Logistic Regression and Feed Forward Neural Network (FFNN) analysis on the Wisconsin breast cancer dataset and Franke's function dataset

Aram Salihi[1], Martine Tan [2] & Andras Filip Plaian [2]

[1]Department of Informatics,University of Oslo, N-0316 Oslo, Norway

Department of Mathematics[2], University of Oslo, N-0316 Oslo, Norway

November 15, 2019

Abstract

In this project we have developed our own implementation of logistic regression and feed-forward neural network in Python, and use these implementations to analyze the Wisconsin breast cancer dataset, as well as the Franke's function dataset. The latter dataset is one we have generated using Franke's function. It consists of $10,000$ observations, to which we have added noise drawn from the Gaussian distribution with mean 0.001 and standard deviation 0.1. We concluded that the feed-forward neural network outperforms logistic regression at classification. The feed-forward neural network was able to predict whether patients had breast cancer or not with an 99.1% accuracy score. With logistic regression, the accuracy score was 96.5%. The feed-forward Neural Network was able to estimate Franke's function with a mean squared error of 0.01002, and R2-score of 0.89386. This score is very close to the score we obtain with ridge regression: a mean squared error of 0.00999 and an R2-score of 0.89165. However, the neural network algorithm has potential for further improvement, and performance may be improved by fine tuning the network parameters. Both the logistic regression and feed-forward neural network implementation use stochastic gradient descent as optimization method.

Contents

1 Introduction

In this study, we will look at two data sets. The first is the well known Wisconsin breast cancer data set, which can be downloaded from `https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)`. The second data set is generated using Franke's function (see section 2.8).lo The Wisconsin breast cancer data set is a classification problem, and the goal is to accurately predict whether a person has breast cancer or not, based on a number of relevant measurements. To this end, we will develop two methods for classification: Logistic regression, and Feed-Forward Neural Network (FFNN) i.e. Multi-layer Perception (MLP). The generated Franke's function data set is a regression problem, and so the goal is to predict the value $z = f(x, y)$, based on given x and y values. We will use our Feed-Forward Neural Network to attempt to estimate Franke's function.

In our previous study of regression methods, we developed algorithms for linear, ridge, and lasso regression, with k-Fold Cross-Validation (see [Salihi Aram, 2019]). We will compare the results of our Feed-Forward Neural Network for regression to these methods, to determine which is able to estimate Franke's function most accurately.

2 Theory

2.1 Logistic Regression:

Logistic regression is a way of classifying a input to a correct output for a given problem for some arbitrary parameters. In other words we want to model the posterior probabilities of K classes as linear functions for a given input x and ensuring that the sum is one and remain in $[0, 1]$. This form of regression is widely used when the problem has two outputs, for example true or false. For example does the patience have diabetes given age, gender, height and weight? The answer is either yes or no. The mathematical model of this type of regression for K classes is given as

$$\log\left(\frac{\Pr(G = 1|X = x)}{\Pr(G = K|X = x)}\right) = \beta_{10} + \beta_1^T x \tag{1}$$

$$\vdots$$

$$\log\left(\frac{\Pr(G = K-1|X = x)}{\Pr(G = K|X = x)}\right) = \beta_{(k-1)0} + \beta_{k-1}^T x \tag{2}$$

2.2 Cost function for logistic regression

Let $\boldsymbol{\beta} = [\beta_0, \beta_1, ..., \beta_p]$ be the vector of regression coefficients, and $X_i^* = [1, X_{i,1}, X_{i,2}, ..., X_{i,p}]$ the vector of predictor values for the i-th observation. For each pair of data points (y_i, X_i) we have the probability

$$p(y_i = 1|X_i^*\boldsymbol{\beta}) = \frac{\exp(\beta_0 + \beta_1 X_{i,1} + ... + \beta_p X_{i,p})}{1 + \exp(\beta_0 + \beta_1 X_{i,1} + ... + \beta_p X_{i,p})} = \frac{\exp(X_i^*\boldsymbol{\beta})}{1 + \exp(X_i^*\boldsymbol{\beta})}$$

and the corresponding probability

$$p(y_i = 0|X_i^*\boldsymbol{\beta}) = 1 - p(y_i = 1|X_i^*\boldsymbol{\beta}).$$

Then for our data set $D = \{(y_i, X_i^*)\}_{i=1}^n$ we have the following probability of seeing the observed data

$$P(D|\boldsymbol{\beta}) = \prod_{i=1}^{n} \left[p(y_i = 1|X_i^*\boldsymbol{\beta})\right]^{y_i} \left[1 - p(y_i = 1|X_i^*\boldsymbol{\beta})\right]^{1-y_i}.$$

We want to maximize this probability. The log-likelihood function is then

$$\begin{aligned}
l(\boldsymbol{\beta}) &= \sum_{i=1}^{n} \Big(y_i \log \Big[p(y_i = 1|X_i^*\boldsymbol{\beta})\Big] + (1-y_i)\log\Big[1 - p(y_i = 1|X_i^*\boldsymbol{\beta})\Big]\Big) \\
&= \sum_{i=1}^{n} \Big(y_i \log \Big[\frac{\exp(X_i^*\boldsymbol{\beta})}{1+\exp(X_i^*\boldsymbol{\beta})}\Big] + (1-y_i)\log\Big[1 - \frac{\exp(X_i^*\boldsymbol{\beta})}{1+\exp(X_i^*\boldsymbol{\beta})}\Big]\Big) \\
&= \sum_{i=1}^{n} \Big(y_i\Big(X_i^*\boldsymbol{\beta} - \log\big[1+\exp(X_i^*\boldsymbol{\beta})\big]\Big) + (1-y_i)\Big(-\log\big[1+\exp(X_i^*\boldsymbol{\beta})\big]\Big)\Big) \\
&= \sum_{i=1}^{n} \Big(y_i X_i^*\boldsymbol{\beta} - \log\big[1+\exp(X_i^*\boldsymbol{\beta})\big]\Big)
\end{aligned}$$

The cost function is just the negative log-likelihood, and thus we want to minimize this cost function

$$C(\boldsymbol{\beta}) = -\sum_{i=1}^{n} \Big(y_i X_i^*\boldsymbol{\beta} - \log\big[1+\exp(X_i^*\boldsymbol{\beta})\big]\Big)$$

To find the optimal parameter $\boldsymbol{\beta}$ that minimizes the cost function, we will use stochastic gradient descent (SGD). For further detailed explanation of this topic, please see [Hastie et al., 2001].

2.3 Gradient/steepest descent (GD)

For a given continuous and differentialable multivariate function $F(\mathbf{x})$ there exist a set of solution, where this function has global and local minimas. This method is based on the idea that the gradient of $F(\mathbf{x})$ is where the function decreases fastest in the negative direction of point $\mathbf{a}$. Thus $-\nabla F(\mathbf{a})$. It follows that

$$\mathbf{a}_{n+1} = \mathbf{a}_n - \gamma \nabla F(\mathbf{a}_n) \tag{3}$$

Where $\gamma \in \mathbb{R}^m$ is a small number. With this in mind, we have that $F(\mathbf{x}_0) \geq F(\mathbf{x}_1 \geq F(\mathbf{x}_2) \geq ....$ Thus a decreasing monotonic sequence. The down side with numerical method is when the size of the evaluated data points is large. In machine learning we want to minimize the gradient of the cost function by finding a solution set of $\boldsymbol{\beta}$ which achieve this. The data sets involved are often large, and thus the evaluation of the gradient can cost a lot of computational time. This decreases the efficiency of this method when evaluating large data sets. Thus we want to use a variant of this method which is more optimized for such tasks. In the next section we will present the stochastic gradient descent which is an optimized version of gradient descent.

2.4 Stochastic gradient descent (SGD)

A variant of the famous gradient descent/steepest descent (GD) is the stochastic gradient decent (SGD). The main difference between steepest/gradient descent and stochastic gradient is that we divide matrix $C(\boldsymbol{\beta})$ into mini batches of size M, where the number of vectors are n/M. We then want to randomly pick out one row vector from our matrix and evaluate the gradient of it. To do so we want to write the cost function as of row vectors

$$C(\boldsymbol{\beta}) = \sum_{i=1}^{N} \mathbf{c}(x_i, \beta_i) \tag{4}$$

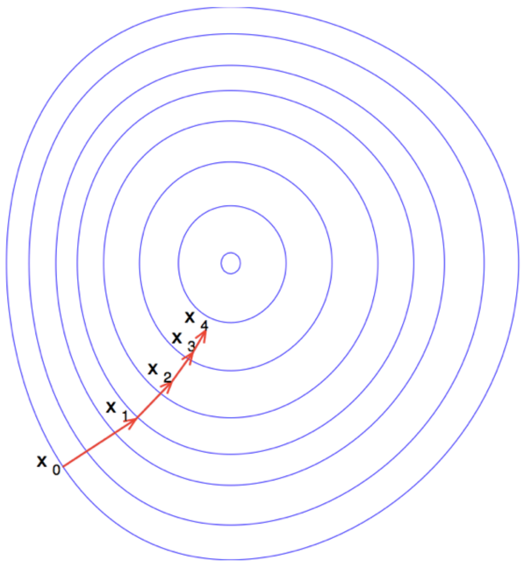


Figure 1: An illustration of how gradient descent works

We then define a batch vector as B_k where k is picked at random, thus we want to sum over the i'th index which chooses a random picked vector from B_k

$$\sum_{i \in B_k} \mathbf{c}(x_i, \beta_i) \tag{5}$$

Sgd will then look like

$$\beta_{j+1} = \beta_j + \eta \nabla \sum_{i \in B_k} \mathbf{c}(x_i, \beta_i) \tag{6}$$

Where η is the learning rate. To apply this gradient method, we need the derivative of the cost function. Let $\boldsymbol{p}$ be the vector containing the probabilities $p(y_i = 1 | X_i^*, \boldsymbol{\beta})$. Then we have the derivative

$$\begin{aligned}
\frac{\partial C(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} &= -\sum_{i=1}^{n} \left[y_i X_i^* - \frac{\exp(X_i^* \boldsymbol{\beta})}{1 + \exp(X_i^* \boldsymbol{\beta})}) X_i^* \right] \\
&= -\sum_{i=1}^{n} X_i^* \left(y_i - \frac{\exp(X_i^* \boldsymbol{\beta})}{1 + \exp(X_i^* \boldsymbol{\beta})} \right) \\
&= -\sum_{i=1}^{n} X_i^* (y_i - p_i) \\
&= -X^T (\boldsymbol{y} - \boldsymbol{p})
\end{aligned}$$

We split the data into mini batches of size M such that there are n/M mini batches B_k for $k = 1, 2, ..., n/M$. Each stochastic gradient descent step is then

$$\beta_{j+1} = \beta_j - \gamma_j \sum_{i \in B_k} -X_i^* (y_i - p_i)$$

where the k is picked at random, with equal probability, from the interval [1, n/M].

2.5 Feed Forward Neural Network, single layer preceptor

The most simple method for deep learning is the so called Feed Forward Neural Network. We will now consider three layers, one for input, hidden and output layer.

- Input layer: Consider $i \in \{1, ..., N\}$ inputs nodes, where each of these nodes takes one value from a data set $\mathbf{x}_i$. This node will then pass the input information to the next node (in our case the hidden node)
- Hidden layer: We only have one layer consisting of some nodes which evaluates the information from all the input nodes. This is simply done by summing over all of the input values. The quantity which is calculated is called the activation

  $$z_i^1 = \sum_{j=1}^{N} w_{ij} x_i + b_i$$

  , where w_{ij} is the weight to fit the data and b_j is the bias. The task of these nodes is then to activate a so called activation function $f(z_i^1$ in order to get the processed data a_i^1.

  $$a_i^1 = f^1\left(z_i^1\right) = f^1\left(\sum_{j=1}^{N} w_{ij}^1 x_i + b_i^1\right) \tag{7}$$

  The activation function depends of what type neural network this is, in our case this is a classification neural network. For a classification network, this function is a sigmoid function (to avoid overflow, we have used scipy.special.expit)

  $$f(z_l^1) = \frac{e^{z_l^1}}{e^{z_l^1} + 1} \tag{8}$$

  For regression network, it is enough just to pass the data through. I.e to reproduce ordinary linear regression, but this is not enough in most cases. This will be presented in detail in section (**??**). When all of this is done, the processed data is then sent to another layer for further evaluation, in this case the output layer.
- Output layer:

  #### 2.5.0.1 Classification

  When the data is evaluated by the hidden layer, we must check the probability of which class the evaluated data point belongs to. This is given by the soft max function. Consider the a_i^1 from the hidden layer, the soft max function is

  $$\sigma(a_i^1) = \frac{\exp\left(a_i^1\right)}{\sum_{i=1}^{G-1} \exp(a_i^1)} \tag{9}$$

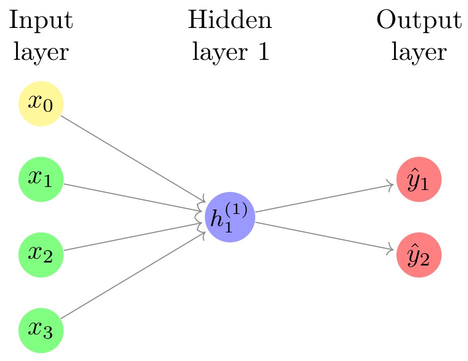
  Where G is the number of categories which depends on the problem.

  #### 2.5.0.2 Regression

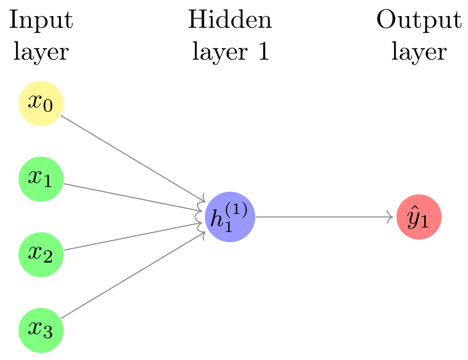
  When it comes to regression Neural Network the need of a evaluation function is not necessary, meaning we just want a linear activation function in the last layer. Thus $f(z_i^L) = z_i^L$. What this function does only passing the information of the last hidden layer as the final result in the output layer.

This can be illustrated as an figure shown down below

2.5.0.3 Classification SLP



2.5.0.4 Regression SLP



2.6 Multilayer perceptron FFNN

Instead of having one input, hidden and output layer. We now want to create a multilayer system with L layers. One layer for input, with N nodes, $L-2$ hidden layers with M_l nodes, and one ouput layer with 2 nodes. The idea behind this is that each neuron from the input sends the input to another neuron in the hidden layer.

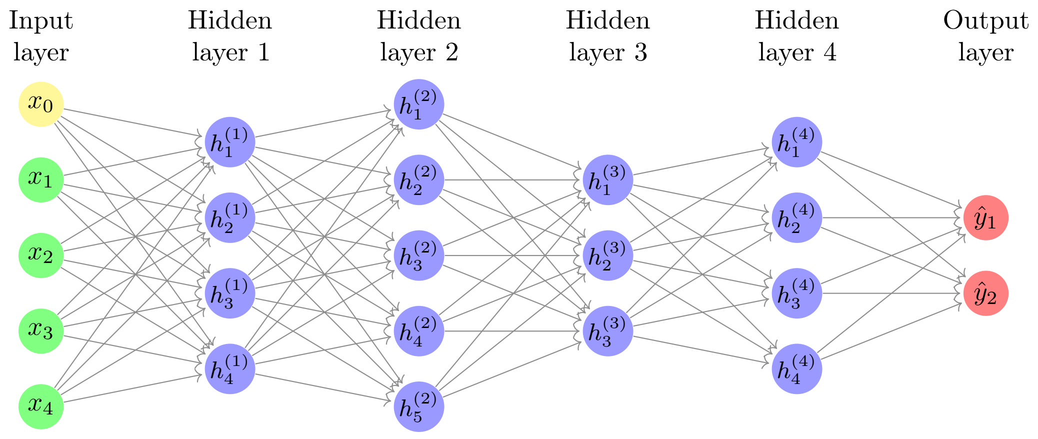
When the information comes the hidden layer, the information is evaluated and processed and sent to next node in another hidden layer. We will denote the processed information as, $a_i^l = f^l(z_i^l)$ where $l \in \{2, ..., l-1\}$ layers, and $i \in \{1, ..., M_l\}$ nodes. Where the activation is mathematically defined as

$$a_i^l = f^l(z_i^l) = f^l\left(\sum_j^{M_{l-1}} w_{ij}^l a_j^{l-1} + b_i^l\right) \tag{10}$$

Keep in mind that this can be formulated as matrix multiplaction

$$\mathbf{a}^l = f^l(\mathbf{z}^l) = f^l\left(W^l \mathbf{a}^{l-1} + \mathbf{b}^l\right) \tag{11}$$

When the output of all the nodes in the hidden layers are computed, the values of the subsequent layer can be calculated and so forth until the output is obtained. In this layer the, we have the softmax as previous mentioned, which normalizes and sets up an probability distribution. We then want to back propagate in order to achieve a best fit of the weights W and the bias b. An example of mlp FFNN is illustrated down below.



This is neural network consisting of one input (7 nodes), four hidden and one output layer (2 nodes).

2.7 Backpropagation in MLP FFNN

The goal of any supervised learning is to find a function or weights which maps the best set of inputs to their correct output. In neural network, we use backpropagation to achieve this. The idea is: when we have proceeded through each layer, and arrived at the output layer at layer L, we then wish to minimize the error by calculating the gradient to the cost/loss function $C(W)$ at layer L. From this we want to propagate backwards and compute all error and use stochastic gradient descent to update the weights W^l and bias $\mathbf{b}^l$ until an optimal solution is found. As explained, before we apply stochastic gradient descent, we need to compute the gradient of the output at layer L, consider the following equation for the error at the output level.

$$\delta_j^L = f'(z_j^L) \odot \frac{\partial C}{\partial a_j^L} \qquad \boldsymbol{\delta}^L = f'(\mathbf{z}^L) \odot \frac{\partial C}{\partial \mathbf{a}^L} \tag{12}$$

This holds for any general cost function C, and activation function f. The function C can represent a loss function for binary classification or regression problem. We then want to compute the error for each layer $L-1 \rightarrow 2$ as follows

$$\delta_j^l = \sum_k \delta_k^{l+1} w^{l+1} kj f'(z_j^l) \qquad \boldsymbol{\delta}^l = \boldsymbol{\delta}^{l+1} (W^{l+1})^T f'(\mathbf{z}^l) \tag{13}$$

After computing the errors, we update the weight W^l and bias $\mathbf{b}^l$ for each layer $l = L-1 \rightarrow 2$ by using stochastic gradient descent as explained in section (2.4). The algorithm in matrix notation is given as

$$W^{l+1} = W^l - \eta(\mathbf{a}^{l-1})^T \boldsymbol{\delta}^l \tag{14}$$

$$b^{l+1} = b^l - \eta \boldsymbol{\delta}^l \tag{15}$$

2.7.1 Backpropagation for binary classification problem

For a binary classification problem, we are looking at a function cost function which is the same as logistic regression.

$$C(\boldsymbol{\beta}) = -\sum_{i=1}^{n} \left(y_i X_i^* \boldsymbol{\beta} - \log\left[1 + \exp(X_i^* \boldsymbol{\beta})\right] \right)$$

For a neural network this can be changed to

$$C(W) = -\sum_{i=1}^{n} (t_i \log(a_i^L) + (1 - t_i) \log(1 - a_i^L)) \tag{16}$$

Where t_i is the so called target variable, and as before a_i^L is the activation in layer L. We want to look at the following expression

$$\mathbf{z}^l = (W^l)^T \mathbf{a}^{l-1} + \mathbf{b}^l,$$

where $a_j^{l-1} = f(z_j^{l-1})$. In the case of classification, we choose the activation function f to be the sigmoid function (8). From this, the gradient of the cost function differentiated with respect to the activation a_i^L at layer L is

$$\frac{\partial C(W)}{\partial a_i^L} = \frac{a_i^L - t_i}{a_i^L(1 - a_i^L)}$$

We can now use the previous expression to find a new expression for the gradient of the cost function with respect to w_{ij},

$$\frac{\partial C(W)}{\partial w_i j^L} = (a_i^L - t_i)a_j^{L-1} \tag{17}$$

and the derivative with respect to the bias is

$$\frac{\partial C(W)}{\partial b_j^L} = \delta_j^L. \tag{18}$$

The error generated in the L'th layer is given by

$$\delta_j^L = t_j - y_j^L \rightarrow \boldsymbol{\delta^L} = \mathbf{t} - \mathbf{y} \tag{19}$$

For more detailed explaination about back propagation, please see [David E. Rumelhart, 1986]

2.7.2 backpropagation for regression analysis

The difference now is the cost/loss function which we replace with the MSE. Recall that the MSE is defined as

$$MSE = \frac{1}{n}(\mathbf{y} - \mathbf{t})^T(\mathbf{y} - \mathbf{t}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - t_i) \tag{20}$$

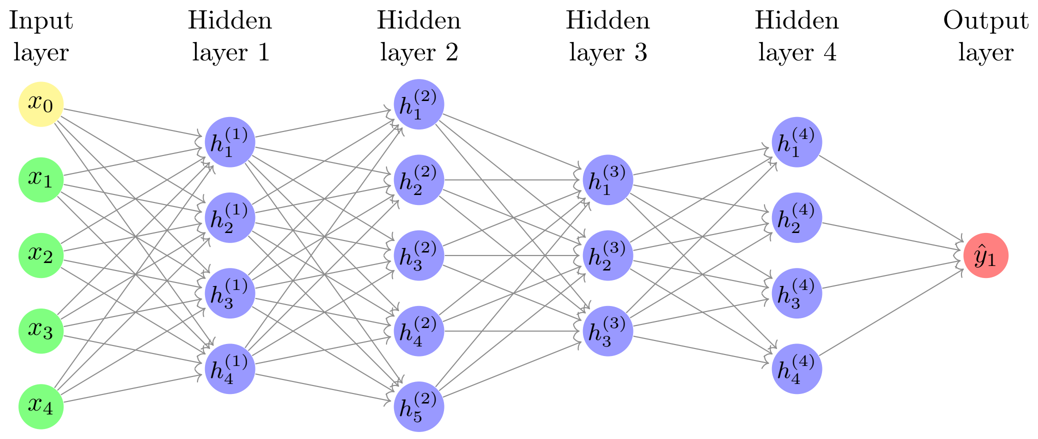
We now want to use this as the cost function, and from this we can find the derivative of this with respect to weights $w_i j^l$ and bias b_j^l, thus

$$\frac{\partial C}{\partial w_{jk}^l} = \delta_j^l a_k^{l-1} \quad \frac{\partial C}{\partial b_j^l} = \delta_j^l$$

Where as previous δ_j^l is expressed as

$$\delta_j^l = \sum_k \delta_j^{l+1} w_{jk}^{l+1} f'(z_j^l)$$

These are then feeded into SGD in order to update the weights $w_i j^l$ and bias b_j^l. Notice that the activation function is now changed since this is a regression case. The most popular activation function are ReLu or the LeakyReLu (there are few others too which are quite popular, but these are the most common ones), as mentioned previously (and how they used in the network). Which activation function are used depends on the problem and how the gradient behaves.



In order to have converging gradient, we must use a activation function which are stable when differentiated and the gradient is larger than zero. Thus, we always want to move into a direction where the cost function/MSE minimizes error. If not we are risking to encounter a problem called a "vanishing gradient". Meaning the network stops learning and we do not get any sensible information. This problem arises when we have dead neurons, i.e when the gradient becomes zero. When multiplying zero we get zero, thus resulting a neuron not getting trained. To avoid this problem, LeakyRelu is quite popular since the derivative is always a constant, thus we will avoid dead neurons. Compared with ReLu, this problem arises quite often since the derivative is either a constant or zero. For more detailed explaination about back propagation, please see [David E. Rumelhart, 1986]

2.8 Franke's function

As mentioned in the introduction, the data set we will use for regression analisys is generated from Franke's function. This function is a two dimensional Gaussian function, with two peaks, defined on the interval $x, y \in [0, 1]$.

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

3 Method and implementation

3.1 The data sets

3.1.1 The Wisconsin breast cancer data set

The Wisconsin breast cancer data set retrieved from `https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(original)`. The data set has dimension 569×30, meaning 569 observations and 30 predictors. For the 569 patients in the data set, 212 have benign (non-cancerous, or not dangerous) tumors and 357 have malignant (cancerous) tumors. Thus 62.7% of the patients in the data set have breast cancer, while the remaining $37,3\%$ do not. This means that this data set is slightly biased towards patients with cancerous tumors, but not to such a degree that it is a cause for worry. The ratio is about $60 - 40$, which is a good ratio to have when training a model. When developing, and later evaluating the performance of a model, it is important that the data used for training the model is not the same as the data used to test and evaluate it. If the same data is used, the model may *overfit*. This means that the model becomes too sensitive to random fluctuations in the data set, which leads to worse performance when completely new data is introduced. To avoid this, we split the data set into two parts; one for training the model, and one for testing. We chose to use 80% of the data for training, and the remaining 20% for testing.

In general, it is always wise to normalize/scale the data before proceeding. This is because the features most often have different ranges, because they were measured with widely different units. The standard method for scaling the data is by removing the mean and scale it to the unit variance. This can be done using sickit-learn StandardScaler class. First we need to fit the StandardScaler instance to the training data. Then, we use the StandardScaler to transform the training data as well as the testing data. This is shown down below

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

We will use this data to train and test our classification models to see whether Neural Network or Logistic Regression is the best and efficient algorithm to use.

3.1.2 The Franke's function data set

The second data set we will use in this article is self made, using Franke's function (see section 2.8). We created 100 x-values, and 100 y-values, both uniformly spaced on the interval $[0, 1]$. This results in $100 \times 100 = 10000$ z-values $z = f(x, y)$, corresponding to each unique pair of x and y. Figure 2 shows a plot of Franke's function. To simulate real life data, we applied a noise term to each $z - value$. The noise terms were drawn from a Gaussian distribution centered at 0, with a scale (standard deviation) of 0.1. As before, we split this data set into 80% training data, and 20%

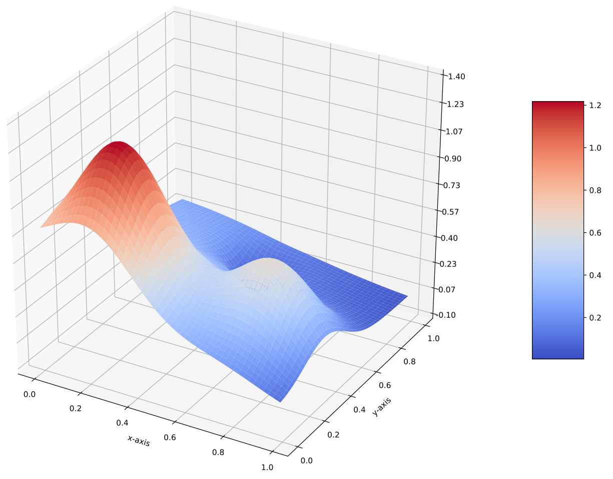


Figure 2: Franke's function

testing data.

3.2 Accuracy, Mean Squared Error, and the R2s-score

We need a way to measure the performance of the trained models. For classification, this means a measure of how accurate the predictions givenby the model are, for the given test data. This is checked by looking at so called accuracy score. Consider the number of samples, $n_{samples}$, the predicted variable $\hat{y}_i$, and the corresponding "true" value y_i. Then the accuracy is defined as

$$accuracy(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n_{samples}} \sum_{i=1}^{n_{samples}} 1(\hat{y}_i = y_i) \tag{21}$$

Where $\hat{y}_i$ is the predicted value and y_i is the corresponding true value.

For regression, the performance is measured by how close the predicted values are to the true values in the test set. We do this by the mean squared error (MSE),

$$MSE(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n_{samples}} \sum_{i=1}^{n_{samples}} (\hat{y}_i - y_i)^2. \tag{22}$$

Additionally, we use the R2-score, which is a measure of the degree to which the variance in the target variables is predictable by the variance in the independent predictor variables. The R2-score is defined as

$$R2(\mathbf{y}, \hat{\mathbf{y}}) = 1 - \frac{\sum_{i=1}^{n_{samples}} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{n_{samples}} (y_i - \bar{y})^2}, \tag{23}$$

where $\bar{y} = \frac{1}{n_{samples}} \sum_{i=1}^{n_{samples}} y_i$

3.3 Python Logistic Regression Class

Using the theory from section (2.1), (2.2) and (2.4) we built a class called LogReg. The program is available on our GitHub page, in the file "LogReg.py". The code snippet below demonstrates how to use this class:

```
logreg = LogReg(X_train, y_train)
logreg.sgd(n_epochs=epochs, n_minibatches=n_minibatches)
pred = logreg.predict(X_test)
```

The LogReg-instance is initiated with the data upon which we want to train the model, separated into the predictors (the x-values), and the response variables (the y-values). In this case, the training data is in the variables *X_ train* and *y_ train.* The model is then trained by calling the sgd-function, which performs stochastic gradient descent with the given number of minibatches, and epochs. The predict-function takes a set of predictors (x-values), and returns the corresponding predictions for the response (y-values).

In our implementation of the stochastic gradient descent, we let the learning rate be a function of two fixed numbers $t_0, t_1 > 0$ and a parameter $t = e \cdot M + i$ where e is the current epoch index, M is the total number of minibatches, and i is the index of the current minibatch s.t. $i = 0, ..., M - 1$. Thus, the learning rate is not constant, and therefore not an input parameter. This method have shown to increase the stability in the system as the gradient converges better. We chose default $t_0 = 5$ and $t_1 = 50$. See *LogReg.py* on our GitHub page for the full implementation.

Additional arguments that can be given to the class instance are *seed*, and *predictor_ names.* The seed parameter sets a seed so that the results obtained can be reproduced. The predictor_names parameter prompts the class to print a table of the estimated predictor coefficients after the model has been trained with stochastic gradient descent. The parameter predictor_names should be a list/array containing the feature names pertaining to the x-values. By default, this parameter is set to None.

3.4 Python Neural Network Class

We have similarly built a class called NeuralNetwork, based on the theory from sections 2.5, 2.6, and 2.7. This class performs classification by default, but can be set to do regression by giving the parameter *problem = 'reg'* when initializing an instance of the class. This is a demonstration of how to use the class to construct a network for classification:

```
network = NeuralNetwork(X_train, y_train, eta=0.1, lmbd=0, epochs=100,
                        batch_size=50, n_layers=3, n_hidden_neurons=[50,40,30],
                        activation='sigmoid', problem='class')
network.train()
pred = network.predict(X_test)
```

The parameters to be set are the number of hidden layers (*n_ layers*), the number of hidden neurons in each layer (*n_ hidden_ neurons*), the activation function, the learning rate (*eta*), the regularization parameter λ (*lmbd*), and lastly the number of epochs and the batch size for the stochastic gradient descent algortihm.

We have implemented four options for the activation function: the sigmoid function, the hyperbolic tangent function (*tanh*), the rectified linear unit function (*ReLU*), and the leaky rectified linear unit function (*leakyReLU*). The sigmoid and tanh functions are logistic, and intended for classification, while the ReLU functions are used for regression. We quickly found out that the sigmoid and leaky ReLU functions gave better results for classification and regression, respectively. For the remainder of this paper, we will therefore use sigmoid for classification, and leakyReLU for regression.

Lastly, we have automated the selection of the number of neurons in the output layer based on the problem type. For classification, it is set to 2 (i.e. binary classification). If a higher number of categories is needed, it can be specified with the parameter *n_ categories* when initializing the network. For regression, it is set to 1.

3.4.1 Classification

The number of hidden layers and neurons are set by the user. The neurons in the hidden layers have sigmoid function as their activation function. The neurons in the output layer have the softmax function as activation. When reaching the final layer L, we then want to calculate the error given by $\boldsymbol{\delta}^L = \mathbf{a}^L - \mathbf{t}$. As before the cost function for this classification problem is simply the logistic regression cost function, but slightly changed. Consider expression (16). From this we can update the weights and bias in the output layer as

$$W^L = (\mathbf{a}^{L-1})^T \boldsymbol{\delta}^L, \quad \mathbf{b}^L = \boldsymbol{\delta}^L \tag{24}$$

We can then proceed by updating the error in the hidden layers, recall from previously in our theory section

$$\boldsymbol{\delta}^l = \boldsymbol{\delta}^L (W^L)^T \odot f'(\mathbf{z}^{L-1})$$

Where $L-1$ indicates the layer before the output layer. We can then use

$$\Delta \mathbf{W}_h = \mathbf{X}^T \boldsymbol{\delta}_h \quad \mathbf{b}_h = \boldsymbol{\delta}_h \tag{25}$$

To update all the weights and bias in the hidden layer. As previously mentioned, for a classification problem the number of neurons in the output layer is 2, and using softmax function as the activation function in the output layer.

3.4.2 Regression

As mentioned before there is a minor difference between regression and classification but most of the implementation is the same. In this case we have used MSE as the loss function and LeakyRelU as the activation function in the hidden layers. This function solves problems where we can encounter vanishing gradient/dead neurons (neurons which is not activated due to zeros in the gradients). The function is mathematically defined as

$$f(x) = \begin{cases} x, & x > 0 \\ 0.01x, & x \leq 0 \end{cases} \rightarrow \nabla f(x) = \begin{cases} 1, & x > 0 \\ 0.01, & x \leq 0 \end{cases} \tag{26}$$

As we can see the gradient of this never becomes zero, thus we will avoid dead neurons in our network. These are used in layer $2, ..., L-1$, and as input and output we have used the simple linear identity function $f(x) = x$ as input and output activation.

4 Results

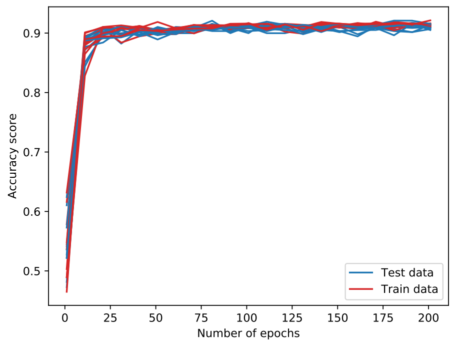
4.1 Classification of the breast cancer data

In the following subsection we are going to present results for the Wisconsin breast cancer data.

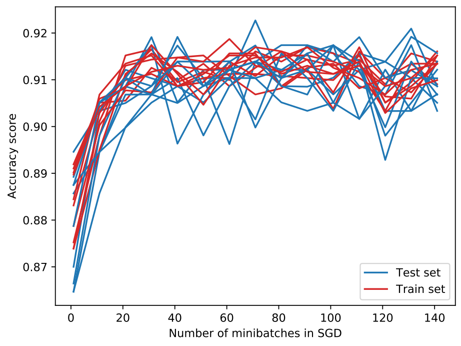
4.1.1 Logistic regression

We want to develop a logistic regression model, where we train the model on the training data. The observations in the test set will be used for testing how well the model is able to predict whether a new patient has breast cancer or not.

There are two parameters that affect how well the logistic regression model is able to learn from the training data: the number of minibatches used in the stochastic gradient descent method, and the number of epochs. Recall that the number of epochs is the number of times the algorithm iterates over the minibatches. Choosing optimal parameters for our logistic regression model can significantly improve the performance. Figure 3 shows how the accuracy score of the logistic regression model behaves for increasing number of epochs and minibatches. We calculated the accuracy for increasing number of epochs and minibatches, respectively. To account for stochastic variability, this was done for 10 iterations, as can be seen in the figure. This gives a more complete picture of how the values plotted behave. In figure 3a, we see that a higher number of epochs gives a higher accuracy score. However, for 20 epochs and more, the increase in accuracy is slow. With limited computing power, the improvement in accuracy may not be worth the increase in computing time after a certain number of epochs. We chose to use 100 epochs. In figure 3b, we similarly see that the increase in accuracy tapers off after a certain number of minibatches. We chose 30 minibatches, as this is the point where the increase in accuracy subsides. With the number of epochs set to 100, and the number of minibatches set to 30, we trained the logistic regression model on the training data. This model was able to predict the outcomes of the patients in the test set with a 0.964912281 accuracy.



(a) Accuracy vs. number of epochs in logistic regression. Here, the number of mini-batches was set to 30.



(b) Accuracy vs. the number of minibatches in logistic regression. Here, the number of epochs was set to 100.

Figure 3

4.1.2 Neural Network for classification

Even though we obtained a 96.5% accuracy with logistic regression, we want to see if we can further improve the results with a classification Neural Network. The neural network has several more parameters that can be optimized to increase the accuracy performance. Most notably is the learning rate and the regularization parameter λ. The other parameters that can be optimized is the number of epochs, as well as the batch size used in the stochastic gradient descent, and the number of hidden layers and hidden neurons in each layer. Note that the batch size is inversely proportional to the number of minibatches. We used batch size as the deciding parameter to reflect the usage of scikit-learn's MLPClassifier method.

Finding the optimal learning rate and regularization parameter λ

We performed a grid search to find which combination of learning rate and lambda gives the best network. Figure 4 shows a heat map where the yellow area represents the best accuracy scores achieved.
This plot has three occurences of the maximum accuracy score 99.1%: at learning rate $\eta = 0.1$ and the three regularization parameter values $\lambda = 1 \cdot 10^{-7}$, $\lambda = 1 \cdot 10^{-11}$, and $\lambda = 1 \cdot 10^{-14}$. This indicates that the optimal learning rate is approximately 0.1, but that the choice of regularization parameter is not as significant as long as it is small. For simplicity, we chose to use $\lambda = 1 \cdot 10^{-7}$

An interesting fact to note is that the Neural Network algorithm for classification with one hidden node and layer will reproduce the exact same result as logistic regression. Having an accuracy-score equal to 0.9649122 for both the Neural Network and Logistic Regression. In theory this make sense, since a Neural Network is just a generalized logistic regression model, but which is highly efficient for non-linear model. Thus, when a classification Neural Network reduced the amount of hidden layers and nodes to 1, we have basically a regular logistic regression.

Deciding the number of epochs, batch size, and number of hidden layers

Figure 5 shows how the accuracy behaves as the number of epochs and batch size increases. In figure 5a, we see that after 100 epochs, the accuracy is virtually unimproved. Thus, we once again chose 100 as an optimal number of epochs. Figure 5b shows that the algorithm becomes unstable for larger batch sizes. However, a very small batch size results in more minibatches, which can increase the computation time. Figure 6 shows that the network performs best when the number

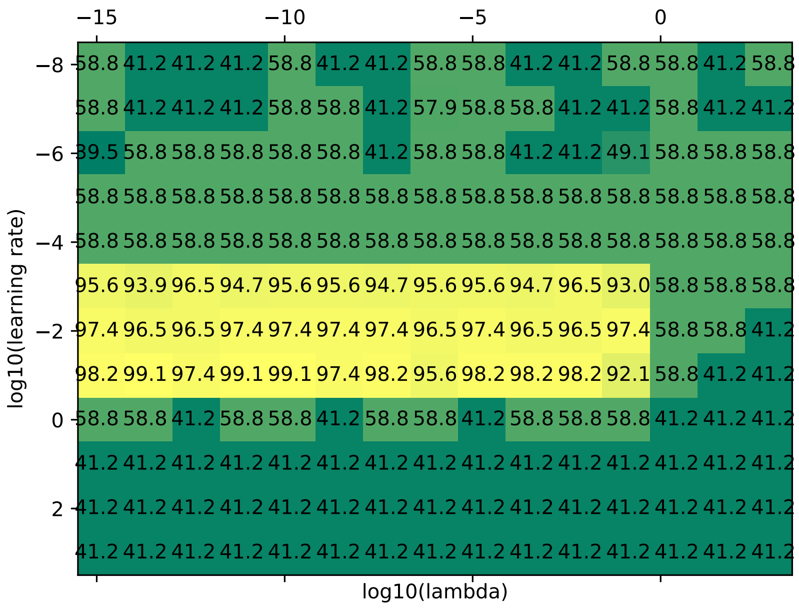
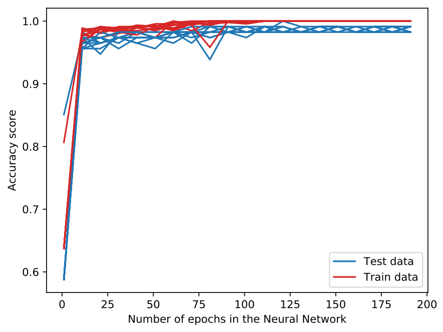
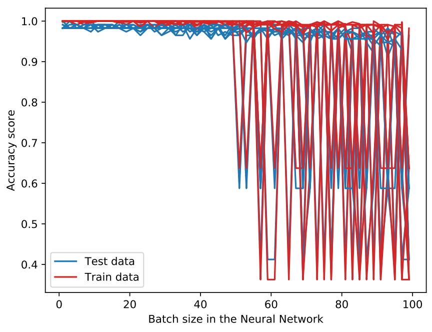


Figure 4: The accuracy score obtained, using various choices of learning rate and λ. The x axis represent log_{10} of the regularization constant λ and the y-axis is the log_{10} of the learning rate η.



(a) Accuracy vs. the number of epochs for neural network.



(b) Accuracy vs. the batch size in neural network.

Figure 5

of hidden layers is between 2 and 5. Based on this discussion, we let the neural network have 3 hidden layers, learning rate 0.1, regularization parameter $\lambda = 1\dot{1}0^{-7}$, batch-size 15, and 150 epochs. The result is an accuracy of 0.99122807, which is much better than the results from logistic regression.

4.2 Regression on Franke's function data set

4.2.1 Neural Network for regression

We trained Neural Networks for regression on the training data set produced by Franke's function. The Neural Network had 5 hidden layers, with 50 neurons each. After some testing, we got good

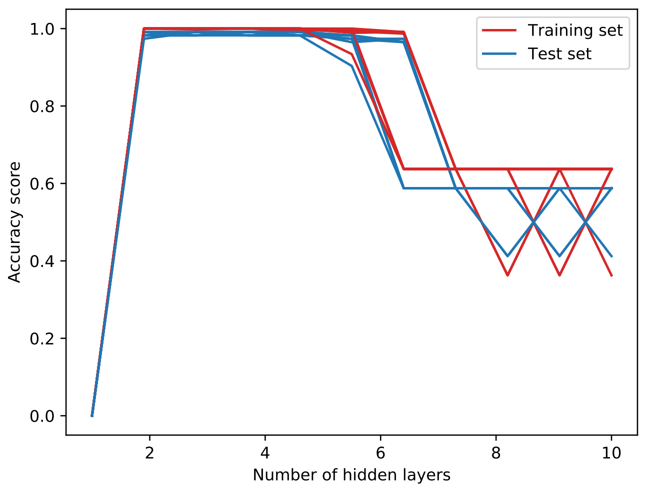


Figure 6: Accuracy vs. the number of hidden layers. The other parameters were kept constant.

results with a learning rate of 0.1, regularization parameter $\lambda = 0$, 100 epochs, and a batch size of 200. For the predictions made on the test set, the mean squared error was 0.0102859071856 and the R2-score was 0.890997369818. Figure 7 shows a plot of the predicted values. How much can this result be improved by tuning the neural network parameters?

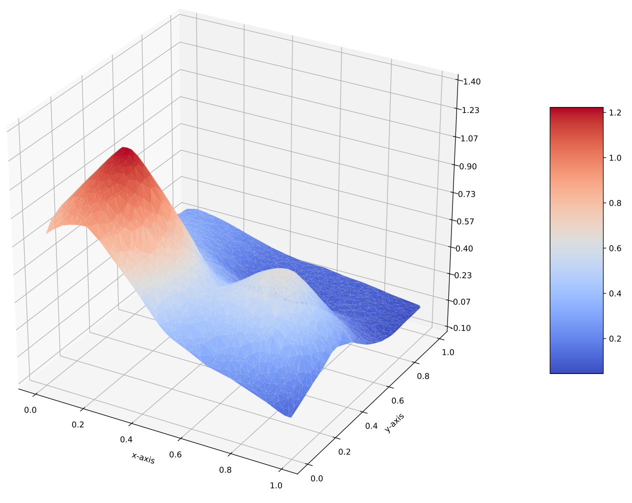
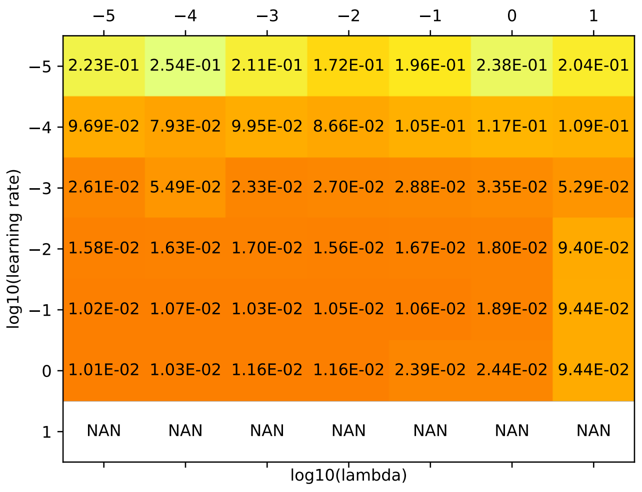


Figure 7: Estimation of Franke's function by Neural Network of 5 hidden layers, with 50 neurons each, with learning rate 0.1, $\lambda = 0$, 100 epochs, and batch size 200.



log10(learning rate) \ log10(lambda)	−5	−4	−3	−2	−1	0	1
−5	2.23E-01	2.54E-01	2.11E-01	1.72E-01	1.96E-01	2.38E-01	2.04E-01
−4	9.69E-02	7.93E-02	9.95E-02	8.66E-02	1.05E-01	1.17E-01	1.09E-01
−3	2.61E-02	5.49E-02	2.33E-02	2.70E-02	2.88E-02	3.35E-02	5.29E-02
−2	1.58E-02	1.63E-02	1.70E-02	1.56E-02	1.67E-02	1.80E-02	9.40E-02
−1	1.02E-02	1.07E-02	1.03E-02	1.05E-02	1.06E-02	1.89E-02	9.44E-02
0	1.01E-02	1.03E-02	1.16E-02	1.16E-02	2.39E-02	2.44E-02	9.44E-02
1	NAN	NAN	NAN	NAN	NAN	NAN	NAN

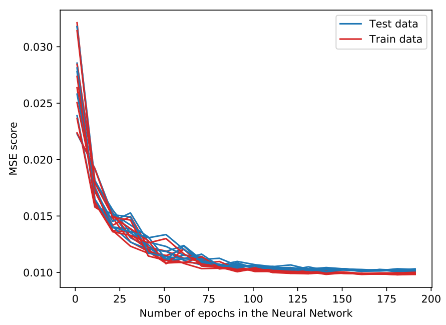
Figure 8: The mean squared error obtained with various choices of learning rate and λ. Darker color indicate a lower MSE.

4.2.2 Finding optimal learning rate and regularization parameter λ

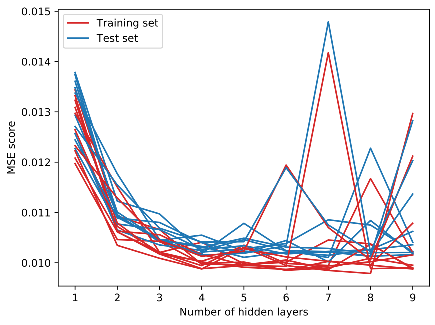
To find the optimal choices for the learning rate and regularization parameter λ, we performed a grid search, and stored the resulting mean squared error for each pair of parameters. Figure 8 shows the results. We see that a higher learning rate yields better mean squared error, however, at a learning rate of 10 the algorithm becomes unstable. The lowest mean squared error obtained in the grid test was 0.0100921308, at $\lambda = 1 \times 10^{-5}$ and learning rate 1. The accompanying R2-score is 0.893050872. We can further optimize our choice of parameters by performing another grid search, this time for learning rate values close to 1 and λ values close to 1×10^{-5}. The results of this grid search gives the even lower mean squared error 0.010016167, for learning rate 0.68129207 and $\lambda = 0.00215443$. The accompanying R2-score is 0.89385588. This is a slight improvement over the results obtained with $\lambda = 1 \times 10^{-5}$ and learning rate 1.

4.2.3 Number of epochs, and number of hidden layers

Figure 9 shows how the mean squared error behaves for varying number of epochs and hidden layers in the network. Again, we see that for a higher number of hidden layers, the algorithm starts to become unstable. Other parameters, such as the regularization parameter λ can help prevent this for deeper networks. In the plot, all other parameters except the number of hidden layers were kept constant.



(a) Mean squared error vs. the number of epochs.



(b) Mean squares error vs. the number of hidden layers.

Figure 9

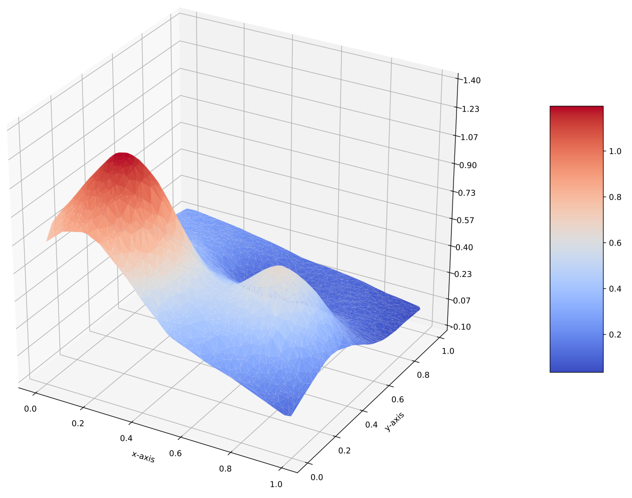


Figure 10: Estimation of Franke's function by Neural Network of 5 hidden layers, with 50 neurons each, with learning rate 0.68129207, $\lambda = 0.00215443$, 100 epochs, and batch size 200

5 Discussion

5.1 Logistic regression vs Neural Network classification

As we can see from the results, the Neural Network for classification performs better than logistic regression on the Wisconsin breast cancer data set. This is no surprise, as the Neural Network can simulate Logistic regression if it has one hidden layer, with only one node, and uses the sigmoid activation function. The optimized Neural Network, which in our case has 3 hidden layers, and 50 nodes, performs better than this 'base case'. Furthermore, by design, logistic regression cannot explain a more complex non-linear relationship with respect to the features. This is something the Neural Network is able to do by use of more layers, and nodes.

With this data set kept in mind we have managed to increase the accuracy by approximately $\approx 3\%$. In other cases where the relationship between the predictors is far more complex, we can expect to increase the accuracy significantly when comparing with ordinary Logistic Regression.

5.1.1 Convergence of accuracy for increasing number of epochs and batch size

When comparing how the accuracy as function number of epoch and and mini batches in logistic regression ,figure 3a and (3b) with neural network (5a) and (5)(Here we used 30 mini batches in botch cases, and for neural network we used 3 hidden layers with 50 neurons each) .From this we can see the convergence is almost alike when looking at the accuracy (neural network is 5% more accurate than logistic regression) as function of epochs, where the optimal epoch start at $5 - 8$ for both cases, which is a bit strange since we expect logistic regression to converge faster than Neural Network, since logistic regression is a special case of Neural Network. Why this happens is unknown. One of the idea could be Wisconsin breast cancer data set has really high correlation between the features and target variables.

However when analyzing the accuracy as function of mini batches the Neural Network outperforms logistic regression starting at best possible accuracy 99.7%, and then decreasing with increasing of batch size. The strange thing behaviour seen after 50 batch size in figure(5b), is that the accuracy oscillate very hard, meaning with increasing number of batches for 100 epoch the system get very unstable. One of the idea is, increasing number of epoch will lead to a better stability in the system. When looking at figure (3b) representing logistic regression, the accuracy gets better with increasing batch size, however it will also oscillate hard but not varying as much as neural network since the accuracy is between $0.90 - 0.915$. In this numerical study we have not taken benchmarks of the time spent when producing this results. However if we only consider the accuracy and the the time spent, we can with high accuracy say that neural network is a better supervised learning method to use than logistic regression.

5.2 Neural network and other linear regression methods

5.2.1 Compare results with other regression methods

To find out which regression method is better suited for the Franke's function data set, we compare the results of our optimal neural network with the results of linear, ridge, and lasso regression. We performed grid searches to find the optimal parameters for each method. The optimal degree for linear regression was $d = 20$, while ridge regression had optimal degree $d = 28$ and $\lambda = 3.162278 \times 10^{-10}$. Lastly, the optimal parameters for lasso regression were degree $d = 29$ and $\lambda = 1 \times 10^{-4}$.

- Linear regression with 5-fold cross validation using degree 20, gave a mean squared error of 0.009993417, and an R2-score of 0.89137248398.
- Ridge regression with 5-fold cross validation, using degree 28 and $\lambda = 3.162278 \times 10^{-10}$ gave mean squared error 0.0099867698 and R2-score 0.8916470204.
- Lasso regression with 5-fold cross validation, using degree 29 and $\lambda = 1 \times 10^{-4}$ gave mean squared error 0.0156377246, and R2-score 0.8303203787.

Notice that these result are retrieved from a previous project, please see [Salihi Aram, 2019] for more detail. With this in consideration and comparing with out neural network for regression analysis, the results of the grid search was: lower mean squared error 0.010016167, for learning rate 0.68129207 and $\lambda = 0.00215443$. The accompanying R2-score is 0.89385588. From this we can see that Ridge regression is slightly better than neural network when it comes to MSE score, but neural network is better when looking at R2 score. From this we must choose the algorithm with is the most efficient to use, thus the algorithm which uses least time to run. However, as previously mentioned we did not do any time benchmarks to which algorithm performs fastest with these result. This part is mentioned in future work down below.

6 Conclusion

6.1 Conclusion on the Wisconsin breast cancer data

The Neural Network algorithm clearly outperforms Logistic regression when applied to the Wisconsin breast cancer data set. The highest score we managed to obtain with our logistic regression was 96.5%, while our neural network implementation obtained 99.1% accuracy with potential for further improvement. Even though we did not benchmark the time when logistic regression and neural network is running, we can see from (5b) that Neural network scores way more better for small batch, with a accuracy score beginning at 99.1% and then decreasing. As mentioned previously, this could be the fact that increasing batch size requires more epochs for a neural network. Ignoring the accuracy as function of the number of epochs, we can conclude that a classification neural network is better than logistic regression, and should be used in these kinds of classification problems, if prediction accuracy is more prioritized than time of computation.

6.2 Conclusion on the Franke's function data set

The best algorithm for estimating Franke's function is ridge regression. This conclusion supports the results of our previous project (please see [Salihi Aram, 2019]), in which we studied linear, ridge, and lasso regression on a smaller Franke's function data set in detail. Linear regression performed almost as good as ridge regression. The difference in MSE is only about 7×10^{-6}. This is not surprising, as the optimal ridge MSE was obtained with $\lambda = 3.162278 \times 10^{-10}$, which is rather small, and ridge regression will behave like linear regression at $\lambda = 0$. However, allowing a small ridge parameter gave us slightly better results. Neural Network regression yielded an MSE about 3×10^{-5} larger than ridge regression.

The R2-score reflects the proportion of the variance in the response variable z that is predictable from the independent variables x and y. The fact that all the regression methods obtained an R2-score just under 0.9 is no coincidence. These scores coincide with the fact that we applied a noise term of factor 0.1 to the $z = f(x, y)$ values obtained by Franke's function. As this noise was randomly generated, the independent variables can only explain ca. 90% of the variance in the z-values.

7 Future work

The results of our neural network algorithm can potentially be improved by further fine-tuning of the parameters. We analyzed how each parameter affects the performance of the network while holding (most of, or all) other parameters constant. This gave some insight into how the network behaves. However, we have not investigated how each parameter affects the performance of the network when the other parameters are allowed to vary. Ideally, we want to find the best combination of parameters by testing all viable combinations - but this is very computationally heavy. Further we also want to take the time for each algorithms to see which one performs fastest with a good MSE/accuracy/R2-score.

Lastly, we could improve the Neural Network by changing the algorithm itself. The stochastic gradient descent method is often not the best or the most stable optimization method. Scikit-learn's neural network classes use the "Adam" algorithm, which more often than not performs better than stochastic gradient descent. Other optimization methods to consider are the Quasi-Newton methods.

8 Appendix

Coefficient estimations by logistic regression on the Wisconsin breast cancer data set

```
+---------------------------+-------------------+
|         Predictor         |    Coefficient    |
+---------------------------+-------------------+
|         intercept         |  0.0926848062436  |
|        mean radius        |   0.728272756616  |
|        mean texture       |  -0.607104829626  |
|       mean perimeter      |  -0.616036115015  |
|         mean area         |  -0.907683427173  |
|      mean smoothness      |  -0.288481872898  |
|      mean compactness     |   0.229736548197  |
|       mean concavity      |   -1.66770777314  |
|    mean concave points    |  -0.984408931493  |
|       mean symmetry       |   -1.08648897857  |
|  mean fractal dimension   |   0.503309402957  |
|        radius error       |  -0.958389022445  |
|       texture error       | 0.00858420605307  |
|      perimeter error      |   -1.28172728768  |
|         area error        |   -1.54595086368  |
|      smoothness error     |   0.14707401641   |
|     compactness error     |   1.06136112972   |
|      concavity error      |   0.672142283693  |
|    concave points error   |  -0.148537391304  |
|       symmetry error      |   0.414653314333  |
| fractal dimension error   |   1.08604033957   |
|        worst radius       |  -0.368117094179  |
|       worst texture       |   -1.39446655337  |
|      worst perimeter      |   -2.16328316367  |
|         worst area        |   -1.4802112775   |
|      worst smoothness     |  -0.414292133097  |
|     worst compactness     |  -0.712391918517  |
|      worst concavity      |  -0.640121631576  |
|    worst concave points   |  -0.605205792298  |
|       worst symmetry      |  -0.279272581291  |
|  worst fractal dimension  |  -0.710389646984  |
+---------------------------+-------------------+
```

References

[David E. Rumelhart, 1986] David E. Rumelhart, G. E. H. . R. J. W. (1986). *Learning representations by back-propagating errors*, volume 1.

[Hastie et al., 2001] Hastie, T., Tibshirani, R., and Friedman, J. (2001). *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA.

[Salihi Aram, 2019] Salihi Aram, T. M. . F. A. (2019). *Regression analysis on Norwegian terrain map with Franke's function as test function*, volume 1.