Project 2

Edvard B. Rørnes* and Isak O. Rukan†
Institute of Physics, University of Oslo,
0371 Oslo, Norway
(Dated: November 3, 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. 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-scores 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. The flexibility of neural networks in approximating complex, non-linear relationships has made them indispensable across diverse fields, such as biology, engineering, finance, physics etc. For just a few examples, see [1–3].

Neural networks are a powerful tool for handling large datasets where traditional modeling may fall short. This allows us to extract patterns and make accurate predictions. As the applications of these techniques continue to expand, so does the demand for a deeper understanding of their underlying principles and implementation details. Thus, we aim to investigate the foundational aspects of neural networks, explore various activation functions and learning algorithms, and study their effects on model performance. By doing so, we hope to develop a robust framework for applying neural networks effectively to complex real-world data.

The main goal of this project is understand the various techniques behind machines learning, and investigate they can be applied to the healthcare system, specifically in diagnosing malignant breast cancer tumors based on the tumor's several features based on the data from `sklearn`'s datasets [4]. We do this by testing the performance of a neural network by comparing it with logistic regression as we are working with discrete data.

We begin by introducing the relevant background necessary to understand the implementations. Then we outline said implementations before discussing the results. The results go over both linear and non-linear regression on the Franke function where performance issues in the implementation are easier to visualize. Finally the neural network and logistic regression are applied to the cancer data.

2. METHODS

In this section we present the various methods used in this report.

2.1. Linear Regression

As discussed in a previous project [5], linear regression is the simplest method for fitting a continuous given a 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 regression methods:

$$C_{\text{OLS}}(\boldsymbol{\beta}) = \frac{2}{n}(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^2, \tag{1}$$

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

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

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

Doing this we arrive at:

$$\boldsymbol{\beta}_{\text{OLS}} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}, \tag{3}$$

$$\boldsymbol{\beta}_{\text{Ridge}} = (\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}^T\boldsymbol{y}. \tag{4}$$

2.2. Regularization Terms

Regularization is a technique to prevent overfitting by adding a penalty to the cost function that discourages complex models. The two common regularization methods that we inspected previously are Ridge and Lasso regularization. In this project we will only be considering Ridge, where the cost function is given by

$$C_{\text{Ridge}}(\boldsymbol{\beta}) = C_{\text{OLS}}(\boldsymbol{\beta}) + \lambda\|\boldsymbol{\beta}\|_2^2, \tag{5}$$

where the hyperparameter λ controls the magnitude of the penalty to large coefficients. For more details see [5].

2.3. Logistic Regression

Whilst linear regression is quite successful in fitting continuous data such as terrain 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 a given input belongs to a particular class, typically using the sigmoid function to map any real-valued number to a value between 0 and 1. Given an input vector $\boldsymbol{X}$ and weights $\boldsymbol{\beta}$, logistic regression predicts the probability of the class as:

$$P(y = 1|\boldsymbol{X}) = \sigma(\boldsymbol{X}\boldsymbol{\beta}) = \frac{1}{1 + e^{-\boldsymbol{X}\boldsymbol{\beta}}} \tag{6}$$

where σ represents the sigmoid function. To find optimal weights, logistic regression minimizes the cross-entropy cost function:

$$C(\boldsymbol{\beta}) = -\frac{1}{n}\sum_{i=1}^{n}\left(y_i \ln(\hat{y}_i) + (1 - y_i)\ln(1 - \hat{y}_i)\right) \tag{7}$$

where y_i is the class label and $\hat{y}_i$ is the predicted probability for sample i. To relate the predicted probability from β-coefficients (which are non-binary), and 'activation function' is used (see sec. 2 2.8), which effectively converts $X\beta$ to a binary output.

Furthermore, to penalize overfitting, and regularization term may be added to $C(\boldsymbol{\beta})$. This terms adds a penalty for large weights, trying to keep the weights (relatively) small. Adding the ℓ^2 regularization term results in

$$C(\boldsymbol{\beta}) = -\frac{1}{n}\sum_{i=1}^{n}\left(y_i \ln(\hat{y}_i) + (1 - y_i)\ln(1 - \hat{y}_i)\right) + \lambda\sum_{i=1}^{n} w_j^2. \tag{8}$$

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.

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:

$$\theta_i^{(j+1)} = \theta_i^{(j)} - \eta\frac{\partial C}{\partial \theta_i} \tag{9}$$

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

$$\theta^{(j+1)} = \theta^{(j)} - \eta\nabla_\theta C \tag{10}$$

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

The learning rate η does not necessarily be of a constant value, 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

$$\eta(e, i; N; b; t_0; t_1) = \frac{t_0}{e \cdot N/b + it_1}, \tag{11}$$

where e is the current epoch, N the data-size, b the batch size (see sec. 2 2.6) and i the current batch-iteration. The parameter t_0 is related to the initial magnitude of the learning rate, making it possible to faster or larger learning rates at the beginning of the algorithm. Keeping this parameter fairly large can be beneficial in scenarios where multiple local minimas are present, as the learning rate will sort of '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.

'Plane' gradient descent or stochastic gradient descent may also keep track of a '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 minimas. It is implemented in the following way:

$$m_i = \beta m_i + (1-\beta)(\nabla_\theta C)_i, \tag{12}$$

and modifies the new θ_{i+1} like

$$\theta_{i+1} = \theta_i - \eta m_i. \tag{13}$$

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 minimas. 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;

$$\eta_i \to \eta_i \frac{1}{\epsilon + \sqrt{\sum_{j=1}^{i} (\nabla_\theta C)_j^2}}, \tag{14}$$

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 effect of (too) large gradients. However, RMSprop does this a bit differently, by calculating a sort of 'decaying average of the squared gradients'. Specifically, it keeps track of a parameter G_i which represents how the average of the squared gradients 'move', and uses a parameter r which controls the 'rate of decay'. Typically, r is set very close to 1, and we have used $r = 0.99$ troughout. The RMSprop algorithm 'modifies' the learing rate like

$$G_i = rG_{i-1} + (1-r) \cdot (\nabla_\theta C)_i^2, \tag{15}$$

$$\eta_i \to \frac{\epsilon + \eta_i}{\sqrt{G_i}}, \tag{16}$$

where ϵ is 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 learing rate by computing estimates of the mean ('first momentum' m) and the variance ('second momentum' v) of the gradients. The two momentums are updated in each itteration like

$$m_i = \beta_1 m_{i-1} + (1-\beta_1)\nabla_\theta C \tag{17}$$

$$v_i = \beta_2 v_{i-1} + (1-\beta_2)(\nabla_\theta C)^2, \tag{18}$$

for some parameters β_1 and β_2 which are typically close to one. Given these values for β_1, β_2, eq. (17) implies that m and v initially starts out close to zero. To account for this, Adam includes corrections terms,

$$\hat{m}_i = \frac{m_i}{1-\beta_1^i} \tag{19}$$

$$\hat{v}_i = \frac{v_i}{1-\beta_2^i}. \tag{20}$$

Adam calculates the next θ_{i+1} as

$$\theta_{i+1} = \theta_i - \frac{\eta}{\sqrt{\hat{v}_i} + \epsilon} \hat{m}_i \tag{21}$$

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, tanh, 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:

$$\sigma(z) = \frac{1}{1 + e^{-z}}. \tag{22}$$

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

$$R(z) = \begin{cases} 0 & \text{if } z \leq 0 \\ z & \text{if } z > 0 \end{cases}. \tag{23}$$

  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 a known issue known as "dying ReLU" where neurons become inactive due to consistently recieving negative inputs., helping to mitigate issues with inactive neurons. It is given by:

$$LR(z) = \begin{cases} az & \text{if } z \leq 0 \\ z & \text{if } z > 0 \end{cases} \tag{24}$$

  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 2.7. The optimization algorithms (plane GD/SGD, Adagrad, RMSprop or Adam) are implemented as classes inheriting from a parent class `Optimizer`. The `Optimizer` 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 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 (11).

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.

Further, the Feedforward Neural Network (FFNN) was implemented to be able 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 using the Adam optimizer. 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, whilst for the breast cancer data the input layer corresponds to the number of features, i.e. 30.

We tested out various different ways of layering the hidden layers. In the end, for the Franke function, we settled with using a pyramid architecture for the hidden layers $[4, 8, 16, 32, 16, 8, 4, 2]$ whilst for the breast cancer data we used $[15, 30, 15, 8, 4, 2]$. These were decided on

based on the size of both the input and output, where the latter 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 activation function to the weighted sum of inputs at each layer. The backward propagation algorithm updates the weights and biases using gradient descent, where the mean squared error serves as the loss function. We 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. The test size used throughout is 25%. 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 and accuracy for Franke and cancer data respectively.

4. RESULTS & DISCUSSION

We represent the results and compare the various methods. The large parameter space plots have been placed in Appendix A for the sake of preserving space.

4.1. Franke

4.1.1. Linear Regression

4.1.2. Neutral Netwrok

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 across the board for $\lambda = 0$ seems to lie in the range $[10^{-3}, 10^{-1}]$.

Further we used the result for the best learning rate to plot the MSE after each epoch, given in Fig. 2. As expected, all activation functions perform poorly on low epochs, and gradually improve as you go along. Here we can however see a potential downside of our implementation of the LReLU activation function. It sharply spikes after a certain amount of epochs before settling again. The reason for this is not clear to us, but this is something worth noting when using this activation, as one may be unfortunate enough to land at the top of this spike at the last epoch, causing terrible results.

Overall these metrics seem to imply that ReLU and LReLU are outperforming `keras`, but when explicitly

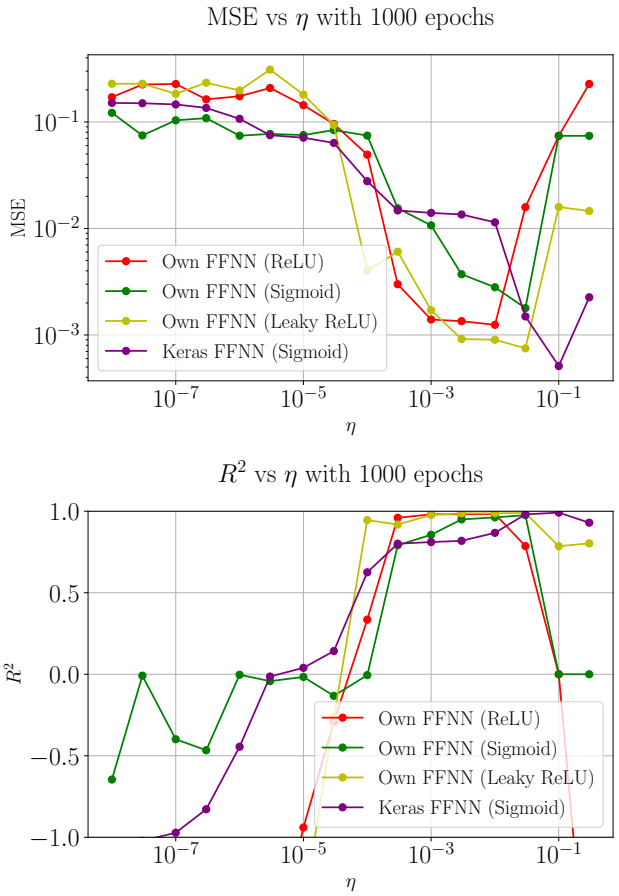


Fig. 1: MSE and R^2 regression results for the FFNN as a function of η with 1000 epochs. The results quickly diverge past 0.3 thus we only plot that far.

plotting the predictions this does not seem to be the case, as can be seen in Fig. 8a. Note that this plotted prediction is a rerun with the best parameters for each activation type. Thus their performance being worse here compared to `keras` may simply be due to large variation from one run to another.

We then did the above again, but now with a regularization parameter λ and excluding `keras` due to performance issues. The MSE for various combinations of η and λ are given in Fig. 5. 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. 8b. Clearly the sigmoid activation function performs worse overall, whilst ReLU and LReLU have a close competition between them. The convergence is much faster here, and we arrive at roughly the same performance with 1/4 of the epochs, implying that the regularization parameter is improving out performance.

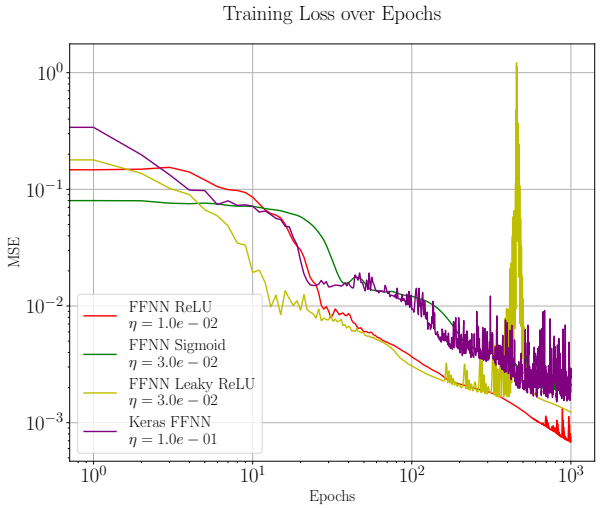


Fig. 2: The MSE regression results for the FFNN as a function of number of epochs for ReLU, LReLU, Sigmoid and keras.

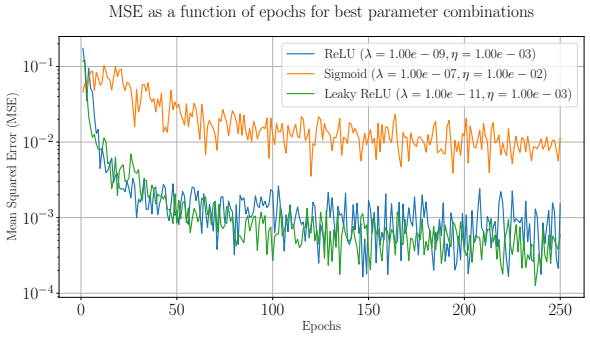


Fig. 3: The MSE regression results for the FFNN as a function of number of epochs for ReLU, LReLU, Sigmoid with the best performing combination of λ and η.

4.2. Cancer Data

4.2.1. Logistic Regression

4.2.2. Neutral Netwrok

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. 6 and 7 respectively.

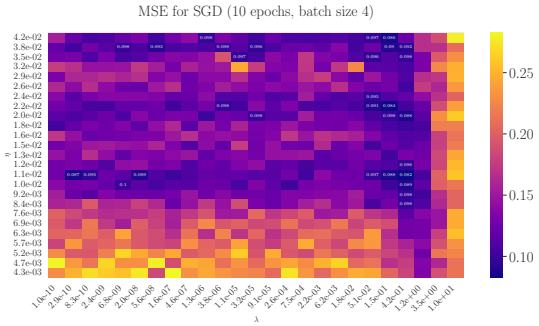
5. CONCLUSION

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.

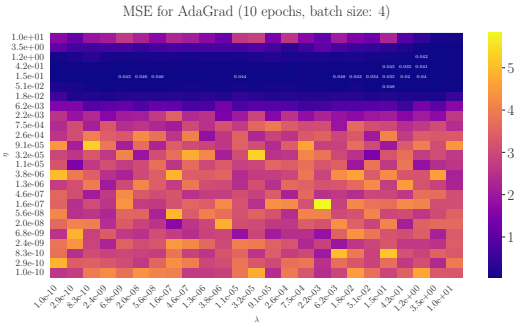
A step forward would for example be testing out different values of a in LReLU and seeing whether it may achieve a higher level of performance.

[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.

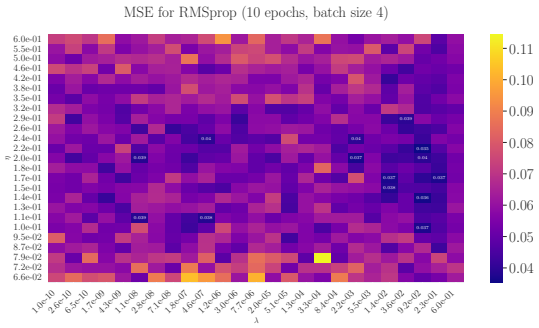
Appendix A: Large Figures



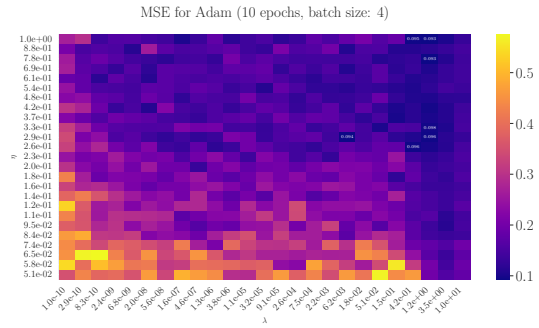
(a) MSE score using SGD for franke data with 10 epochs, batch size 4, $\lambda \in [10^{-10}, 10^{1}]$ and $\eta \in [4.2 \cdot 10^{-2}, 4.3 \cdot 10^{-3}]$



(b) MSE score using AdaGrad for franke data with 10 epochs, batch size 4, $\lambda \in [10^{-10}, 10^{1}]$ and $\eta \in [10^{-10}, 10^{0}]$

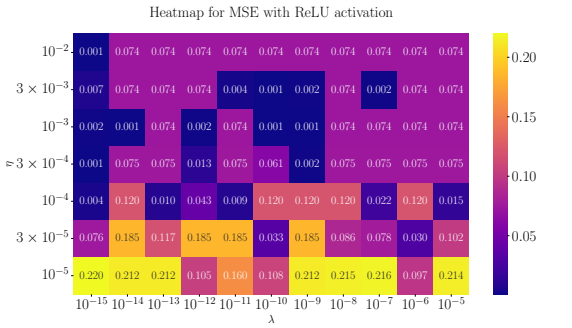


(c) MSE score using RMSprop for franke data with 10 epochs, batch size 4, $\lambda \in [10^{-10}, 6 \cdot 10^{-1}]$ and $\eta \in [6.6 \cdot 10^{-2}, 6 \cdot 10^{-1}]$

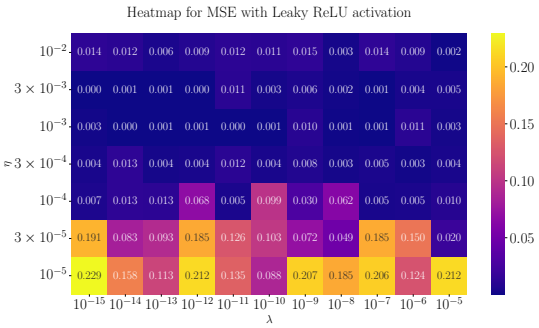


(d) MSE score using Adam for franke data with 10 epochs, batch size 4, $\lambda \in [10^{-10}, 10^{1}]$ and $\eta \in [5.1 \cdot 10^{-2}, 10^{0}]$

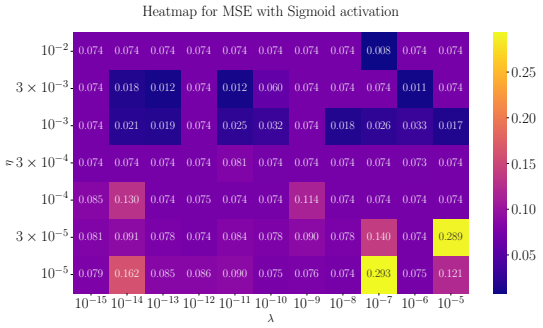
Fig. 4: Accuracy score for logistic regression, for number of epochs $N = 10$ (left), $N = 100$ (right). The figures to the right are zoomed in versions of the ones to the left.



(a) ReLU

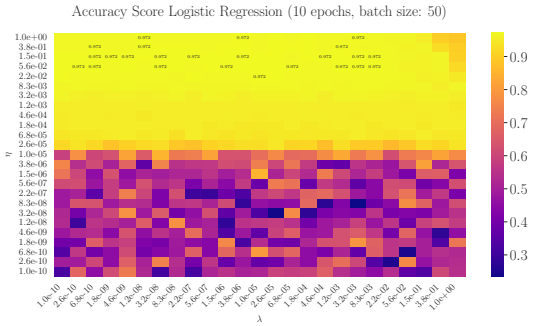


(b) LReLU

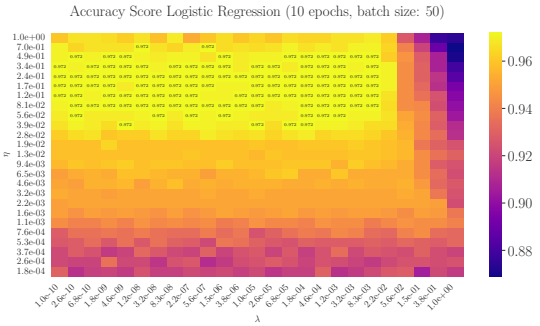


(c) Sigmoid

Fig. 5: MSE for various combinations of η and λ for ReLU, LReLU and Sigmoid activation functions with 250 epochs on the Franke function.

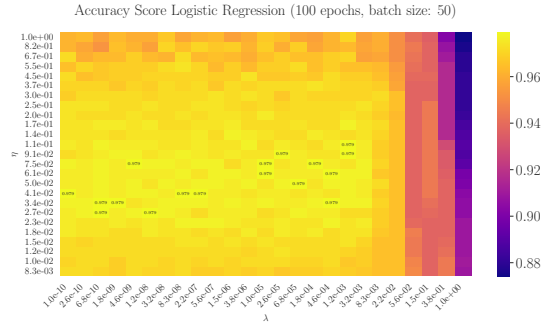


(a) Logistic regression for cancer data with 10 epochs, batch size 50, $\lambda \in [10^{-10}, 10^{1}]$ and $\eta \in [10^{-10}, 10^{1}]$



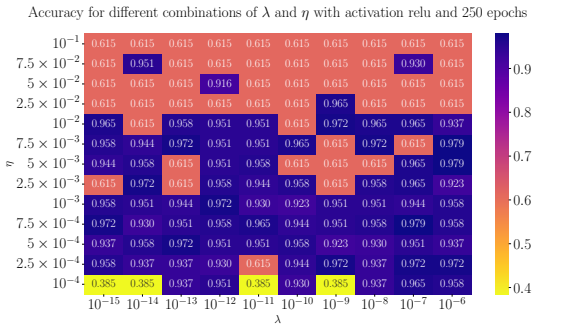
(b) Logistic regression for cancer data with 10 epochs, batch size 50, $\lambda \in [10^{-10}, 10^{0}]$ and $\eta \in [10^{-4}, 10^{0}]$

(c) Logistic regression for cancer data with 100 epochs, batch size 50, $\lambda \in [10^{-10}, 10^{0}]$ and $\eta \in [10^{-10}, 10^{0}]$

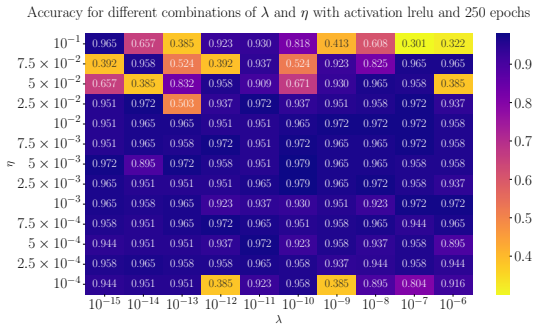


(d) Logistic regression for cancer data with 100 epochs, batch size 50, $\lambda \in [10^{-10}, 10^{0}]$ and $\eta \in [8.3 \cdot 10^{-3}, 10^{0}]$

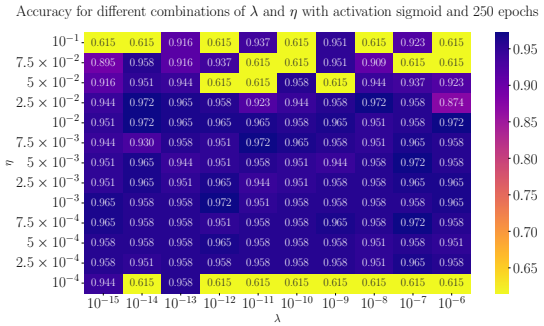
Fig. 6: Accuracy score for logistic regression, for number of epochs $N = 10$ (left), $N = 100$ (right). The figures to the right are zoomed in versions of the ones to the left.



(a) ReLU

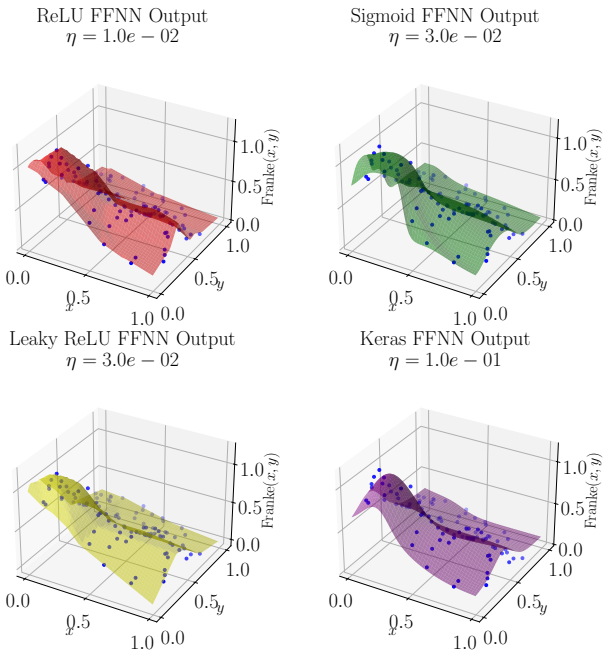


(b) LReLU

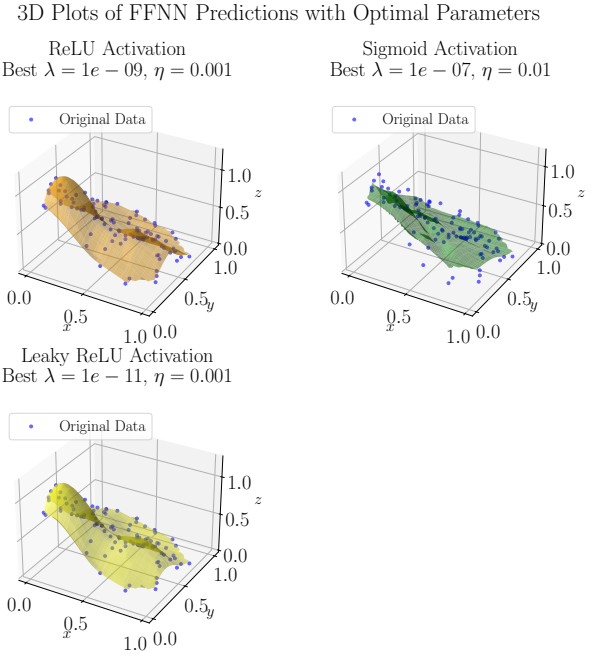


(c) Sigmoid

Fig. 7: Accuracy for various combinations of η and λ for ReLU, LReLU and Sigmoid activation functions with 250 epochs on the breast cancer dataset.



(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.