Classification using a Feed-Forward Neural Network
FYS-STK4155

(Dated: December 20, 2022)

Link to Github repo: https://github.com/hafos/FYS-STK4155/project2

In this project, our aim is to investigate classification and regression problems using a Feed-Forward Neural Network (FFNN). We will compare this method this to linear and logistic regression whilst studying two data sets: The Franke function and the Wisconsin Breast Cancer data set. The goal is to study which methods are best with regards to computational time and accuracy. Our results show that the neural networks code is better suited to the classification problem, achieving a higher accuracy than our logistic regression code and scikit learn's logistic regression function. Meanwhile, for the regression problem the logistic regression code performed slightly better. Various activation functions point towards the Sigmoid function producing the best results in this case, but the logistic regression code still performed better. The implementation and material relevant to this project can be found at the Github repo referenced above.

I. INTRODUCTION

The field of artificial neural networks has a long history of development, starting with McCulloch and Pitts developing a model of artificial neurons in in 1943 in order to study signal processing in the brain. As computer science and computers themselves become more advanced, the field of artificial neural networks has been refined and will continue to advance [1]. Today, it is used in many technological fields, from medicine where trained models can assist in diagnostic and treatment decisions, to providing entertainment companies with models for what products a costumer is more likely to consume.

The neural nets are neural-inspired nonlinear models for supervised learning, which attempt to mimic the neural networks of an animal brain, composed of billions of neurons that communicate by sending electrical signals. The signals must exceed a threshold in order to yield output, or else the neuron remains inactive. The method offers a simple way of analyzing large amounts of data when an exact model is not applicable, and it is often used within regression and classification problems. Neural nets can be viewed as natural, more powerful extensions of supervised learning methods such as linear and logistic regression and soft-max methods [1].

The aim of this project is to study classification and regression problems by developing our own Feed-Forward Neural Network (FFNN) in python. In order to analyze the efficiency and accuracy of each method, we compute the Mean-Squared Error (MSE) and accuracy score.

Previously, we analyzed and developed algorithms for two linear regression methods which we will make use of in this project: The Ordinary Least Square (OLS) method and Ridge regression. We will also include logistic regression for classification problems and write an algorithm for the FFNN for studying both regression and classification problems.

In section II we provide a short summary of the linear regression methods we use, OLS and Ridge regression, as well as an overview of logistic regression and gradient descent. Additionally, we explain relevant theory behind the FFNN and present the datasets we will be working with. A selection of results relevant to our understanding are presented together with a discussion of the results in section III, and in section IV we provide a short summary and outlook.

II. METHOD

Linear and Logistic Regression

When using linear regression we approximate a function $f(\boldsymbol{x})$ by $\tilde{\boldsymbol{y}} = \mathbf{X}\boldsymbol{\beta}$, where the matrix $\mathbf{X}$ is the design matrix and $\boldsymbol{\beta}$ are the unknown parameters we want to determine. The model is fitted by finding the values of $\boldsymbol{\beta}$ which minimize the cost function $C(\mathbf{X},\boldsymbol{\beta})$ where the cost function is a function which allows us to judge how well the model $\boldsymbol{\beta}$ fits the matrix $\mathbf{X}$. The minimum is usually found using numerical methods, as analytical methods are generally not possible.

A common linear regression model is the ordinary least squared (OLS), where we assume a cost function

$$\mathcal{C}_{\mathrm{OLS}}(\boldsymbol{\beta}) = \frac{1}{n}\left\{(\boldsymbol{y}-\mathbf{X}\boldsymbol{\beta})^T(\boldsymbol{y}-\mathbf{X}\boldsymbol{\beta})\right\}, \tag{1}$$

which, when minimized, yields the OLS expression for the optimal parameter $\hat{\boldsymbol{\beta}}$. Another common model is Ridge regression, where we include a regularization parameter λ, and for which the cost function becomes

$$C(\mathbf{X},\boldsymbol{\beta})_{\mathrm{Ridge}} = \left\{(\boldsymbol{y}-\mathbf{X}\boldsymbol{\beta})^T(\boldsymbol{y}-\mathbf{X}\boldsymbol{\beta})\right\} + \lambda\boldsymbol{\beta}^T\boldsymbol{\beta}. \tag{2}$$

For the linear regression analysis our main interest was around leading the coefficients of a functional fit in order to be able to predict the response of a continuous variable on some unseen data. Linear regression resulted in analytical expressions for standard OLS or Ridge regression for several quantities, ranging from the variance and thereby the confidence intervals of the parameters β to the mean squared error (MSE) [1]. By inverting the product of the design matrices we could fit our data.

Classification problems, on the other hand, are concerned with outcomes which take the form of discrete variables. Obtaining such a discrete output can be done by using the perceptron model, which is a "hard classification" model where each data point is deterministically assigned to a category. In many cases, however, it is favorable to use a "soft classifier" that outputs the probability of a given category rather than a single value, which is where we apply logistic regression.

When we apply logistic regression the most common situation is having two possible outcomes, normally denoted as a binary outcome [1]. The probability that a data point x_i belongs to a category $y_i = \{0,1\}$ is given by the logistic function, also known as the Sigmoid function,

$$p(t) = \frac{1}{1+\exp -t} = \frac{\exp t}{1+\exp t}, \tag{3}$$

which is meant to represent the likelihood of a given event [1]. Assuming that we have two categories with $y_i \in \{0,1\}$ and that we only have two parameters β in the fit of the Sigmoid function, we define the probabilities

$$p(y_i = 1|x_i, \boldsymbol{\beta}) = \frac{\exp(\beta_0 + \beta_1 x_i)}{1+\exp(\beta_0 + \beta_1 x_i)} \tag{4}$$

$$p(y_i = 0|x_i, \boldsymbol{\beta}) = 1 - p(y_i = 1|x_i, \boldsymbol{\beta}), \tag{5}$$

where x is an input set and $\boldsymbol{\beta}$ are the weights we wish to extract from data, in this case β_0 and β_1 which are the coefficients we use to estimate the data.

Our aim is now to maximize the probability of seeing the observed data. Using the Maximum Likelihood Estimation (MLE), we define the total likelihood for all possible outcomes from a dataset $\mathcal{D} = \{(y_i, x_i)\}$ with the binary labels $y_i \in \{0,1\}$:

$$P(\mathcal{D}|\boldsymbol{\beta}) = \prod_{i=1}^{n} [p(y_i = 1|x_i, \boldsymbol{\beta})]^{y_i} [1 - p(y_i = 1|x_i, \boldsymbol{\beta})]^{1-y_i}, \tag{6}$$

which then is an approximation of the likelihood in terms of the individual probabilities of a specific outcome y_i [1]. From this we obtain the log-likelihood

$$\begin{aligned}\mathcal{C}(\boldsymbol{\beta}) = \sum_{i=1}^{n} (y_i \log p(y_i = 1|x_i, \boldsymbol{\beta}) \\ + (1-y_i)\log[1 - p(y_i = 1|x_i, \boldsymbol{\beta})]),\end{aligned} \tag{7}$$

which is a cost function. The maximum likelihood estimator is defined as the set of parameters that maximize the log-likelihood where we maximize with respect to β. The cost function is the negative log-likelihood, and so by reordering the logarithms, it can be rewritten as

$$C(\boldsymbol{\beta}) = -\sum_{i=1}^{n} (y_i(\beta_0 + \beta_1 x_i) - \log(1 + \exp(\beta_0 + \beta_1 x_i))). \tag{8}$$

This cost function, known as cross entropy, is what we use for logistic regression, and it is often supplemented with additional regularization terms.

We minimize the cross entropy cost function with respect to the two parameters β_0 and β_1, keeping in mind that this is a convex function of the weights $\boldsymbol{\beta}$, thereby making any local minimizer a global minimizer. By defining a vector $\boldsymbol{y}$ with n elements y_i, an $n \times p$ matrix $\mathbf{X}$ which contains the x_i values and a vector $\boldsymbol{p}$ of fitted probabilities $p(y_i|x_i, \boldsymbol{\beta})$, we find that the first derivative of the cost function becomes

$$\frac{\partial \mathcal{C}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -\boldsymbol{X}^T \left(\boldsymbol{y} - \boldsymbol{p}\right). \tag{9}$$

By defining a diagonal matrix $\mathbf{W}$ with elements $p(y_i|x_i, \boldsymbol{\beta})(1 - p(y_i|x_i, \boldsymbol{\beta}))$, we obtain an expression for the second derivative

$$\frac{\partial^2 \mathcal{C}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta} \partial \boldsymbol{\beta}^T} = \boldsymbol{X}^T \boldsymbol{W} \boldsymbol{X}. \tag{10}$$

In order to measure the performance of the classification problem we will use the accuracy score, which is the number of correctly guessed targets t_i divided by the total number of targets n,

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} I(t_i = y_i)}{n}, \tag{11}$$

where I is the indicator function, 1 if $t_i = y_i$ and 0 otherwise for a binary classification problem.

When performing the linear regression analysis we solved for the best value for $\boldsymbol{\beta}$ by taking the inverse. However, this is not always possible, and in such cases we can apply a method which takes advantage of numerical optimization, called gradient descent.

Gradient Methods

Previously, we have solved OLS and Ridge regression using an algorithm for matrix inversion. We now study another method for minimizing a function $\boldsymbol{f}(\boldsymbol{x})$.

Gradient descent, also known as Steepest Descent, is an optimization algorithm we use in order to find the minima of $\boldsymbol{f}(\boldsymbol{x})$, where $\boldsymbol{x} = (x_1, ..., x_n)$. A function such as this is expected to decrease fastest while going from x towards the direction of the negative gradient $-\Delta \boldsymbol{f}(\boldsymbol{x})$.

The method can be used in the training of a machine learning model, where it is applied to the convex cost function in order to minimize this to its local minimum. For a certain amount of iterative steps towards the direction of the minima, we will eventually reach a point where the cost function is at its smallest if

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \gamma_k \Delta \boldsymbol{f}(\boldsymbol{x}_k),$$

where the step length/learning rate $\gamma_k > 0$. If γ_k is sufficiently small we are always moving towards smaller function values, $\boldsymbol{f}(\boldsymbol{x}_{k+1} \leq \boldsymbol{f}(\boldsymbol{x}_k))$.

Ideally the sequence $\{\boldsymbol{x}_k\}_{k=0}$ converges towards a global minimum of the function $\boldsymbol{f}$, and this is always the case when $\boldsymbol{f}$ is a convex function, as all local minima are also global minima. While this scheme is simple and straightforward to implement, it has several limitations such as being sensitive to the chosen initial condition and being expensive to compute numerically.

The gradient descent method is sensitive to the choice of learning rate γ_k, due to the fact that we require a sufficiently small γ_k to reach the minima. Choosing a learning rate that is too small leads to the method taking a long time to converge, while choosing a too large learning rate can lead to erratic behaviour.

Stochastic Gradient Descent

Many of these shortcomings can be alleviated by introducing randomness. One such method is that of Stochastic Gradient Descent (SGD).

The cost function, which we want to minimize, can often be written as a sum over n data points $\{\boldsymbol{x}_i\}_{i=1}^n$,

$$\mathcal{C}(\beta) = \sum_{i=1}^{n} c_i(\boldsymbol{x}_i, \beta),$$

which means that the gradient can be computed as a sum over i-gradients,

$$\Delta_\beta \mathcal{C}(\beta) = \sum_{i}^{n} \Delta_\beta c_i(\boldsymbol{x}_i, \beta). \tag{12}$$

Stochasticity is then introduced by taking the gradient on a subset of the data called minibatches, denoted by B_k where $k = 1, ..., n/M$, with n being the number of data points and M being the size of each minibatch. We approximate the total gradient by replacing the sum over all data points with a sum over the data points in one of the minibatches, where the minibatches are chosen at random in each gradient descent step. For a number of batches $M < 1$ we have SGD with mini batches, while for $M = 1$ we simply have SGD. The gradient step then becomes

$$\beta_{j+1} = \beta_j - \gamma_j \sum_{i \in B_k}^{n} \nabla_\beta c_i(\mathbf{x}_i, \beta),$$

where k is chosen at random with equal probability from $[1, n/M]$ and n_j is the learning rate at the jth step.

By iterating over the gradients and weighting them with the learning rate γ_k we can find the minima,

$$\beta \leftarrow \beta - \gamma_k \Delta \mathcal{C}(\beta). \tag{13}$$

The algorithm iterates through the training set, updating β until it begins converging, where the convergence is calculated from the cost function.

Momentum based Gradient Descent

The SGD is usually used with a momentum term that served as a memory of the direction we are moving in parameter space. This algorithm is called Gradient Descent with Momentum (GDM), and is presented in algorithm 1.

Algorithm 1 Gradient Descent with Momentum

```
$k_1 \leftarrow hf(t_i, y_i)$                                        ▷ Define a variable $k_1$
while in epochs do                                                   ▷ Iterate through epochs
    while in mini-batches do                                         ▷ Iterate through the mini-batches
        $\Delta\boldsymbol{\theta}_{t+1} \leftarrow \gamma\Delta\boldsymbol{\theta}_t - \eta_t \nabla_\theta E(\boldsymbol{\theta}_t)$    ▷ $\Delta\theta_t = \theta_t - \theta_{t-1}$.
```

From the GDM algorithm we see that we have introduced a momentum parameter γ, with $0 \leq \gamma \leq 1$, for which we have that when $\gamma = 0$ this reduces to the ordinary SGD.

In SGD, both with and without momentum, we have to specify a schedule for tuning the learning rate η_t as a function of time. If the learning parameter is too small, the computations will be slow, and if it is too high we will never achieve acceptable loss. The learning rate is limited by the steepest direction which might change. We therefore keep track of curvature, taking large steps in flat directions and small steps in steep directions. The common method for achieving this, where we approximate the Hessian and normalize the learning rate by curvature, this can be computationally expensive for large models. Therefore, it is often preferable to use one of the several methods introduced that adaptively changes the step size to match the landscape without paying the steep computational price of calculating or approximating Hessians. Common methods used to do this for neural networks are the AdaGrad algoritm, Root Mean Squared Propagation (RMSprop), and Adam, summarized in the algorithms 2, 3, and 4 in appendix A, respectively.

Neural Networks

Artificial Neural Networks (ANN) are computational systems which learn to perform tasks based on examples, generally without being programmed with any task-specific rules. The aim is to mimic a biological system, wherein interconnected neurons send signals in the form of mathematical functions between layers, where each layer contains and arbitrary number of neurons and each connection is represented by a weight variable. An example of a simple neural network is the single perceptron model, which consists of one node with two inputs and one output, visualized in figure 1.

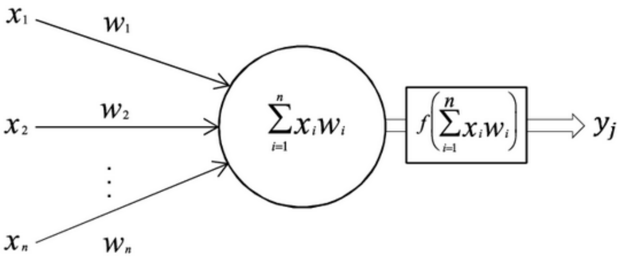

Figure 1: Illustration of the single perceptron model. The image is from the lecture notes.

Each node accumulates its incoming signals, which must exceed an activation threshold to yield an output. The input has a weight associated with it, W_x and W_y, and each node has a bias b and an activation function $\sigma(z)$. The output of the node is determined by the activation function, which takes a weighted sum of signals $x_i, ..., x_n$ received by n other neurons:

$$y = f\left(\sum_{i=1}^{n} w_i x_i\right) = f(u), \tag{14}$$

where the output y of the neuron is the value of its activation function, which will be discussed in further detail later [1].

We allow one node to take n inputs, enabling us to solve a linear regression problem of degree n, by viewing each input i as x_i, and each weight as a coefficient in the linear expression. By scaling the output and interpreting this as a probability, we may solve binary classification problems.

Feed-Forward Neural Network

If we add an additional layer between the input and output layer, we build a multi-layer perceptron model, visualized in figure 2. This method, FFNN, is a simple type of ANN which enables us to solve more complex models. In this case, the information only moves forward through the layers.

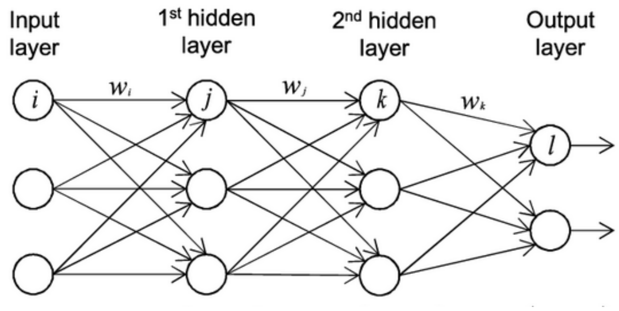

Figure 2: Illustration of the multi-layer perceptron model.

For this type of network, nodes n_l are arranged in an input layer, an output layer L and hidden layers $1, L-1$, where each layer can contain an arbitrary number of nodes, and each connection between two nodes is associated with a weight variable W_l. Each node, defined by a model function, passes information to the nodes ahead if it, causing input information to move without backtracking through the network from the input layer, through the hidden layers in between, and out to the output layer.

Activation Functions

The output of the neural networks will be a linear function of the inputs, and we therefore introduce the activation function to add some kind of non-linearity to the the neural network in order to fit non-linear functions. There are several typical choices for activation functions, of which we will use the sigmoid function, the Rectified Linear Unit (ReLU), and the Leaky ReLU.

The sigmoid function,

$$f(x)_{\text{sigmoid}} = \sigma(x) = \frac{1}{1+e^{x}}, \tag{15}$$

is inspired by probability theory and is commonly used in models where the output is a measure of probability. It is usually applied to the output layer, as applying it to the hidden layers often leads to vanishing gradients.

Another common activation function is the ReLU,

$$f(x)_{\text{ReLU}} = \max(0, x), \tag{16}$$

which has output in $[0, \infty]$. While the function is efficnent and does not saturate for positive values, it suffers from a problem known as the dying ReLUs, where some neurons effectively die during training. In such cases, the neurons stop outputting anything other than 0 [1]. There have been several attempts to solve this issue, one of which is known as the Leaky ReLU,

$$f(x)_{\text{Leaky ReLU}} = \begin{cases} x, & \text{if } x \geq 0 \\ x\alpha, & \text{if } x \leq 0' \end{cases} \tag{17}$$

where $\alpha = 0.01$ is a parameter that increases the range of the function such that it becomes $[-\infty, \infty]$.

The weights and biases in a network can be initialized randomly, however this makes them unlikely to produce an accurate prediction. We therefore adjust the weights and biases by training them, where we have to use gradient methods in order to find the minimum of the model's cost function. In order to compute the gradients of the cost function with respect to every weight and bias in the network, we use an algorithm called backpropagation, presented in appendix **??**.

Datasets

In order to test the optimization methods discussed in this project, we test the models on two datasets. For the regression model we use the Franke function, while for the classification model we use the Wisconsin Breast Cancer Data [2]. For both datasets we split the data into a train and test set, where 80% of the data will be used to train the set.

Franke Function

The dataset we use to analyze the regression models is the 2 dimensional Franke function, which is a weighted sum of four exponentials given as,

$$\begin{aligned} f(x,y) = &\frac{3}{4}\exp\left(-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}\right) \\ &+\frac{3}{4}\exp\left(-\frac{(9x+1)^2}{49} - \frac{(9y+1)}{10}\right) \\ &+\frac{1}{2}\exp\left(-\frac{(9x-7)^2}{4} - \frac{(9y-3)^2}{4}\right) \\ &-\frac{1}{5}\exp\left(-(9x-4)^2 - (9y-7)^2\right), \end{aligned} \tag{18}$$

defined for $x, y \in [0, 1]$. This function is common to use in order to evaluate different surface interpolation techniques. We sample the function at 100 uniformly distributed data points, including stochastic noise ϵ,

$$z = f(x,y) + \epsilon,$$

where f is the Franke function and the noise is generated from a normal distribution $\epsilon \sim N(0, \sigma = 0.25)$. The data is fitted to a polynomial of degree 6.

Breast Cancer data

The second dataset we study is the Wisconsin Breast Cancer data set, which is a typical binary classification problem with one single output, making it useful for testing machine learning algorithms. The set was created in 1995 consists of images representing various features of tumors. The number of instances is 569, and of these 212 are malignant, 0, and 357 are benign, 1. The data is collected from the University of California Irvine (UCI) Machine Learning Repository [2].

In order to study the Wisconsin Breast Cancer data set, we change the cost function for the neural network code such that it can perform a classification analysis. We also want to compare the FFNN code to Logistic regression, and therefore define the cost function and the design matrix before writing a Logistic regression code using the SGD algorithm.

To measure the performance of the classification problem we use the accuracy score, which is the number of correctly guessed targets t_i divided by the total number of targets,

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} I(t_i = y_i)}{n},$$

where I is the indicator function, 1 if $t_i = y_i$ and 0 otherwise for a binary classification problem. Here, t_i represents the target and y_i the output of the FFNN code, while n is simply the number of targets t_i.

III. RESULTS AND DISCUSSION

Regression Analysis

Problem with argument: SDG takes longer. Need number of iterations to be equal to epoch times batches, because then If we take the same time, 1000 iterations, 100 epochs

SGD should approach GD when we only have 1 batch (how we check that we ar right At a certain point the MSE is no better for greater batch size, batch size 10, 100 epochs

SGD with mini-batches and a given number of epochs (tunable learning rate) 3) batch size against learning rate If the learning rate is too high, it does not converge Higher batch size, lower error Time vs benefit

Ridge: Same result as for linreg probably. Error is lower for ridge -¿ 0 finally compare with what we did last time

We began by analyzing a regression problem using gradient descent and stochastic gradient descent, which replaced the matrix inversion algorithm previously tested in project 1. The models were first tested on the Franke function. As we previously found that the linear regression codes with matrix inversion worked well for an order 6, we continue to use this throughout this report.

In figure 3 we study how the MSE of the GD method varies with epochs, for three fixed learning rates; $\eta = 0.1$, $\eta = 0.01$ and $\eta = 0.001$. The MSE decreases earlier and more rapidly for a higher learningrate. If the learningrate is made too high, the method would be unable to converge due to overflow, while for lower learningrates the model becomes less accurate. This is further supported by figure 4, where we visualize the MSE as a function of both the l2 parameter and the learningrate. The number of epochs is set to 300, and the number of batches is set to 64. The batches and number of operations have been plotted as a colormap where we see the MSE in figure **??** in the appendix. A greater number of batches mean that the convergence rate increases, and we spend less time on each operation. From figure 4, we see that we obtain an MSE of 0.06854 for $\eta = 0.1$, and the lowest l2 parameter.

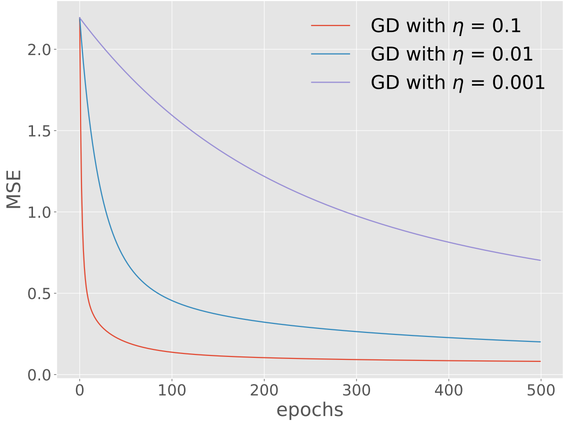

Figure 3: How the MSE for Franke's function varies with epochs for the GD method, for fixed learning rates $\eta = 0.1$, 0.01 and 0.001.

We implemented several methods for tuning the learning rate, presented in figure **??** in the appendix. All new methods converge faster than the GD method does, with the RMSprop method being the fastest and most stable method. Both the Adagrad with and without momentum also perform well. However, after a certain number of operations the MSE is low for all methods, which can also be seen in table **??**.

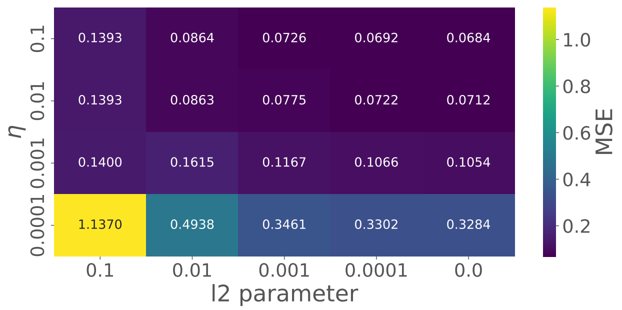

Figure 4: The MSE for Franke's function for the SGD method as a function of η and the l2 parameter, where the number of epochs is 500 and the number of batches is 64.

Method	GD		SGD	
η	0.1	0.01	0.1	0.01
Time per epochs [ms]	0.1224	0.1097	0.2733	0.2616
Time per operation [ms]	0.1224	0.1097	0.0683	0.0554

Table I: Speed per epochs of GD and SGD, for $\eta = 0.1$ and $\eta = 0.01$. The number of epochs was set to 500 and the number of batches to 4.

In table I we have studied the speed of the GD and SGD in further detail, by calculating the speed per epoch and speed of each operation for $\eta = 0.1$ and $\eta = 0.01$. While the SGD is slower per epochs, it is much faster per step, as gradients are computed from 5 poins at a time, rather than 1000 (N/M) ? Due to the randomness of SGD, it is possible for a local minima to be overcome as there is no confinement. Overall, the SGD method seems to overall be an improvement to the GD method, achieving the same MSE as GD in a shorter amount of time.

We analyze the same regression problem for the Franke function using the FFNN implementation. We begin by using the Sigmoid function as an activation function for the hidden layers, initializing the weights using a normal distribution.

The method will be affected by the number of hidden layers, nodes, batches, epochs and the learning parameters. We began by analyzing the number of hidden layers and nodes, using the values based on what we found for SGD in the previous analysis for the remaining parameters. The grid search for the optimal MSE depending on hidden layers and nodes is presented in figure **??**, in the appendix. The best results are obtained for 3 hidden layers and 30 neurons, and we use these values when investigating the choices for the batches and epochs, presented in figure **??** in the appendix. Based on this analysis, we see how we need fewer iterations if the number of batches is high enough. Using a high number of batches spends more computational time, but it can also save time for the right number of iterations. Compromising between minimizing the MSE and keeping the computational time satisfactory, we set the batches to 32.

Now we may perform the grid search for the learning rate η and optimization parameter λ, presented in figure 5. We see how, for higher values for the learning parameter, the accuracy increases. However, for too high values of η we see that no convergence happens, represented by the gray areas. Similar to what we saw for the logistic regression analysis, the optimization parameter has the lowest MSE for $\lambda = 0.00001$, and the lowest MSE overall is where $\eta = 0.1$. (Ridge is no improvement then?)

Next, we test how different activation functions affect these results, for the same parameters. There is some improvement to the MSE when using the ReLu as an activation function, seen in figure 6, while the Leaky ReLu activation function shows no improvement to the MSE, seen in figure 7. The lowest MSE=0.0691 is obtained when using ReLu is not much of an improvement on the Ridge regression code from the previous project where we had MSE=0.0112 when using cross-validation as the resampling technique. However, both the SGD and NN are better than the OLS and Ridge regression codes when not using cross validation.

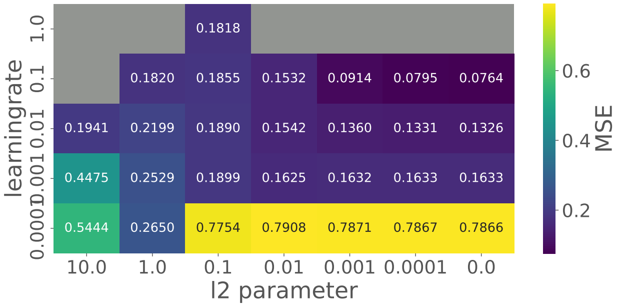

Figure 5: How the MSE for Franke's function varies with η and λ, using the Sigmoid function as an activation function. Hidden layers = 3, Neurons = 30, number of batches = 32, epochs = 300.

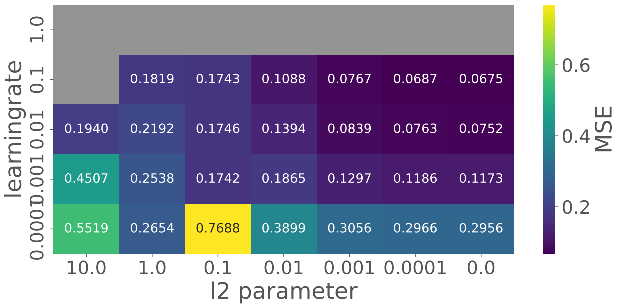

Figure 6: How the MSE for Franke's function varies with η and λ, using the ReLu as an activation function. Hidden layers = 3, Neurons = 30, number of batches = 32, epochs = 300.

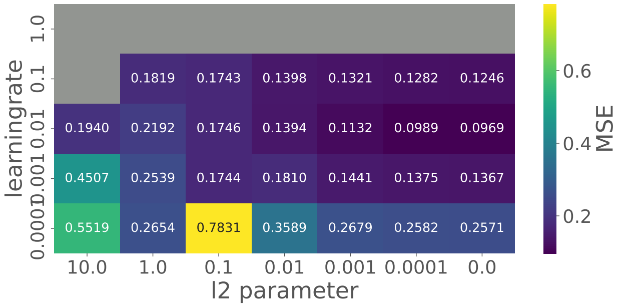

Figure 7: How the MSE for Franke's function varies with η and λ, using the Leaky ReLu as an activation function. Hidden layers = 3, Neurons = 30, number of batches = 32, epochs = 300.

Classification Analysis

When we study the Wisconsin Breast Cancer data set, we change the cost function for the neural network code in order to perform a classification analysis. The performance is measured with the accuracy score.

We begin by finding suitable parameters for the number of hidden layers and neurons, seen in the heatmap in figure **??**, and then the number of batches and epochs, presented in figure **??**, both in the appendix. The resulting lowest MSE for the neural network code is 0.9825, for $\eta = 0.1$ and an l2 parameter of 0.00001.

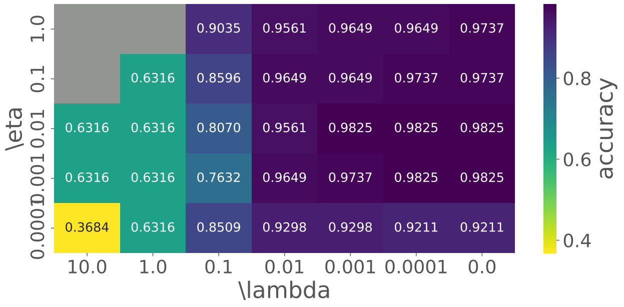

Figure 8: How the MSE for the Wisconsin Breast Cancer data varies with η and λ, using the Sigmoid function as an activation function. Hidden layers = 1, neurons = 25, number of batches = 64, epochs = 768

Finally, we want to compare the neural network classification results with the results we are able to obtain using another method, in this case the results we obtain from the logistic regression code using the SGD algorithm.

We begin by looking at the accuracy for SGD for different l2 parameters and different learningrates, comparing the GD algorithm, seen in figure 9 with ADAM seen in figure 10. The accuracy when using ADAM are somewhat higher than the results with SGD, with a difference of around 0.01 between the highest accuracy scores for both. The best scores with SGD are for a low learningparameter, while the best scores with ADAM are for a slightly higher learningparameter, at 0.0001.

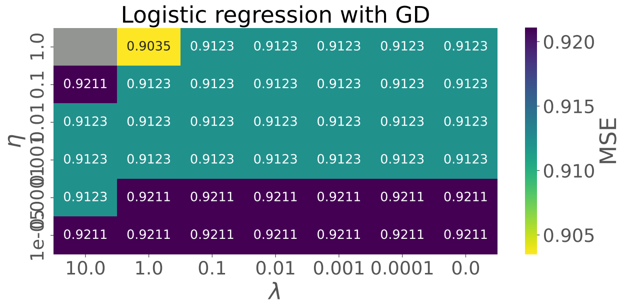

Figure 9: Accuracy score for SGD as a function of the learningparameter η and the l2 parameter λ, for epochs = 1024 and batches = 1.

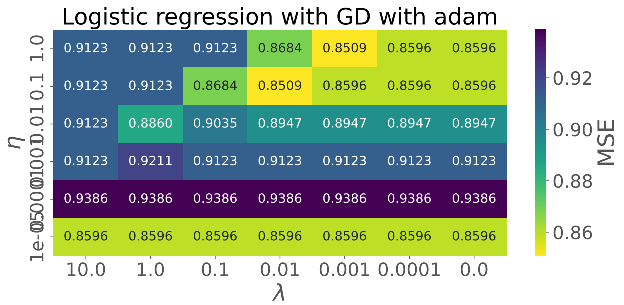

Figure 10: Accuracy score for ADAM as a function of the learningparameter η and the l2 parameter λ, for epochs = 1024 and batches = 1.

Next, we study the accuracy for SGD with ADAM for a different number of batches and number of operations, seen in figure 11.

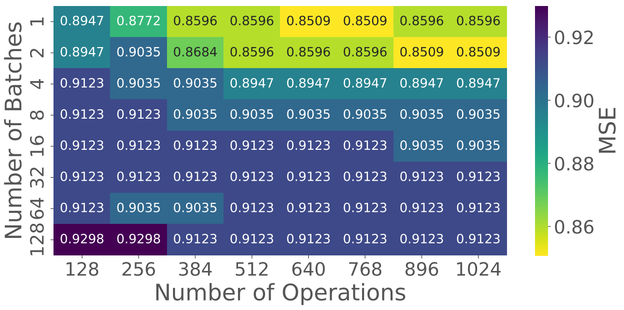

Figure 11: Accuracy score using logistic regression as a function of the number of batches and the number of operations, for epochs = 1024, batches = 1, eta=0.001, and l2=0.0

Finally, we can compare our results with the accuracy we obtain when using scikit learn's logistic regression function, for which we obtain a test set accuracy of 0.9474. This is higher than what we obtained using our own logistic regression code, where we obtain an accuracy of 0.9386 when using ADAM. It is still lower than the score we got for the neural networks code, where we obtained an accuracy score of 0.9825.

IV. CONCLUSION

When studying the regression problem for the Franke function with the FFNN implementation, we have used a fixed number of hidden layers and nodes. Therefore, we are limited by the layers and nodes we set based on the grid search where we search for the best combination of these two parameters. As the parameters are selected based on values for the number of batches, epochs, and learning parameter that might not be the most optimal for the model, this somewhat limits the accuracy of the analysis, and it is possible that even better paramaters could be found for the FFNN implementation. While the MSE values were satisfactory and the models can still be considered accurate, further studies could attempt to implement these in a manner that allows us to study the effect of every parameter in further detail.

REFERENCES

[1] M. Hjorth-Jensen. *Applied Data Analysis and Machine Learning*. Jupyter Book. 2021.

[2] University of California at Irvine. *Breast Cancer Wisconsin (Diagnostic) Data Set*. Last accessed 8. november 2022. 2022. URL: https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+\%28Diagnostic\%29.

Appendix A: Algorithms

Adagrad

The AdaGrad algorithm adaptively scales the learning rate for each dimension. We implement it by iterating over the epochs, and then for every epoch we iterate through the mini-batches, as explained in algorithm 2.

Algorithm 2 Adagrad

```
while in epochs do                                 ▷ Iterate through epochs
    while in mini-batches do                       ▷ Iterate through the mini-batches
        $\mathbf{g}_t \leftarrow \nabla_\theta \mathcal{C}(\boldsymbol{\theta})$
        $\theta_{t+1} \leftarrow \mathbf{g}_t \eta \frac{1}{\sqrt{\delta + \sum^t (\mathbf{g}_t)^2}}$
```

Root Mean Squared Propagation

RMSprop provides an exponentially decaying average rather than the sum of the gradients. The decaying average is realized by combining the momentum algorithm and the Adagrad algorithm with a new term. The RMSprop method is restricted to the sum of the past gradients, in addition to the gradients for the recent time steps, meaning that it changes the learning rate slowly while converging relatively fast [1].

Algorithm 3 RMSprop

```
while in epochs do                                 ▷ Iterate through epochs
    $k \leftarrow 0$
    while in mini-batches do                       ▷ Iterate through the mini-batches
        $\mathbf{g}_t \leftarrow \nabla_\theta \mathcal{C}(\boldsymbol{\theta})$
        $k \leftarrow (\rho k + (1-\rho)\mathbf{g}_t \mathbf{g}_t)$          ▷ Scaling with $\rho$
        $\theta_{t+1} \leftarrow \mathbf{g}_t \eta \frac{1}{\sqrt{\delta + k}}$          ▷ Inverting the diagonal
```

Adam

Another related algorithm is the Adam optimizer, which is efficient when working with large problems involving a lot of data and parameters. The algorithm 4 keeps a running average of both the first and second moment of the gradient and uses this information to adaptively change the learning rate for different parameters.

Algorithm 4 ADAM

```
$\mathrm{m}_0 \leftarrow 0.0$                                    ▷ Initialize first moment
$\mathrm{s}_0 \leftarrow 0.0$                                    ▷ Initialize second moment
while in epochs do                                 ▷ Iterate through epochs
    while in mini-batches do                       ▷ Iterate through the mini-batches
        $\mathbf{g}_t \leftarrow \nabla_\theta \mathcal{C}(\boldsymbol{\theta})$                       ▷ Get gradients
        $\mathbf{m}_t \leftarrow \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t$       ▷ Update biased 1st moment
        $\hat{\mathbf{m}}_t \leftarrow \frac{\mathbf{m}_t}{1-\beta_1^t}$                    ▷ Compute bias-corrected 1st moment
        $\mathbf{s}_t \leftarrow \beta_2 \mathbf{s}_{t-1} + (1-\beta_2)\mathbf{g}_t^2$     ▷ Update biased 2nd moment
        $\hat{\mathbf{s}}_t \leftarrow \frac{\mathbf{s}_t}{1-\beta_2^t}$                    ▷ Compute bias-corrected 2nd moment
        $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \eta_t \frac{\mathbf{m}_t}{\sqrt{\mathbf{s}_t}+\epsilon}$         ▷ Update parameters
```

The parameters β_1 and β_2 set the memory lifetime of the first and second moment and are typically taken to be 0.9 and 0.99 respectively, and η is the learning rate typically chosen as 10^{-3}, and ϵ is a small regularization constant to prevent divergences.

Appendix B: Back Propagation

Train data based on the cost function (compare network output with training data) minimize the cost output (method of adjusting weights and biases by estimating the gradient of the cost function, back propagation)

f is the activation function f'(zj) is the derivative for activation z j is node k is class or entry L is layer w and b are the weights and biases

weight and biase in teh network are typically initialized randomly -¿ highly unlikely to find good predictions Need to iteratively adjust the weights and biases until the predictions are satisfactory Called training: We use gradient methods to find the minimum of the models cost functions. Now need to compute the gradients of the cost function with respect to all the weights and biases in the network, which has potentially many weights and biases all dependent on one another, and can become coputationaly heavy We therefore make use of the algorithm backpropagation

Initializing the weights and biases. to get fat covergence

Method	Time [s]	
	299 iterations	1025 iterations
GD	0.0922	0.0731
GD w/ momentum	0.0841	0.0717
Adagrad	0.0875	0.0722
Adagrad w/momentum	0.0796	0.0705
RMSprop	0.0710	0.0691
ADAM	0.0689	0.0670

Table II

or non convergence of the gradient method. Initialize the biases to some small value eprioon (can experiment to find it) and the weights with iniform distribution between -1 and 1 .

The four equations provide us with a way of computing the gradient of the cost function, for which the method we use is presented in the algorithm **??** for back propagation, where the parameter η is the learning parameter discussed in connection with the gradient descent methods. XXX: Here it is convenient to use stochastic gradient descent (see the examples below) with mini-batches with an outer loop that steps through multiple epochs of training.

Algorithm 5 Back Propagation

```
XXX    ▷ Set up the input data $\hat{x}$ and the activations $\hat{z}_1$ of the input layer and compute the activation function and the pertinent outputs $\hat{a}^1$
XXX ▷ Perform the feed forward until we reach the output layer. Compute all $\hat{z}_l$ of the input layer and the activation function and the pertinent outputs $\hat{a}^l$ for $l = 1, 2, 3, ..., L$.
$\eta = XXX$    ▷ Initialize the learning rate
$\delta_j^L \leftarrow f'(z_j^L)\frac{\partial \mathcal{C}}{\partial (a_j^L)}$    ▷ Compute the output error $\delta^L$
$\delta_j^l \leftarrow \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l)$    ▷ Compute the back propagate error for each $l = L-1, L-1, ..., 2$
$w_{jk}^l \leftarrow w_{jk}^l - \eta \delta_j^l a_k^{l-1}$, ▷ Update the weights and the biases using gradient descent
$b_j^l \leftarrow b_j^l - \eta \frac{\partial \mathcal{C}}{\partial b_j^l = b_j^l - \eta \delta_j^l}$
    =0
```

When all weights and biases throughout the network have been updated a number epoch times, the cost function preferably decreases.

The lambda is to prevent overfitting. Preventing overditting. Introduce a regularization term (lambda)

Appendix C: Other Figures

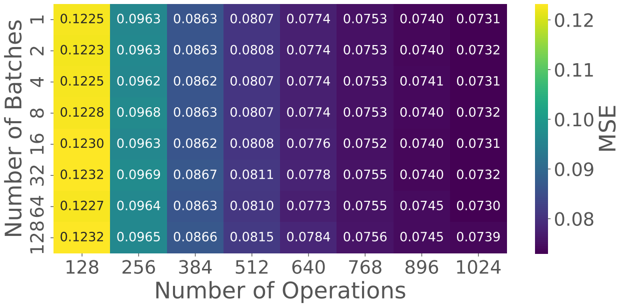

Figure 12: How the MSE for Franke's function varies with the batch size and number of operations, for $\eta = 0.1$.

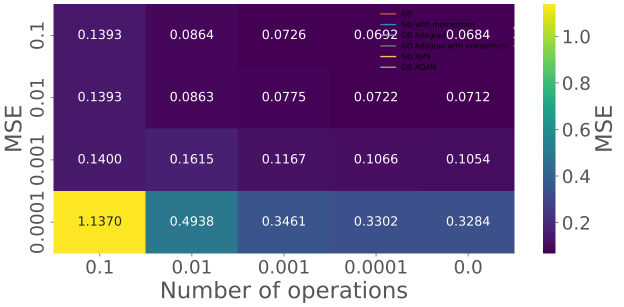

Figure 13: How the MSE for Franke's function varies with epochs for the GD method, for $\eta = 0.01$.

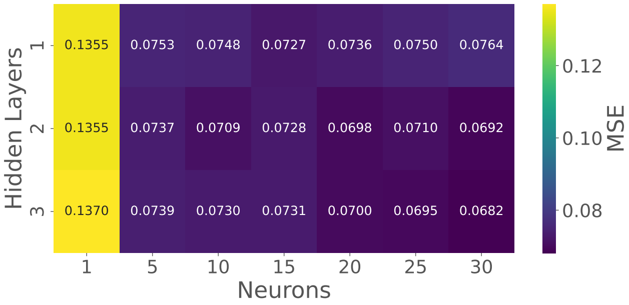

Figure 14: How the MSE for Franke's function varies with the number of hidden layers and neurons, using the Sigmoid function as an activation function. Number of batches = 64, epochs = 300, $\eta = 0.1$.

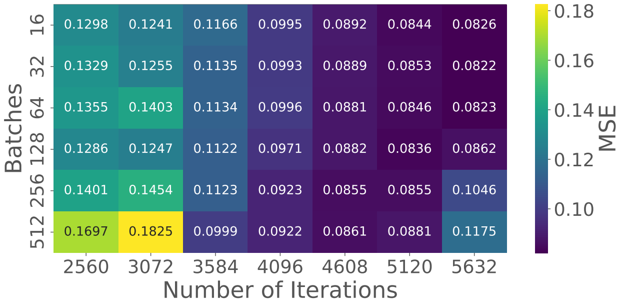

Figure 15: How the MSE for Franke's function varies with the batches and epochs, using the Sigmoid function as an activation function. Hidden layers = 3, Neurons = 30, $\eta = 0.1$, and l2 = 0.00001

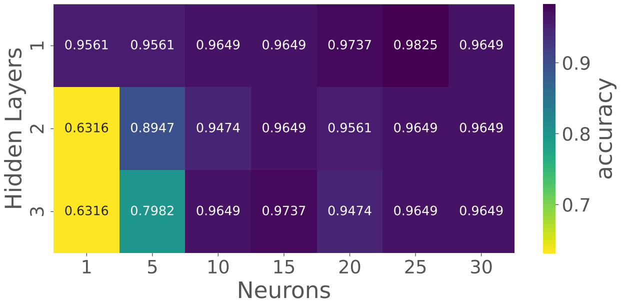

Figure 16: How the MSE for the MSE for the Wisconsin Breast Cancer data varies with the batches and epochs, using the Sigmoid function as an activation function. Batches = 1, epochs = 640, η = 0.1, l2=0.0

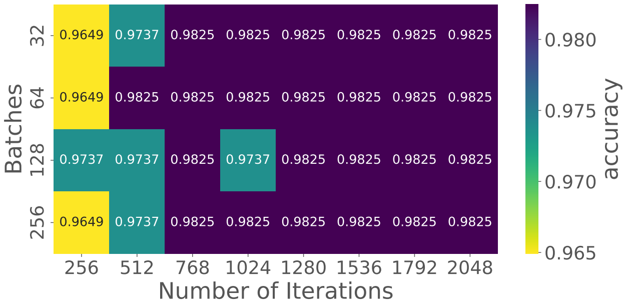

Figure 17: How the MSE for the MSE for the Wisconsin Breast Cancer data varies with the batches and epochs, using the Sigmoid function as an activation function. Hidden layers = 1, neurons = 25, η = 0.1, l2=0.0.