UNIVERSITETET I OSLO

Department of Physics

FYS-STK4155
Applied Data Analysis and Machine Learning

Project 2: Classification and Regression, From Linear and Logistic Regression to Neural Networks

Author:
Mathias Svoren & Jonas Semprini

Date: 19.11.2023

Table of Contents

List of Figures

List of Tables

1 Abstract

In this project, we developed a versatile Feed Forward Neural Network (FFNN) to rigorously assess regression and classification models. Our FFNN was tailored for various architectures, activation functions, and schedulers. For polynomial regression of Franke‘s Function, Adam and Adagrad outperformed others, achieving more stable and lower Mean Squared Errors (MSE). For the classification tasks of the neural network, Adam proved superior, achieving the most stable classification results and a test accuracy higher than logistic regression on average for the Wisconsin Breast Cancer Dataset. However, logistic regression almost met this accuracy at a much lower computational cost. Shifting to Franke Function regression, our neural network's MSE, though performing as well as a simple polynomial fit, prompts consideration of simpler alternatives with comparable or superior performance. We thence conclude the study emphasizes on practical trade-offs between neural network complexity and the effectiveness of simpler methods.

2 Introduction

This study aims to evaluate the efficiency of a feed-forward neural network (FFNN) implementation through addressing regression and classification problems. Specifically, we seek to establish the FFNN's comparative advantages over conventional linear and logistic regression approaches. The initial analysis focuses on Franke‘s function for regression, followed by a more practical examination of breast cancer data for classification.

Breast cancer is a significant global health concern for women. It is principle when addressing a tumour, to be able to distinguish it as either malignant (1) or benign (0). This specific problem is nothing more than a binary classification problem in which this project aims to employ a FFNN to study its effect on the Breast Cancer Wisconsin (Diagnostic) Data Set (WDBC). The entire data set is examined so that it is possible to study the performance of the FFNN with logistic regression.

Furthermore, the paper explores gradient descent optimization, comparing full batch and stochastic gradient descent, along with various scheduling methods (constant, adaptive, and momentum-enhanced) to reduce computational costs. We then examine supervised learning models employing gradient descent, including linear and logistic regression. Additionally, our research will encompass comparisons of activation functions and model architectures to understand their trade-offs. We aim to address questions as: the required model complexity for satisfactory results and the necessary trade-offs while considering the computational effort.

Finally, the neural network architecture, while conceptually simple, confronts a variety of challenges, notably the issues of gradients vanishing and exploding. Nevertheless, methods employed to both reduce and mitigate this will be handled. Moreover, the optimization of hyper-parameters remains a central challenge in achieving model fine-tuning, which will be facilitated through the model in accordance to the classification problem. Notably, our approach involves multiple FFNNs, accomplished through bootstrap iterations, a methodology elaborated upon in subsequent sections.

3 Theory

In this section there will be elaborate theory and clearer details on how neural networks are constructed and how they work. Further, there will also be clarifications regarding logistic regression and components employed in combination with the neural networks for these methods.

3.1 Descent Methods

3.1.1 Gradient Descent (GD)

The Gradient Descent (GD) algorithm is a fundamental optimization method employed to locate local extrema (either minima or maxima) of differentiable functions. Consider a vector-valued function $F(\boldsymbol{x})$ with $\boldsymbol{x} \in \mathbb{R}^n$. The gradient of F, denoted as $\nabla F(\boldsymbol{x}) = \boldsymbol{J}_F$, is represented as:

$$\begin{bmatrix} \frac{\partial F_1}{\partial x_1} & \cdots & \frac{\partial F_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_m}{\partial x_1} & \cdots & \frac{\partial F_m}{\partial x_n} \end{bmatrix}. \tag{1}$$

The gradient naturally describes how F changes with respect to $\boldsymbol{x}$, which is highly valuable when one wants to, for example, scrutinize where F decreases the most. The GD method operates on the principle of following the steepest descent in the direction of $-\nabla F(\boldsymbol{x})$. This requires to iteratively update the parameters in the direction opposite to the gradient, which, in the context of minimization, corresponds to the direction of the greatest decrease in the function F. In this manner, GD aims to reach a local minimum of the function, from a given starting point $\boldsymbol{x}_0$. It can thus be shown that

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \gamma \nabla F(\boldsymbol{x}_n), \quad n \geq 0$$

for $\gamma > 0$ (sufficiently small) [1] which implies that $F(\boldsymbol{x}_n) \leq F(\boldsymbol{x}_{n+1})$. Meaning we are always moving towards a possible minimum.

3.1.2 Stochastic Gradient Descent (SGD)

As opposed to GD, which utilizes the whole training set to locate extrema, stochastic gradient descent (SGD) computes the gradient over subsets named 'minibatches' (Géron 2019). The gradient is approximated by the sum of the data points in one of the minibatches randomly selected in each gradient descent step and this introduces stochasticity/randomness.

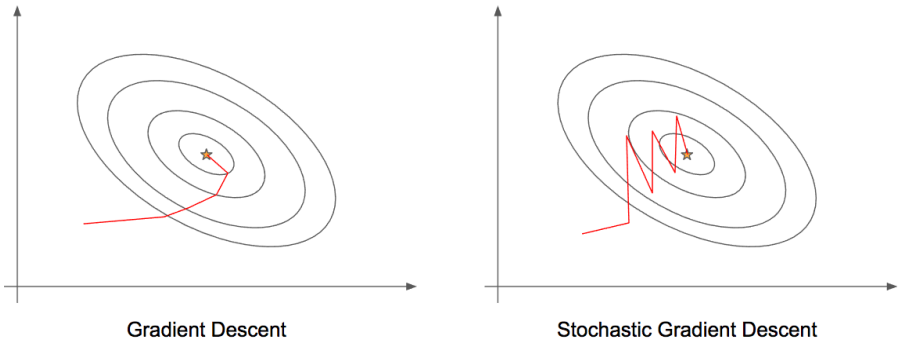


Figure 1: The GD method (left) follows the steepest descent in the direction of $-\nabla F(\boldsymbol{x})$, while each iteration with SGD (right) is much more stochastic.

By using SGD rather than GD, we achieve computational efficiency. The stochasticity allows the optimization scheme to escape local minima, making SGD better at finding global minima than GD.

[1] γ is often times denoted and referred to as the learning rate

3.2 Learning Schedules

The GD method is sensitive to the choice of learning rate, and it is thus important to find a sufficient one (Hjorth-Jensen 2023a). Too high of a γ may cause the training to diverge, and too low of a γ makes the convergence very time-consuming and computationally heavy. *Learning schedules* are strategies that allow us to exploit a variety of learning rates during training so that it is possible to find the most optimal one.

3.2.1 Momentum

Momentum is an extension to the gradient descent optimization algorithm, which incorporates inertia from past updates to navigate local minima and reduce noisy gradient oscillations. Analogous to physical momentum, it enables broader exploration through a variative landscape [2], preventing possible convergence in local minima (Lee et al. 2023). Including historical updates accelerates the optimization process. The hyperparameter, ranging from 0 to 1, determines the impact of previous updates, with 0 equivalent to standard gradient descent. A higher momentum value integrates a wider scope of past gradients.

A general outline of the algorithm can be presented as below:

Algorithm 1: Gradient Descent with Momentum (General idea)

Data: Initial parameters θ, learning rate α, momentum parameter β, number of iterations N, cost function $J(\theta)$
Result: Optimized parameters θ
Initialize velocity vector $\mathbf{v}$ with zeros;
Initialize iteration counter $i = 0$;
while $i < N$ **do**
  Compute gradient $\nabla J(\theta)$ using the current parameters θ;
  $\mathbf{v} \leftarrow \beta \cdot \mathbf{v} + (1 - \beta) \cdot \nabla J(\theta)$;
  `// Update parameters using the velocity`
  $\theta \leftarrow \theta - \alpha \cdot \mathbf{v}$;
  Increment iteration counter: $i \leftarrow i + 1$;
end

3.2.2 AdaGrad

The Adaptive Gradient Algorithm or AdaGrad for short, is a stochastic optimization method, that dynamically adjusts the learning rate for parameters by implementing smaller updates for frequently occurring features, and larger updates for infrequently occurring ones. The update rule modifies the overall learning rate for each parameter at each time step based on the historical gradients. The full algorithm with the update scheme can be expressed as follows (Algorithm 8.4 from Goodfellow et al. 2016).

[2] "landscape" is here referred to as search-landscape in machine learning terminology

Algorithm 2: The AdaGrad algorithm

Data: Global learning rate ϵ, decay rate ρ
Data: Initial parameters $\boldsymbol{\theta}$
Data: Small constant δ, somewhere in the region of $10^{-}7$ to account for numerical stability
Initialise gradient accumulation variable $\boldsymbol{r} = 0$;
while termination criterion not met **do**
  Sample a minibatch of m examples from the training set $\left\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\right\}$ with corresponding targets $\boldsymbol{y}^{(i)}$;
  Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L((f(\boldsymbol{x}^{(i)}; \theta),\ \boldsymbol{y}^{(i)})$;
  Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$;
  Compute parameter update: $\Delta\boldsymbol{\theta} - \frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise) ;
  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end

The issue with Adagrad however is that as the squared gradients keep adding up, the learning rate may become very small, hindering significant progress.

3.2.3 RMSprop

Root Mean Squared propagation or RMSprop, is an adaptive learning rate optimization algorithm, which extends the Adaptive Gradient Algorithm (Adagrad), efficiently reducing computational effort in neural network training. By exponentially decaying the learning rate when the squared gradient falls below a specified threshold, RMSProp adjusts the learning rate for each network parameter, outperforming standard Gradient Descent.(Algorithm 8.5 from Goodfellow et al. 2016)

Algorithm 3: The RMSprop algorithm

Data: Global learning rate ϵ, decay rate ρ
Data: Initial parameters $\boldsymbol{\theta}$
Data: Small constant δ, somewhere in the region of $10^{-}6$ to account for numerical stability
Initialise gradient accumulation variable $\boldsymbol{r} = 0$;
while termination criterion not met **do**
  Sample a minibatch of m examples from the training set $\left\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\right\}$ with corresponding targets $\boldsymbol{y}^{(i)}$;
  Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L((f(\boldsymbol{x}^{(i)}; \theta),\ \boldsymbol{y}^{(i)})$;
  Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho\boldsymbol{r} + (1-\rho)\boldsymbol{g} \odot \boldsymbol{g}$;
  Compute parameter update: $\Delta\boldsymbol{\theta} - \frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise) ;
  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end

3.2.4 ADAM

The Adam Optimizer or simply ADAM is a gradient descent optimization algorithm known for its efficiency in handling large datasets and numerous parameters, requiring minimal memory. Conceptually, it combines aspects of the 'gradient descent with momentum' and 'RMSprop' algorithms.

ADAM maintains an exponentially decaying average of past squared gradients and past gradients, akin to momentum, which can be pictured as adding friction to a rolling object down an incline plane.(Algorithm 8.7 from Goodfellow et al. 2016)

Algorithm 4: The ADAM optimizer/algorithm

Data: Step size $\boldsymbol{\epsilon}$ (Suggested default: 0.001

Data: Exponential decay rates for moment estimates, $\rho_1, \rho_2 \in [0, 1)$ (Suggested defaults: 0.9 and 0.999 respectively)

Data: Small constant δ, somewhere in the region of 10^-8 to account for numerical stability

Data: Initial parameters $\boldsymbol{\theta}$

Initialize 1.st and 2.nd moment variables $\boldsymbol{s} = \boldsymbol{r} = 0$;

Initialize time step $t = 0$;

while termination criterion not met **do**

Sample a minibatch of m examples from the training set $\left\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\right\}$ with corresponding targets $\boldsymbol{y}^{(i)}$;

$t \leftarrow t + 1$;

Update biased first moment estimate: $\boldsymbol{s} \leftarrow \boldsymbol{s}\rho_1 + (1 - \rho_1)\boldsymbol{g}$;

Update biased second moment estimate: $\boldsymbol{r} \leftarrow \boldsymbol{r}\rho_2 + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$;

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L((f(\boldsymbol{x}^{(i)}; \theta),\ \boldsymbol{y}^{(i)})$;

Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\rho_1^t}$;

Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2^t}$;

Compute update: $\Delta\boldsymbol{\theta} = -\boldsymbol{\epsilon}\frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}}+\delta}$ (operations applied element-wise) ;

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end

3.3 Feed Forward Neural Network (FFNN)

3.3.1 Overview

A neural network, is a foundational element in artificial intelligence, which emulates human brain functions to process data. Operating as a variety of machine learning known as deep learning, neural networks utilize layered structures of interconnected nodes, resembling the organization of the human brain. This configuration establishes an adaptive system, enabling computers to learn from errors and progressively enhance their performance through the given models. In such manner they can be employed to solve various tasks from facial recognition to regression analysis and classification problems, which is what this study is chiefly regarding.

3.3.2 General setup and Layers

The first model of artificial neurons were introduced in 1943 to investigate brain signal processing (Hjorth-Jensen 2023b). The concept emulates the neural networks in the human brain, where billions of neurons communicate via electrical signals. Each neuron accumulates incoming signals, requiring them to surpass an activation threshold for output. Failure to meet the threshold results in neuron inactivity, signifying zero output. This behavior has motivated a basic mathematical model for an artificial neuron, which could be presented as follows

$$y = f\left(\sum_{i=0}^{n} w_i x_i\right)$$

where y is the output of that neuron, with corresponding signals $x_0, x_1, \ldots, x_n$ and weights $w_0, w_1 \ldots, w_n$. The function f is often defined as the activation function or threshold function.

Now extending the concept of a neuron, we can look at multiple neurons interacting through layers, as illustrated in Figure 2.

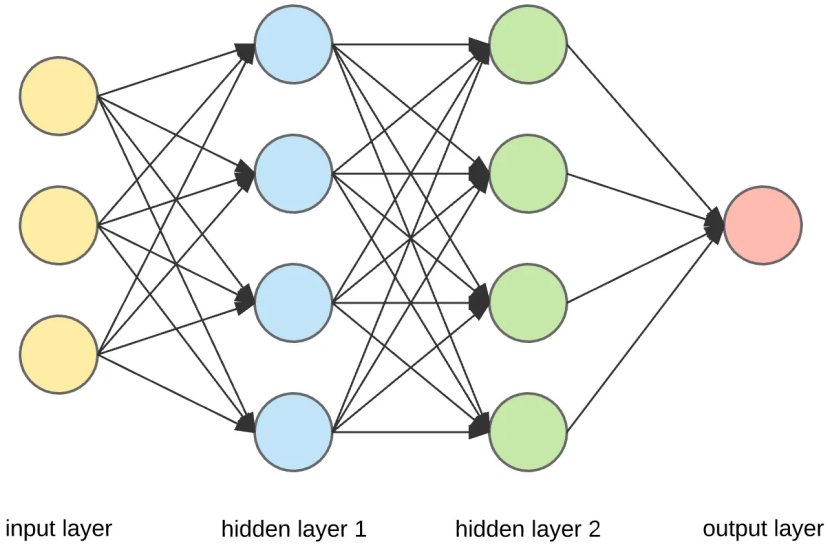


Figure 2: A visual of how a simple Artifical Neural Network looks in concept, with two hidden layers.

For each node i in the inital hidden layer, we compute the weighted sum z_i^1 of the input coordinates x_j, such that

$$z_i^1 = \sum_{j=1}^{M} w_{i,j}^1 x_j + b_j^1$$

where b_i is known as the bias which typically is required when activation weights or inputs are zero. The value z_i^1 serves as the argument for the activation function f_i of each node i. M represents all potential inputs to a node i in the first layer. The collective output y_i^1 of all neurons in layer 1 is then defined as:

$$y_i^1 = f(z_i^1) = f\left(\sum_{j=1}^{M} w_{i,j}^1 x_j + b_j^1\right)$$

where we have assumed identical activation functions for all nodes within the same layer, denoted as f. If on the other hand there happens to be different or several activation functions in different hidden layers then we can identify the output as:

$$y_i^l = f^l(u_i^l) = f^l\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l y_j^{l-1} + b_i^l\right)$$

where N_l is the node count in layer l. After computing the initial hidden layer nodes' output, the subsequent values of later layers are determined iteratively until obtaining the final output.

Considering this, we can further extend the network by examining the output of neuron i in layer 2, described as follows:

$$\begin{aligned} y_i^2 &= f^2\left(\sum_{j=1}^{N} w_{ij}^2 y_j^1 + b_i^2\right) \\ &= f^2\left[\sum_{j=1}^{N} w_{ij}^2 f^1\left(\sum_{k=1}^{M} w_{jk}^1 x_k + b_j^1\right) + b_i^2\right] \end{aligned}$$

Generalizing this to an Artifical Neural Network (ANN) with l hidden layers, then yields the comprehensive functional form

$$y_i^{l+1} = f^{l+1}\left[\sum_{j=1}^{N_l} w_{ij}^3 f^l\left(\sum_{k=1}^{N_{l-1}} w_{jk}^{l-1}\left(\dots f^1\left(\sum_{n=1}^{N_0} w_{mn}^1 x_n + b_m^1\right)\dots\right) + b_k^2\right) + b_1^3\right]. \tag{2}$$

The functional form 2 encompasses what is known as a Feed-Forward Neural Network (FFNN), which is a type of neural network that considers uni-directional flow of information [3], but as we will observe shortly, can be trained in the backwards direction too. Additionally it is worth noting that in this specific project we are constructing the FFNN to be fully-connected [4] which means that one can, if desirable, label it as a Multi-Layer Perceptron (MLP), but we will refrain from that throughout and use FFNN.

3.3.3 Backpropagation

Artificial neural networks undergo continuous learning through corrective feedback loops to enhance its predictive analysis. The data flows from input to output nodes along multiple pathways, with only one representing the correct mapping from input to output. To identify this path, the neural network employs a feedback loop, involving the following steps:

1. Each node generates a prediction for the subsequent node in the path
2. It verifies the accuracy of the prediction, assigning higher weights to paths with correct guesses and lower weights to those with incorrect ones.
3. For the next data point, nodes make new predictions using the weighted paths and repeat the process from Step 1. This iterative approach allows the network to progressively refine its predictions.

In more mathematical terms the algorithm can be divided into three equations. One for acquiring the change in cost concerning weights, one for changes in biases, and one for obtaining changes concerning the next layer. This is essentially the application of the chain rule. In matrix-vector form, the equations are as follows.

$$\frac{\partial \mathcal{C}}{\partial \boldsymbol{W}} = \boldsymbol{X}^{\boldsymbol{T}} \cdot \frac{\partial \mathcal{C}}{\partial \boldsymbol{Y}} + \lambda \boldsymbol{W} \tag{3}$$

$$\frac{\partial \mathcal{C}}{\partial \boldsymbol{B}} = \frac{\partial \mathcal{C}}{\partial \boldsymbol{Y}} \tag{4}$$

$$\frac{\partial \mathcal{C}}{\partial \boldsymbol{X}_1} = \frac{\partial \mathcal{C}}{\partial \boldsymbol{Y}} \odot a'(\boldsymbol{X}) \quad \& \quad \frac{\partial \mathcal{C}}{\partial \boldsymbol{X}_2} = \boldsymbol{W}^{\boldsymbol{T}} \cdot \frac{\partial \mathcal{C}}{\partial \boldsymbol{Y}} \tag{5}$$

$$\frac{\partial \mathcal{C}}{\partial \boldsymbol{X}} = \frac{\partial \mathcal{C}}{\partial \boldsymbol{X}_1} \cdot \frac{\partial \mathcal{C}}{\partial \boldsymbol{X}_2} \tag{6}$$

Here $\odot$ denotes the Hadamard product, $\mathcal{C}$ is the cost, $\boldsymbol{X}$ is the input (or the output backwards direction), $\boldsymbol{Y}$ is the output (or the input if going backwards), $\boldsymbol{W}$ are the weights, $\boldsymbol{B}$ are the biases, and $a'(\boldsymbol{X})$ is the derivative of the activation with respect to the input (or the output if one is going backwards). There are four equations due to our code structure, where a hidden layer is split into an application layer and an activation layer.

3.3.4 Activation and Initialisation

Activation functions, crucial in the theoretical framework, primarily influence hidden layers in an artificial neural network (ANN). The behavior of the ANN is chiefly shaped by the number of

[3] Generally inferring that the internal information flows in a single direction (forward)
[4] Meaning that each node in a given layer i is connected to all nodes in a subsequent layer j

hidden layers and their respective activation functions. In the output layer, activation selection depends on the problem type (Linear for regression, Sigmoid for binary classification, Softmax for multi-class classification). However, determining activation functions for hidden layers is nuanced, often relying on trial and error due to a lack of comprehensive theoretical guidance (Goodfellow et al. 2016, p.188). The chosen activation functions for this study will be outlined below.

Linear Activation

Linear activation, or the identity function, ensures that the resulting activation remains directly proportional to the input. Mathematically, it is defined as a straightforward linear function, essentially preserving the original input value without introducing any transformation to the weighted sums.

$$f(x) = x, \quad x \in \mathbb{R} \tag{7}$$

However, a linear activation function has two major problems:

1. Backpropagation will have no effect as the derivative of the function is a constant and has no relation to the input x.
2. All layers of the neural network will collapse into one if a linear activation function is used. No matter the number of layers in the neural network, the last layer will still be a linear function of the first layer. So, essentially, a linear activation function turns the neural network into just one layer.

Sigmoid

The Sigmoid activation is mathematically defined as

$$f(x) = \frac{1}{1 + e^{-x}}, \quad x \in \mathbb{R} \tag{8}$$

where the output $f(x) \in (0, 1)$. For especially high values of x, the rate of change is minimal, leading to challenges such as vanishing gradients. This issue can hinder weight updates, potentially halting or fully stagnating the training process. Due to this saturation, the sigmoid is generally avoided as an activation function for hidden layers (Goodfellow et al. 2016, p.191).

Hyperbolic Tangent

The Tanh function, much like the sigmoid activation, presents an S-shaped curve but with an extended output range from -1 to 1. This means that as the input becomes larger (more positive), the output approaches 1.0, and conversely, for smaller inputs (more negative), the output tends towards -1.0. The Tanh function is particularly useful in scenarios where the centered output range around zero is beneficial, offering improved convergence during the training of neural networks. Mathematically it is represented as:

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}, \quad x \in \mathbb{R} \tag{9}$$

ReLU and LReLU

The ReLU function, denoted by

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}, \quad x \in \mathbb{R} \tag{10}$$

is a linear unit. Its computational efficiency comes from its simple gradient calculation. ReLU avoids saturation for positive values, making it a favorable choice in deep neural networks. Despite that, an apparent challenge with ReLU is the issue of "dying ReLUs": during training, if a neuron's weights get updated such that the weighted sum of the neuron's inputs is negative, the output is 0, and the neuron may stay at 0 since the gradient of the ReLU function is 0 when its input is negative (Géron 2019).

The Leaky ReLU is a modified version of ReLU designed to address zero gradient issues and the "dying ReLU" problem by incorporating a small slope in the negative part. It is defined by the function

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{otherwise} \end{cases}, \quad x \in \mathbb{R} \tag{11}$$

where α is a nonnegative parameter close to zero, preventing zero gradients (Efron and Hastie 2016). Both ReLU and Leaky ReLU are non-differentiable at $x = 0$, but assigning a derivative value at 0, typically 1 or 0, is practically feasible.

ELU

The Exponential Linear Unit (ELU) is an activation function that facilitates quicker convergence to minimal (or zero) cost with sufficient accuracy. Similar to LReLU, ELU incorporates an additional positive constant, α. ELU shares similarities with RELU for non-negative inputs, both adopting an identity function form. However, ELU gradually becomes smooth as its output approaches $-\alpha$, contrasting with the abrupt smoothing of RELU. In mathematical terms it can be defined as

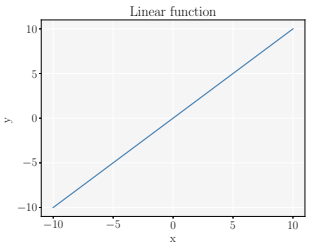
$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{otherwise} \end{cases}, \quad x \in \mathbb{R} \tag{12}$$

Softmax

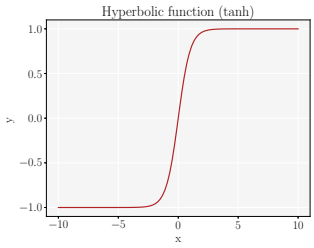
The softmax function is prevalent in data science, particularly in machine learning classification tasks. In these scenarios, a machine learning model generates a vector of real-valued scores for each class. The softmax function transforms this vector into a normalized vector, representing probabilities associated with each class. These probabilities indicate the likelihood of a specific class. Mathematically it has the expression of

$$f(\boldsymbol{x}) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}, \quad \boldsymbol{x} \in \mathbb{R}^n. \tag{13}$$

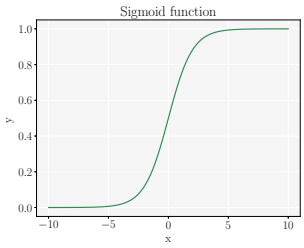
With the defined activation functions thoroughly explained, a summary visualizing their functional behavior is presented below (ref Figure 9 and 4).



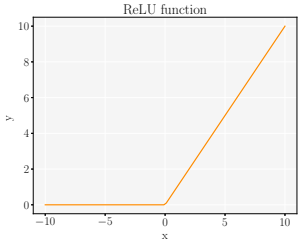
(a) Linear activation function



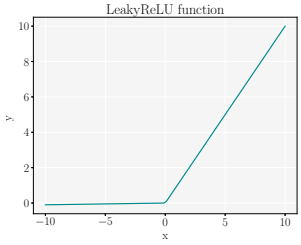
(b) Tanh activation function



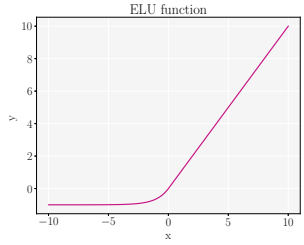
(c) Sigmoid activation function



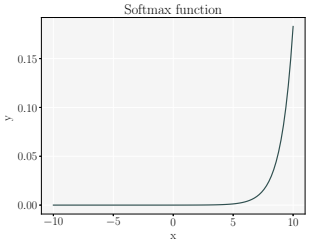
(d) ReLU activation function



(e) LeakyReLU activation function.

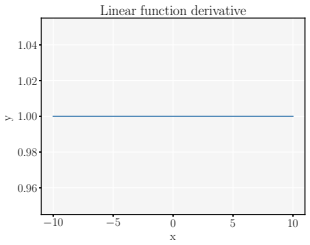


(f) ELU activation function.

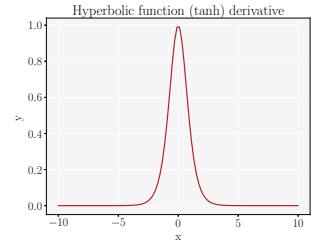


(g) Softmax activation function.

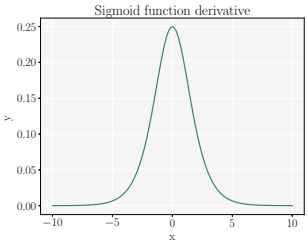
Figure 3: Structured subplot to showcase the behaviour of the different activation functions.



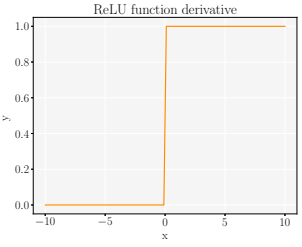
(a) Linear activation derivative



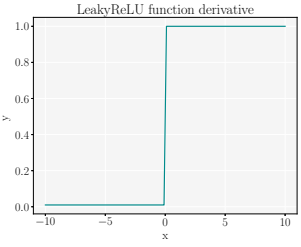
(b) Tanh activation derivative



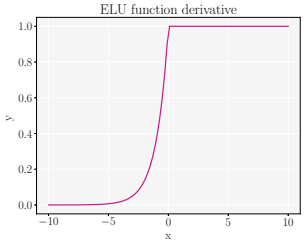
(c) Sigmoid activation derivative



(d) ReLU activation derivative



(e) LeakyReLU activation derivative



(f) ELU activation function derivative

Figure 4: Structured subplot of the derivatives to the respective activation functions

3.4 Training and Error measurement

In neural networks, assessing errors during training is vital for accurate regression and binary classification. This process enables adjustments to the model, improving its performance and making it more effective for real-world applications. In this section there will be appropriate explanations and clarifications to the cost functions chosen to measure the performance of our models.

3.4.1 Cost Functions

MSE

To train the network, an essential step involves establishing a performance metric. In regression, the mean sum of squares of errors (MSE) is employed, while binary classification utilizes Binary Cross-Entropy (BCE). The MSE cost function is initially defined as follows:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(y_{d_i} - \hat{y}_{NN_i}(\boldsymbol{\theta})\right)^2 := \mathcal{C}(\boldsymbol{\theta}). \tag{14}$$

where y_{d_i} is the i-th data point, $\hat{y}_{NN_i}$ is the network's prediction for this point, and $\boldsymbol{\theta}$ represents the weights and biases of the network. Minimizing the MSE for model training can be accomplished using solvers based on gradient descent methods, which we covered in Section 3.1.

Referring to prior work by the authors (Svoren et al. 2023) , the derivative of the MSE cost function $\mathcal{C}(\boldsymbol{\theta})$ can be succinctly expressed in matrix notation:

$$\mathcal{C}(\boldsymbol{\theta}) = \frac{1}{n}\|X\boldsymbol{\theta} - y\|_2^2 \tag{15}$$

$$\frac{\partial \mathcal{C}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = -\frac{2}{n}X^T y + 2X^T X\boldsymbol{\theta} \tag{16}$$

Binary Cross-Entropy (BCE)

MSE is unsuitable for classification problems due to it taking values on the real number line, contrasting with the binary case that adheres to a Bernoulli distribution, and not Gaussian since it only produces outputs of either 0 or 1. Additionally, while MSE is convex for OLS, its restriction by the sigmoid function in classification makes it non-convex, hindering convergence to global minima in gradient descent. Conversely, cross entropy is convex, rendering it a preferable choice (Goodfellow et al. 2016). This project exclusively adopts the binary case with the binary cross entropy cost function (BCE).

BCE measures the difference between predicted probability (p) and the true value ($y = 0$ or $y = 1$). To establish intuition and identify terms defining the Binary Cross Entropy (BCE) function, consider the case where $y = 1$. Define the loss function $L(y, p)$ as

$$L(y, p) = -y\log(p) = -\log(p) \tag{17}$$

The logarithm implies $L \to 0$ as $p \to 1$ and $L \to \infty$ as $p \to 0$. This results in a substantial penalty for incorrect predictions, and a small loss value for probabilities close to the correct $y = 1$, aligning with the desired behavior. Assuming $y = 0$, shifting the value by 1 and utilizing the same equation as above with substitutions for $1 - y$ and $1 - p$ yields:

$$L(y, p) = -(1 - y)\log(1 - p) = -\log(1 - p). \tag{18}$$

As these terms cannot coexist under the binary condition $y = 0$ or $y = 1$, their summation yields:

$$L(y, p) = -y\log(p) - (1 - y)\log(1 - p). \tag{19}$$

Currently, we've examined the loss for a single prediction. By aggregating n loss values through element-wise products, our Binary Cross Entropy cost function is then formulated as:

$$\mathcal{C}(y,p) = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(p_i) + (1-y_i)\log(1-p_i)]. \tag{20}$$

When we hence differentiate it with respect to p we obtain

$$\frac{\partial L(y,p)}{\partial p} = -\frac{y}{p} + \frac{1-y}{1-p}. \tag{21}$$

For the binary classification classifier, the output layer employs the sigmoid activation to determine the probability $p = \sigma(z)$. The derivative of the sigmoid function is obtained through the application of the chain rule and quotient rule of derivation:

$$\sigma'(z) = \sigma(z)(1-\sigma(z)) \tag{22}$$

$$= p(1-p) \tag{23}$$

$$= \frac{\partial p}{\partial z} \tag{24}$$

By the chain rule, we can combine these two derivatives

$$\frac{\partial L(y,p)}{\partial z} = \frac{\partial L(y,p)}{\partial p}\frac{\partial p}{\partial z} = \left(\frac{-y}{p} + \frac{1-y}{1-p}\right)(p(1-p)) \tag{25}$$

$$= -y(1-p) + (1-y)p \tag{26}$$

$$= p - y \tag{27}$$

Which naturally for the cost function yields

$$\frac{\partial \mathcal{C}(\boldsymbol{y},\boldsymbol{p})}{\partial z} = \boldsymbol{p} - \boldsymbol{y} \tag{28}$$

in vector-matrix notation.

3.5 Logistic Regression

Logistic regression is a supervised machine learning model, which is primarily applied in classification tasks, predicting the probability of an instance belonging to a specific class. It utilizes the sigmoid function to estimate this probability based on the output of a linear regression function. Unlike linear regression, which produces a continuous output, logistic regression focuses on predicting the probability of class membership.

In classification, categories can vary, but we will focus on the simplest case of binary categories, such as true or false, familiar to outcomes from a coin flip or filtering emails as either spam or not. In our scenario, the dependent variables, denoted as responses or outcomes (y_i), are discrete, taking values from $k = 0, 1$. The objective is to predict output classes from the design matrix $\boldsymbol{X} \in \mathbb{R}^{n\times p}$, composed of n samples, each featuring p predictors. The primary aim is to establish the classes to which new, unseen samples belong. For binary classes this can be denoted as $y_i = 0$ and $y_i = 1$.

In this respect, when presented with binary categories, it would be favorable to choose a "soft" classifier when determining the membership of respective samples. Soft classifiers have the ability to perform estimates of conditional probabilities relating to respective categories and hence classify the samples based on the estimates. Logistic regression, the most prominent example of a soft classifier, calculates the probability that a data point x_i belongs to a category (i.e $y_i = \{0, 1\}$) using the logit function (or Sigmoid, from equation 8). This function serves to represent the likelihood of a specific event.

4 Method

After establishing a comprehensive theoretical framework, we transition to implementation, detailing the construction and development of our models, along with the corresponding management of according data-sets.

4.1 Data

4.1.1 Splitting the data

The first step we need to take before doing any analysis is to randomly split our data into two mutually exclusive sets. One training set and one testing set. We split our data such that 80% of the data is in the training set and use the remanding 20% for testing the performance of our models. To being able to compare our results we use a set seed so that our data is split the same way each time. Splitting the data provides an unbiased estimate for the performance of the model.

4.1.2 Polynomial

In testing both gradient descent models and our FFNN implementation, a simple one-dimensional polynomial of order 3, defined as $f(x) = 1 + x + 3x^2 - 4x^3$ (Equation 29), was utilized. We generated 100 linearly spaced x-values within the interval $(-1, 1)$. The attempt was made to fit a 2nd order design matrix with an intercept, resulting in a design matrix with 3 features of size $(100, 3)$. Essentially, we aimed to fit a 2nd order polynomial to our initial 3rd order polynomial.

$$f(x) = 1 + x + 3x^2 - 4x^3 \tag{29}$$

4.1.3 Franke's function

In Project 1, it was determined that using a 7th degree design matrix of the x and y values for Franke's function yielded optimal results. While uncertain if this holds true for our GD and FFNN models, we opted for the same size design matrix to compare.

Hence, the data to be fitted will constitute a 7th degree design matrix of x and y values. The x and y data are generated by uniformly spacing values between 0 and 1 with a step size of 0.05. Noise is introduced to the data, with a normal distribution having $\mu = 0$ and $\sigma^2 = 0.1$. Consequently, we obtain a design matrix of size $(400, 36)$. After data splitting, a training set with 320 elements and a test set with 80 elements are produced, replicating the exact data used for Franke's function in Project 1 (Svoren et al. 2023). This facilitates a meaningful comparison between our models.

4.1.4 Logic gates

We used logic gates for a general testing of our classification implementation of the FFNN. We implemented a test for classifying 4 sets of two numbers based on OR, XOR and AND gates.

$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \qquad yOR = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} \qquad yXOR = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \qquad yAND = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

4.1.5 Wisconsin Breast Cancer data set (WBC)

The Wisconsin Breast Cancer dataset, comprising 569 samples with 30 features each, is obtained from `scikit-learn`'s library. The binary output indicates the presence (1) or absence (0) of cancer for each patient.

During model testing, significant improvement in prediction performance was observed with data scaling. We employed scikit-learn's `StandardScaler` class, which standardizes the data by subtracting the mean and dividing by the standard deviation of the training set for both the training and test sets.

4.2 Implementation

4.2.1 Descent methods with schedulers

To establish a robust framework for our FFNN, we began constructing our neural network from the ground up, starting with fundamental principles and gradually adding features, which meant developing our own gradient descent (GD) method with a fixed learning rate (LR). Subsequently, we incorporated momentum and assessed convergence in comparison with a fixed LR. Further enhancements included the integration of adaptive time-based decay for the LR, and once again, convergence was evaluated.

We hence implemented Stochastic Gradient Descent (SGD) with specified mini-batch size and epochs, testing it with and without momentum, and using both fixed and adaptive learning rates. Additionally, we implemented the Adagrad method for learning rate tuning, considering variations with and without momentum for both plain GD and SGD. Accordingly, RMSprop and Adam were incorporated into our library of methods for further learning rate adjustment and moment addition.

4.2.2 The Feed-Forward Neural Network

We implement the FFNN as a class using NumPy's various matrix functions, allowing specification of an arbitrary architecture, activation functions, and cost function. The model can be utilized for regression, binary classification, linear, and logistic regression by adjusting the architecture. The instantiation sets weights with random floating point numbers sampled from a univariate normal distribution of mean 0 and variance 1, while bias is set to a small number of 0.01, following (Goodfellow et al. 2016, 189). This unified implementation allows for direct metric comparison across different models.

We then implemented a forward method that feeds the input through the network by applying weights and activation functions on the inputs for each layer of the network. For our model to be able to train and update the weights of the network, we implemented a backpropagation method traversing the network in reverse by applying the gradient of the activation function. (Theory 3.3.3)

Model training involves repeated feed-forward and backpropagation calls per batch per epoch. The choice of scheduler can be specified during training, with schedulers implemented as classes following the reasoning in our theoretical framework and [Hjorth-Jensen 2023a]. To find optimal parameters, we utilize a custom grid search, exploring combinations of λ, γ (learning rate), and scheduling parameters to minimize error.

4.3 Binary classification

4.3.1 Logistic regression

The logistic regression model we implement on the same structure as our SGD. The difference is that we calculate the gradient using the gradient of the sigmoid function. So for each batch the gradient is given by

$$\nabla = -X_{\text{batch}}^T(y_{\text{batch}} - \sigma(X_{\text{batch}}\beta)) \tag{30}$$

Here, X_{batch} represents the input features for the current batch, y_{batch} is the corresponding target values and σ denotes the sigmoid function. The sigmoid function is crucial in logistic regression as it transforms the output into a probability distribution between 0 and 1. The model assigns the probabilities of $X\beta$ above 0.5 to 1, and those below to 0. This results in an output of classified predictions.

4.3.2 Classification through the FFNN

To employ our FFNN as a classifier, the output layer was modified to utilize the sigmoid activation function, and the Binary Cross Entropy was adopted as the cost function, with its corresponding derivative detailed in the theory section. We then included a new classify method which assigns the probabilities above 0.5 to 1 and those below to 0.

5 Results and Discussion

The results section is split into two segments. Initially, the findings from the regression portion of the project are presented, succeeded by the outcomes from the classification segment. There will additionally be results of thorough grid searches, from the two sections respectively, presented to sufficiently substantiate our values.

5.1 Regression

5.1.1 Grid-search (Descent Methods)

We initally conducted a grid-search for the gradient and stochastic descent methods on Franke's Function, considering two features without an intercept. The MSE scores obtained with all schedulers were satisfactory (though some for especially specific parameters), but Adam emerged as the optimal scheduler, demonstrating consistent and non-varying values across different regularization and learning rate parameters. Adagrad exhibited comparable MSE values, although with slightly more variability.(Figure 13 and Figure 15)

We thus proceeded to select Adam as our scheduler while grid searching stochastic gradient descent for the optimal epochs to batch size ratio.

5.1.2 Grid-search (FFNN)

The grid search for our FFNN aimed to identify optimal configurations, considering the number of hidden layers, nodes, and activation functions appropriate to effective performance in regression tasks. Building on the insights gained from our previous grid search on descent methods, we conducted a subsequent grid search for the FFNN using Franke's Function with 36 features, excluding an intercept. The results revealed fair MSE scores across all activation functions, with Sigmoid and LeakyReLU standing out as the most effective and proficient, providing accurate fits and maintaining stability throughout all epochs of training.(Figure 14)

In an extended grid search analysis (Figure 14a), optimal performance for fitting Franke's function data was observed with 3 layers and 100 hidden nodes in this specific case. A general trend suggests that the optimal number of layers ranges from 2 to 3, while the number of hidden nodes is optimal between 10 and 50. Notably, selecting more layers and fewer nodes can yield a good fit with lower computational cost. Conversely, choosing too many nodes and layers tends to lead to overfitting of the training data.

Results from gridsearch of the WBC-classification shows that this problem is best predicted with a model containing only 1 hidden layer, and is good for anything between 10 and 50 hidden nodes. (Figure 16)

5.1.3 Descent Methods

In our regression methods when applying gradient descent, the Ordinary Least Squares (OLS) cost function showed greater performance. As illustrated in Figure 5, OLS outperformed the Ridge cost function throughout 2000 epochs. Notably, the Ridge cost function with a very low λ value of 1×10^{-8} exhibited better performance than when $\lambda = 10^{-4}$. As λ approaches very low values, the Ridge cost function tends to behave similarly to OLS. However, for this complex problem, OLS exhibits signs of converging to local optima, posing issues regarding stability of our models. Hence, we conclude that, for both regression fitting methods, Ridge with a small λ value is the preferred choice.

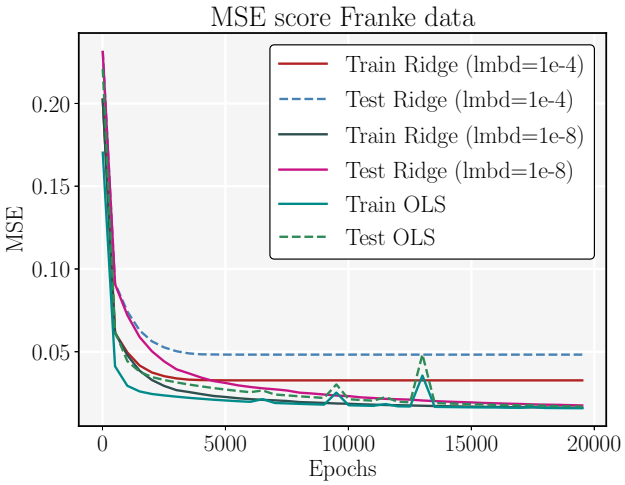


Figure 5: **Comparing OLS and Ridge.** Gradient Descent on the design matrix for Franke's function with 36 features. Learning rate of 0.1 using the Adam optimizer.

Polynomial:

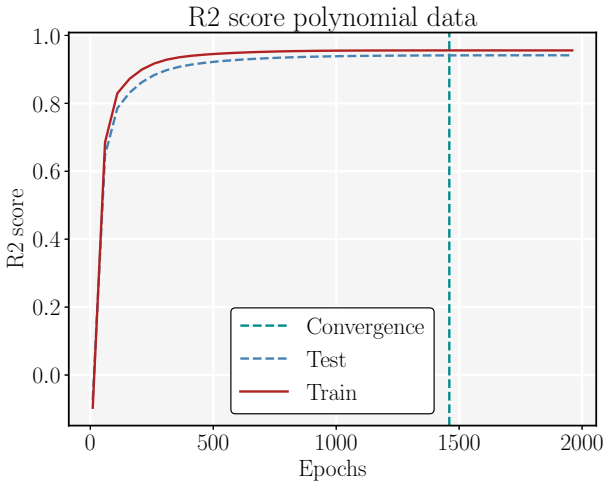
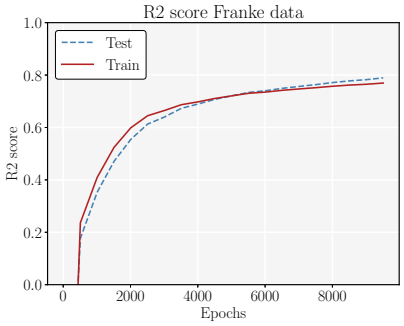


Figure 6: **R2 score for GD.** R2 score doing Gradient Descent for fitting a 2nd order polynomial to the polynomial data. The vertical "Convergence" line shows where the absolute difference of two subsequent R2-scores are less than 10^{-4}.

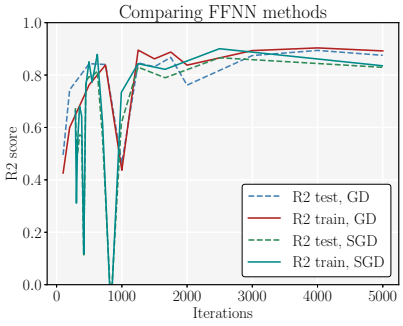
Figure 6 shows that the specific Gradient Descent model reaches a convergence point around 1500 epochs.

We refrain from comparing Stochastic Gradient Descent to Gradient Descent in this scenario. The ridge cost function is convex in one dimension, suggesting that we need several iterations before the model reaches convergence in this specific problem.

5.1.4 Results on Franke's Function



(a) **R2 score for GD.** For design matrix of franke data with 36 features, i.e. 7th order design matrix. Using Adam optimizer and a learning rate of 0.01.



(b) **R2 score for the FFNN** Comparing the FFNN with GD and SGD. The SGD neural network uses a batchsize of 20. The plot compares number of iterations. For SGD we multiply batches with epochs. Both models use Adam optimizer, the ELU activation function, learning rate of 0.01 and regularization parameter of 0.01.

Figure 7: **R2 score comparison**

Figure 7b indicates that, for the regression problem, utilizing Gradient Descent (GD) is the optimal choice. We actually found that GD performed better than SGD. The divergence in performance could be due to a suboptimal SGD implementation. In theory, its reduced computations should lead to quicker convergence. However, for convex regression problems, where SGD may be unnecessary, longer convergence times could result.

	GD	FFNN
Cost function	OLS	MSE
Learning rate	0.01	0.01
Optimizer	Adam	Adam
Epochs	10 000	1 000
Activation function	–	Sigmoid
Regularization	–	0.01
Layers	–	3
Hidden nodes	–	100

Table 1: **Best model parameters.**

	GD	FFNN
MSE	0.01151	0.0096

Table 2: **Best MSE.** For same amount of epochs.

Showing the resulting fit on the test set for our best implementations.

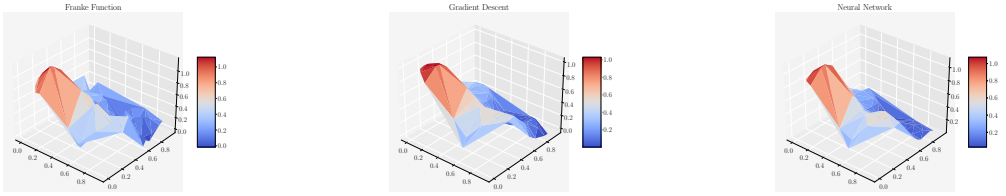


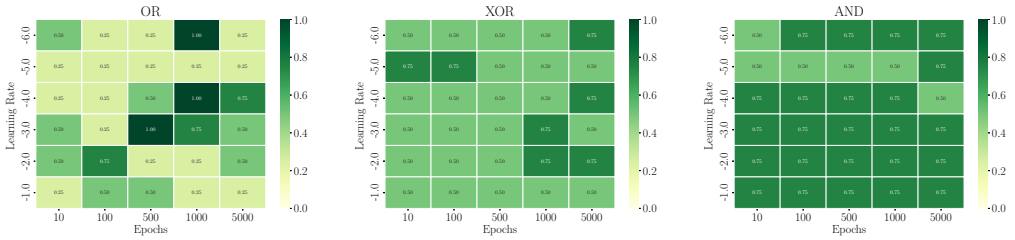
Figure 8: 3D surface plot of our best predictions on the test set compared with the actual test set of Franke's function.

5.2 Classification

To assess the FFNN classifier's performance, we employed the accuracy score as a quantitative measure, indicating the proportion of correct outputs for test examples (Goodfellow et al. 2016). During the grid search, we observed similar performance across a broad range of possible combinations.

It should be noted that we had some problems implementing the softmax function's derivative. This should be the preferred activation function for the output layer. We instead opted for using the sigmoid function also in the output layer, still yieled good accuracy in our models.

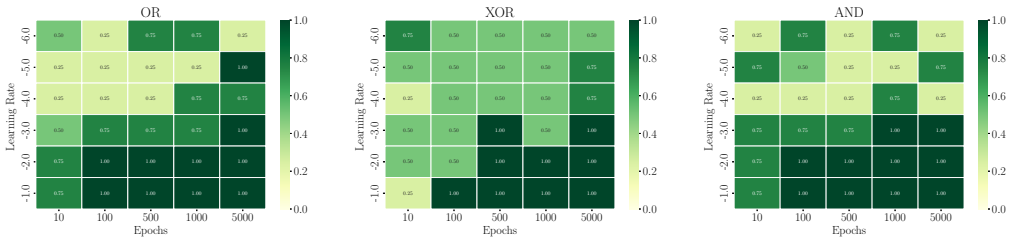
5.2.1 Logic gates



(a) (b) (c)

Figure 9: **Logistic Regression**

Logistic regression outperforms linear regression slightly. Moreover, we examine predictions from our neural network model.



(a) OR gate (b) (c)

Figure 10: **Neural network.** Fixed batch size of 20, using Adam optimizer and 1 hidden layer with 30 nodes.

Figure 10 suggests that with an appropriate learning rate and sufficient iterations, our FFNN accurately predicts all logic gates.

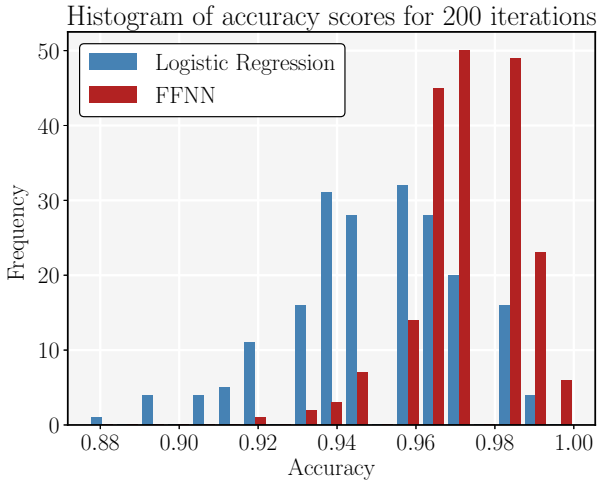


Figure 11: Histogram displaying the frequency with which each accuracy was obtained through logistic regression and our FFNN. The accuracy displayed is the accuracy of the test set for 200 random iterations of splitting the data. For logistic regression we used a learning rate of 0.1. For our FFNN we used a learning rate of 0.01 and regularization parameter of 1.0. For both we used the Adam optimizer, a batch size of 30 and 1000 epochs.

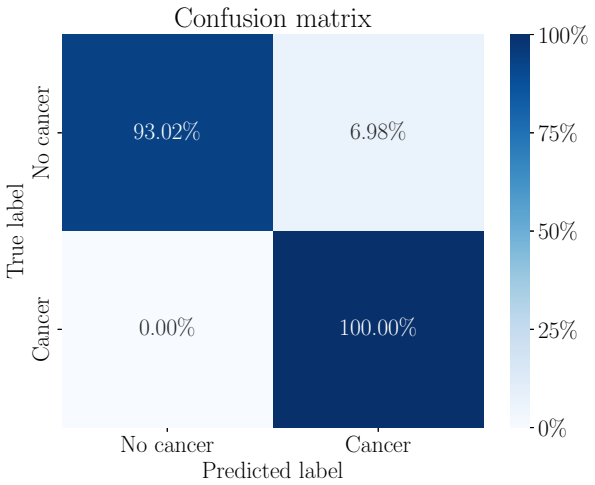


Figure 12: Confusion matrix of the test set predictions. Using the Adam optimizer, 1000 epochs, batch size of 50, learning rate of 0.1 and regularization parameter of 1.0.

Figure 11 reveals that our neural network exhibits higher mean accuracy compared to our logistic regression implementation. The FFNN achieves higher minimum and maximum accuracy scores, with instances of 100% accuracy in multiple test predictions.

In this instance, our predictions yielded 100% true positives, 0% false negatives, 6.98% false positives, and 93.02% true negatives. This implies that we correctly identified all individuals with cancer, albeit at the expense of a few false positive predictions, leading to potentially alarming messages for some healthy individuals.

6 Conclusion

In this study, we investigated regression and classification problems using a custom-developed feed-forward neural network (FFNN) code, comparing its performance against traditional methods like linear regression and logistic regression. In the regression task for Franke´s function, the descent methods acheived a respectable MSE of 0.0115, while our FFNN achieved an MSE of 0.0096. The FFNN outperformed the analytical methods from the previous project (MSE: 0.014) (Svoren et al. 2023), showing improvements in both MSE and R2 score. In the classification task, while the NN achieved a mean accuracy score of aproximately 94.0%, calculated from a small number of random samples, logistic regression achieved a mean of approximately 97.5% on the same amount of random samples. Despite the lower accuracy, logistic regression's practicalities and interpretability could make it a preferred choice for end-users. Interestingly, the FFNN with the traditional sigmoid function in the hidden layer outperformed models with more recent activation functions, like the SoftMax. Further exploration of the hyperparameter space may yield improved results, highlighting the potential for future optimization studies.

Bibliography

Efron, Bradley and Trevor Hastie (2016). *Computer Age Statistical Inference*. https://hastie.su.domains/CASI_files/PDF/casi.pdf. Cambridge University Press.

Géron, Aurélien (2019). *Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow*. O'Reilly.

Goodfellow, Ian, Yoshua Bengio and Aaron Courville (2016). *Deep Learning*. http://www.deeplearningbook.org. MIT Press.

Hjorth-Jensen, Morten (2023a). *Applied Data Analysis and Machine Learning*. URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week39.html#rmsprop-for-adaptive-learning-rate-with-stochastic-gradient-descent (visited on 9th Nov. 2023).

— (2023b). *Applied Data Analysis and Machine Learning*. URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week41.html (visited on 9th Nov. 2023).

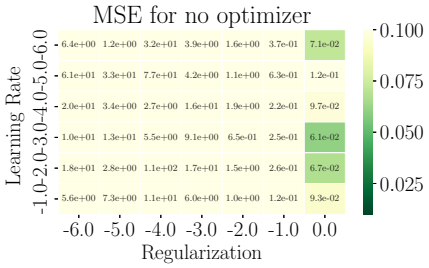
Lee, Thomas, Greta Gasswint and Elizabeth Henning (2023). *Momentum*. URL: https://optimization.cbe.cornell.edu/index.php?title=Momentum (visited on 7th Nov. 2023).

Svoren et al. (2023). *Regression Analysis and resampling methods*. URL: https://github.uio.no/mathihs/Project-1-Regression-analysis-and-resampling-methods/blob/master/Project_1.pdf (visited on 15th Oct. 2023).

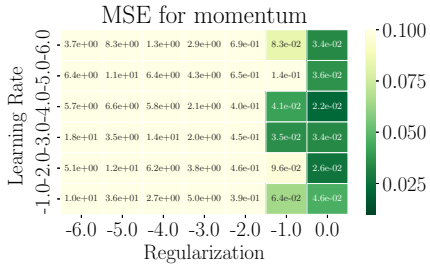
Appendix

All code for this project can be found in GitHub:
https://github.uio.no/mathihs/Project-2-Classification-and-Regression-.git

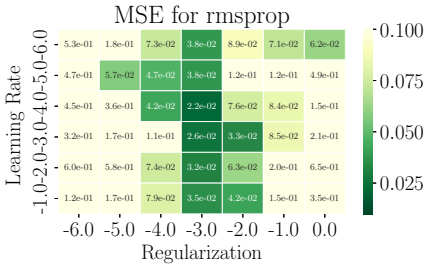
A Gridsearch FFNN GD



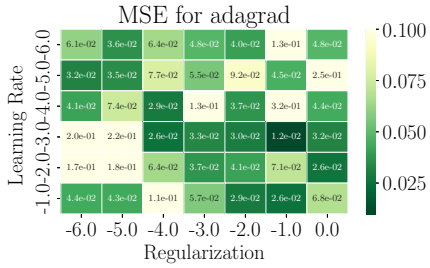
(a) MSE for model with no optimizer.



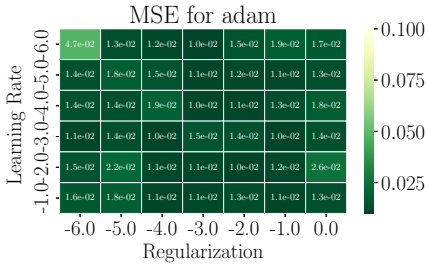
(b) MSE for model using momentum.



(c) MSE for model using RMSProp optimizer.

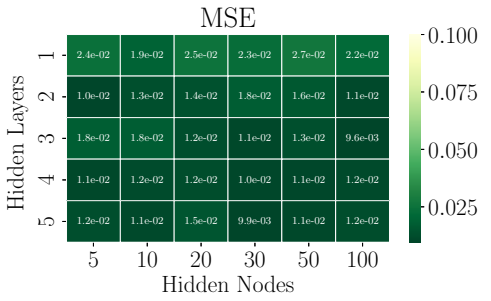


(d) MSE for model using Adagrad optimizer.

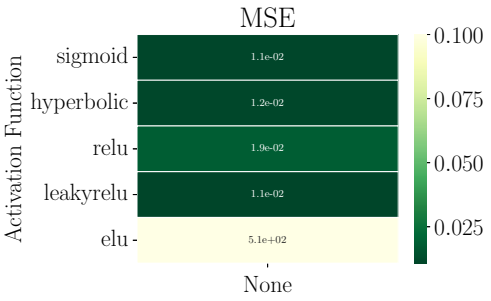


(e) MSE for model using Adam optimizer.

Figure 13: **Comparing optimizers.** Heatmap comapring optimizers, learning rate and regularization parameter on design matrix fit of Franke's function with 36 features.



(a) **Comparing layers and nodes.** Activation function sigmoid, Adam optimizer, learning rate of 0.01 and regularization parameter of 0.01. Doing 1000 epochs.



(b) **Comparing layers and nodes.** Activation function sigmoid, Adam optimizer, learning rate of 0.01 and regularization parameter of 0.01. Doing 1000 epochs.

Figure 14: Comparing optimizer, learning rate and regularization parameter for FFNN on Franke's function using GD learning method.

B Gridsearch FFNN SGD

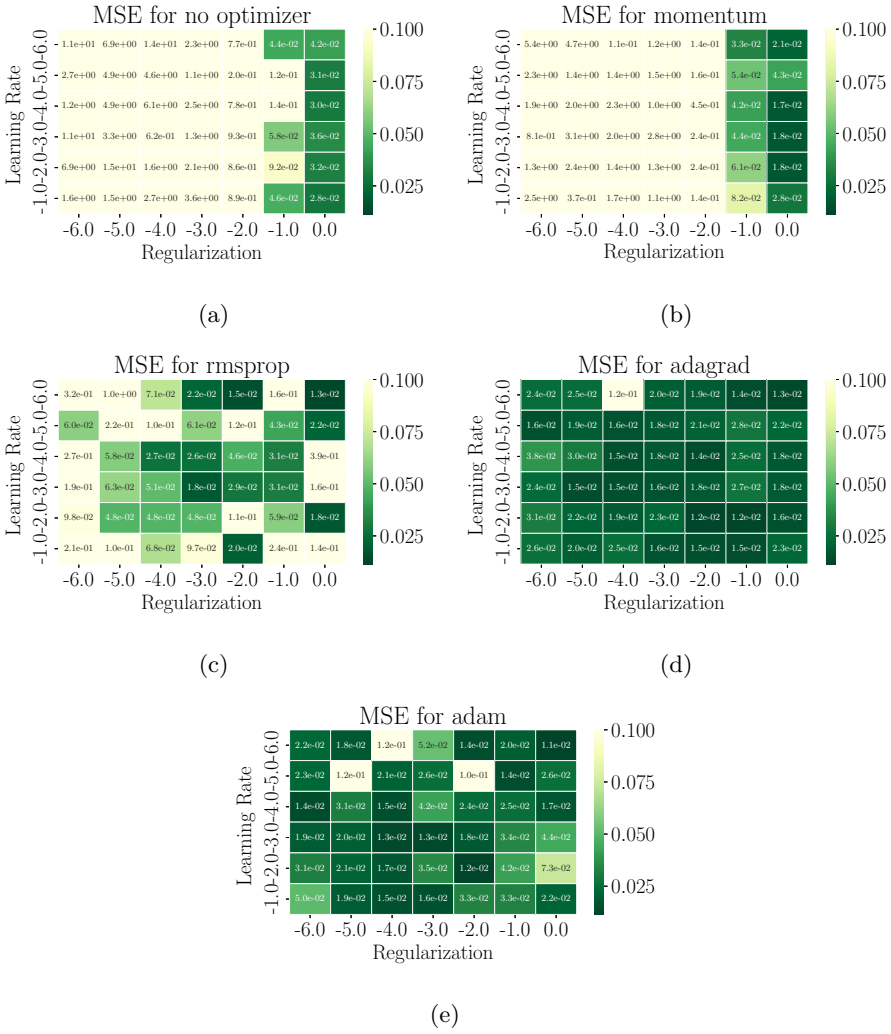


Figure 15: Comparing optimizer, learning rate and regularization parameter for FFNN on Franke's function using SGD learning method.

C Wisconsin Breast Cancer

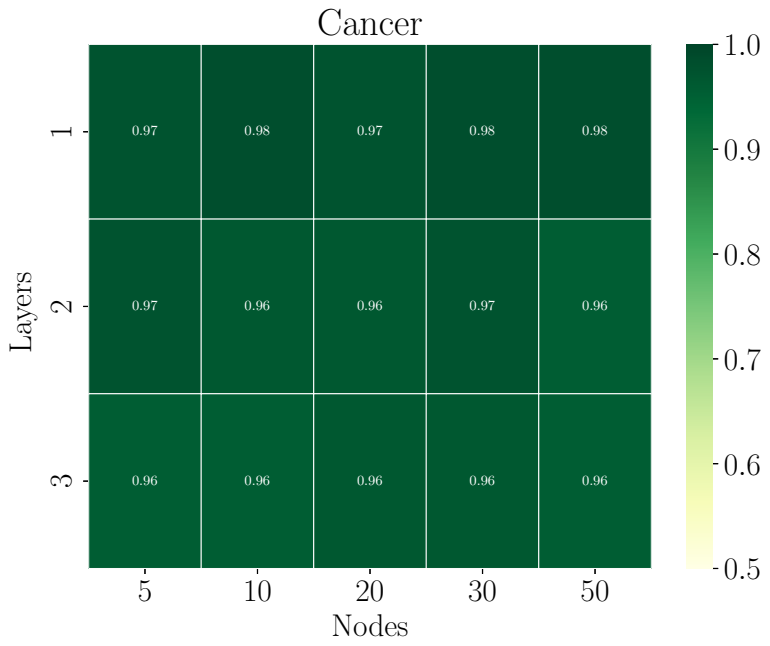
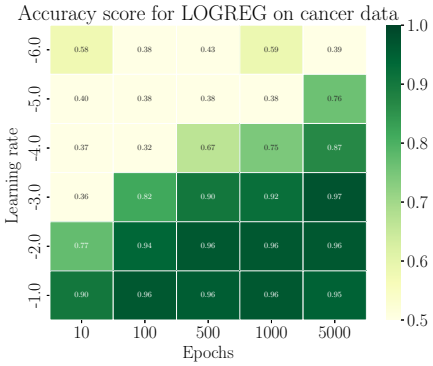
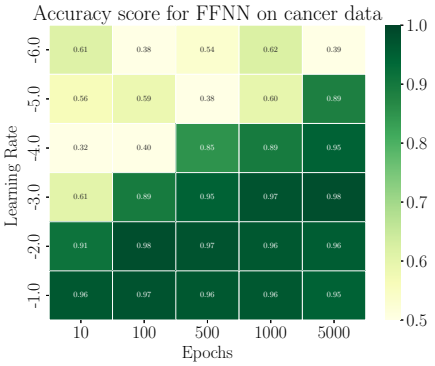


Figure 16: **Gridsearch for finding best number of hidden layers and nodes.**



(a) **Logistic regression on breast cancer data.** Heatmap of accuracy, comparing number of epochs and learning rate. Logistic regression using Adam optimizer and a batch size of 20.



(b) **FFNN on breast cancer data.** Heatmap of accuracy, comparing number of epochs and learning rate. FFNN using the sigmoid activation function, th Adam optimizer, and a batch size of 20. Network has 1 hidden layer and 30 nodes