UiO : University of Oslo

FYS-STK4155 – Applied Data Analysis and Machine Learning

# Project 2: Classification Methods with Neural Networks and logistic regression

November 13, 2020


*Authors:*

Håkon Berggren Olsen

Elisaveta Dombrovski

Lasse T. Keetz

*Lecturer:*

Prof. Morten Hjorth-Jensen

**Abstract**

The field of Machine Learning provides state-of-the-art tools to tackle classification and regression problems. In this report, we are evaluating two different classification algorithms, namely Neural Networks and Multi-Class Logistic Regression on the MNIST dataset, which contains grey scale matrix values depicting hand written numbers with the goal to correctly assign each image with the shown integer. These methods are evaluated using common accuracy metrics. We are also exploring how different activation functions affect neural network performance. Moreover, we investigate the dependency of the penalty factor on the Neural Network by imposing an $l2$-regularization on the cost function with the aim to reduce overfitting. Additionally, we illustrate how different parametrizations of Stochastic Gradient Descent methods minimize the cost function in machine learning, exemplified by a standard linear regression use case. With regards to optimizing the weights and biases of a neural network, we find that the Adam optimization method outperforms SGD, SGDM and RMSprop, resulting in both a higher accuracy and faster convergence of the cost function. Revisiting the regression algorithm for the Franke Function and the Terrain data from Project 1, we demonstrate the strength of Neural Networks that can be easily adapted to linear regression and lead to better model performance. We find that the ReLU activation function outperforms the traditional sigmoid activation function in both $R2$ and MSE with a more quickly converging cost function. Finally, even though the final model performance is satisfactory for our applications, we acknowledge that even more sophisticated approaches have been discussed in the literature that motivate additional exploration.

# Contents

# 1 Introduction

The field of Machine Learning (ML) provides a powerful toolbox for tackling classification and regression problems across a wide range of scientific disciplines. The associated methodologies are commonly considered state-of-the-art due to their ability to automatically detect patterns and relationships between input parameters for vast amounts of data (Murphy, 2013). Enabled by recent advances in computer science, the field is currently undergoing perpetual and rapid developments, designed to even further improve model performance and interpretability. ML is, therefore, the basis for many applications that facilitate daily life in the modern era such as biomedical research, information technologies, and transportation - just to name a few (Marsland, 2009). In this project, we will present an application of a subset of impactful methods and their underlying principles. In particular, we focus on a case study for regression and classification using neural networks (NN). In this context, we will also show an example of how machine learning models in general improve their performance over time using Gradient Descent methods. Finally, we will compare the results of using NN to a less complex approach, namely Logistic regression.

Data classification can be broadly defined as the process of assigning data to homogeneous groups based on their characteristics. This can help to reduce the dimensionality of data, e.g. making it more easily interpretable and comparable. It also enables associated additional statistical analyses and focusing on desired data properties. A simple example is to assign places to a specific country based on their geographical location. Regression, on the other hand, explores the mathematical relationship between a set of dependant variables or outcomes to associated independent variables or predictors. This also allows to make predictions for outcomes outside the initial training data feature space. An example would be to relate the price of a house to properties such as location and size (Marsland, 2009).

Neural networks can be used for both purposes, i.e. classification and regression analyses. Essentially, they were designed to emulate the human brain and learning processes. A set of nodes (analogy: brain cells or neurons) is connected (analogy: synapses) by passing the weighted input or output of mathematical functions to other nodes. Thus, the training data that is fed into the "input nodes" of the NN gets transformed within its architecture and produces an output at the "output nodes". Thereby, also non-linear relationships between feature and outcome can be detected. The output can either be continuous numbers, but also discrete numbers or classes in case of classification. Commonly, the nodes are grouped into "layers" that share the same direction of input/output they get/pass on (Figure 1). Here, the "activation function" determines how the input a node receives is mathematically processed. By iteratively changing the weights

that ultimately determine the output, or alternating the activation functions and other hyper-parameters such as the number of nodes in a layer, this principle is used to reproduce the training target data in a desired way (i.e. based on minimizing a chosen "cost function" or error). As a consequence, the model automatically "learns" how input and output are connected, given there is a meaningful relationship between the two (see e.g. Nielsen, 2015). We present the details of this procedure, i.e. how the model minimizes the error between calculated output and the observed targets mathematically, in section 2.

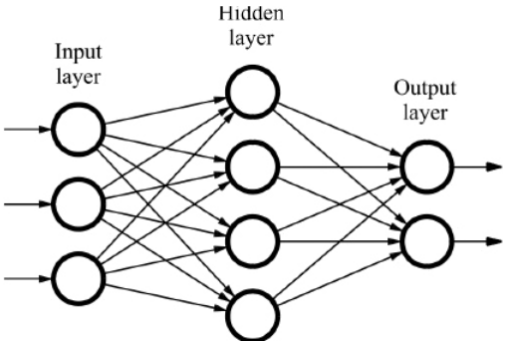



**Figure 1:** *The basic architecture of a Feed Forward Neural Network. Obtained from https://databricks.com/glossary/neural-network [accessed 10-11-2020].*

We will apply the aforementioned concepts both to "artificially" generated data, where the distribution is hence known, and to a "real world" classification problem. The former is based on adding random noise to the Franke Function (cf. Project 1) and mainly used to illustrate SGD. The latter consists of image files showing hand written numbers [0-9] where we train a neural network to automatically assign the corresponding depicted integer values. The performance is critically evaluated and compared against logistic regression results.

The structure of the project report is as follows. Section 2.1 will present the theory behind the methods and metrics we use for evaluation. Section 3 contains additional background information on the data sets. In section 4.1, we will present and discuss the results. Finally, we will summarize the key findings and possible limitations in section 5.

# 2 Theory and Methodology

## 2.1 Stochastic Gradient Descent

Gradient descent is an iterative way of finding the minimum of a desired function. Starting the function at some initialized points, often randomized within a certain range, the next step is found by subtracting the point with the gradient of that point, multiplied with some step size hereafter also referred to as the "learning rate". The iterative scheme can be written as

$$f(\boldsymbol{x}_{n+1}) = f(\boldsymbol{x}_n) - \eta \nabla f(\boldsymbol{x}_n) \tag{1}$$

For regression and classification, one aims to find a certain set of parameters, $\beta$ which minimizes some function $C$. The iteration scheme will then take the form

$$\beta_{n+1} = \beta_n - \eta \nabla C(\boldsymbol{X}, \beta_n) \tag{2}$$

Computing gradient descent for all the data points is the most straight forward, naive way of minimizing a function, but the thorough and tedious process of evaluating every increment until it converges to a certain threshold is computationally heavy. A compromise is therefore to take randomized subsets of the data, called mini-batches, multiple times through epochs. This is called stochastic gradient descent, and its pseudo code is shown below.

**Data:** x, y
**Result:** Stochastic Gradient Descent
Initialize the weights and biases;
**for** *epoch* $\leftarrow$ 1 **to** *max epoch* **do**
  Randomly Shuffle data set;
  **for** $x_b \leftarrow$ 1 **to** *n batches* **do**
    Calculate Gradient $G(x_b, \beta)$;
    $\beta \leftarrow \beta - \eta \mathrm{G}(\mathrm{x}_b, \beta)$;
  **end**
**end**

The optimization method provided in the pseudo code conveys the core idea, but there are also several other popular optimization methods which can prove more effective. Some of them are summarized in the following sections.

### 2.1.1 SGD Momentum

SGD momentum (Sutskever, 2013) calculates the next update of the estimator by adding the previous change of the estimate, multiplied with a coefficient $\mu$ called the "momentum coefficient".

$$v_{i+1} = \rho v_i - \eta \nabla C(\boldsymbol{X}, \beta_n) \tag{3}$$

$$\beta_{n+1} = \beta_n + v_{i+1} \tag{4}$$

### 2.1.2 RMSprop

RMSprop is an advancement of SGD with momentum, where the momentum is squared. $v$ is here a moving average of the gradients. Dividing the gradient by the moving, squared average of the gradient helps combats large values in the gradient not overstepping.Hinton (2012)

$$v_{i+1} = \rho v_i + (1-\rho)(\nabla C(\boldsymbol{X}, \beta_n))^2 \tag{5}$$

$$\beta_{n+1} = \beta_n - \frac{\eta}{\sqrt{v_i + \epsilon}} \nabla C(\boldsymbol{X}, \beta_n) \tag{6}$$

The gradient is squared elementwise, an $\epsilon$ is a small number ($\approx 10^{-8}$) to avoid division by zero. $\rho$ is the forgetting factor which usually is set to 0.9 (Hinton, 2012).

### 2.1.3 Adam

Adam optimzers is short for Adaptive Moment Estimation. It computes moving averages, $m$ and $v$ of the gradient and the gradient squared, respectively. $m$ is an estimate of the mean, while $v$ is an estimate of the variance (Ba, 2014).

$$m_{n+1} = \rho_1 m_n + (1-\rho_2)\nabla C(\boldsymbol{X}, \beta_n) \tag{7}$$

$$v_{n+1} = \rho_2 v_n + (1-\rho_1)(\nabla C(\boldsymbol{X}, \beta_n))^2 \tag{8}$$

$$\hat{m}_{n+1} = \frac{m_{n+1}}{(1-\rho_1^{n+1})} \tag{9}$$

$$\hat{v}_{n+1} = \frac{v_{n+1}}{(1-\rho_2^{n+1})} \tag{10}$$

$$\beta_{n+1} = \beta_n - \eta \frac{\hat{m}_{n+1}}{\sqrt{\hat{v}_{n+1}} + \epsilon} \tag{11}$$

Here $\epsilon$ is a small scalar to avoid division by zero, $\approx 10^{-8}$, and $\beta_1, \beta_2$ are "forgetting factors", usually having values $0.9, 0.999$ respectively (Ba, 2014).

## 2.2 Logistic Regression

Logistic regression is a classification algorithm where the objective is to estimate to which target class $\boldsymbol{y}$ a set of given explanatory variables $\boldsymbol{X}$ most probably can be associated with. The output vector $\boldsymbol{y}$ contains "1" for the predicted class with highest probability and 0 for the others (i.e. 100% that this is the correct class). This way of writing $\boldsymbol{y}$ is a called "one-hot-encoded"-vector.

Logistic regression aims to fit a polynomial parameterization with the logit odds given in 12. For convenience, the the theory discussed will be a binary case where the "one-hot-encoded" vector is just a scalar.

$$\log\left(\frac{p}{1-p}\right) = \boldsymbol{\beta X} \tag{12}$$

Here $p$ denotes the probability of the model that the outcome class is "0" or "1" and is the desired output of our classification. Rearranging with respect to $p$ gives

$$\boldsymbol{p} = \frac{1}{1 + exp(-\boldsymbol{\beta X})} \tag{13}$$

This is the sigmoid function and squishes values from $(-\infty, \infty)$ to $(0, 1)$. The loss function can be found by taking the log of Maximum Likelihood. For a binary case, the maximum likelihood is given as

$$L(\mathcal{D}|\beta) = \prod_{i=n}^{N} p^{y_i}(1-p)^{(1-y_i)} \tag{14}$$

Taking the logarithm

$$C(\beta) = \sum_{i=n}^{N} y_{ic} \log(\sigma\left(\boldsymbol{\beta X}\right)) + (1-y_i)\log(1-\sigma\left(\boldsymbol{\beta X}\right)) \tag{15}$$

The gradient for logistic regression (Hjorth-Jensen, 2020b), i.e. the partial derivative of C with respect to $\beta$, can then be written as

$$\frac{\partial C(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -\boldsymbol{X}^T\left(\boldsymbol{y} - \boldsymbol{p}\right). \tag{16}$$

### 2.2.1 Accuracy

To evaluate the performance of the classification methods, both logistic regression and Neural Network, the accuracy is calculated and is defined as

$$\text{Accuracy} = \frac{\sum_{i=0}^{N} I(p_i = y_i)}{N} \tag{17}$$

where I is the indicator function with 1 if $p_i = y_i$ ($p$=target, $y$=modelled) and 0 otherwise for a binary classification problem. For regression, the evaluation metrics will be $R2$ and $MSE$, as already presented and discussed in Project 1.

## 2.3 Feed Forward Neural Network

A feed forward neural network is network of affine linear transformations being fed into different activation functions at each layer, resulting in a desired output space at last layer.

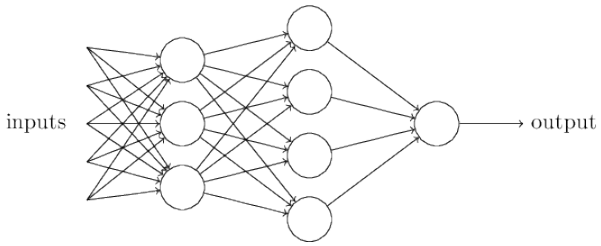



**Figure 2:** *Overview of Feed Forward Neural Network architecture with a single output node as can be used for binary classification.*

The notation discussed here will be used in the python code and the discussion of the theory of neural networks:

1. Input features will be denoted as $X$
2. The desired output will be denoted as $y$
3. The weights and biases will be denoted as:
    - $W^l$, as a matrix of the weights

- $w_{jk}^l$, an entry in the matrix
- $b^l$, as a vector of the biases
- $b_k^l$ as an entry in the vector

4. Here, the indices are:
    - The direction to the next layer: $j$
    - The neuron where the weight and bias is applied: $k$
    - The layer in the network: $l$

### 2.3.1 Forward Propagation

Forward propagation is the way neural network models the output and gives a prediction. The features, $X$, are entered into the input layer $l = 0$. All the nodes in the next layer have their own weights and biases, $W^l$ and $b^l$ gets multiplied and added to give a weighted sum. The weights are directed

$$z_j^l = \sum_k w_{jk}^l x + b_k^l A \tag{18}$$

This sum then gets fed into an activation function $f$, and subsequently forwarded to the next layer.

$$a_k^{l+1} = f(z^l) = f(\sum_k w_{jk}^l x + b_k^l) \tag{19}$$

Written in vector form, this is written as:

$$\boldsymbol{a^l} = f\left(\boldsymbol{w^l a^{l-1}} + \boldsymbol{b^l}\right) = f\left(\boldsymbol{w^l} f\left(\boldsymbol{w^{l-1}} f(...) + \boldsymbol{b^{l-1}}\right) + \boldsymbol{b^l}\right) \tag{20}$$

## 2.4 Cost/Loss Function

The cost function is the objective function we want the neural network to minimize based on the type of problem. The individual functions are presented below.

### 2.4.1 MSE

For regression problems, we will use the MSE as the cost function for the NN given as

$$C = \frac{1}{2}\left(y - a_L\right)^2 \tag{21}$$

Here $a_L$ is the activation of the last neuron in the layer, as we are using the identity activation function to be able to predict all values the regression can have. Taking the derivative wrt. z:

$$\frac{\partial C}{\partial z} = -f'(z_L)(y - f(z_l)) = (f(z_L) - y) \tag{22}$$

Where the $f'(z_L) = 1$ as we will be using a linear output for the regression.

### 2.4.2 Cross Entropy

For the classification neural network, the cross entropy cost function will be used. The cross entropy stems from the Maximum Likelihood Estimator (MLE) in which we want to maximize. The MLE can be written as

$$P(\mathcal{D} \mid \hat{\theta}) = \prod_{i=1}^{n} \prod_{c=0}^{C-1} [P(y_{ic} = 1)]^{y_{ic}}. \tag{23}$$

Where $y_i c$ $i$-th and $c$th element of the y vector being transformed into a "one-hot-dummy"-vector as a K-length binary digit string. Taking the negative logartihm of this gives us the cross entropy cross function:

$$C = -\log P(\mathcal{D} \mid \hat{\theta}) \tag{24}$$

Defining the $P(y_{ic} = 1)$ as the softmax $p(\boldsymbol{z})$

$$p(\boldsymbol{z}) = \frac{e^{z_c}}{\sum_{j=1}^{C} e^{z_j}} \tag{25}$$

where $z_c$ is the highest value in $\boldsymbol{z}$, we can rewrite the cost function for cross entropy as

$$C = -\sum_k \boldsymbol{y} \log(a_L) = -\sum_k \boldsymbol{y} \log(\frac{e^{z_c}}{\sum_j e^{z_j}}) \tag{26}$$

To find the derivative of the softmax $p(\boldsymbol{z})$ wrt. $a_L$, we use the quotient rule. $z_m$ is the $m$-th output value from the last layer, and $p_c$ which is the $c$-th softmax value. For $m = c$ we have

$$\frac{d}{dz_m} p_y = \frac{d}{dz_m} \frac{e^{z_c}}{\sum_j e^{z_j}} = \frac{e^{z_c} \sum_j e^{z_j} - e^{z_c} e^{z_m}}{\sum_j e^{z_j} \cdot \sum_j e^{z_j}} \tag{27}$$

$$\frac{d}{dz_m} \frac{e^{z_c}}{\sum_j e^{z_j}} = \frac{e^{z_c}}{\sum_j e^{z_j}} \frac{\sum_j e^{z_j} - e^{z_m}}{\sum_j e^{z_j}} = p_c(1 - p_m) \tag{28}$$

For $m \neq c$:

$$\frac{d}{dz_m} p_y = \frac{d}{dz_m} \frac{e^{z_c}}{\sum_j e^{z_j}} = \frac{0 \cdot \sum_j e^{z_j} - e^{z_c} e^{z_m}}{\sum_j e^{z_j} \cdot \sum_j e^{z_j}} = -p_m p_c \tag{29}$$

The derivative of the cost function with respect to $z_m$ is

$$\frac{\partial C}{\partial z_m} = -\sum_{c=1}^{C} y_c \frac{\partial}{\partial z_m} \log(p_c) \tag{30}$$

$$= \sum_{c=1}^{C} y_c \frac{1}{p_c} \frac{\partial p_c}{\partial z_m} \tag{31}$$

$$= -y_m(1 - p_m) - \sum_{c \neq m}^{C} \frac{y_c}{p_c} \frac{\partial p_c}{\partial z_m} \tag{32}$$

$$= -y_m + y_m p_m + \sum_{c \neq m}^{C} y_c p_m \tag{33}$$

$$= -y_m + \sum_{c=1}^{C} y_c p_m = -y_m + p_m \left( \sum_{c=1}^{C} y_c \right) = p_m - y_m \tag{34}$$

Comparing with the $\partial C / \partial z$ from the MSE with linear regression (section 2.4.1), we get the same answer and will therefore keep the error $\delta_L$ for the last layer always equal to

$$\delta_L = y - a_L \tag{35}$$

### 2.4.3 Regularization

Adding regularization to prevent over fitting, can be generally described by Nielsen (2015) as

$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2 \tag{36}$$

Where $C_0$ is the un-regularized cost function, $\lambda$ is the penalty factor and $\sum_w w^2$ is the sum of all the weights. As the network aims to minimize the cost function, the inclusion of a regularization term will incentivize the network to learn smaller weights. With the $l2$ norm of the weights being low, changes in the input vector by some small amount would intuitively not result in a large change in the final output, and can therefore reduce overfitting (Nielsen, 2015).

The gradient of the cost function wrt. the weights then get added a second term

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} w \tag{37}$$

and the gradient of the bias remains the same.

## 2.5 Back Propagation

The back propagation algorithm is, in part with the optimization method, how the neural network learns. It is a method to calculate the gradients of the cost function wrt. all the weights and biases in each layer. For a given neural network with $L$ number of layers, one can find the gradient by applying the chain rule on the derivative of the cost function.

Starting with the last layer $l = L$, we have

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial w_{jk}^{L-1}} = \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} \frac{\partial z_j^L}{\partial w_{jk}^l} \tag{38}$$

The first term on the right hand side is called the error term and can be written as

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C}{\partial z_j^L} \tag{39}$$

Written in vector notation this becomes

$$\delta^L = \nabla_a C \odot f'(z^L) \tag{40}$$

where $\odot$ represents the harmand product which is element wise matrix multiplication (Nielsen, 2015). With the error term, eq. 38 can be rewritten as

$$\frac{\partial C}{\partial w_{jk}^l} = \delta_j^L a_j^{L-1} \tag{41}$$

because $\partial z_j^L / \partial w_{jk}^l$ is just equal to the activation output of the previous layer. We then have definition for the derivative of the cost function with respect to the cost function (Nielsen, 2015). It is given wrt. the bias as

$$\frac{\partial C}{\partial b_k^l} = \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} \frac{\partial z_j^L}{\partial b_k^l} = \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} = \delta_j^l \tag{42}$$

As the $\partial z_j^L / \partial b_k^l$ is equal to 1.

To express the error term for the hidden layers $l < L$, expanding it with the chain rule and summing over the $k$ nodes (neurons) gives

$$\delta_j^l = \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1} \tag{43}$$

The $\partial z_k^{l+1} / \partial z_j^l$ term can be written as

$$\frac{\partial}{\partial z_j^l}(z_j^{l+1}) = \frac{\partial}{\partial z_j^l} \left( \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} \right) = \frac{\partial}{\partial z_j^l} \left( \sum_j w_{kj}^{l+1} f(z_j^l) + b_k^{l+1} \right) \tag{44}$$

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{ij}^{l+1} f'(z_j^l) \tag{45}$$

And the error term for layer $l$, finally be written as

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} f'(z_j^l) \tag{46}$$

And in vector notation

$$\delta^l = ((W^{l+1} \delta^{l+1})^T \odot f'(z^L) \tag{47}$$

The last equation, eq. 47, shows the power of the back propagation as the deeper layers $\delta^l$ can be calculated only by applying the harmand product of $(W^{l+1}\delta^{l+1})^T$ the "previous" error term $\delta^{l+1}$, instead of calculating the gradient at every step (Nielsen, 2015).

So in eq. 48, eq. 49, eq. 50 and eq. 51 we have all four formulas needed to write the back propagation algorithm. To recap:

$$\delta^L = \nabla_a C \odot f'(z^L) \tag{48}$$

$$\delta^l = ((W^{l+1} \delta^{l+1})^T \odot f'(z^L) \tag{49}$$

$$\frac{\partial C}{\partial b_k^l} \tag{50}$$

$$\frac{\partial C}{\partial w_{jk}^l} = \delta_j^L a_j^{L-1} \tag{51}$$

### 2.5.1 Pseudo code for the Back Propagation

The pseudo code for a generalized neural network is given below. The pseudo code highlights the back-propagation method and does neither take into account what cost function is being used, the cost functions' regularization, nor the activation functions. Therefore this method generalizes to both classification and regression oriented networks.

**Data:** x, y

**Result:** Back Propagation Pseudo Code

Intilialize the weights and biases;

**for** *epoch* $\leftarrow 1$ **to** *max epoch* **do**

Feed x data through all layers;

$\mathrm{a}_l = f(w_{l-1}f(w_{l-2}f(w_{l-3}...f(X)... + b_{l_3}) + b_{l_2}) + b_{l_1})$

Evaluate loss function;

Compute the error term of the last layer $L$;

$\delta^L = \frac{\partial C}{\partial(a^L)f'(z^L)}$

Back propagating the error term;

**for** $l \leftarrow (L-1)$ **to** 2 **do**

$\delta_j^l = \sum_k w_{kj}^{l+1}\delta_k^{l+1}f'(z_j^l)$**end**

Update the parameters;

$\mathrm{w}_{jk}^l \leftarrow w_{jk}^l - \eta\delta_j^l a_k^{l-1}$

$\mathrm{b}_k^l \leftarrow b_k^l - \eta\delta_j^l$

**end**

## 2.6 Hyperparameters

In machine learning, a hyperparameter is a fixed parameter that regulates the learning process and is, hence, not derived during the training procedure. Here, the hyperparameters include the layer dimensions (i.e. number of hidden layers and number of neurons in the corresponding layers), regularization to the cost function (e.g. $l_1$, $l_2$ norm), activation functions, and the learning rate. The learning rate is a highly tuned parameter which is more explained in the subsection of stochastic gradient descent, the other ones are described below.

## 2.7 Activation Functions

The different activation functions utilized in this study are explained below. For an activation function to be used in a neural network, one needs to explicitly know its derivative before training the network as discussed in 2.5. The different function derivatives are included in the respective sections. The choice of activation function depends on each use case. For instance, the sigmoid function (see below) is more biologically plausible (with respect to "emulating" the human brain) as the output of inactive neurons would be "0". On the other hand, it was shown that the hyperbolic tangent function (see below) often leads to higher accuracy in deep neural network applications (Hjorth-Jensen, 2020c). It should therefore be critically considered when interpreting results.

### 2.7.1 Sigmoid

The sigmoid function, or the logistic or the "logit" function is one of the oldest activation functions.

The sigmoid function is given as

$$\sigma(z) = \frac{1}{1+e^{-x}} \tag{52}$$

and its derivative as can after a bit of algebra be shown to be equal to

$$\sigma'(z) = \sigma(z)(1-\sigma(z)) \tag{53}$$

takes values from $(-\infty, \infty)$ and outputs $[0, 1]$.
A problematic feature with using this activation function is evident by the derivative of the function. When the input is very large, the derivate is $\approx 0$ and the error term discussed in 2.5 will vanish. This hinders the ability for the network to learn (Hjorth-Jensen, 2020d).

### 2.7.2 tanh

The tanh function is given as

$$\tanh(z) = \frac{e^x - e^{-x}}{e^x + -x} \tag{54}$$

$$\tanh'(z) = 1 - \tanh^2(z) \tag{55}$$

### 2.7.3 ReLU

The ReLU stands for Rectified Linear Unifier and is equal to its input if $z > 0$ and 0 for every value else.

$$\text{ReLU}(z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases} \tag{56}$$

and its derivative is just the Heaviside step function

$$\text{ReLU}'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases} \tag{57}$$

The leaky ReLU allows a small slope on the values $z \leq 0$, denotes as *alpha*. If $alpha = 0$, this is just the regular ReLU.

$$\text{leaky ReLU}(z) = \begin{cases} z & \text{if } z > 0 \\ \alpha & \text{if } z \leq 0 \end{cases} \tag{58}$$

and its derivative

$$\text{leaky ReLU}'(z) = \begin{cases} 1 & \text{if } z > 0 \\ \alpha & \text{if } z \leq 0 \end{cases} \tag{59}$$

### 2.7.4 Softmax function

The softmax function will be used when predicitng the probability of multi-class classification (Hjorth-Jensen, 2020a). It takes a vector and normalizes it as

$$p(z) = \frac{exp(z)}{1 + \sum_j^C exp(z_c)} \tag{60}$$

# 3 Data

The datasets we use for our classification exercise is the MNIST dataset, details follow in the next sections. For the data used in the regression problem, we refer to Project 1 where the Franke function and the terrain data from southern Norway are presented in detail.

### 3.1 MNIST

The dataset we will study for the classifcation methods is the MNIST dataset (Dua and Graff, 2017), which is optically scanned handwriting of numbers compressed down to an 8x8 matrix, see Figure 3.The values in matrix range from 0 to 16, with higher value denoting blacker values. There are a total of 1797 images, all with correct classified labels, i.e. the desired number assigned to the handwritten image. All the pixels in the $8x8$ matrix are treated as an explanatory variabel (logistic regression)/ feature (neural network), and is ravelled, and the labels are the desired output.

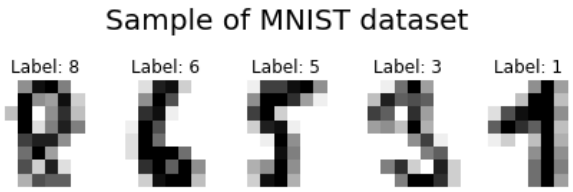



**Figure 3:** *Sample from the MNIST database, showing the features as a* $8x8$ *grid with corresponding labels indicating the number drawn.*

## 4 Results and discussion

### 4.1 Stochastic Gradient Descent

As shown in section 2.1, Stochastic Gradient Descent (SGD) is an iterative approach to minimize a chosen loss function that is frequently used in machine learning. Here, we implemented SGD to replace the analytical solution for the same linear regression problem presented in Project 1. The main intention is to illustrate its basic principles and to explore the implications of using different SGD hyperparameters on model performance. More specifically, we generate the target data by passing an array of x and y values to the Franke Function and subsequently adding normally distributed random noise of the form $\mathcal{N}(0, 0.1)$ to the output. Then, the feature matrix is constructed using a two-variable input polynomial function of a given degree $n$ (here: $n = 5$). Additional details about the model setup, which is consistently used for the results throughout this section, are summarized in Table 1. For in-depth background information on the Franke function and the polynomial fit refer to Project 1.

**Table 1:** *Parameters used to generate the feature matrix (2-variable polynomial of degree 5) that is fitted to the Franke Function with added random noise.*

| $n$ observations | $n$ features ($\beta$) | Scaling (Pedregosa et al., 2011) | Test set proportion |
|---|---|---|---|
| 10,000 | 21 | StandardScaler() | 0.2 |

#### 4.1.1 SGD for OLS

The first experimental setup was to use standard SGD parameters for the OLS method, thus, omitting a regularization term. Figure Figure 4 shows how the MSE gradually decreases over time (i.e. with each iteration or epoch) as the SGD method changes the input feature weights $\beta$ in small steps, thereby approaching a minimum of the cost function.

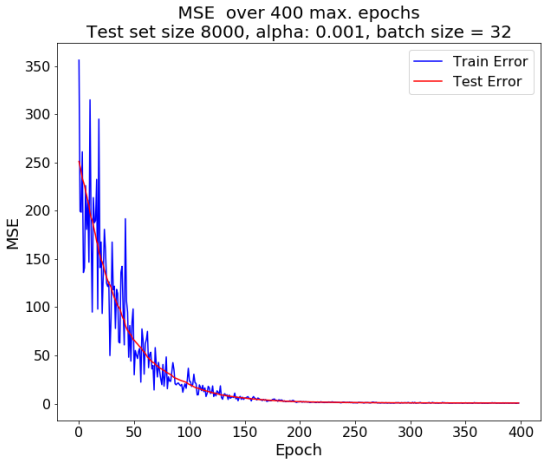



**Figure 4:** *Mean Squared Error (MSE) per epoch when minimizing the cost function (MSE for 2-variable polynomial linear fit to Franke function) using the Stochastic Gradient Descent (SGD) method. SGD standard hyperparameters are indicated in the title.*

With the chosen parameters, SGD converges towards low MSE values after approx. 200 epochs (Fig. Figure 4). Note how the MSE values for the training set are oscillating with high magnitude relative to the respective values for the test set. This can be explained by the fact that these error values are obtained using only a small subset of the data as each epoch optimizes a mini batch $m$ of the full training data set (i.e. here only 32 observations), whereas this error gets smoothed or "averaged out" when applied to the larger test set ($n = 2000$).

### 4.1.2 Learning rate

Next, we illustrate how the learning rate $\alpha$ affects the optimization procedure in our example. Figure 5 depicts how the MSE evolves using SGD over time when varying $\alpha$ within the given ranges. Figure 5 (a) shows the results for a range of sufficiently small learning rates ($\alpha <= 0.1$). It is evident that the MSE is gradually decreasing for each of the chosen values. However, for very small $\alpha$ (e.g. 0.0001), the cost function decreases more slowly and it did not yet converge to a stable minimum value under the chosen maximum amount of iterations (max epochs = 400). This underlines that too small learning rates result in additional time requirement for the algorithm to converge, which can be problematic if the amount of data is large and computational time is a limiting factor. Accordingly, under the given conditions, a learning rate of 0.1 results in an almost immediate minimization. On the other hand, if the learning rate is too large as shown in Figure 5 (b) (cf. $\alpha = 0.2$), SGD can fail completely ("overshoot") and diverge. Consequently, it does not minimize the cost function. It is, therefore, worthwhile to carefully tune $\alpha$ to an optimum value to achieve best performance with respect to computational time using this implementation of SGD.

**(a)** **(b)**

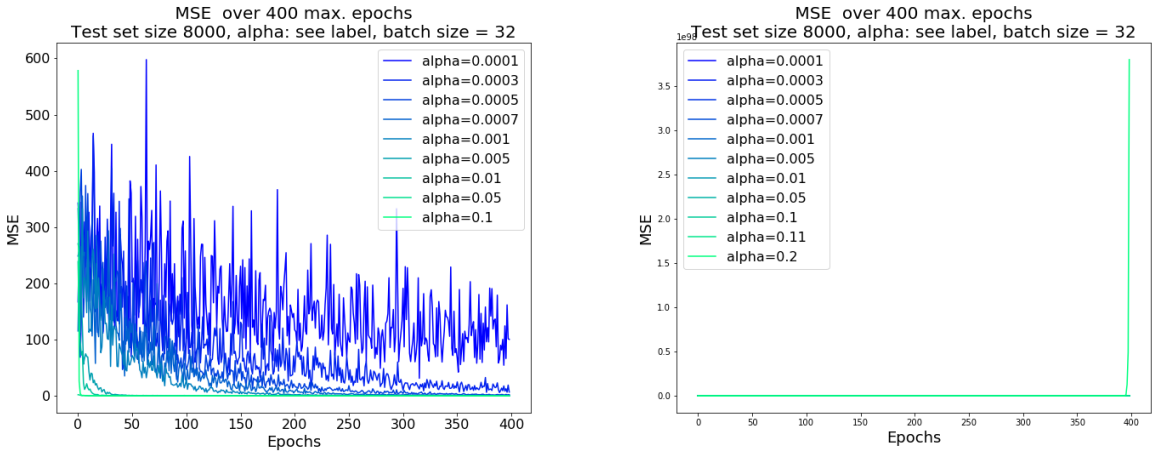



**Figure 5:** *Mean Squared Error (MSE) per epoch when minimizing the cost function (2-variable polynomial linear fit to Franke function) using the Stochastic Gradient Descent (SGD) method. The colored lines represent different values of the learning rate $\alpha$. (a) shows convergence for sufficiently small $\alpha$, (b) depicts divergence when $\alpha$ is too large.*

### 4.1.3 Batch size

Figure 6 shows how different mini batch sizes affect the performance of SGD when keeping the other hyper-parameters constant. To reiterate, a mini batch is a randomly chosen subset of the shuffled training data that is used for parameter optimization in each individual epoch, which helps to avoid minimizing the cost function to a local minimum as opposed to a desired global minimum. Note how all batch sizes eventually lead to a minimization of the cost function.

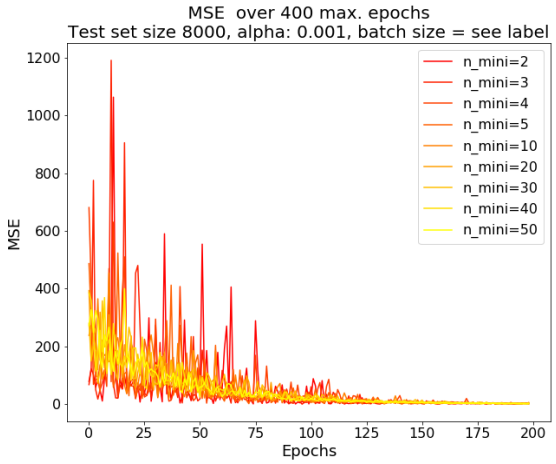



**Figure 6:** *Mean Squared Error (MSE) per epoch when minimizing the cost function (2-variable polynomial linear fit to Franke function) using the Stochastic Gradient Descent (SGD) method. The colored lines represent different values of the mini batch size (here: $n_{mini}$).*

The different shades of yellow and red represent sequentially changing mini batch sizes. It becomes clear that larger mini batch sizes result in relatively less oscillation of the MSE curve over time, which means that the results during early epochs can be regarded as more robust. On the other hand, the mini batch size directly influences the computation time of each epoch (less samples mean faster operations). Therefore, it is important to tune the parameter to get a good balance between robust results for low amounts of epochs, fast convergence, and fast computations within each epoch.

### 4.1.4 SGD for Ridge Regression

In this section we present how including a regularization term and the associated factor $\lambda$ affect the performance of SGD in Ridge regression. Figure 7 (a) shows the performance over time similar to the representations in the previous sections for standard values of the hyperparameters. Two main observations should be highlighted: firstly, the MSE approaches low values much faster compared to using SGD for OLS (Figure 4) using the same hyperparameter values. This could be an indication for faster performance when generalizing, i.e. penalizing over-fitting, the mini batch training data fit. Secondly and in accordance with this finding, the oscillation of the train data fit is substantially reduced.

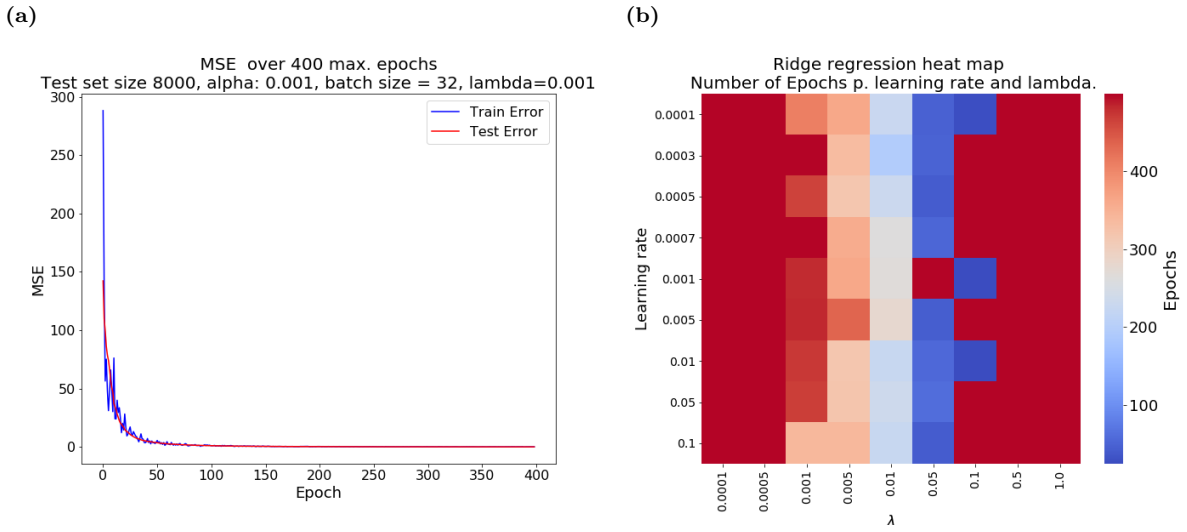



**Figure 7:** *Ridge regression by minimizing the cost function (2-variable polynomial linear fit to Franke function) using the Stochastic Gradient Descent (SGD) method. (a) shows MSE using standard parameter values. (b) shows epochs until reaching an arbitrary low threshold value as a function of varying $\lambda$ and learning rates $\alpha$.*

Figure 7 (b) illustrates how different values of $\lambda$ and $\alpha$ in concert affect the amount of epochs that were needed until SGD reaches an arbitrary low threshold value for the cost function. Firstly, we can conclude that the choice of $\lambda$ in this use case and given range had a more dominating effect on how fast SGD reached the low threshold value (never reached for both too low and too high lambdas with given max. epochs, but reached at least once for each value of $\alpha$). Secondly, a $\lambda$ of 0.01 seems to provide consistent, fast reduction regardless of the chosen learning rates. Even though $\lambda = 0.05$ overall led to faster run times, there is one combination ($\alpha = 0.001$) that failed to approach the minimum threshold completely, a trend that also

be observed for $\lambda = 0.1$. This could indicate a trade-off between computational speed and reliable model performance and one should ensure that SGD provides robust results for ridge regression with respect to these parameters.

## 4.2 MNIST

The accuracy for the different methods utilized on the MNIST dataset Dua and Graff (2017) are shown in Table 2. For the logistic regression, the learning rate was found to give best results with $\eta = 0.01$, 100 max epochs and batch size 32. For our Neural Network, the hyper-parameters were the same expect for an increased learning rate $\lambda = 0.05$. We also found it to work best with two hidden layers with firstly a ReLU followed by a leaky ReLU with $\alpha = 0.001$, of sizes 20 and 10 respectively. The table below utilizes the *Adam* optimizer (which is disussed more below) for both the logistic regression and the neural net.

**Table 2:** *Accuracy on training and testing set of the MNIST dataset.*

| | NN | Tensorflow NN | Log Reg | Sci-Kit Learn Log Reg |
|---|---|---|---|---|
| Train Accuracy | 1.000 | 1.000 | 0.930 | 0.999 |
| Test Accuracy | 0.961 | 0.950 | 0.930 | 0.964 |

Comparison between the different optimizers on the MNIST data set can be seen in table 3. For the neural network we can see that SGDM perform quite poorly (slightly better than a just random guessing), and SGD, RMSprop and Adam achieving the around the same test accuracy. The Adam optimizer gave the best test accuracy while also achieving 100% train accuracy. As for the speed of convergence, Figure 8 shows the cost function over each epoch of the training data. We can see that the Adam optimizer not only gives the highest accuracy, but also has the most effective decrease in the cost function.

In contrast to the neural network, the logistic regression seems to have much lower accuracies, with the SGD and the ADAM optimization methods performing best. The SGDM performs better for the logistic regression than the Neural network. The cost function per epoch performance of the logistic regression shown in Table 3 are shown in Figure 8, with the logistic regression greatly outperformed by the Neural Network.

The logistic regressions have a much smaller learning rate than that for the neural network, and even though numerical stability measures have been implemented in the logistic regression, we additionally implemented a high $l2$ penalty term $\lambda = 1$ to curb the beta matrix of growing too large and to prevent

**Table 3:** *Accuracy on training and test set, comparing Neural Network and Logistic Regression*

| | Neural Network | | Logistic Regression | |
|---|---|---|---|---|
| Optimizer | Train Accuracy | Test Accuracy | Train Accuracy | Test Accuracy |
| SGD | 0.994 | 0.939 | 0.915 | 0.915 |
| SGDM | 0.163 | 0.153 | 0.889 | 0.889 |
| RMSprop | 0.990 | 0.936 | 0.907 | 0.907 |
| Adam | 1.000 | 0.964 | 0.917 | 0.917 |

overflow.

An interesting consequence of the cross entropy cost function is that it does not penalize the final outputs that are wrong, but rather tweaks the the correct one to have a higher value. From the logistic regression on the MNIST dataset, the softmax output can have multiple outputs which are higher than 0.90, and we have therefore been interpreting the outputs with the argmax function from the numpy library.

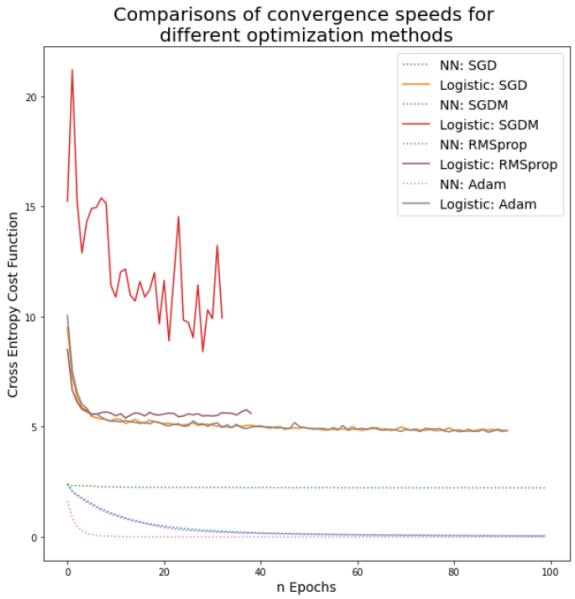



**Figure 8:** *Cost function after each epoch for the Neural Network (stripled line) and the logistic regression (solid line) on the MNIST dataset. Test and training accuracy is shown in 3.*

Regularization on the Neural Network is shown in Figure 9. With a low penalty, the training accuracy is

approximately 1 with a relatively large separation of testing and training accuracy. This could indicate that we are somewhat over-fitting the network, although the separation is very small in this case and the network seems well behaved. However, when increasing the lambda, we can see that the separation between testing and training accuracy decreases, removing the over-fitting as discussed in 2.4.3. And with furthermore heavy increase in the penalty we see that the accuracy drops significantly, as the weights have to become smaller to accommodate for the regularization.

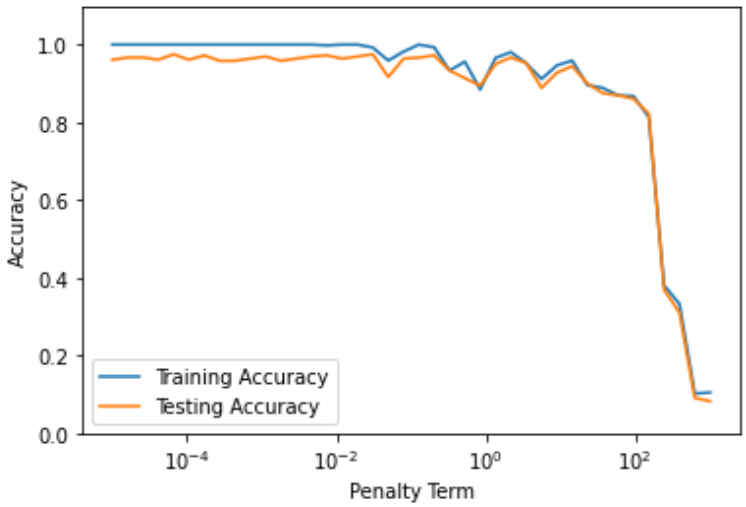
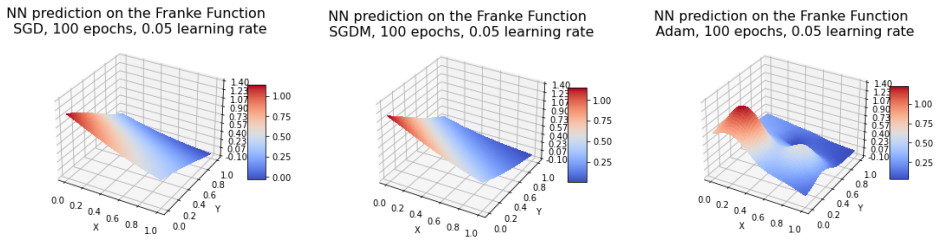



**Figure 9:** *Testing and training accuracy after 100 epochs for different l2 regularization penalties* $\lambda \in [10^{-5}, 10^{3}]$.

## 4.3 Regression

### 4.3.1 Franke Function

The neural network on the Franke Function gave greatly differing results regarding the optimization method utilized. Two layers with sigmoid functions and a final one with the identity function showed greatest dependence on the optimization method, where SGD and SGD with momentum gave a planar fit. The Adam optimization method proved to be the optimal method, capturing the shape of the Franke function as seen in Figure 10.

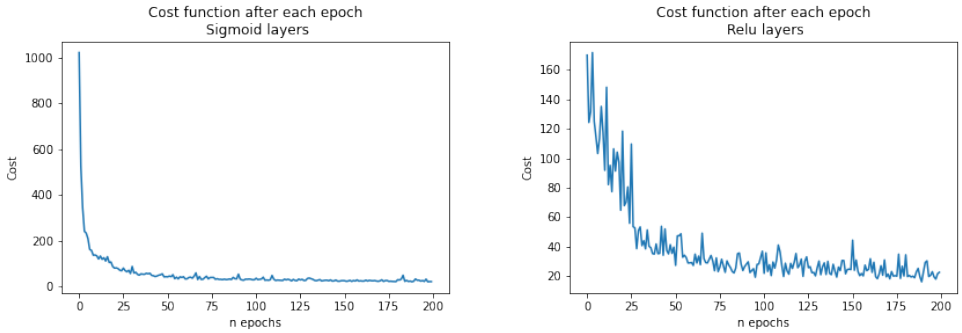



**(a)** ***SGD*** **(b)** ***SGDM*** **(c)** ***Adam***

**Figure 10:** *Neural Network prediction on the Franke Function with SGD, SGDM and Adam optimizers, all after 100 epochs and learning rate* $\eta = 0.05$

### 4.3.2 Terrain Data

The prediction of the terrain data by the Neural Network is shown in figure 11. Two identitcal networks with hidden layers of size $[100, 50]$, one with sigmoids as activation functions and the other with leaky ReLU and ReLU activation function. Both had the same learning rate $\eta = 0.01$ and Adam as the optimization method. Relu gave slightly higher MSE and R2 scores than sigmoid activation when the neural network predicted the terrain data. Scaling of the terrain data was an absolute necessity as the non scaled predictions had skyhigh R2 values and MSE and struggeled to learn.




**(a)** ***Sigmoid activation functions.*** **(b)** ***ReLU activation functions.***

**Figure 11:** *Cost function on the terrain data with different activation functions.*

Evaluating different activiation functions, the sigmoidal activation function proved slightly more accurate than using relu and leaky relus in combination, as seen table 4 In figure 11 we can see that, while both networks train with the same hyper-parameters and using Adam optimization, the sigmoids learn faster.

**Table 4:** *Regression metrics on the training data using in part only sigmoids and only ReLu and leaky ReLU as activation functions.*

| | Train MSE | Test MSE | Train R2 | Test R2 |
|---|---|---|---|---|
| Relu | 0.004 | 0.004 | 0.996 | 0.996 |
| Sigmoid | 0.006 | 0.006 | 0.994 | 0.994 |

# 5 Conclusion

Our findings show better model performance when classifying using neural networks compared to logistic regression. Adam optimizer shows great convergence at few epochs while simultaneously yielding the highest accuracies. We also generally showed how important critical evaluation of Gradient Descent methodology and chosen hyperparameters is, as illustrated for the Linear regression use cases.

Additionally, we have explored how to use regularization to reduce over fitting in classification, thereby decreasing the difference between train and test accuracies. We have also experienced that the relu functions, both standard and leaky relu, work better for regression, providing a lower cost function value faster than the traditional sigmoid function.

While our methods gave satisfactory accuracies, other sophisticated additions to the neural network architecture could be considered. Dropout layers, batch normalization, and feature engineering are topics which could improve not only this model, but also further implementations of Neural Networks (Goodfellow et al., 2016). For the logistic regression, comprehensive numerical stability and exploring more optimization methods are interesting avenues for further research.

# 6 Documentation

The figures and python scripts can be found in the GitHub of one of the group members:
https://github.com/haakools/FYS-STK4155AUTUMN/tree/main/Project%202
The main script can be found at:
https://github.com/haakools/FYS-STK4155AUTUMN/blob/main/Project%202/Project2_main.ipynb
The SGD scripts, main and library, can be found at:

https://github.com/haakools/FYS-STK4155AUTUMN/tree/main/Project%202/SGD

The function library can be found at:

https://github.com/haakools/FYS-STK4155AUTUMN/blob/main/Project%202/FunctionsLibrary.py

The test run for benchmarking:

https://github.com/haakools/FYS-STK4155AUTUMN/blob/main/Project%202/testrun.ipynb

And the figures can be found at:

https://github.com/haakools/FYS-STK4155AUTUMN/tree/main/Project%202/Images

# References


Ba, D. P. K. . J. (2014). Adam: A method for stochastic optimization. https://arxiv.org/abs/1412.6980.

Dua, D. and Graff, C. (2017). Optical recognition of handwritten digits data set, UCI machine learning repository.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

Hinton, G., S. N. . S. K. (2012). On the importance of initialization and momentum in deep learning. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

Hjorth-Jensen, M. (2020a). Fys-stk4155 lecture notes week 38. https://compphysics.github.io/MachineLearning/doc/pub/week38/html/week38.html.

Hjorth-Jensen, M. (2020b). Fys-stk4155 lecture notes week 39. https://compphysics.github.io/MachineLearning/doc/pub/week39/html/week39.html.

Hjorth-Jensen, M. (2020c). Fys-stk4155 lecture notes week 40. https://compphysics.github.io/MachineLearning/doc/pub/week40/html/week40.html.

Hjorth-Jensen, M. (2020d). Fys-stk4155 lecture notes week 41. https://compphysics.github.io/MachineLearning/doc/pub/week41/html/week41.html.

Marsland, S. (2009). *Machine Learning - An Algorithmic Perspective*. Chapman and Hall / CRC machine learning and pattern recognition series. CRC Press.

Murphy, K. P. (2013). *Machine learning : a probabilistic perspective*. MIT Press, Cambridge, Mass. [u.a.].

Nielsen, M. A. (2015). How the backpropagation algorithm works - neural networks and deep learning.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Sutskever, I., M. J. D. G. . H. G. (2013). On the importance of initialization and momentum in deep learning. http://www.cs.utoronto.ca/~ilya/pubs/2013/1051_2.pdf.