CLASSIFICATION & REGRESSION

From Regression to Neural Networks

FYS-STK4155

November 10, 2019

Teah Kaasa McLean
Marius Helvig Havgar
Åsmund Danielsen Kvitvang

Abstract

In this paper we compare and contrast the basic machine learning methods; linear regression, logistic regression and the multilayer perceptron model (MLP). We look at two data sets, the Franke function on x and y in $[0,1] \times [0,1]$ with and without added noise and the Wisconsin breast cancer data. We have seen in the Franke function case that a simple regression model more often than not gives better results, although the best MLP model had a slightly better MSE, 0.0238 vs 0.0416. In the cancer case, an MLP model with ReLu activation and 200 neurons in one hidden layer seemed to perform best, with an almost perfect accuracy score of 98.8%, compared to an accuracy score of 97.1% with logistic regression.

Contents

1 Introduction 1

2 Theory 1

2.1 Classification and Regression . . . 1

2.2 Accuracy Score . . . 3

2.3 Logistic Regression . . . 3

2.4 Gradient Descent and Stochastic Gradient Descent . . . 5

2.5 Multilayer Perceptron Network . . . 6

2.5.1 Terminology and Definitions . . . 6

2.5.2 Universal Approximation Theorem . . . 7

2.5.3 The Backpropagation Algorithm . . . 8

2.5.4 Activation Functions . . . 10

3 Method 11

3.1 Logistic Regression . . . 12

3.2 Multilayer Perceptron Network . . . 13

3.2.1 Classification . . . 13

3.2.2 Regression with MLP . . . 14

4 Results 14

4.1 Logistic Regression . . . 14

4.2 Classification with an MLP Model . . . 16

4.3 Regression . . . 20

5 Discussion 27

6 Conclusion 28

1 Introduction

Machine learning (ML) has been a buzzword the last few years. Machine learning techniques impact almost all disciplines and there are countless applications, like for example virtual personal assistants such as Apple's Siri or traffic predictions while commuting. In order to get a better understanding of the more complicated methods that lie beneath these AI systems, we must first understand the simplest machine learning methods. The focus in this paper will therefore be to study these methods; linear regression, logistic regression and the multilayer perceptron model (MLP), a type of neural network (NN), by examining their performance on both *regression* and *classification* problems. For this we will use two different datasets; the Franke function on some **x** and **y** with and without added noise, a typical regression problem, and the Wisconsin breast cancer data from scikit-learn [Ped+11], a typical classification problem.

The simplest machine learning methods, the linear regression methods, take in some data and return a continuous output, with a value from the real number line. Logistic regression returns a value between 0 and 1. The two different types of regression will therefore have different uses. If you wanted to predict which value **z** the Franke function would give based on some input **x** and **y**, one would use linear regression. Whereas if you want to classify a tumor as benign or malignant, you would use logistic regression to find the probability that it is one or the other. The beauty of neural networks is that they can be used for both regression and classification problems, and it is therefore of interest to compare the neural networks with linear regression on the Franke function data and with logistic regression on the breast cancer data. For the linear regression we will use the Ordinary Least Squares (OLS), Ridge and Lasso regression methods.

We will start by introducing the machine learning methods and algorithms and give a brief description of the implementation. Further, we will look at the accuracy scores and mean squared errors (MSE) for the various methods. Finally, we will discuss the differences between the methods and compare them to determine which model best fits the data.

2 Theory

2.1 Classification and Regression

Regression analysis is used for estimation of the relationship between a dependent variable and one or more independent variables, usually called *outcome* and *predictors*, respectively. In this paper the focus is on regression using an MLP model, but we will compare these results with results obtained from the linear regression models *Ordinary Least Squares*, *Ridge regression* and *Lasso regression*. Results from these regression models are not included but we refer to an earlier written article for extensive explanation of these methods [HK19].

A related problem to regression is the *classification* problem. In classification, we wish to identify which category or class(es) some input data belongs to. An example could be to make inferences on whether a person has a certain disease or not, given a set of symptoms or health data. The classes might be, but are not limited to, binary classes. In the binary case, we would have a positive class (1) and a negative class (0).

Binary classification can be viewed as identifying a boundary, known as the *decision boundary*, that separates data of the different classes from each other. If there exists a (hyper)plane that perfectly separates data of the two classes, the data is *linearly separable*. Linearly separable data would be the simplest classification problem, and a simple linear model would be sufficient to solve the problem. However, as the decision boundary becomes more complicated, the need for a more complicated model arises.

As a consequence of the universal approximation theorem [Cyb89], we know that classification problems can indeed be solved by MLPs. In the binary case, the output $\mathbf{F}(\mathbf{x})$ from the MLP is used to decide what class $\mathbf{x}$ is predicted to belong to. Typically we can view the output of the MLP as a probability of the input belonging to a certain class. This is considered “soft classification.” Alternatively, we do inference based on the output value by rounding the number by some rule, e.g. a cut-off threshold, and letting the final rounded number represent the predicted class of the input. This is considered “hard classification”.

It is often the case that finding a model, e.g. an MLP that perfectly separates all data into correct classes is not ideal. As is commonly the case in machine learning there is a fine balance between generalization and overfitting. We want to find a model that with a high accuracy will classify unseen data correctly, but at the same time avoid estimating an overly complicated decision boundary. If we trained a suitable MLP for an arbitrary number of iterations on an entire dataset, we could expect the model to correctly learn the the class of each instance of our dataset. The model might learn the specific features of our data perfectly, but it would probably not be learned to generalize well. Hence, we could typically not expect the model to perform very well on new data.

To monitor the problem of overfitting, we need a way to evaluate how well the model performs on data that has not been used in training. There are several viable options to estimate this. A simple approach is to set aside a testing set, which would contain for example 30% of the original dataset. This set would be used specifically to test how accurately the model classifies data that has not been used in training. The remaining 70% of the data would be used to train the model. One would then evaluate the model performance on the test set, and observe how the evaluation changes as the training progresses, and try to identify the point where more training does not systematically lead to increases in model performance. It is important to be aware that the performance on the test set is a stochastic quantity. Hence it might not be the case that the best test accuracy reflects the true model performance on unseen data after that level of training. It is common that a complicated model will have a tendency to overfit as training progresses. In light of the stochasticity of the test accuracy, one would prefer to stop training as early as possible, when the test accuracy begins to converge.

2.2 Accuracy Score

The accuracy score is a metric for measuring the performance of a classifier. The accuracy is the number of correctly predicted 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, returning 1 if $t_i = y_i$ and 0 otherwise.

A perfect classifier will have an accuracy score of 1.

2.3 Logistic Regression

The theory in this section is based on the lecture notes on logistic regression by M. Hjorth-Jensen [Hjo19a].

Logistic regression is a statistical model that is used when the output data is categorical, i.e. when we have a classification problem.

The simplest case of classification is binary classification, when the observation belongs to one of only two classes. Take the example from the previous section, predicting whether a person has a certain disease or not, given a set of symptoms. In this case, we want to classify whether an observation x is in the class “has disease” or “is healthy”. Then our output value y would be 1 if x is sick and 0 if healthy.

However, in most real-world applications, it is more useful to perform “soft classification”, i.e. that our model gives us the *probability* that the observation is in a certain class, rather than just the hard values 1 or 0. This will require continuous output. The linear regression model gives us continuous output, but the range is the real number line. In order to map our input to values between 0 and 1, we can use a *sigmoid* function, the so-called *logistic function*, defined as

$$S(z) = \frac{1}{1+e^{-z}} = \frac{e^z}{e^z+1}.$$

Let $y_i = \beta_0 + \beta_1 x_{i,1} + \ldots + \beta_n x_{i,n}$, then we get the probabilities

$$P(y_i = 1 | x_i, \boldsymbol{\beta}) = \frac{e^{\beta_0+\beta_1 x_{i,1}+\ldots+\beta_n x_{i,n}}}{e^{\beta_0+\beta_1 x_{i,1}+\ldots+\beta_n x_{i,n}}+1}, \text{ and}$$
$$P(y_i = 0 | x_i, \boldsymbol{\beta}) = 1 - P(y_i = 1 | x_i, \boldsymbol{\beta}).$$

This can be expanded to more classes, but for the sake of simplicity we stick with the binary case.

To ensure that the regression gives us the most accurate prediction, we need a cost-function. In order to obtain this cost-function, we compute the total likelihood of all possible outcomes from a dataset $\mathscr{D} = \{(X_{i,*}, y_i)\}$ with $y_i \in \{0,1\}$. We use the so-called Maximum Likelihood Estimation (MLE) principle. The aim then is to maximize the probability of seeing the observed data. We can approximate the probability by taking the product of all the individual probabilities of a specific outcome y_i,

$$P(\mathscr{D}|\boldsymbol{\beta}) = \prod_{i=1}^{n} [P(y_i = 1 | X_{i,*}, \boldsymbol{\beta})]^{y_i} [1 - P(y_i = 1 | X_{i,*}, \boldsymbol{\beta})]^{1-y_i}.$$

We then obtain the cost function

$$C(\boldsymbol{\beta}) = \sum_{i=1}^{n} (y_i \log p(y_i = 1|X_{i,*}, \boldsymbol{\beta}) + (1-y_i)\log[1 - p(y_i = 1|X_{i,*}, \boldsymbol{\beta})]),$$

from the MLE, with the gradient

$$\frac{\partial C(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -X^T(\mathbf{y} - \mathbf{p}) \tag{1}$$

where $\mathbf{y}$ is a vector containing the target values and $\mathbf{p}$ is a vector containing the probabilities $p(y_i = 1|x_i, \boldsymbol{\beta})$.

Our goal is to find the parameters $\boldsymbol{\beta}$ that minimize our cost-function, using some kind of optimizer. The most popular choice is the gradient descent method, which is described in more detail in section 2.4. The gradient in equation 1 is used to update the parameters for each iteration in the gradient descent method [Hjo19a].

2.4 Gradient Descent and Stochastic Gradient Descent

The theory in this section is based on the lecture notes on gradient methods by M. Hjorth-Jensen [Hjo19b].

Gradient descent (GD), or *steepest descent*, is an iterative optimization method for finding the minimum of a function. To find a minimum, which could be a local or global minimum, one starts with an initial guess for the parameters of the function. Then, for a number of iterations, one takes steps in the direction of the negative of the gradient of the function at the current point and updates some parameters of the function. The function must be differentiable with respect to these parameters. The steps we take are of a chosen length, called the *learning rate*, often denoted by η. The number of iterations and the learning rate are hyperparameters that can be tuned.

The algorithm for minimizing the cost-function $C(\boldsymbol{\beta})$ with learning rate η is summarized in algorithm 1.

Algorithm 1 Gradient Descent

```
Initialization of parameter $\boldsymbol{\beta}_0$
**for** each iteration $i$ **do**
    Compute
        $$\boldsymbol{\beta}_i = \boldsymbol{\beta}_{i-1} - \eta \frac{\partial C(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}}.$$
```

For small enough learning rate η, we will ensure that we get smaller and smaller function values, and thus approach a minimum. As mentioned, this could be a local minimum. Since GD is deterministic, we can get stuck in a local minimum, unless we have a good initial guess for the parameters. If the function is convex, then all minima are global minima and the GD algorithm is sufficient. However, we do not always have convex functions and will sometimes require a more robust optimizer. To combat the problem of getting stuck in local minima, we introduce some randomness and get the Stochastic Gradient Descent (SGD) method.

The idea for SGD comes from the observation that the cost function $C(\boldsymbol{\beta})$ can almost always be written as a sum of n datapoints,

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

Then the gradient can be written as a sum of the n gradients corresponding to to n cost-functions,

$$\frac{\partial C}{\partial \boldsymbol{\beta}} = \sum_{i=1}^{n} \frac{\partial c_i}{\partial \boldsymbol{\beta}}(\mathbf{x}_i, \boldsymbol{\beta}) \tag{2}$$

We want to approximate this gradient sum in equation 2 by a sum over a subset of the datapoints. Therefore we split the data into subsets called *minibatches*. If there are n datapoints and the batch size is M, then there are n/M minibatches, denoted by B_k where $k = 1, \dots, n/M$. We pick a minibatch at random for each iteration, and approximate the gradient by replacing the sum over all the datapoints by a sum over the datapoints in the chosen minibatch [Hjo19b].

The algorithm for SGD is described in algoritm 2.

Algorithm 2 Stochastic Gradient Descent

```
Initialization of parameter $\boldsymbol{\beta}_0$.
**for** each epoch $j$ **do**
    **for** $k = 1, \dots, n/M$ **do**
        Pick random minibatch $B_k$.
        Compute
            $$\boldsymbol{\beta}_j = \boldsymbol{\beta}_{j-1} - \eta \sum_{i \in B_k} \frac{\partial c_i}{\partial \boldsymbol{\beta}}(\mathbf{x_i}, \boldsymbol{\beta}).$$
```

2.5 Multilayer Perceptron Network

2.5.1 Terminology and Definitions

In the following we will establish terminology to describe the *multilayer perceptron network* (MLP), which is a type of artificial neural network (ANN).

An MLP is a directed graph structure that serves as a computational system, and can be utilized as a function approximator, see Section 2.5.2. Our goal is to use an MLP to approximate a function $f : \mathbb{R}^m \to \mathbb{R}^n$ in order to perform classification or regression. As implemented in this paper, the MLP consists of a set of neurons organized in layers. The number of neurons in the first layer and last layer equal m and n respectively, corresponding to the number of features in the input and output data. Between the input and output layers are a number of hidden layers with a number of neurons in each layer. These numbers will be treated as hyperparameters in the MLP. We will take an empirical approach in the tuning of these, but take for granted that the MLP has at least one hidden layer, since this is a sufficient criterion for the MLP to approximate any non-linear functions [Cyb89]. Each neuron not in the output layer is connected to each neuron in the next by a weighted connection.

We will use the following notation. W^l will denote the set of weights that connect the neurons in layer $l-1$ to the neurons in layer l. If there are n neurons in layer $l-1$ and m neurons in layer l, then

$$W^l = \begin{pmatrix} w^l_{11} & w^l_{12} & \cdots & w^l_{1n} \\ \vdots & \ddots & & \\ w^l_{m1} & w^l_{m2} & \cdots & w^l_{m,n} \end{pmatrix},$$

where w_{jk} is the weight connecting neuron k to neuron l. The collection of all weight matrices in the MLP will be denoted $\mathbf{W}$. The total signal going out of the neurons in a layer l is denoted $\mathbf{a}^l$, and is taken to be an *activation function* $\varphi(\cdot)$ of the weighted sum of the inputs to the neurons. The total signal from layer $\mathbf{a}^l$ into neuron j in layer $l+1$ is given by

$$z^l_j := W^l_{j,*}\mathbf{a}^{l-1} + b^l_j. \tag{3}$$

It follows that $a^l = \varphi\left(z^l\right)$. The quantity b^l_j acts as a bias, which is necessary to enable the MLP to map $\mathbf{0}$ onto a non-zero output. We can hence take an input $\mathbf{x} \in \mathbb{R}^m$, pass it through each layer of the MLP, and get an output $\mathbf{a}^L \in \mathbb{R}^n$ in the output layer, which we will think of as the prediction. L denotes the output layer. We will refer to this as a *forward pass* of an input in the MLP, see algorithm 3 for full algorithm. We will denote this $F(\mathbf{x}) = \mathbf{y}$.

Algorithm 3 Feed-forward

```
Initialize the input layer.
for $i = 1, \dots, L-1$ do
    Calculate the output in each hidden layer.
Calculate the output in the output layer.
```

2.5.2 Universal Approximation Theorem

The universal approximation theorem [Cyb89] shows that an MLP with a finte number of neurons and a bounded, non-linear activation function can approximate continuous functions on compact subsets of $\mathbb{R}^n$. This implies that for a given dense subset of $\mathbb{R}^n$ and continuous target function we aim to approximate, there exists a single layer MLP with some set of nodes, weights and activation function that approximates the target function. Even though this results speaks to the flexibility and usefulness of MLPs, finding a suitable network for a given problem is highly non-trivial, and introduces an optimization problem. See section 2.5.3 for a more extensive discussion on this subject.

2.5.3 The Backpropagation Algorithm

The backpropagation algorithm (BPA) is an optimization algorithm for neural networks. The goal is to use pairs $(\mathbf{x}, \mathbf{t})$ of inputs and targets to adjust the weights in the MLP in order to improve it's accuracy. The full details of the BPA can be found in [Mar09, p.105]. As the name suggests, the idea of the algorithm is to evaluate the error between the output and target values. The error is then propagated backwards through the weights to assign errors to the hidden neurons in each layer. This requires a repeated application of the chain rule to calculate the error terms for each neuron. There are several viable functions that can be used to calculate the error in the output layer. The simplest choice of cost function is the sum of square differences,

$$C(\mathbf{y}, \mathbf{t}; \mathbf{W}) = \frac{1}{2}\sum_{k=1}^{N}\left(\mathbf{y}_k - \mathbf{t}_k\right)^2, \tag{4}$$

where $\mathbf{y} = F(\mathbf{x})$ for a corresponding $\mathbf{x}$. The cost function increases as the difference between $\mathbf{y}$ and $\mathbf{t}$ increases, and $\mathbf{y}$ is implicitly a function of $\mathbf{W}$. Finding a $\mathbf{W}'$ that minimizes C corresponds to finding an MLP that maps $\mathbf{x}$ onto a point as near as possible to $\mathbf{t}$. The gradient of C with respect to the weights of layer l is given by

$$\frac{\partial C}{\partial W^l} = \frac{\partial C}{\partial \mathbf{a}^l}\frac{\partial \mathbf{a}^l}{\partial \mathbf{z}^l}\frac{\partial \mathbf{z}^l}{\partial W^l}. \tag{5}$$

In particular, the gradient of the cost function with respect to the weights in the last layer is given by

$$\begin{aligned}\frac{\partial C}{\partial W^L} &= \frac{\partial C}{\partial \mathbf{a}^L}\frac{\partial \mathbf{a}^L}{\partial \mathbf{z}^L}\frac{\partial \mathbf{z}^L}{\partial W^L}\\ &= \frac{\partial C}{\partial \mathbf{y}}\frac{\partial \mathbf{y}}{\partial \mathbf{z}^L}\mathbf{a}^{L-1}\end{aligned}$$

The gradient of the cost function with respect to the bias in layer l is given by

$$\frac{\partial C}{\partial \mathbf{b}^l} = \frac{\partial C}{\partial \mathbf{a}^l}\frac{\partial \mathbf{a}^l}{\partial \mathbf{z}^l}\frac{\partial \mathbf{z}^l}{\partial \mathbf{b}^l} \overset{(*)}{=} \frac{\partial C}{\partial \mathbf{a}^l}\frac{\partial \mathbf{a}^l}{\partial \mathbf{z}^l} = \frac{\partial C}{\partial \mathbf{z}^l}, \tag{6}$$

where eq. (3) is applied in $(*)$. We now define the error term

$$\delta^l := \frac{\partial C}{\partial \mathbf{z}^L} = \frac{\partial C}{\partial \mathbf{a}^l}\frac{\partial \mathbf{a}^l}{\partial \mathbf{z}^L} = \frac{\partial C}{\partial \mathbf{a}^l} \odot \varphi'(\mathbf{z}^l), \tag{7}$$

where $\odot$ denotes the Hadamard operator. Next, $\frac{\partial C}{\partial \mathbf{a}^l}$ may be written as

$$\frac{\partial C}{\partial \mathbf{a}^l} = \frac{\partial C}{\partial \mathbf{z}^{l+1}}\frac{\partial \mathbf{z}^{l+1}}{\partial \mathbf{a}^l} = \left(W^{l+1}\right)^T \delta^{l+1}.$$

Finally, combining eqs. (6) and (7) yields

$$\delta^l = \frac{\partial C}{\partial \mathbf{a}^l} \odot \varphi'(\mathbf{z}^l) = \left(W^{l+1}\right)^T \delta^{l+1} \odot \varphi'(\mathbf{z}^l). \tag{8}$$

Substituting eq. (8) into eqs. (5) and (6) gives gives

$$\frac{\partial C}{\partial W^l} = \delta^l \frac{\partial \mathbf{z}^l}{\partial W^l} \quad \text{and} \quad \frac{\partial C}{\partial \mathbf{b}^l} = \delta^l. \tag{9}$$

From equation eq. (9) we now have a method to calculate the gradient of the cost function with respect to the bias and weights in each of the hidden layers in the network, and suggests a way to adjust these to minimize the cost function. See algorithm 4 for pseudo code on the backpropagation algorithm.

Algorithm 4 Backpropagation

```
Initialization of weights and biases.
for $i = 1, \dots, N$ do
    Run feed-forward algorithm on input.
    Compute the error term $\delta^L$.
    for $l = L-1, \dots, 1$ do
        Compute $\delta^l$.
    Update weights and biases by computing

        $W^l_{next} \leftarrow W^l_{prev} - \eta\delta^l \frac{\partial \mathbf{z}^l}{\partial W^l}$
        $\mathbf{b}^l_{next} \leftarrow \mathbf{b}^l_{prev} - \eta\delta^l,$

    where $\eta$ is the learning rate.
```

It is in general not possible to find an analytical expression that minimizes C and it is usually not convex. As a consequence of non-convexity, any local minima of the cost function might not be a global one. In fact, for a complex MLP model the search space can be very complex as well, and there might be a multitude of local minimas. Therefore it might not be the case that strictly following the gradient along the weight where the cost function decreases the most will result in the best model, and we will in practice often need to use a stochastic gradient method, e.g. the SGD, to improve the chances of training a good model.

2.5.4 Activation Functions

In the following we describe several commonly used *activation functions*. An activation function is used to modify the signal that a neuron fires. It is common to use activation functions that are bounded, i.e. there is a lower and an upper bound on the values the function can take. If the activation function between all layers in the model is linear, the MLP is in fact equivalent to a model where the output is a linear combination of the input, i.e equivalent to a simple linear model. It is therefore common to use a non-linear activation function between at least one of the layers in the network. We will use φ as shorthand notation for the activation function. We will limit our discussion to the following activation functions:

The *logistic* activation function has range $(0,1)$, and it is differential everywhere. It is given by $\varphi(x) = \frac{1}{1+e^{-x}}$, and it has a derivative $\varphi'(x) = \varphi(x)(1-\varphi(x))$.

ELU is an abbreviation for exponential linear units, and is given by

$$\varphi(x) = \begin{cases} \alpha(e^x - 1), & 0 \leq x \\ x, & x > 0. \end{cases}$$

It has derivative

$$\varphi'(x) = \begin{cases} \alpha e^x, & 0 \leq x \\ 1, & x > 0. \end{cases}$$

In this paper we set $\alpha = 1$, which means that φ is differential everywhere. This φ has range $(-\alpha, \infty)$.

Leaky ReLU is an abbreviation for Leaky rectified linear unit. It is defined as

$$\varphi(x) = \begin{cases} 0.01x, & 0 < x \\ x, & x \geq 0. \end{cases}$$

It has derivative

$$\varphi'(x) = \begin{cases} 0.01x, & 0 < x \\ 1, & x > 0. \end{cases}$$

This φ has range $\mathbb{R}$. Since it is not differentiable at the point $x = 0$, we will for implementation purposes define its derivative there to be 1.

ReLu is an abbreviation for rectified linear units, and is defined as follows.

$$\varphi(x) = \begin{cases} 0, & 0 \leq x \\ x, & x > 0. \end{cases}$$

It has derivative

$$\varphi'(x) = \begin{cases} 0. & 0 < x \\ 1, & x > 0. \end{cases}$$

This φ has range $[0, \infty)$. Since it is not differentiable at $x = 0$, we define its derivative there to be 1.

tanh is the hyperbolic tangent function. It has range $(-1, 1)$, and has derivative $\varphi'(x) = 1 - \varphi(x)^2$

A nice property of the activation functions ELU and leaky ReLu is that their gradients do not go to 0 as the input becomes large or small. This property can sometimes be very important for the ability to effectively apply the backpropagation algorithm.

3 Method

Dataset

The datasets studied in this paper is the Wisconsin breast cancer data, that can be imported from scikit-learn, and the Franke function on input $\mathbf{x}, \mathbf{y} \in [0, 1]$.

The Franke function is defined as

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

The testing has been performed on three different input pairs $(\mathbf{x}, \mathbf{y})$, with different sizes. We have tested with 20, 50 and 100 points in $\mathbf{x}$ and $\mathbf{y}$.

The breast cancer data was created by Dr. William H. Wolberg, physician at the University Of Wisconsin Hospital at Madison, Wisconsin, USA. It describes the nature of tumors in breast masses. There are ten features, such as the radius and texture of the tumor, and the mean value, extreme value and standard error of each feature have been calculated, resulting in 30 predictors. The dataset consists of two classes, "malignant" (value 0 or "negative") and "benign" (value 1 or "positive"). Of the total 569 samples, 212 samples are malignant (37.3%) and 357 samples are benign (62.7%), thus the data is biased in favor of benign tumors. 569 samples and 30 predictors yields a 569×30 design matrix for logistic regression. [SWM93]

3.1 Logistic Regression

Logistic regression was implemented to be performed on the breast cancer data. The implementation contains a `fit` function, that optimizes the regression coefficients $\boldsymbol{\beta}$ with the standard gradient descent method. The coefficients have been initialized with random values from the uniform distribution. The number of iterations, or epochs, and learning rate in the gradient descent are parameters to the `fit` function, and have been tuned to improve the classifier.

The `predict` function in the implementation evaluates the logistic function on $X\boldsymbol{\beta}$, the product of the design matrix and the coefficients from the `fit` function. To convert the probabilities returned from the logistic function to 0 or 1, we use a cut-off threshold, such that we get a "hard classification." The default threshold is 0.5, meaning that any probability over 0.5 will become the value 1 and anything below will become 0. This threshold can be chosen to favor one class over the other in the prediction, for example predict more tumors to be malignant than benign, to lower the chance of missing someone who has cancer. However, the logistic regression performed well with the 0.5 as the threshold, so this is what we used.

The data was split into training and test sets, with 70% of the data allocated to training and the remaining 30% for testing. The test set then contains 171 samples. The data was scaled using scikit-learn's `StandardScaler`. The scaler constructs a normal scaling transformation using the training input data, such that the transformation transforms the training data to be standard normally distributed. Then, using the same transformation generated by the training input data, the test input data was scaled.

The logistic regression was performed on all the possible combinations of learning rates and number of epochs, with learning rates ranging from 0.0001 to

1, and number of epochs from 1 to 101. The accuracy score was used as a metric to measure the performance, and a heatmap of the accuracy for the different combinations can be seen in section 4.1.

3.2 Multilayer Perceptron Network

It is no trivial task to find an MLP that is suitable for a given problem. There are many choices and considerations to be made, such as the number of hidden layers, the number of neurons in each hidden layer, the type of activation function, to mention a few. There is no way to know a priori what will be the optimal choice for the problem at hand. Finding good parameters for the MLP is in itself a matter of optimization or trail and error. We will mainly opt for less complex models, and take an empirical approach to parameter selection. We initialize the MLP with weights that are normally distributed with mean 0 and standard deviation $\sqrt{\frac{2}{N}}$, where N is the number of weights in the given layer, and the biases were initialized to 0, as is suggested in [GB10].

The dataset was split using scikit-learn's `train_test_split`-method, with 70% allocated to training.

3.2.1 Classification

The aim was to implement and train an MLP model on the breast cancer data set to perform binary classification. We initially choose an MLP with a single hidden layer with 50 hidden neurons, trained for 60 epochs with backpropagation with batch size 1, i.e. the model is trained with one pair $(\mathbf{x}, t)$ of input/target at a time. Each pair was used 60 times for training. The learning rate was set to 0.1, and choice of activation function in the hidden layer varied between ELU, ReLu, Leaky ReLu, tanh and logistic. We measure the quality of the model in terms of the accuracy score as the training progresses. The result of this preliminary experiment will be used as a benchmark for further model selection and parameter tuning. We then select the best performing models, and vary the network size and learning rate to explore how the performance changes with these parameters. The number of epochs used in training will be decided from the convergence behaviour observed in the first test. The best test accuracy for each model and each set of parameters observed within the decided number of training epochs will be recorded. We will also compare plots of how the accuracy of the most promising models changes as the training progresses. We will consider two different MLP architectures, namely simple models with few hidden neurons, and complex models with many hidden neurons and several hidden layers.

Lastly we will investigate how the model performs when we change the threshold that decides if we identify a certain prediction as class 0 or class 1, i.e. rounding prediction values to 0 or 1 according to the specified threshold. Incorrectly classifying a tumor as class 1, i.e. a benign tumor, when it is in fact malignant, which corresponds to a false positive, might be more harmful than a false negative, i.e. classifying a benign tumor as malignant. With this in mind, we will evaluate the tradeoff between model accuracy and a preference for false negatives.

3.2.2 Regression with MLP

The aim in this part was to implement an MLP regressor to approximate the Franke function with input variables $\mathbf{x}, \mathbf{y} \in [0, 1]$.

As in the case of classification we initially chose an MLP model with a single hidden layer. To begin with it is tested against the Franke function data without noise, and with different numbers of epochs, to be sure that the model works as it should. Then noise is added to the data and the results are compared to the linear regression results obtained from OLS, Ridge and Lasso, see [HK19].

4 Results

4.1 Logistic Regression

Figure 1 shows a heatmap of the accuracy scores for the different combinations of learning rates and numbers of epochs. We see that the learning rates $\eta = 1, 0.1, 0.01$ give decent accuracy scores regardless of number of epochs. In fig. 2 we examine the accuracy for these three learning rates as a function of number of epochs. The training accuracy scores all increase as number of epochs increases, as expected. The higher the number of epochs, the better the fit is to the training data. However, the test accuracy scores increase until a certain point and then start to decrease. This is a sign of overfitting of the training data. As we want our model to be able to perform well on new data, we want to avoid overfitting and therefore choose the lowest number of epochs that give a desirable accuracy score. We choose the model with 21 epochs and learning rate $\eta = 0.01$.

Epochs	Learning rate	Accuracy	False positives	False negatives
21	0.01	0.97076	2	3

Table 1: The performance of logistic regression on the breast cancer data with 21 epochs and $\eta = 0.01$.

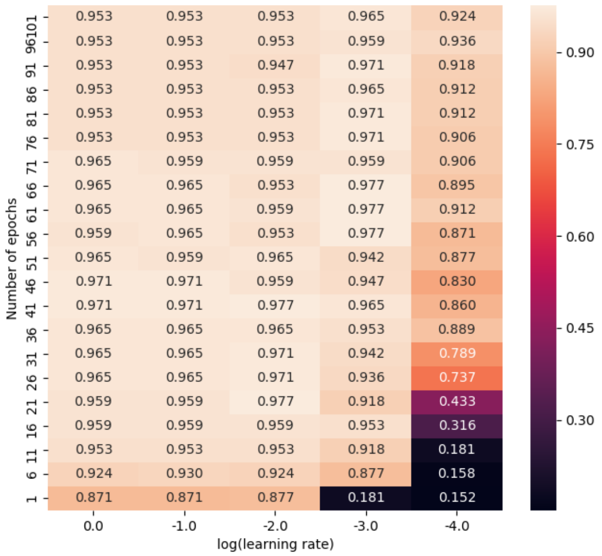


Figure 1: Test accuracy score as a function of number of epochs and learning rate, using logistic regression on the breast cancer data.

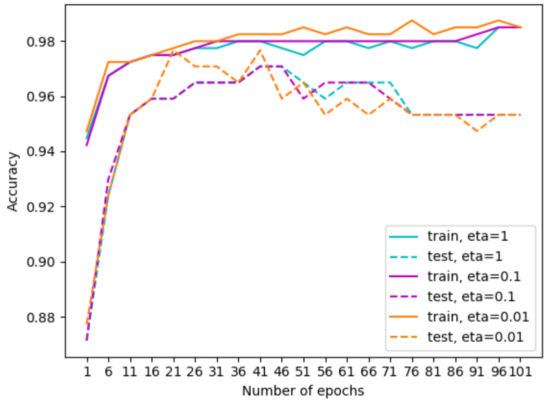


Figure 2: Accuracy score as a function of number of epochs trained, using logistic regression on the breast cancer data.

From table 1, we see that with the chosen model, we have an accuracy of 97%. Of the 171 samples in the test set, 108 samples are benign, or 1's, and 63 samples are malignant, or 0's. The model predicts two observations to be benign that are actually malignant (false positive) and three observations to be malignant that are actually benign (false negatives). We will use this model as a benchmark when comparing logistic regression to neural networks.

4.2 Classification with an MLP Model

Figure 3 shows that that all the models except the logistic activated MLP successfully achieve an accuracy score of around 97% on the test set and the training set, measured after 60 epochs of training. All the models reach peak accuracy on the test set within 40 epochs of training. The plots also show a tendency that the test accuracy does not improve when training is prolonged beyond 40 epoch. In fact, the MLPs with tanh and elu activation functions show a decline in test accuracy as training progresses past 20 epochs. This might indicate that the models have been overfitted, and that it could be beneficial to stop the training at an earlier point. Nevertheless, all the models achieve very good accuracy.

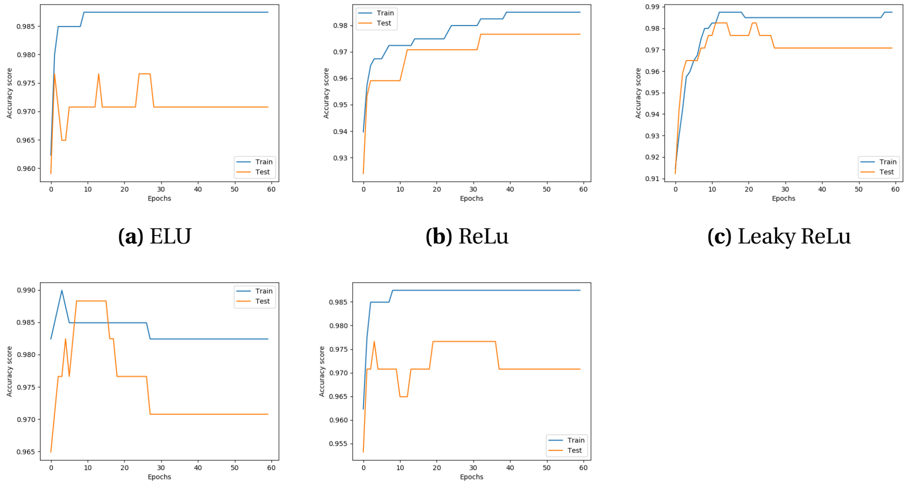
Based on these results, we performed further tests on MLPs with activation function elu, tanh and logstic activation functions. Even though the logistic activation MLP performed worse than the other models, it is included in further analysis as a benchmark since it is a widely used activation function. As there was no indication that training the networks past 40 epochs was beneficial, the selected models were trained for 40 epochs. The set of simple models was chosen to be MLPs with a single hidden layer, with 10, 20 or 30 neurons in the hidden layer respectively, and the mentioned activation functions. The best accuracy on the training set during training was recorded. The results are shown in fig. 4a, while the results for the more complex models are shown in fig. 4b.

The best performing simple model was an MLP with tanh activation function and 10 or 20 hidden neurons. As seen in fig. 5a, it achieved an accuracy score of 0.988, or 98.8%. Table 4 shows how the misclassifications are distributed for the best performing simple models. In the case of the tanh-MLP with 20 hidden neurons, 1 false positive and 1 false negative were observed. Similarly, the number of false positives and false negatives were 1 and 1 respectively for the best performing complex model, which was an MLP with 200 neurouns in the hidden layer.

The two best performing models seem to be MLP with tanh activation function and 20 hidden neurons and MLP with ReLu activation function and 200 hidden neurons. For both models, increasing the threshold for classifying data into class 1 to $0.5, 0.6, 0.7, 0.8$ did not result in a systematic decrease in false positive misclassifications. When the threshold was set to 0.9, the average misclassifications for the tanh MLP with 20 hidden neurons was 0.9 false positives and 7.4 false negatives. For the MLP with ReLu activation function and 200 hidden layers, the misclassifications were on average 1.1 false positives and 8.2 false negatives, calculated by training each and testing a similar model 10 times, trained for 20 epochs on uniquely split and shuffled data. Without the threshold, the results were 2.1, 2.4 and 2.1, 1.8 false positives and false negatives respectively for the two model types. These findings suggests that even though the models were initialized with the same set of hyperparameters, there was significant variation in the performance of these models. Hence, one needs to do extensive testing and often re-run the same initialization and training algorithms on similar models to achieve the best accuracy.

	False positives	**False negatives**	**accuracy score**
ReLu	4	0	0.9766
ELU	3	1	0.9766
leaky ReLu	3	1	0.9766
logistic	11	0	0.9356
tanh	2	2	0.9766

Table 2: Accuracy score and misclassifications for MLPs with activation functions ReLu, ELU, logistic and tanh activation functions.



(a) ELU

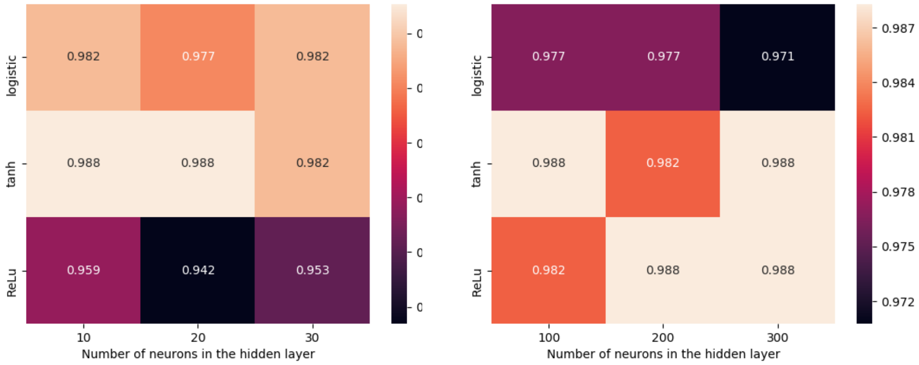
(b) ReLu

(c) Leaky ReLu

(d) Hyperbolic tangent

(e) Logistic

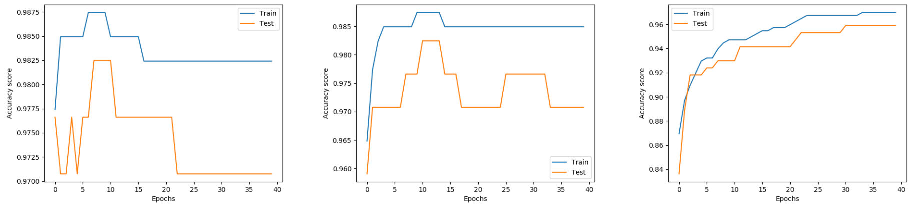
Figure 3: Accuracy score as a function of number of epochs trained. We use a variety of different activation functions.



(a) Simple models

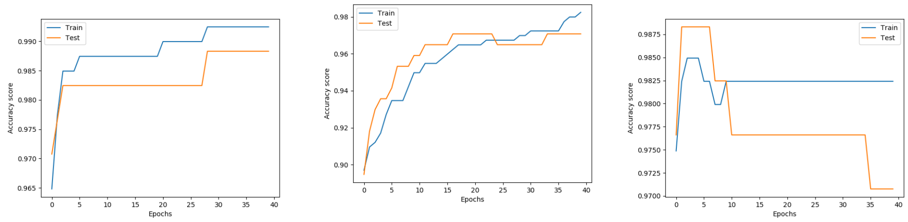
(b) Complex models

Figure 4: Best recorded accuracy score after 40 epochs of training on simple models (a) and complex models (b). A single hidden layer with 10, 20, 30 neurons and 100, 200, 300 neurons respectively.



(a) tanh, 20 hidden neurons **(b)** Logistic, 30 hidden neurons **(c)** ReLu, 10 hidden neurons

Figure 5: Accuracy score on training and testing set as training progresses.



(a) tanh, 300 hidden neurons **(b)** Logistic, 100 hidden neurons **(c)** ReLu, 200 hidden neurons

Figure 6: Accuracy score on training and testing set as training progresses for more complex MLPs

	False positives	**False negatives**
ReLu, 10 neurons	5	2
logistic, 30 neurons	2	1
tanh, 20 neurons	1	1

Table 3: Misclassifications for selected, well-performing simple models with different number of hidden neurons.

	False positives	**False negatives**
ReLu, 200 neurons	1	1
logistic, 100 neurons	2	2
tanh, 300 neurons	2	0

Table 4: Misclassifications for selected, well-performing complex models with different number of hidden neurons.

4.3 Regression

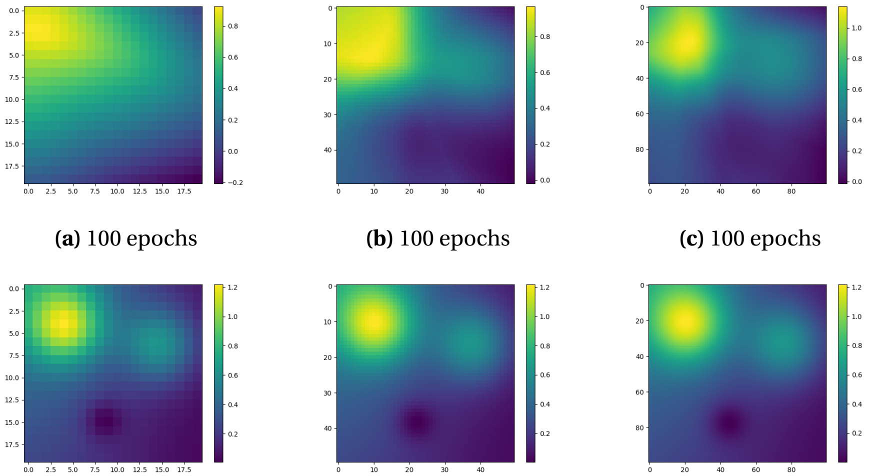
Table 5, together with figs. 7 and 8, show examples of the neural network running as a regressor, with ReLu as the activation function. It was run with different numbers of points, namely 20 × 20, 50 × 50 and 100 × 100, on both noisy and non-noisy data. In fig. 7 we see that all the different choices of numbers of points seem to converge to the Franke function already after 100 epochs. It is not as clear in the case where it is run with 20 × 20, but running more epochs would lead to convergence as well. In fig. 8 the convergence towards the Franke function is not the case, since the data has added noise. In this illustration the plot is shown for both 100 and 2000 epochs, together with the true plots of the Franke function.

Table 6, together with figs. 9 and 10, show a series of runs with different choices of activation functions and different numbers of hidden layers. The program is tested with 1, 2 and 3 hidden layers, with hyperbolic tangent, ELU, ReLu, and Leaky ReLu as activation function. The number of epochs in table 6 is chosen to show approximately where the improvement diminishes, over three different periods. By looking at the MSE and R^2-score in the table, as well as the plots in figs. 9 and 10, we can conclude that this neural network is not the best approach for approximating the Franke function with added noise. Then, it would be more efficient to use a simpler regression method, such as the OLS, Ridge regression or Lasso [HK19].

Some disadvantages when using this FFNN model to approximate the Franke function is that it is easier to obtain overfitting, less chance of convergence, higher complexity of the model and greater time complexity. Figure 10 shows cases of overfitting in the case where the activation function is set to be Leaky ReLu. Otherwise, the plots seem to indicate that the MSE decreases steadily.

	EXACT			NOISY		
Points	**Epochs**	**MSE**	**R2**	**Epochs**	**MSE**	**R2**
20 × 20	1	0.02827	0.5895	1	0.03354	0.6467
	500	0.00549	0.9201	50	0.02243	0.7637
	1000	0.00392	0.9430	100	0.02176	0.7708
	2000	0.00252	0.9633	200	0.02159	0.7726
	3000	0.00184	0.9732	500	0.02170	0.7714
	8000	0.00086	0.9874	2000	0.02171	0.7713
50 × 50	1	0.02076	0.7355	1	0.02056	0.7302
	50	0.00676	0.9139	50	0.01809	0.7626
	100	0.00466	0.9406	100	0.01793	0.7647
	200	0.00295	0.9624	200	0.01770	0.7677
	500	0.00168	0.9785	500	0.01764	0.7685
	2000	0.00120	0.9846	2000	0.01774	0.7671
100 × 100	1	0.01400	0.8251	1	0.01787	0.7794
	50	0.00175	0.9780	50	0.01604	0.8021
	100	0.00089	0.9888	100	0.01587	0.8041
	600	0.00049	0.9938	200	0.01581	0.8049
	800	0.00042	0.9946	500	0.01579	0.8051
	2000	0.00031	0.9960	2000	0.01579	0.8051

Table 5: Regression on Franke function, learning rate = 0.2, test size = 0.2, 1 hidden layer with 512 neurons, activation function=ReLu, SGD, varying number of epochs.



(a) 100 epochs

(b) 100 epochs

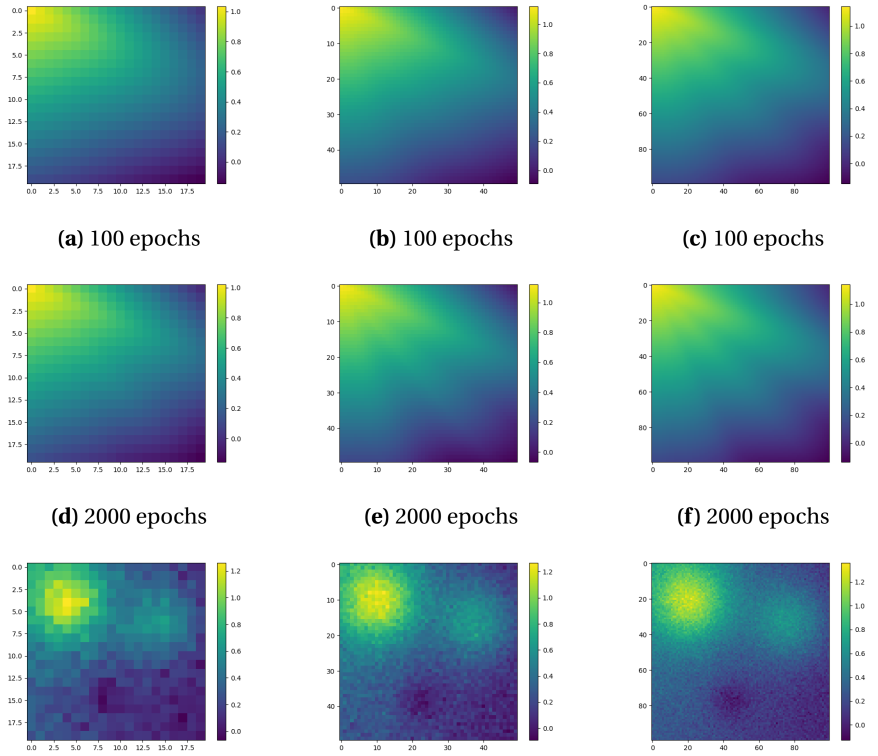
(c) 100 epochs

(d) Original Franke function

(e) Original Franke function

(f) Original Franke function

Figure 7: Regression on Franke function (left: 20 × 20 points, middle: 50 × 50, right: 100 × 100) learning rate = 0.2, test size = 0.2, 1 hidden layer with 512 neurons, activation function=ReLu, SGD, varying number of epochs.



(a) 100 epochs

(b) 100 epochs

(c) 100 epochs

(d) 2000 epochs

(e) 2000 epochs

(f) 2000 epochs

(g) Original Franke function

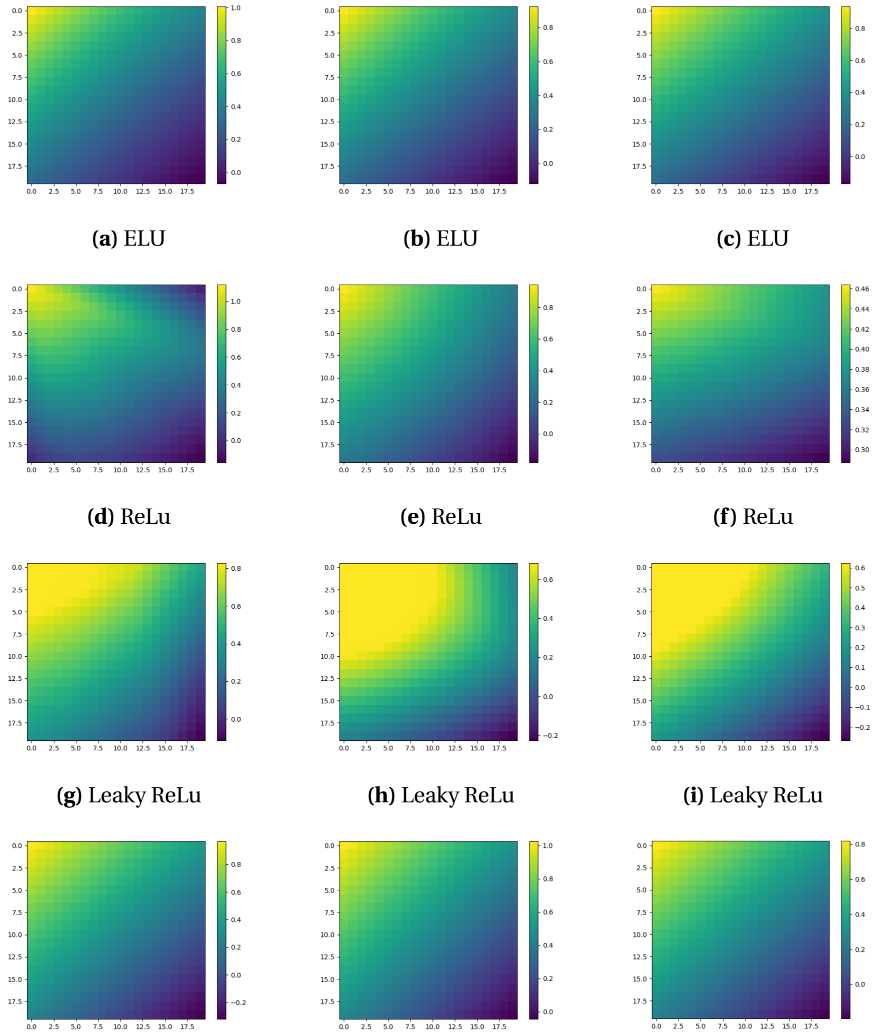
(h) Original Franke function

(i) Original Franke function

Figure 8: Regression on Franke function WITH noise (left: 20 × 20 points, middle: 50 × 50, right: 100 × 100) learning rate = 0.2, test size = 0.2, 1 hidden layer with 512 neurons, activation function=ReLu, SGD, varying number of epochs.

Hidden layers	**Activation function**	**Epochs**	**MSE**	**R2**
1 Layer	tanh	1	0.028835	0.668771
		200	0.023843	0.672104
		500	0.026778	0.692399
	ELU	1	0.062870	0.397360
		200	0.058697	0.437358
		500	0.057523	0.448615
	ReLu	1	0.064451	0.407582
		200	0.054406	0.499913
		500	0.054189	0.501906
	Leaky ReLu	1	0.070486	0.402766
		200	0.055560	0.529236
		500	0.062248	0.472568
2 Layers	tanh	1	0.065443	0.233397
		2	0.058955	0.309402
		500	0.057985	0.320763
	ELU	1	0.062555	0.451449
		200	0.057493	0.495834
		500	0.056641	0.503309
	ReLu	1	0.091376	0.005762
		200	0.049276	0.463842
		500	0.048875	0.468207
	LeakyReLu	1	0.133774	-0.004755
		200	0.073239	0.449916
		500	0.075397	0.433707
3 Layers	tanh	1	0.125677	-0.109847
		20	0.057331	0.493714
		500	0.055152	0.512960
	ELU	1	0.173303	0.011925
		8	0.101564	0.420941
		500	0.093682	0.465878
	ReLu	1	0.090376	-0.005028
		200	0.084581	0.059410
		500	0.077584	0.137221
	LeakyReLu	1	0.112544	-0.003855
		277	0.073029	0.348605
		500	0.078414	0.300573

Table 6: MSE and R^2-score corresponding to figs. 9 and 10.



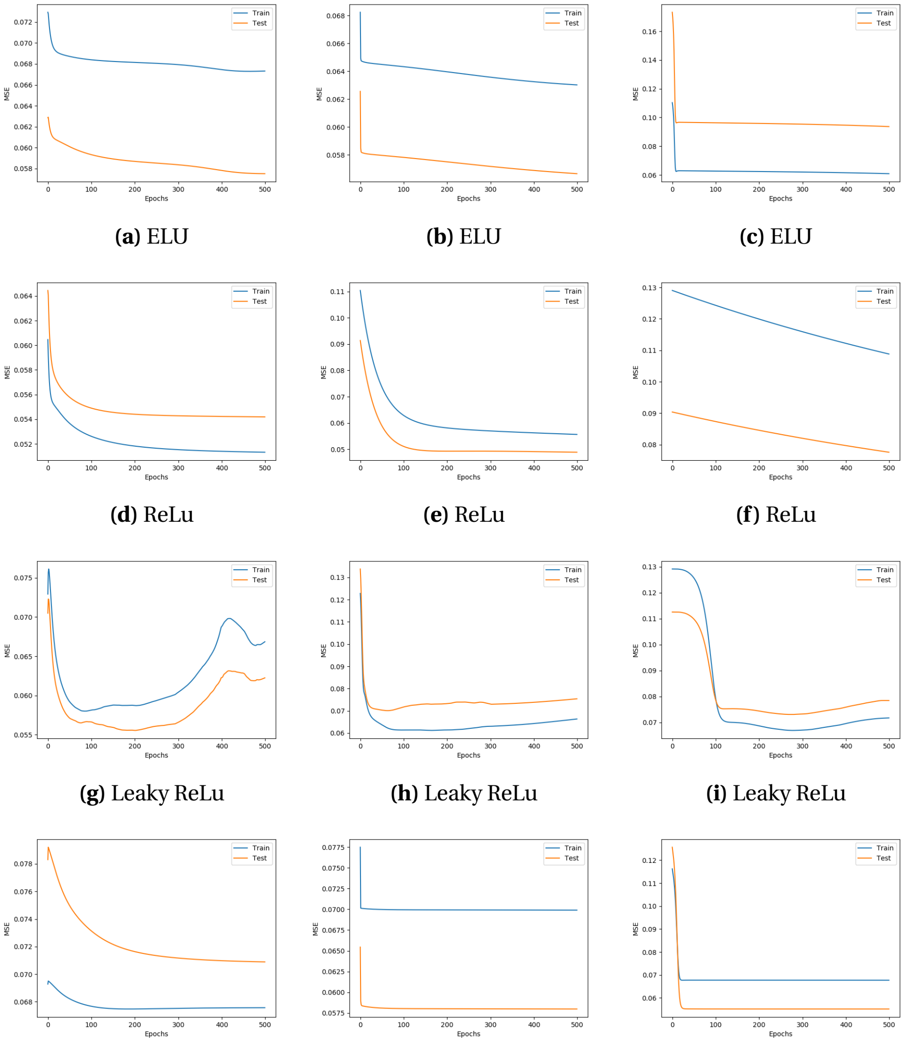
(**a**) ELU (**b**) ELU (**c**) ELU

(**d**) ReLu (**e**) ReLu (**f**) ReLu

(**g**) Leaky ReLu (**h**) Leaky ReLu (**i**) Leaky ReLu

(**j**) tanh (**k**) tanh (**l**) tanh

Figure 9: Regression on Franke function with normally distributed noise ($\mu = 0, \sigma = 0.2$) and 20×20 points. Left: 1 layer with 512 neurons. Middle: 2 layers with 256 and 128 neurons. Right: 3 layers with 128, 64 and 32 neurons. Learning rate 0.2 and test size 0.2. Activation functions ELU, ReLu, Leaky ReLu and tanh on the rows, respectively. Method used is SGD. Run with 500 epochs.



(a) ELU **(b)** ELU **(c)** ELU

(d) ReLu **(e)** ReLu **(f)** ReLu

(g) Leaky ReLu **(h)** Leaky ReLu **(i)** Leaky ReLu

(j) tanh **(k)** tanh **(l)** tanh

Figure 10: MSE from regression on Franke function with normally distributed noise ($\mu = 0, \sigma = 0.2$) and 20×20 points. Left: 1 layer with 512 neurons. Middle: 2 layers with 256 and 128 neurons. Right: 3 layers with 128, 64 and 32 neurons. Learning rate 0.2 and test size 0.2. Activation functions ELU, ReLu, Leaky ReLu and tanh on the rows, respectively. Method used is SGD. Run with 500 epochs.

5 Discussion

The goal of this project was to apply and compare different machine learning algorithms for classification and regression on different datasets.

In the classification case, the two models compared were MLP and logistic regression. We were interested in determining which model was best suited for classifying malignant breast cancer tumors, and to examine the features of these models.

In the regression case we have used an MLP model to predict Franke function values on both noisy and non-noisy data, in comparison to established benchmark results for regression with linear regression models.

As we have discovered, MLP models can be very accurate predictors, and can in many cases outperform logistic regression. However, the models can be exceedingly complicated, difficult to optimize and can often be difficult to interpret. In addition, they are prone to overfitting and rely on lucky initialization.

Logistic regression, on the other hand, is a much simpler and more predictable algorithm, both in terms of implementation and stability in performance. This is why the choice of model depends on application, and we often need to test several different models to find a suitable choice.

An MLP model with ReLu activation, 200 neurons in one hidden layer seemed to perform best. It had an accuracy score of 98.8%. This was better than the best accuracy for the logistic regression model, which was 97.1%.

In the regression case we found no evidence suggesting that an MLP regression model would be more advantageous than a simple linear regression model on the Franke function data set. In the best cases, the MLP performed equally well as the linear regression models. However, it is highly computationally expensive and the outcome of the regression is less stable. In some cases it did not converge to desirable MSE or R^2 scores. The application of an MLP regression model might be more beneficial for more complex datasets.

The best MSE and R^2 score we have achieved with an MLP regression model on noisy data with 20×20 datapoints are 0.027 and 0.692, respectively, achieved with a model with a single hidden layer with a single layer with 512 neurons and tanh activation. This was on data where the noise had $\sigma^2 = 0.04$. The best MSE with ridge regression on the same dataset with the same noise parameter was 0.0416, with a λ parameter of 0.0002. While the MLP model gave better results in the best case, all other combinations of activation function and number of neuron tested were worse than the Ridge regression results.

6 Conclusion

In this project we have looked at several machine learning algorithms and found the performance for each model on some data. We compared an MLP with linear regression models on a regression problem, the Franke function data. The MLP was also compared to logistic regression on a classification problem, the Wisconsin breast cancer data.

In general, both logistic regression and neural networks performed well on the breast cancer data. This is a very small dataset with only 569 samples, and relatively few predictors, which might explain why simple models can perform almost as well as more complex models. It might be tempting to choose a logistic regression model in this case as it is less computationally expensive and easier to interpret. However, basing the choice of model on the highest accuracy score, an MLP model proved to be the best classifier for the cancer data.

In regression problems on relatively well behaved data we have found no grounds to conclude that the more complicated models are better than the simple and well understood regression models. In fact, the simple models did in most cases give better results both in terms of training times and results, and hence are to be preferred in the simple cases.

In conclusion, all the machine learning methods have their advantages and disadvantages and they all have a rightful place. Choosing the optimal model will depend on the dataset and what type of efficiency is prioritized, and will often require domain knowledge. We have found that a complicated ML model might perform very well in the right circumstances, while it fails to compete with simpler models in others.

References

[Cyb89] G. Cybenko. “Approximation by superpositions of a sigmoidal function”. In: *Mathematics of Control, Signals, and Systems (MCSS)* 2.4 (Dec. 1989), pp. 303–314. URL: `http://dx.doi.org/10.1007/BF02551274`.

[GB10] X. Glorot and Y. Bengio. “Understanding the difficulty of training deep feedforward neural networks”. In: *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS’10). Society for Artificial Intelligence and Statistics*. 2010. URL: `http://dblp.uni-trier.de/db/journals/jmlr/jmlrp9.html#GlorotB10`.

[Hjo19a] M. Hjorth-Jensen. *Data Analysis and Machine Learning: Logistic Regression*. Oct. 2019. URL: `https://compphysics.github.io/MachineLearning/doc/pub/LogReg/html/LogReg.html`.

[Hjo19b] M. Hjorth-Jensen. *Data Analysis and Machine Learning: Optimization and Gradient Methods*. Sept. 2019. URL: `https://compphysics.github.io/MachineLearning/doc/pub/Splines/html/Splines.html`.

[HK19] M. H. Havgar and Å. D. Kvitvang. *Regression Analysis and Resampling Methods*. University in Oslo, Nov. 2019. URL: `https://github.com/aasmunkv/PROJECT1_FYS-STK4155`.

[Mar09] S. Marsland. *Machine Learning - An Algorithmic Perspective*. 2009.

[Ped+11] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[SWM93] W. N. Street, W. H. Wolberg, and O. L. Mangasarian. “Nuclear feature extraction for breast tumor diagnosis”. In: *Biomedical image processing and biomedical visualization*. Vol. 1905. International Society for Optics and Photonics. 1993, pp. 861–870.