momentum' m) and the variance ('second momentum' v) of the gradients. The two momenta are updated in each iteration:

$$m_i = \beta_1 m_{i-1} + (1-\beta_1)\nabla_\theta C, \tag{19}$$

$$v_i = \beta_2 v_{i-1} + (1-\beta_2)(\nabla_\theta C)^2, \tag{20}$$

with the parameters β_1 and β_2 which are typically close to one. Given these values for β_1, β_2, eq. (19) implies that m and v initially starts out close to zero. To account for this, Adam includes additional corrections terms:

$$\hat{m}_i = \frac{m_i}{1-\beta_1^i}, \tag{21}$$

$$\hat{v}_i = \frac{v_i}{1-\beta_2^i}. \tag{22}$$

Adam then calculates the next θ_{i+1} as such:

$$\theta_{i+1} = \theta_i - \frac{\eta}{\sqrt{\hat{v}_i} + \epsilon}\hat{m}_i \tag{23}$$

2.8. Neural Networks

Neural networks are computational models inspired by the human brain, designed to recognize patterns and relationships within data. They consist of layers of interconnected neurons or nodes, where each neuron applies a transformation to the input data. In each layer, neurons take a weighted sum of inputs, apply an *activation function* to introduce non-linearity, and pass the result to the next layer. The final layer produces the output, serving as the network's prediction.

2.8.1. Feed Forward Neural Networks

Feed-forward neural networks (FFNNs) are the simplest type of neural network, where data flows forward from input to output without forming cycles. These networks contain one or more hidden layers that apply an activation function to capture complex, nonlinear patterns in the data. The training process adjusts the weights of each connection to minimize a cost function, typically using gradient descent.

2.8.2. Activation Functions

Activation functions play a critical role in neural networks by introducing non-linearity, which enables the network to approximate more complex functions beyond simple linear mappings. In this project, we use three different activation functions: sigmoid, ReLU, and Leaky ReLU. Each function has different properties that can impact training performance and convergence.

- The sigmoid function, suitable for binary classification, takes an input $z \in \mathbb{R}$ and outputs a value in the range $(0, 1)$. This makes it useful for probabilistic interpretations. As mentioned prior, it is given by:

$$\sigma(z) = \frac{1}{1+e^{-z}}. \tag{24}$$

- The ReLU (Rectified Linear Unit) function activates only positive values:

$$R(z) = \begin{cases} 0 & \text{if } z \leq 0 \\ z & \text{if } z > 0 \end{cases}. \tag{25}$$

  This reduces the number of calculations that the network has to perform and can speed up the training.

- The Leaky ReLU (LReLU) function is a variation of ReLU that allows a small gradient when $z \leq 0$. This can help mitigate an issue known as 'dying ReLU' where neurons become inactive due to consistently receiving negative inputs, helping to mitigate issues with inactive neurons. It is given by:

$$LR(z) = \begin{cases} az & \text{if } z \leq 0 \\ z & \text{if } z > 0 \end{cases} \tag{26}$$

  where a is some small number. In this project we consider only $a = 0.01$.

By selecting appropriate activation functions for each layer, FFNNs can effectively capture complex data patterns, enhancing model performance.

2.8.3. Backpropagation

Backpropagation is the key algorithm for training neural networks by optimizing weights to minimize the cost function. It works by propagating the error backward from the output layer to the input layers, computing gradients for each weight based on the error. These gradients are then used to update the weights, enabling the network to learn from its errors and make more accurate predictions over time.

3. IMPLEMENTATION

3.1. Linear and Logistic Regression

Linear and Logistic Regression was used to study data from the Franke function and the breast cancer data, respectively. Both methods rely on some sort of optimization algorithm when applying the gradient descent algorithm, see sec. 2.7. The optimization algorithms (plain GD/SGD, Adagrad, RMSprop or Adam) are implemented as classes inheriting from a parent class `Optimizer`. This class defaults to the parameters $\eta = 0.01, m = 0.9, \epsilon = 1e-8, \beta_1 = 0.9, \beta_2 = 0.999, r = 0.9$, with the possibility of η being a callable, see sec. 2.5. In this report, when comparing constant and varying learning rates, we have set $t_0 = 2$ and $t_1 = 2/\eta$ (for constant η) in the equation for varying learning rate (13).

After an optimization algorithm has been chosen, the class `DescentSolver` is used together with a chosen gradient, the Ridge-gradient (linear regression) or a logistic-gradient (logistic regression), to compute GD or SGD. To analyze the results of `DescentSolver`, the class `DescentAnalyzer` is used. This class computes metrics, MSE/R^2 (linear regression) or accuracy score (logistic regression), for an (equally sized) grid of λ, η-values. The metrics, together with all other parameters, are saved as pickle-files, which can be read at a later time from `DescentAnalyzer.load_data` giving a dictionary of the data.

3.2. Neural Network

Further, the `FFNN` class was implemented to analyze both the Franke function and the breast cancer data. The FFNN is structured with an input layer, one or more hidden layers, and an output layer, with each layer utilizing an activation function, and currently only supports the Adam optimizer. This was due to unknown performance issues when trying to incorporate the `Optimize` class. The architecture is defined by specifying the input size, the number of neurons in hidden layers, and the output size. In the case of the Franke function, the input layer is size 2, one for x and one for y, while for the breast cancer data, the input layer corresponds to the number of features, i.e., 30, representing several properties of the tumor, such as circumference, density, etc.

We tested various configurations for layering the hidden layers. Ultimately, for the Franke function, we settled on a pyramid-like architecture for the hidden layers $[4, 8, 16, 32, 16, 8, 4, 2]$, while for the breast cancer data, we used $[15, 30, 15, 8, 4, 2]$. These choices were based on the sizes of both the input and output layers, where the output layer is 1 in both cases. The weights and biases are initialized using random values scaled by the number of input neurons, which aids in faster convergence during training. The network supports ReLU, Sigmoid, and Leaky ReLU as activation functions.

The forward propagation computes the output of the network by applying the chosen activation function to the weighted sum of inputs at each layer. Depending on the task, the FFNN uses mean squared error (MSE) as the loss function for regression tasks (like the Franke function) and binary cross-entropy (BCE) for classification tasks (like breast cancer data). The backward propagation algorithm updates the weights and biases using gradient descent, which considers the specific loss function being used. We also implement regularization techniques to mitigate overfitting.

The training process includes splitting the data into training and test sets, followed by iterating through a specified number of epochs during which the network adjusts its weights to minimize the error. After training, the network can predict outputs for new data, allowing for evaluation against known values from the Franke function. The overall performance of the model is assessed using MSE for the Franke function and accuracy for the breast cancer data, reflecting the different loss functions applied during training.

3.3. Data Parameters

We tried various different learning rates η and λ-values. Specifically, we used $\eta_{\text{const}}, \lambda \in [10^{-10}, 10^1]$. For varying learning rate, we settled on $t_0 = 2$ and $t_1 = 2/\eta_{\text{const}}$, see (13). We sampled 20 datapoints from the Franke function and looked at epoch sizes $e \in \{10, 100\}$, and batch sizes $b \in \{4, 5, 10\}$. We used a test size of 25% throughout.

On the cancer data we use StandardScaling to scale the data after splitting it to avoid data leakage, whereas the Franke function was normalized to the range $[0, 1]$ using min-max scaling. On the Franke function we also implemented `tensorflow.keras` to compare its results with our own FFNN.

4. RESULTS & DISCUSSION

We represent the results and compare the various methods. The larger plots have been placed in Appendix A to preserve space for the sake of readability. More figures can be found in the `Figures` folder.

4.1. Franke

4.1.1. Linear Regression

MSE scores from linear regression for plain SGD and the three more advanced optimization methods (AdaGrad, RMSprop and Adam), for different epoch and batch sizes, and for both constant and learning rates, can be found in the `Figures`-folder, look for files on the form *LinReg...pdf*. Fig 4 is composed of one figure each for each

of the optimization methods, for epochs $N \in \{10, 100\}$ and a batch size of 4.

For plain SGD with a varying η, higher epoch sizes and smaller batch sizes lowered the MSE scores, suggesting a stable convergence. However, with a constant learning rate, the MSE scores blew up, implying that the algorithm skipped the global minima and continued either towards a local minimum or straight up diverged. This can be seen from plots in `Figures` (look for *LinRegplainSGD_constEta...pdf*), where all MSE scores are equal to ~ 0.207. This feature was not seen in any of other optimization methods. Both RMSprop and Adam showed a clear convergence when increasing (decreasing) the epoch size (batch size). However, RMSprop did not converge for constant learning rates. For varying learning rate, RMSprop gave low MSE scores, that is, with $t_1 = 2/\eta_{\text{const}} \sim [2, 2 \cdot 10^2]$. In this range, RMSprop gave quite good results, averaging an MSE score of $\sim [0.1, 0.01]$.

On the other hand, Adam converged for both varying and constant learning rates, and essentially outperformed all the other algorithms by giving lesser MSE scores overall. In fact, Adam seemed to treat varying and constant learning rates the same, in the sense that for $\eta_{\text{const}} \sim [2 \cdot 10^{-2}, 1]$, it averaged an MSE score of $\lesssim 0.1$.

The odd one out was AdaGrad. Giving the largest MSE scores, AdaGrad performed the worse out of all the optimization methods. In fact, AdaGrad seemed to not converge at all, rather producing sort of random data. We found this somewhat strange, and we suggest that there could be that there is a flaw in the implementation of AdaGrad. On the other hand, AdaGrad showed good results for logistic regression (see sec. 4.2.1).

Together, the results show an overall better performance for varying learning rate, compared to that of a constant learning rate. plain SGD is especially effected by this, compared to Adam which seems to converge for both varying and constant learning rates. Common for all optimization methods, except plain SGD, was that an increase (decrease) in number of epochs (batch size) gave better MSE scores.

4.1.2. Neural Network

The results for the MSE and R^2 as a function of the learning rate η with 1000 epochs with our own FFNN with different activation functions and keras NN with the sigmoid activation function are given in Fig. 1. In this particular plot we are not using any regularization terms. The reason for this is simply due to `keras` being very slow, but we still wanted to include it as a reference. The optimal learning rates for $\lambda = 0$ can easily be read off this plot: $\{10^{-3}, 10^{-3}, 3 \times 10^{-3}, 2 \times 10^{-2}\}$ for {ReLU, Sigmoid, LReLU, `keras`}.

Further we used the result for the best learning rate to plot the MSE after each epoch, given in Fig. 2. As

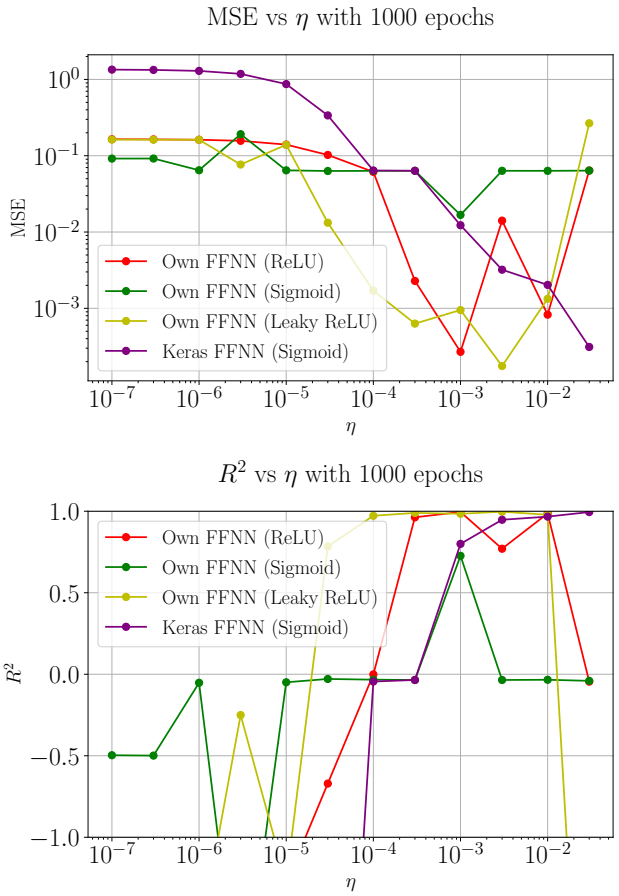


Fig. 1: MSE and R^2 regression results for the FFNN as a function of η with 1000 epochs.

expected, all activation functions perform poorly on low epochs, and gradually improve as you go along. At the end, all seem to achieve a relatively good MSE as the number of epochs increase, with the exception of our own implementation of the Sigmoid function.

Overall these metrics seem to imply that ReLU and LReLU are outperforming `keras`. This is likely due to the choice of using the sigmoid function, which is more suited towards binary classification tasks. As can be seen in Fig. 5a, ReLU, LReLU and `keras` are all quite close to recreating the Franke function, whilst Sigmoid performs about as well as LASSO did in [5].

We then did the above again, but now with a regularization parameter λ and excluded `keras` due to time constraints. The MSE for various combinations of η and λ are given in Fig. 6. The best results from here are then picked to once again plot the MSE over epochs given in Fig. 3 and another 3D plot to visualize the results in Fig. 5b. Clearly the sigmoid activation function performs worse overall once again, whilst ReLU and LReLU

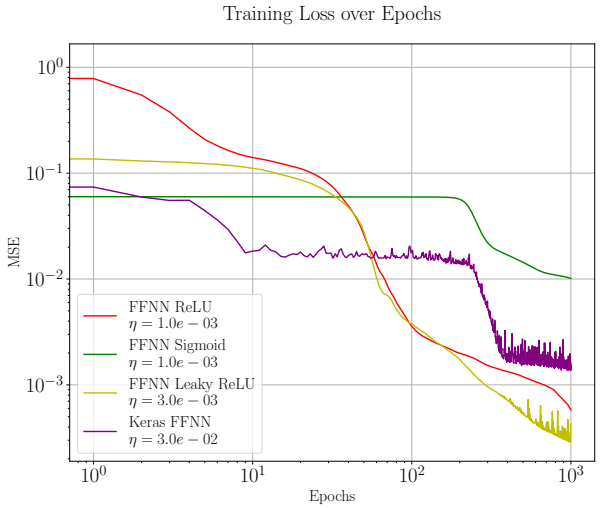


Fig. 2: The MSE regression results for the FFNN as a function of number of epochs for ReLU, LReLU, Sigmoid and keras.

have a close competition between them. The convergence is much faster here, and we same performance with ReLU and LReLU after only 50 epochs. The sigmoid function converges much slower, but this is due to it being suited for binary classification and not regression. The much faster convergence does suggest that the regularization parameter is improving our performance significantly.

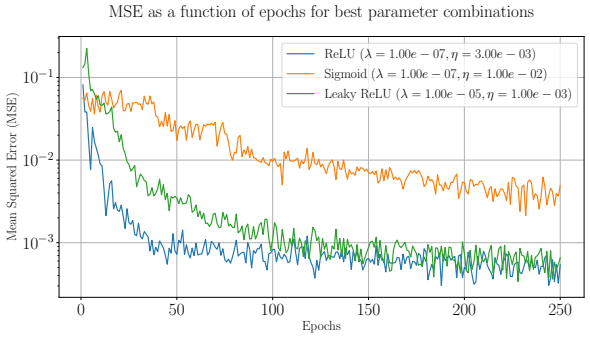


Fig. 3: The MSE regression results for the FFNN as a function of number of epochs for ReLU, LReLU, Sigmoid with the best performing combination of λ and η.

4.2. Cancer Data

4.2.1. Logistic Regression

Fig. 7 shows the accuracy score of (plain stochastic) logistic regression for batch size $N \in \{10, 100\}$ and a batch size $b = 50$, more figures can be found in the `Figures` folder. Note that the figures to the right in fig. 7 are zoomed in versions of the one to the right, and the varying learning rate has been used. This figure shows an average accuracy score of ~ 0.97 for $t_1 = 2/\eta_{\text{const}} \sim [10^{-2}, 10^{-1}]$. The more advanced optimization methods were briefly looked into, showing similar results. However, Adam diverged for $N = 100$, and AdaGrad showed promising results (accuracy scores above 0.9), the opposite behavior of what we saw for linear regression.

4.2.2. Neutral Network

The accuracy for logistic regression and our own FFNN for the three different activation functions and for different values of η and λ are given in Figs. 7 and 8 respectively.

5. CONCLUSION

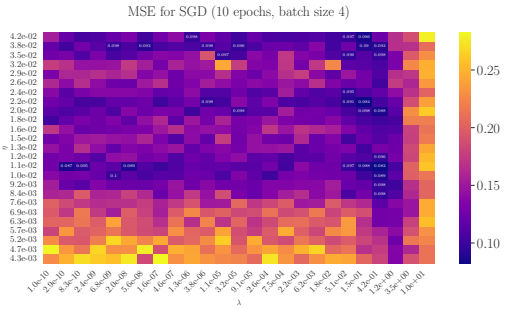
On the Franke function, we found that Sigmoid is not overly sensitive to changes in the learning rate and hyperparameter λ, but also struggles to get good results. On the other hand ReLU has the ability to get great results, but only for very specific combinations of η and λ. LReLU seems to be the best of both worlds, being relatively insensitive to getting the exact right combination of η and λ, whilst still managing to obtain very good MSE scores like ReLU. For logistic regression our FFNN performs quite well, approaching 98% accuracy with certain parameter combinations.

A step forward would for example be testing out different values of a in LReLU and seeing whether it may achieve a higher level of performance. Thoroughly testing out other optimizers such as Adagrad and RMSprop for the FFNN may also yield benefits, but due to time constraints we did not implement this properly.

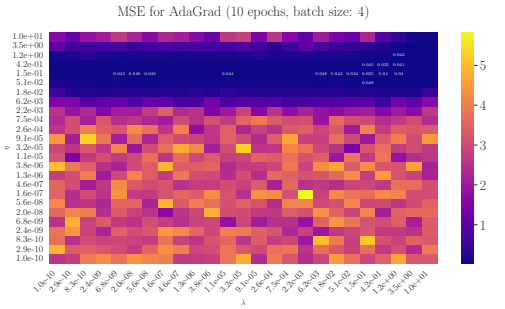
[1] A. Dawid, J. Arnold, *et. al.*, *Modern applications of machine learning in quantum sciences*, 2023.
[2] V. Thapar, *Applications of machine learning to modelling and analysing dynamical systems*, 2023.
[3] F. G. Mohammadi, F. Shenavarmasouleh, and H. R. Arabnia, *Applications of machine learning in healthcare and internet of things (iot): A comprehensive review*, 2022.
[4] F. Pedregosa, G. Varoquaux, *et. al.*, *Scikit-learn: Machine learning in python*, *Journal of Machine Learning Research* **12** (2011) 2825–2830.
[5] I. Rukan and E. R. rnes, *Application of regression and resampling on usgs terrain data*, 2024.

Appendix A: Large Figures

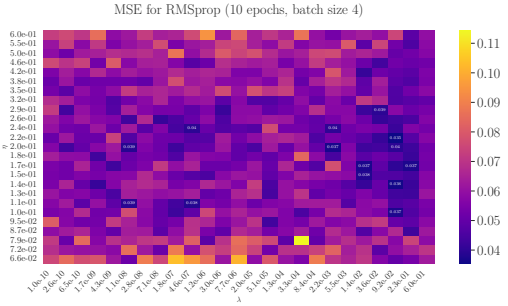
A selection of figures, more can be found in the `Figures` folder.



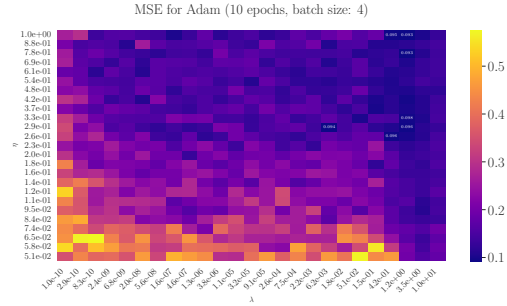
(a) MSE score using SGD for franke data with 10 epochs, batch size 4.



(b) MSE score using AdaGrad for franke data with 10 epochs, batch size 4.

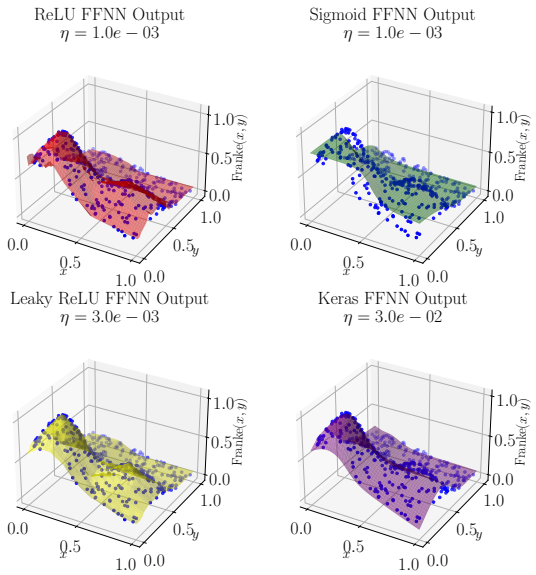


(c) MSE score using RMSprop for franke data with 10 epochs, batch size 4.

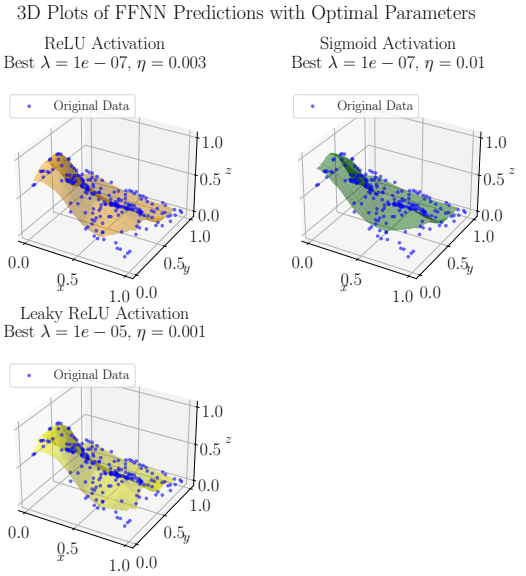


(d) MSE score using Adam for franke data with 10 epochs, batch size 4.

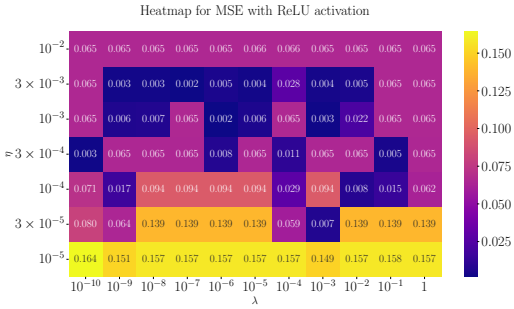
Fig. 4: Accuracy score for logistic regression, for number of epochs $N = 10$ (left), $N = 100$ (right). The figures to the right are zoomed in versions of the ones to the left.



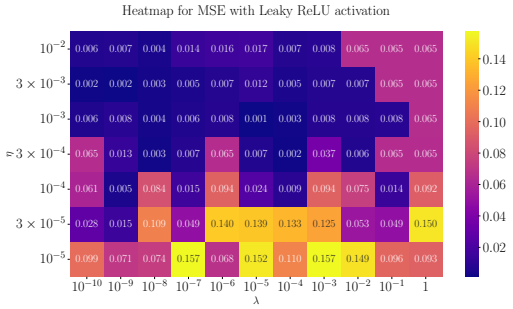
(a) 3D plots for best η with 1000 epochs. The blue dots correspond to the sampled points from the Franke function with 100 total samples.



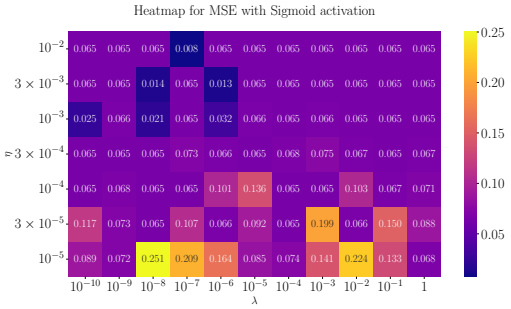
(b) 3D plots for best combination of λ and η for our own neural network with 250 epochs and 100 samples from the Franke function.



(a) ReLU

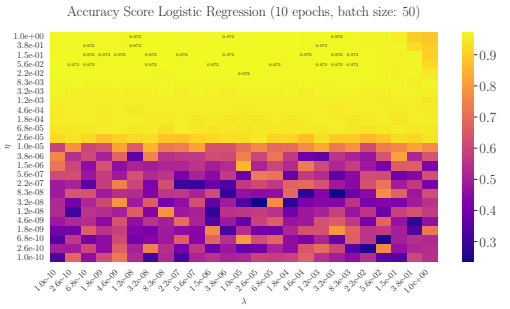


(b) LReLU

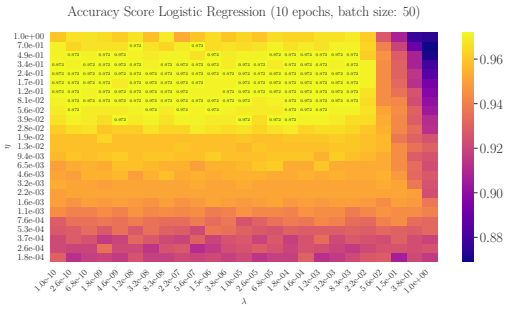


(c) Sigmoid

Fig. 6: MSE for various combinations of η and λ for ReLU, LReLU and Sigmoid activation functions with 250 epochs on the Franke function.

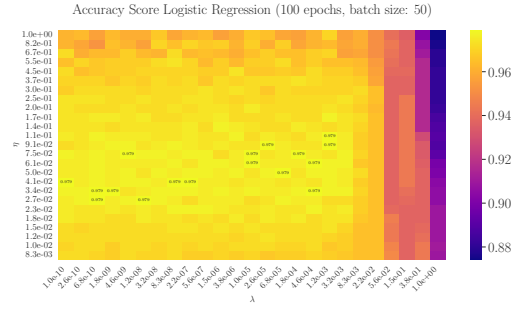


(a) Logistic regression for cancer data with 10 epochs, batch size 50.



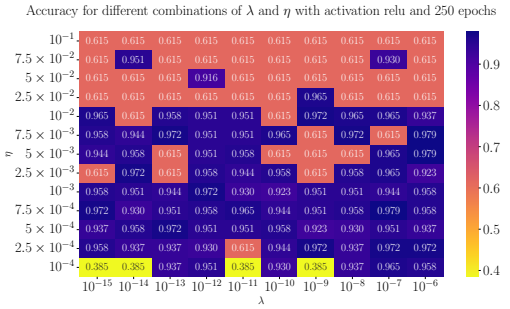
(b) Logistic regression for cancer data with 10 epochs, batch size 50.

(c) Logistic regression for cancer data with 100 epochs, batch size 50.

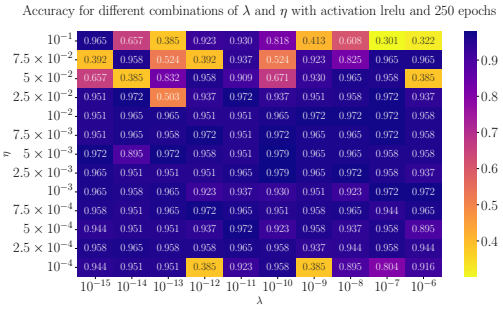


(d) Logistic regression for cancer data with 100 epochs, batch size 50.

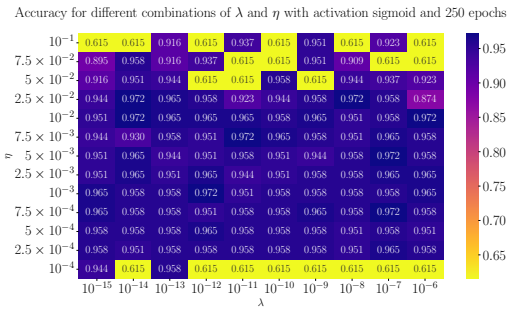
Fig. 7: Accuracy score for logistic regression, for number of epochs $N = 10$ (left), $N = 100$ (right). The figures to the right are zoomed in versions of the ones to the left. A varying learning rate has been used, with $t_0 = 2$, $t_1 = 2/\eta_{\text{const}} \in [10^{-10}, 1]$



(a) ReLU



(b) LReLU



(c) Sigmoid

Fig. 8: Accuracy for various combinations of η and λ for ReLU, LReLU and Sigmoid activation functions with 250 epochs on the breast cancer dataset.