Scientific Report for course FYS-STK4155

Project n°2

Optimizing Neural Network on Regression and Classification Problems

Sofie Vos, Kristian Bugge, Mathieu Nguyen

Department of Physics
University of Oslo
Oslo, Norway, 2023

Abstract

We explored regression problems on Franke's function using (stochastic) gradient descent, feed-forward neural networks and logistic regression. We then tackled classification problems using Wisconsin's breast cancer data as a study case. Our main goal was to analyze the impact of different parameters, methods, and network structures in neural networks. For the regression problem, optimal parameters for gradient descent on the Franke function included learning rates of 0.1 or 0.01, and a regularization parameter between 10^{-5} and 0.01.

Transitioning to neural networks, we found that the ADAM optimization method, with a learning rate of 0.0001 and a lambda of 10^{-5}, provided optimal performance. A 4-layer structure with 64 hidden nodes in each layer produced even superior results. Exploring activation functions, we found that sigmoid functions in hidden layers and an identity function in the output layer led to favorable outcomes.

Comparing our neural network results with those from project 1, we observed either similar or improved performance, depending on the chosen structure and parameters. In our classification problem, we optimized by utilizing approximately 78 hidden nodes, leading to enhanced model performance.

Now introducing and comparing with Logistic Regression, we can recognize that it is also outperformed by our Neural Network, as it is more fitting in deep learning situations such as Wisconsin's. Although still performing very well, it is not a match for our Neural Network powered with ADAM.

In the end, we concluded on saying that Neural Networks tend to be more powerful and accurate than both Ridge Regression and Logistic Regression but that it comes with a time-complexity counterpart, and that it requires more hyperparameters tuning.

Contents

1 Introduction

Neural networks, gradient descent and logistic regression are widely used to solve **classification and regression problems** like detecting spam emails [8], house price [27]and temperature forecasting [32]. They are a subset of Machine Learning, or even Deep Learning in some cases, and are blooming in recent days [22]. Which is why we wonder: What makes them so much better than already introduced linear regression methods ? That is why, as a follow up of our first project our goal is to explore the use of different regression methods and neural networks, in both regression and classification problems.
We will begin our analysis by focusing on the regression case, implementing gradient descent and its variants on Franke's Function. These methods will be integrated within a **Feed Forward Neural Network model**. Our primary emphasis will be on **identifying the most optimal hyperparameters** for each method, such as the learning rate and regularization parameter. Additionally, we will explore further optimization possibilities by determining the **optimal number of hidden layers**, the **composition of neurons** within these layers, and the **functions used** for computing both the loss and activation of neurons.
Once we have identified these optimal parameters, we will compare our best-performing model with the most effective method from the initial project, which was **Ridge Regression**. This comparison will be based on evaluating their error values, leading us to conclude whether the Feed Forward Neural Network outperforms Ridge Regression.
Subsequently, we will extend our analysis to a classification problem using **Wisconsin's Breast Cancer Dataset**. Through a systematic exploration of hyperparameters and network composition, we aim to identify the best configuration to fit our data accurately. Furthermore, we will investigate Logistic Regression and compare it with our optimized Feed Forward Neural Network. This comparative analysis aims to determine whether Logistic Regression or a more complex neural network is better suited for Wisconsin's dataset.
Lastly, we will define on which method is the best for regression and classification cases, and explain why by defining its pros and cons.

2 Definitions and Notions

This project will mostly discuss about the usage and performance of **Neural Networks**. Therefore it is essential to define a few notions in order to properly grasp how it works and how useful it could be in regression problems.

2.1 Neural Network

A neural network is a computational system that can learn to perform tasks given numerous examples. It takes its name from the biological brain network system as it is making its numerous neurons interact with each other by sending mathematical functions as signals. It is composed of multiple layers, each with an arbitrary number of neurons. [14] Computational implementations of Neural Networks are called **Artificial Neural Network (ANN)**. In ANN, the neurons are called nodes, and they are usually composed of input and output layers, as well as an arbitrary number of hidden layers based on how many computations we want to make.

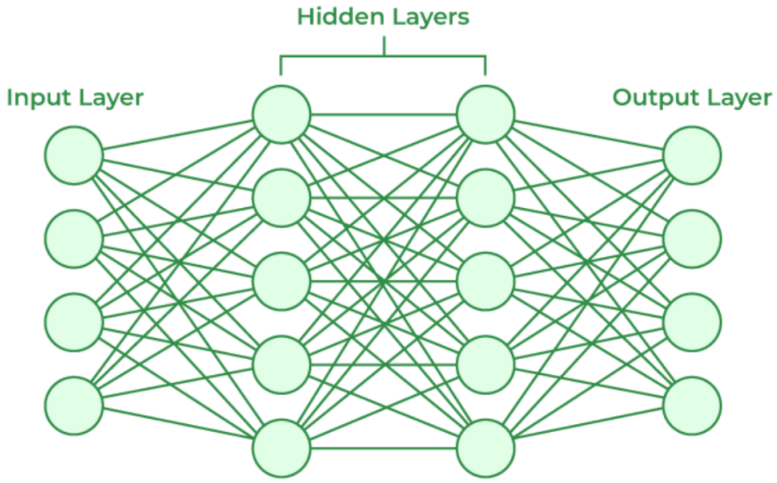

Figure 1: An exemple of a Neural Network [10]. Each node may have a connection with multiple other nodes, depending on how we want them to interact with each other

The connection of two nodes is associated with a **weight**, which determines how we get from one layer to the other. The main purpose of these Neural Networks is that they are **able to perform tasks simply by being exposed to various datasets**, without any prior task-specific rules, **as if they were learning on the spot** [9]. These weights are computed with some propagation algorithms along the nodes and depend on multiple parameters, such as its **batch size** or its **learning rate**.

2.2 Learning Rate

The learning rate is a **hyperparameter** that controls how much a model should adjust its weights given the estimated error of the loss gradient.

$$\Theta_{t+1} = -\eta\Theta_t \tag{1}$$

Where η is the learning rate.

The lower the value, the slower we travel along the downward slope. Finding an optimal learning rate is essential to get the best results. Indeed :

- If it is too big, then the changes would be too consequential and the model might **diverge** and never reach a local minima.

- If it is too small, then the model will converge to the local minima very slowly, and we **might stop the model's computations** before it does reach it.

Therefore, we must put great focus into obtaining a learning rate that is fit for our data [43]. This can be done by **computing the Hessian matrix of the cost function**, which gives a lower bound for the learning rate [42]. Indeed, a relatively good learning rate can be computed as followed :

$$\eta = \frac{1}{\lambda_{max}} \tag{2}$$

Where λ_{max} is the *highest eigenvalue of the Hessian matrix.*

But the Hessian matrix can be **computationally expensive** to obtain. Therefore there are methods that try to **approximate the learning rate** without computing the Hessian matrix. [17]

2.3 Epochs

In machine learning, an epoch is an instance of an **entire training dataset that has been passed through an algorithm**. It is a completely different hyperparameter from the **batch size** since this one only refers to **how many samples are in said dataset**, whereas the epochs are **how many different datasets are used** to train our model [33]. Usually, the more epochs we use, the better our model will result in since it allows our model to stop before **risking overfitting**, but doing so will take even more time to compute and may even overfit if we use too many epochs, which is what we want to prevent [1]. Therefore, it is essential to balance the number of epochs along with the batch size.

2.4 Activation Function

An activation function is a function that decides **whether a neuron should be activated or not** by calculating its weight and adding bias to it, and therefore decides if the neuron's input to the network is **important or not in the process of prediction** [4]. Its purpose is to **introduce non-linearity** in the output of a neuron [38]. Apart from the activation functions being non-linear, they must also have the following properties:

- Continuously differentiable : needed to compute the gradient (see backward propagation)
- Range : A well-defined range for the activation function's output can help **control the scale of activations** throughout the network. It can prevent activations from growing too large (exploding gradients) or vanishing (vanishing gradients).
- Monotonic : Monotonicity can make **optimization more predictable and training more stable**, as the output increases with the input.
- Approximates identity near the origin : it allows the network to **maintain linear behavior for small inputs**. This is important for preserving information during training. [36]

Since they are simply mathematical functions, a lot of different activation functions exist and are used in various models in machine learning.

2.5 Accuracy Score

An accuracy score is a cost function that gives an indication on how many guesses are actually correct out of the total number of targets. Its formula is simply

$$accuracy = \frac{\sum_{i=1}^{N} I(t_i = y_i)}{N} \tag{3}$$

And it is a great tool in Classification, as it is very simple to implement and there exists tools to visualize it.

2.6 Confusion Matrix

A confusion matrix is a performance measurement for machine learning classification problem. It is **usually used for binary cases** as it is really simple to set up with just two classes, but there can obviously be more.
In binary cases, a confusion matrix has four metrics :

- True Positive (TP) : the values predicted positive (say 1) and are indeed positive
- True Negative (TN) : the values predicted negative (say 0) and are indeed negative

- False Positive (FP) : the values predicted positive but are in fact negative
- False Negative (FN) : the values predicted negative but are in fact positive

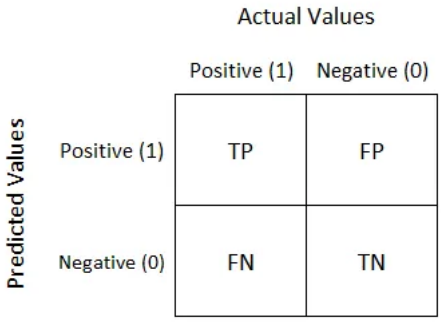

Figure 2: How is constructed a Confusion Matrix on **binary class cases**. [28]

The confusion matrix allows us to compute three accuracy metrics : **the accuracy, the precision and the recall** [28].

$$accuracy = \frac{TP + TN}{Total} \qquad precision = \frac{TP}{TP + FP} \qquad recall = \frac{TP}{TP + TN} \tag{4}$$

The precision is the percentage of positive predictions that were actually positive.

The recall is what percentage of the positive values where correctly predicted.

The accuracy is the **number of correct predictions**, and is what will matter the most in this project as we want to make our model predict values as best as possible. Thus, if we want to have a good accuracy score, we want the **values on the diagonal to be as high as possible**.

2.7 Feed Forward Neural Network

The Feed-Forward Neural Network (or FFNN) is the algorithm on which our Neural Network will be based. The FFNN presents itself with multiple layers of neurons : the input, output and hidden layers.

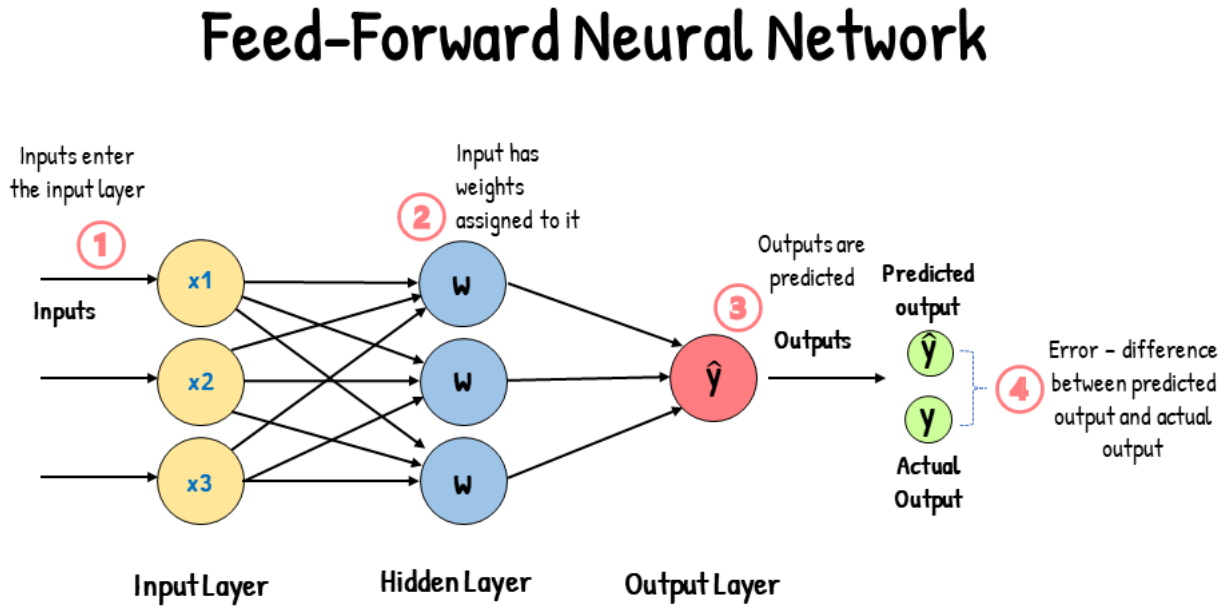

Figure 3: Algorithm and construction of the FFNN. [24] The number of hidden layers is arbitrary and can be set by the developper, but the number of neurons in each layer must match the number of features in our dataset.

Each link within nodes is assigned a weight, and by multiplying inputs by said weight when entering the input, we can determine **how impactful a node is**. The input layer receives input and passes it on to the other layers of the network, while the output layer represents the forecasted feature. Most of the computations are done within the hidden layers, **transforming the inputs** based on the weights' values before entering the following layer. Each neuron creates weighted input sums and **activates them given a specific activation function**, as if they were **making a decision** on a feature's usefulness. [39]

2.8 Backpropagation

We must keep in mind that, although the information cannot be propagated back towards the input, **that is not the case of other variables**. Indeed, the FFNN is often backed up by another algorithm called the **Backpropagation**, a technique used to **minimize cost function** when learning a neural network. All neurons in each layer are used to compute the cost function, and their gradient is **exploited by the Backpropagation iteratively**. By obtaining the gradient of the output layer, we are able to update the weights and biases by **iterating backward on all layers**, reducing the gradient of all neurons.
[39]

The idea behind the FFNN is to run this loop over and over so that the loss function gives a smaller error at every step.

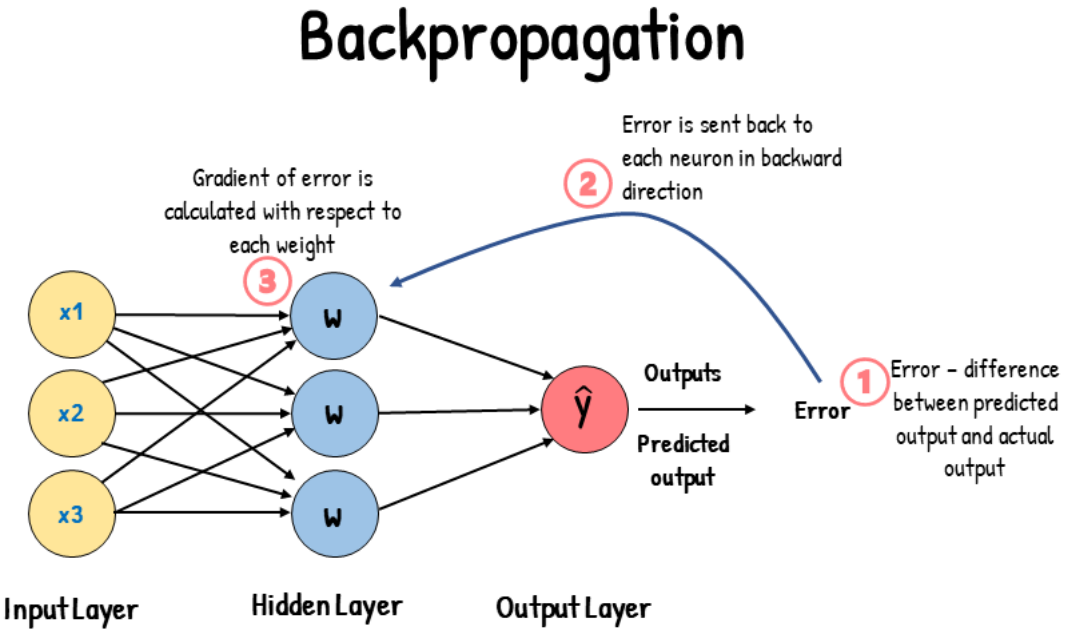

Figure 4: Algorithm and construction of the backpropagation completing the FFNN. [24] The gradient in the output allows us to update all weights and biases by going backward towards the input start.

3 Methods Implemented

Now that we have defined all notions required to understand the construction of a neural network, let us introduce all methods used to create said network. There are numerous way to create a Neural Network, therefore we will explain how did we tackle this project.

3.1 Chain Rule

The chain rule is a method to **calculate the derivative of a composite function**. It is usually used when there's a function within another function (say the function $f(g(x))$), and in this case, the chain rule is defined as :

$$\frac{df}{dx} = \frac{df}{du}\frac{du}{dx} \tag{5}$$

Allowing us to compute the **derivative of f with relation to x** by the intermediate of another function u. [26]

This can be generalized to bigger composition of functions and is greatly used in machine learning. Indeed, the chain rule can be illustrated by a **tree diagram** that looks like :

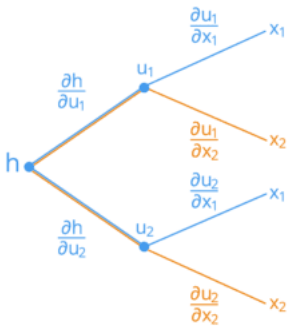

Figure 5: An illustration of the Chain Rule as a **tree diagram**. If we want to compute the derivative of $h(x) = u_1(u_2(x))$, we would need to compute **both the derivative of** u_1 **and** u_2, and so on if these functions were also composed. When every function has been computed as a basic operation, all the results are brought back to the start, just like a **backpropagation algorithm**.[6]

We observe that this tree functions **just like a neural network**, as it passes all of its computations from one way to another.This means that we can apply the chain rule to a neural network through the use of backpropagation. This implementation will be crucial for many computations, especially the **cost and lost functions**, as they requires to compute the gradient of difficult functions. [6]
As a layer's output is the input to the next layer, the chain rule is used to find the gradients.

3.2 Autograd

In order to compute the gradient of the loss function, it is necessary to be able to calculate the partial derivative of it. Sadly, computing the partal derivative can prove to be **computationally expensive**, especially when the number of dataset we're using (and thus, the size of its design matrix) is large. Therefore, neural networks use a different approach for achieving such feats by the means of **Automatic Differentiation**.
Indeed, this can be achieved by the fact that any computer calculation only executes a **series of arithmetic operations and functions**. And by applying the chain rule repeatedly to all operations, we can obtain the partial derivative of any function with working precision. [41]

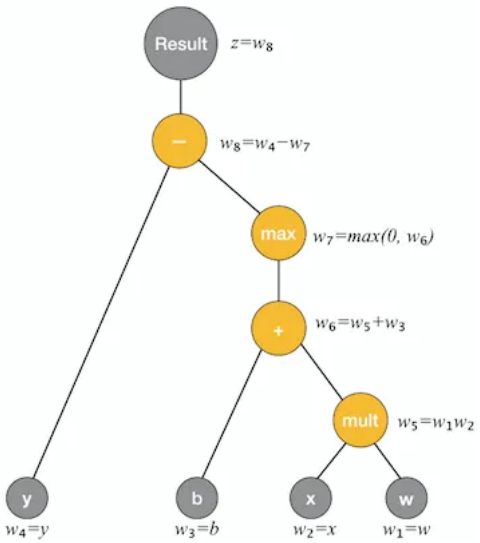

Figure 6: An illustration on how Automatic Differentiation works. Every operation is symbolized by a yellow node, while the values are illustrated by the white node. All operations of the function are linked and computed via **chain rule**. We must take into account that autodiff can only compute the partial derivative of a function **on a certain point**, therefore it is needed to set initial values of the variable. [40]

Autograd is a technique used in machine learning to automatically compute the gradients and is **Python's most famous library** for implementing Automatic Differentiation. It is **less error prone and simpler than manually computing them**. Autograd also "records a graph recording all of the operations that created the data as you execute operations, giving you a directed acyclic graph whose leaves are the input tensors and roots are the output tensors. By tracing this graph from roots to leaves, you can automatically compute the gradients using the **chain rule**." [31]

3.3 Activation Functions

As mentioned previously 2.4, there is a considerable amount of activation functions to choose from, each with their own advantages and drawbacks. We will be listing all the activation functions that we have considered and studied during this project.

3.3.1 Sigmoid

The sigmoid function is a very popular activation function in Neural Network and was introduced in a paper in 1986 by David Rumelhart. It is one of the earliest non-linear functions used to perform the "forward" and "backward" propagation of a Neural Network. [2]

The sigmoid function is defined as follows :

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{6}$$

Resulting in a S-shape function, taking values from 0 to 1.

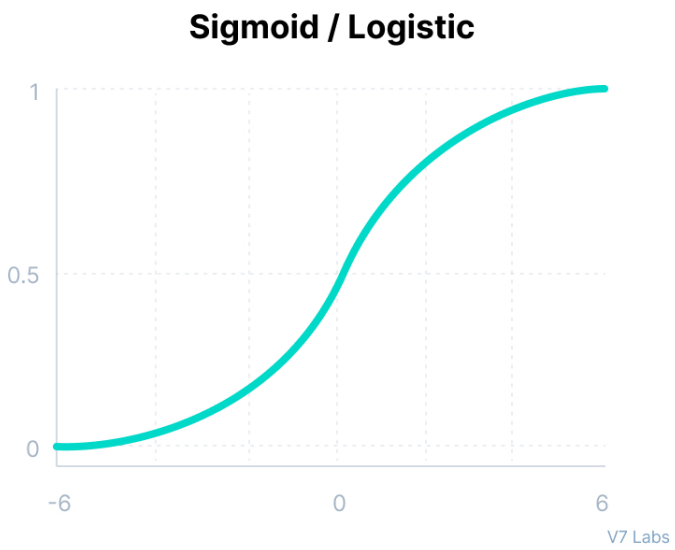

Figure 7: A plot of the sigmoid function. The larger the input is, the closer to 1 the output will be, meaning that said input has a great impact.

Advantages :
- The sigmoid function allows us to have a smooth gradient, preventing "jumps" in the output values.
- It also normalizes the output of each neuron by given values bound between 0 and 1 and gives a clear prediction (classification) by giving values within these bounds.

Drawbacks
- It is however prone to the **vanishing gradient problem.** It is a phenomenon that occurs during the training of deep neural networks when the gradients computed become very small and vanish during the backpropagation part.
- It is also non-centric, as it always gives positive values. Meaning it does not prevent the model from learning a strong positive or negative bias, which can lead to convergence issues.
- It is a computationally expensive function because of the exponential in it.

3.3.2 ReLU

The Rectified Linear Unit or ReLU function is another activation which has a derivative function and allows for backpropagation while simultaneously making it computationally efficient. [3] It is also a function that facilitateed the training of deeper neural network, something that the sigmoid was not able to. [37] It also does not activate all neurons at the same time.

The ReLU function is defined as follows :

$$ReLU(x) = max(0, x) \tag{7}$$

Resulting in a flat curve in negative inputs, and a linear growth afterwards.

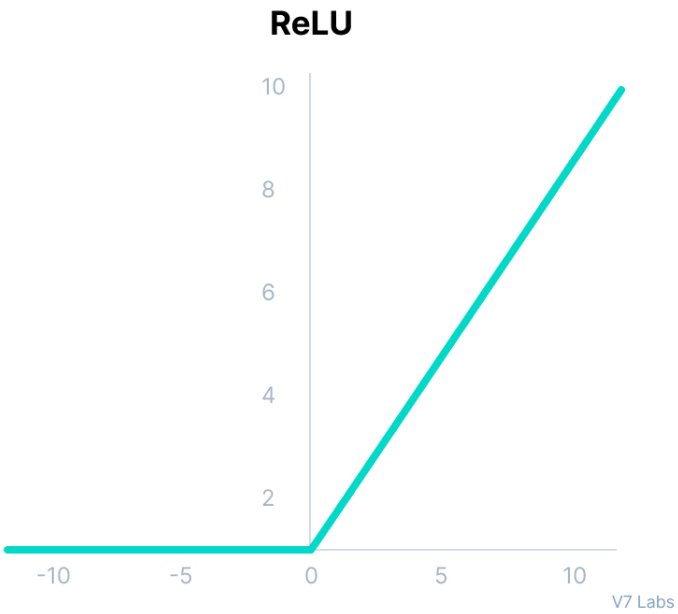

Figure 8: A plot of the ReLU function. Its idea is to deactivate a neuron if the output of the linear transformation is negative.

Advantages :

- the ReLU function fixes the vanishing gradient problem, as the value of the partial derivative of the loss function will be having values of either 0 or 1.
- It is also computationally easy to obtain, as the only operation is the max(), a very computation-friendly task.

Drawbacks

- Just like the sigmoid, it is not a zero-centric function.
- The function has the problem known as the **dying ReLU**. Indeed, during training, some neurons stop **outputting anything other than 0** and 'die', leaving them **completely unactive**.

3.3.3 Leaky ReLU

Leaky RELU is a variant of ReLU, except that instead of fixing negative values as 0, it has a **small positive slope in the negative area**. It is described by the function :

$$LReLU = max(\alpha x, x) \tag{8}$$

With α being a floating value.

Thus giving us the following plot :

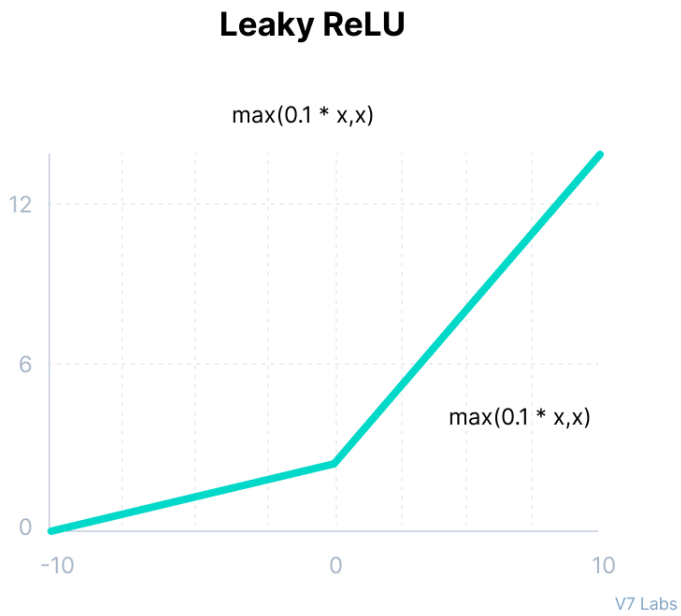

Figure 9: A plot of the LReLU function. This time, instead of a flat value, we output a smaller but present slope to ensure that the neurons still has some activity.

This way, we ensure that all neurons in the network can contribute to the output, even if their inputs are negative. rather than completely killing the neurons, thus **preventing the dying ReLU problem**.

Advantages :
- Brings the same advantages as the ordinary ReLU, without risk of dying ReLU situations.

Disadvantages :
- The α used in this activation function is a **hyperparameter** that needs to be adjusted carefully given the dataset. If the hyperparameter is too high, the function might **act like a regular linear function**. If it is too low, it may **not address the dying ReLU problem** effectively.

3.4 Cost Functions

There is also a lot of cost functions to choose from, but some of them can only be used in specific situations. We will define which cost functions have been used throughout this project.

3.4.1 Ordinary Least Squares

Ordinary Least Squares (or OLS) is the most common and simplest function to define the coefficients of our model. It has already been used in our first project and will mostly be our stepping stone to compare with our previous Linear and Ridge Regression.
The cost function alongside OLS is the **Mean Squared Error** (or MSE), and is defined the following way :

$$C(\tilde{y} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \tilde{y}_i)^2 \tag{9}$$

This cost function can be used for any problem, but will mostly be used on **regression problems**.

3.4.2 Logistic Regression

As the name suggest, this cost function follows the Logistic Regression principle, and **can only be used in classification problems**, as a linear regression cannot properly define a Logistic Regression.
In a Binary Classification Problem, the cost function is defined as :

$$Cost(\tilde{y}) = \begin{cases} -log(\tilde{y}) & \text{if y = 1} \\ -log(1-\tilde{y}) & \text{if y = 0} \end{cases}$$

Combined into a single equation, the Logistic Regression cost function will be defined as : [29]

$$C(\tilde{y}) = -\frac{1}{N}\sum_{i=1}^{N}(y_i log(\tilde{y}_i + \epsilon) + (1-y_i)log(1-\tilde{y}_i + \epsilon)) \tag{10}$$

With ϵ being a very small number (usually 10^{-10}) to make sure that we do not accidentally compute the log of zero, resulting in the system crashing.

3.4.3 Cross Entropy

Cross Entropy is our last cost function and represents the difference between two probability distributions for a given random variable or set of events. This can be given with the function

$$Cost(\tilde{y}) = -log(\tilde{y}) \tag{11}$$

In a set of x in X discrete states, the cost function will result in : [5]

$$C(\tilde{y}) = -\frac{1}{N}\sum_{i=1}^{N}(y_i log(\tilde{y}_i + \epsilon) \tag{12}$$

With ϵ being a very small number for the same reasons as mentionned previously.

3.5 Stochastic Gradient Descent

The Gradient Descent methods are algorithms used to **minimize a cost function.** The idea of the method is that "a function decreases fastest if one goes from in the direction of the negative gradient" : [18]

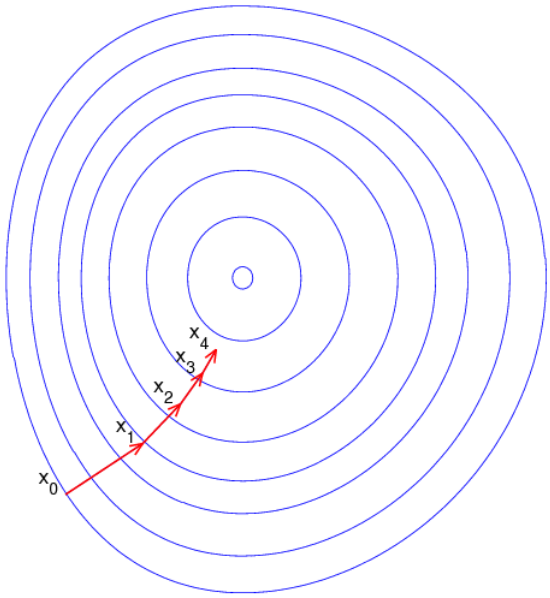

Figure 10: An illustration on how the Gradient Descent method works. Here, the gradient makes his way into the center, where the loss function is at its lowest, therefore minimizing it. [7]

The most basic implementation of the Gradient Descent method (also called Steepest Descent) follows the algorithm :

$$x_{k+1} = x_k - \gamma_k \nabla F(x_k) \tag{13}$$

With γ being the learning rate, a *positive floating value.*

How much to go in the direction of the negative gradient is determined by the **learning rate**. For small enough learning rates we are always moving towards smaller function values and will **eventually find a global or local minimum.** [18]

It can however be **too computationally expensive** to update the gradient over the whole dataset for each iteration. Indeed, it is computationally faster to compute the gradient of multiple small dataset rather than the gradient of one big dataset multiple times thanks to **vectorized code optimization**.
Thus, a variant is introduced, called the **Stochastic Gradient Descent** (or SGD). In this method, the dataset (or batch) is split up into subsets (or mini-batches) and the gradient is **updated over a mini-batch** each iteration. [19]

The SGD algorithm will hence be the following :

$$\beta_{j+1} = \beta_j - \gamma_j \sum_{i \in B_k}^{n} \nabla_\beta c_i(x_i, \beta) \tag{14}$$

where k is an integer picked at random with equal probability from [1, E], referring to an interaction over a mini-batch. In common usage, and also during this project, we will set $E = \frac{n}{M}$, as this value is commonly referred as an **epoch**. [19]

3.6 Addition of Momentum

Usually, we implement SGD alongside a momentum term that **keeps memory of the direction** we are moving in parameter space. [15] The momentum parameter helps the gradient descent algorithm "**gain speed** in directions with persistent but small gradients, while also **suppressing oscillations** in high-curvature directions." The gaining of speed makes GD with momentum **faster than the plain GD** and combined with suppressed oscillation, GD with momentum has a **chance of escaping local minima**. [23]

Therefore, the new Gradient Descent with Momentum implemented follows :

$$\begin{aligned} \mathbf{v}_t &= \gamma \mathbf{v}_{t-1} + \eta_t \nabla_\theta E(\boldsymbol{\theta}_t) \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \mathbf{v}_t, \end{aligned}$$

where γ is the newly introduced momentum, with $0 \leq \gamma \leq 1$ and η is the learning rate. $\mathbf{v}_t$ is a running average of recently encountered gradients and creates momentum for computing our gradient θ_t.

3.7 Appromixating the Learning Rate

Calculating the right learning rate can be accomplished with **second-order methods**, for example, by computing the Hessian matrix. However, these methods become quite computationally expensive as the model grows since computing the Hessian results may **take a lot of CPU memory** [25]. Thus, we use some methods that keep track of the second moment of the gradient and are less computationally expensive. The most popular examples are **RMSProp, AdaGrad and Adam**.

3.8 RMSProp

The **Root Mean Squared Propagation** (or RMSProp) is the first of our second-order methods. In this method, We keep an **average of the first moment of the gradient**, as well as the **second order moment** and introduces a **decay factor** β, putting an emphasis on the more recent gradients [23]. The update rule now becomes :

$$\begin{aligned} \mathbf{g}_t &= \nabla_\theta E(\boldsymbol{\theta}) \\ \mathbf{s}_t &= \beta \mathbf{s}_{t-1} + (1-\beta)\mathbf{g}_t^2 \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta_t \frac{\mathbf{g}_t}{\sqrt{\mathbf{s}_t + \epsilon}}, \end{aligned}$$

where $\mathbf{g}_t$ is the *gradient*, $\mathbf{s}_t$ is the *second order of the gradient* and $\boldsymbol{\theta}_{t+1}$ is the updated position. A small ϵ variable is also introduced to **prevent divisions by 0**. In practice, we tend to use $\beta = 0.9$ and $\epsilon = 10^{-8}$

Thanks to this method, "the learning rate is reduced in directions where the **norm of the gradient is consistently large**. This greatly speeds up the convergence by allowing us to use a **larger learning rate for flat directions**."[16]

3.9 Adagrad

Adaptive Gradient algorithm (or AdaGrad) is similar to RMSProp, except it does not use a decay to deal with sparse features [23]. Indeed, it performs **smaller updates** for parameters associated with **frequently occurring features**, and larger updates for the infrequently occurring ones. The learning rate is updated **inversely proportional to the past gradient** at every step, making the algorithm as follows :

$$\begin{aligned}\mathbf{g}_t &= \nabla_\theta E(\boldsymbol{\theta}) \\ \mathbf{s}_t &= \mathbf{s}_{t-1} + \mathbf{g}_t^2 \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta_t \frac{\mathbf{g}_t}{\sqrt{\mathbf{s}_t + \epsilon}},\end{aligned}$$

With this method, a feature that has been considerably updated will be **less modified in the future**. This allows for sparse features in the model to catch up, preventing bias made from the first steps (as they are usually the more impactful). [23]

3.10 ADAM

Adaptive Moment Estimation (or Adam) "is a combination of the **gradient descent with momentum** algorithm and the **RMSprop** algorithm discussed above." [11] It also combines both their advantages, making it "the **go-to choice** of deep learning problems" nowadays. ADAM estimates the running average of the first and second moment orders of the gradients, along with a bias to make up for the approximations, giving us the final algorithm as :

$$\begin{aligned}\mathbf{g}_t &= \nabla_\theta E(\boldsymbol{\theta}) \\ \mathbf{m}_t &= \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t \\ \mathbf{s}_t &= \beta_2 \mathbf{s}_{t-1} + (1-\beta_2)\mathbf{g}_t^2 \\ \mathbf{m}_t &= \frac{\mathbf{m}_t}{1-\beta_1^t} \\ \mathbf{s}_t &= \frac{\mathbf{s}_t}{1-\beta_2^t} \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta_t \frac{\mathbf{m_t}}{\sqrt{\mathbf{s_t}} + \boldsymbol{\epsilon}},\end{aligned}$$

where $\mathbf{g}_t$ is the *gradient*, $\mathbf{m}_t$ and $\mathbf{s}_t$ are respectively the *running average of the first and second moments of the gradient* and $\boldsymbol{\theta}_{t+1}$ is the updated position.
Usually, we set up $\beta_1 = 0.9$ and $\beta_2 = 0.99$ and ϵ and η just **like in RMSProp**, and just as this latter method, the effective step size **depends on the squared gradient**.
This way, ADAM gets **both the speed from momentum** and **the ability to adapt gradients** in different directions from RMSProp. [11]

3.11 Feed Forward Neural Network

We implemented the **Feed Forward Neural Network** (or FFNN) as defined in 2.7, by creating an input and output layer, as well as an arbitrary number of hidden layers.

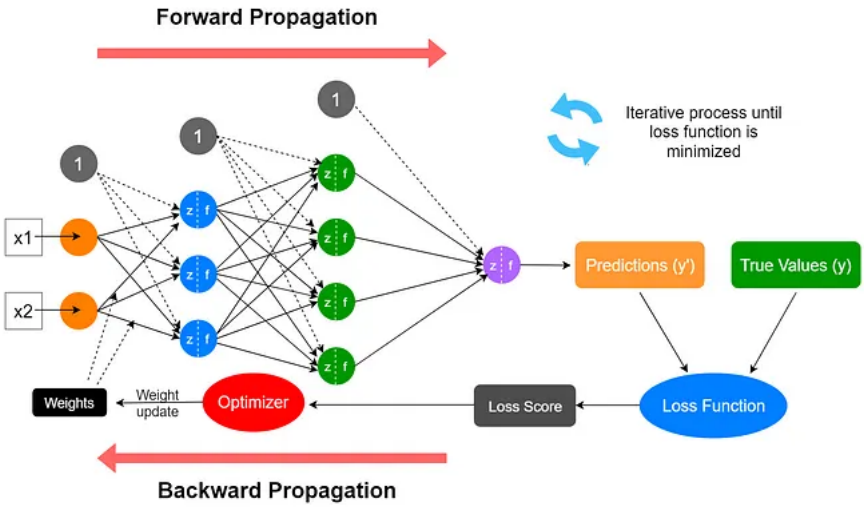

Figure 11: Algorithm and construction of the FFNN. [30] The cost and activations functions can vary depending on if we are discussing a regression or classification problem.

The inputs go into the first layer. There, each input is multiplied by a weight and a bias is added. The result is given to an activation function. Afterwards the result of the activation function is the **input to the next layer**, which again multiplies the input it gets with a weight and adds a bias. This feed-forward process continues until the last layer. [13]
When reaching the last layer, our FFNN computes the loss function. There are multiple loss functions possible and they are defined when constructing the FFNN.

To update the weights, the derivative of the cost function with respect to each weight of the output is calculated using the chain rule. The calculated derivative is used to update the weights with **gradient descent**. The same process is used to update the biases instead of the weights [12].

3.12 Logistic Regression

Our last model is **logistic regression**, a method which consists in modeling the probability of a discrete outcome given an input variable. It is used in **classification problems**, and **mostly in binary cases**, where the output is either True or False (the method is then called *binary logistic regression*).

This method is not based on a neural network, and is rather studied here as a stepping stone to compare its performance against all neural network methods. It is nonetheless still widely used in modern Machine Learning algorithms.

It is used to predict the probability of each outcome, as the categories have a probability of

$$\begin{aligned} p(G=k|X=x) &= \frac{e^{\beta_{0,k}+\beta_k^T x}}{1+\sum_{k'=1}^{K-1} e^{\beta_{0,k'}+\beta_{k'}^T x}} \\ p(G=K|X=x) &= \frac{1}{1+\sum_{k'=1}^{K-1} e^{\beta_{0,k'}+\beta_{k'}^T x}} \end{aligned} \tag{15}$$

with k = 1,...,K-1.

Logistic regression tends to be more reliable when there are only a few classes, like in the **binary case**. However, when dealing with bigger datasets and multiple cases, we should **rather use neural networks**, as they are more fit in deep learning problems. [21]

4 Dataset Used

Since Neural Network can be used for a lot of situations, we wanted to tackle dataset with different problems. Therefore we will introduce the two datasets that we studied during this project.

4.1 Franke's Function

Franke's Function is a 2D function widely used for **interpolation problems**, composed of **two Gaussian peaks** of different heights and a **smaller dip**.

This is the same function that we have used in our previous project as a basic example to code our methods, and will be our candidate dataset for **regression problem**. Like in the last project, Franke's Function is defined by the following function and plot : [35]

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

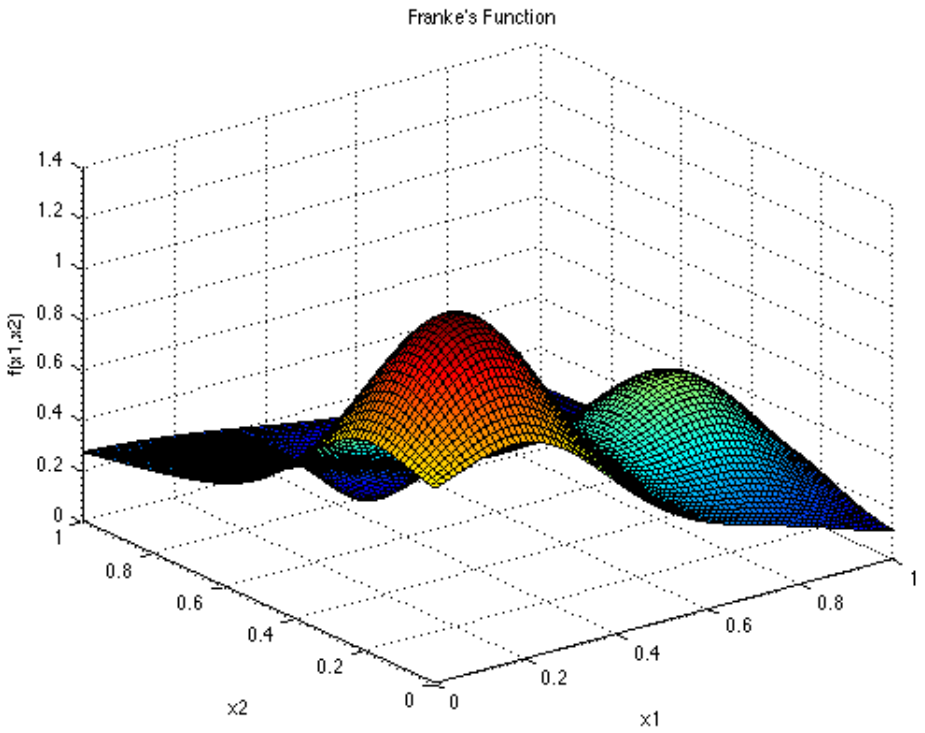

Figure 12: 3D plot of Franke's Function

We can observe that the output here is **only defined by two variables**, so it is unlikely that one variable has little impact on the results compared to the other. Since it takes values in R, we can use it to **adapt our models for regression problems**, therefore the error score can be computed with a **simple mean squared error** or $\mathbf{R^2}$ **error** and visualized with either plotting the predicted output, or the **MSE score**. We can also compare our results with Neural Networks models with our previous regression models to see if they perform better.

4.2 Wisconsin Breast Cancer Dataset

The **Wisconsin Breast Cancer Dataset** is a dataset introduced by the *University of California Irvine (UCI)*, recording the measurements for breast cancer cases. The input variables are called **features** and are computed from a **digitized image of a fine needle aspirate (FNA) of a breast mass**, describing characteristics of the cell nuclei present in the image. [34]

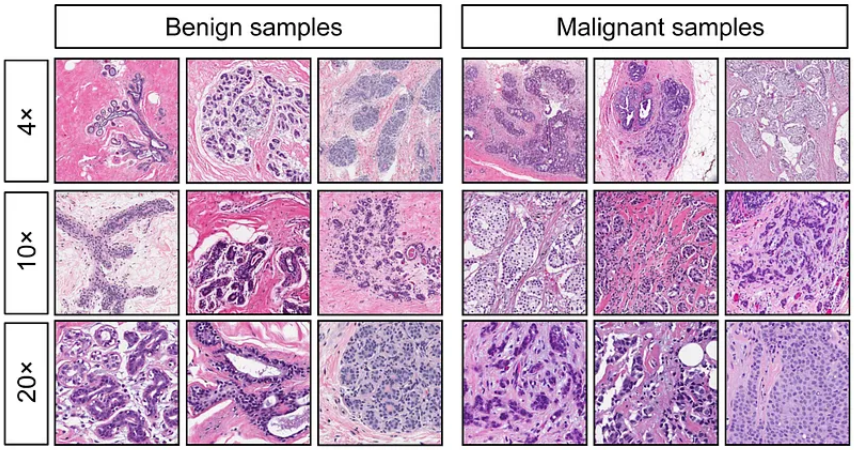

Figure 13: Example of Wisconsin's Dataset's image. It is very difficult to distinguish benign and malign samples, but these images also comes with many variables, helping characterizing every image contained within the dataset.

The output variable is called **target** and is composed of two classes, benign and malign, making this dataset a **binary classification dataset**. WBC Dataset contains 569 instances, divided into 357 benign and 212 malignant. [34] We will also use our Neural Network models to predict targets of this dataset, however this being a classification problem rather than a regression problem, we need to use other means to establish whether our network properly fits the data. This can be achieved by computing the **misclassification error** and plotting the **confusion matrix**.

5 Implementation and Results

Now that we have discussed what model will be presented and how will be manipulate and study them, we will implement them and discuss their results. A great portion of our FFNN has been taken from **Morten Hjorth-Jensen's FFNN implementation**, which can be found here [20], but there are more methods that we've added and that we will present along the way.

5.1 Comparing different Gradient Descent methods

Here, we will compare all of the stochastic gradient descent schedulers discussed in the previous section using the data from the Franke Function. To recall, these were **Constant Gradient Method 3.5, RMSProp 3.8, Adagrad 3.9 and ADAM 3.10**, all with and without the addition of **Momentum** 3.6.

Our goal is to find the optimal hyperparameters for each scheduler, which are the **learning rate** η and the **L2 regularizer** λ. A easy way to find them is to decide a range of value for both and to compute the results for each combination. Thus, we run every scheduler on said range and plot a **heatmap of the results** for each model. The hyperparameters have been taken in these ranges :

$$\eta \in \{0.0001, 0.001, 0.01, 0.1\} \qquad \lambda \in \{0.0, 10^{-5}, 0.0001, 0.001, 0.01, 0.1\} \tag{16}$$

We have also implemented **cross-validation** along the computation, to reduce the chances of outliers and get the best possible results. This implementation can be found in 7.1
When compiled, our gradient descent implementation gives the following results :

Figure 14: Heatmap of the normal Stochastic Gradient Descent

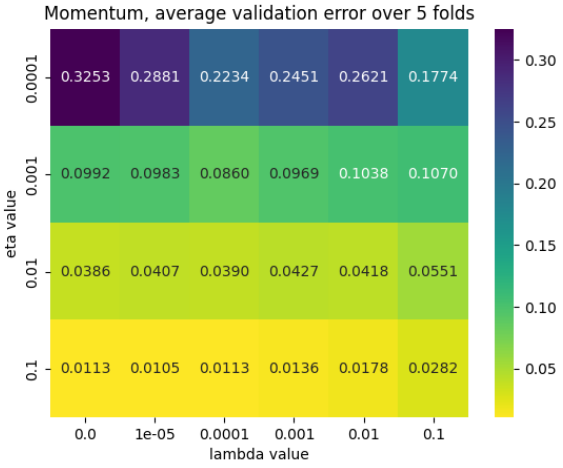

Figure 15: Heatmap of the Stochastic Gradient Descent with Momentum added

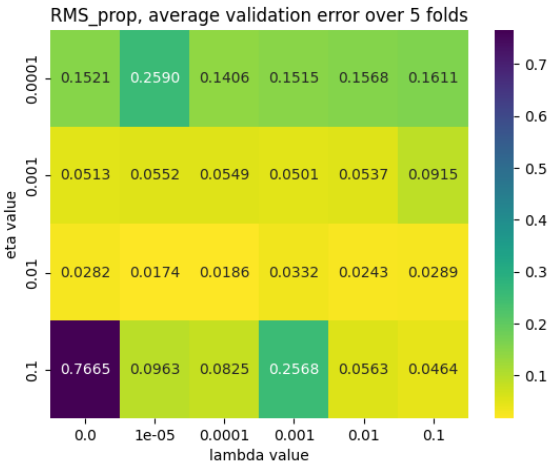

Figure 16: Heatmap of the RMSProp scheduler

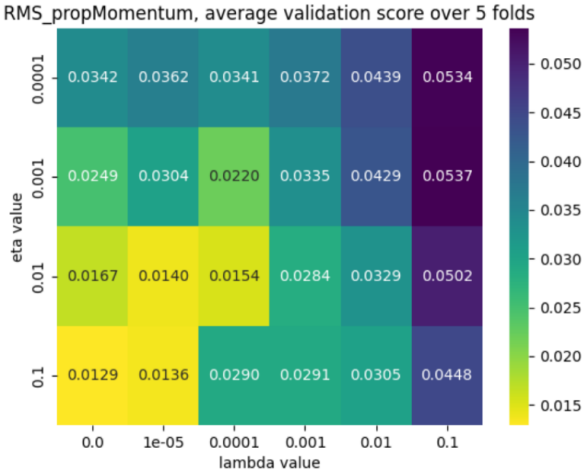

Figure 17: Heatmap of the RMSProp scheduler with Momentum added

Figure 18: Heatmap of the Adagrad scheduler

Figure 19: Heatmap of the Adagrad scheduler with Momentum added

Figure 20: Heatmap of the ADAM scheduler

These heatmaps show the **resulting mean squared error** (MSE) of gradient descent for **500 epochs** on the *Franke function*, with each heatmap representing a different scheduler. We can see that for most of them, a **learning rate of either 0.1 or 0.01 is optimal** and a **regularization parameter in the range** 10^{-5} **- 0.01**. This also counts for the lowest error of all the heatmaps, which is **0.0105 when using momentum**.

Note : We have also implemented ADAM with Momentum, but this scheduler has given us **very high validation error**. Indeed, ADAM has a tendency to **overfit our data when the learning rate becomes too big**, which can be seen on Figure 20. While this is only the case for $\eta = 0.1$, it was a **much worse case** when Momentum was added, so we decided to scrape out that scheduler entirely.

We can also compare our results with Scikit-Learn's implementation. Indeed, their librairies introduce *MLPRegressor*, a modern multilayer perceptron that is made of **input, output and hidden layers** that communicate together, just like a feed forward neural network does. However, not all gradient descent schedulers are implemented, so we will only compare the results of **Plain Gradient Descent (Constant) and ADAM**.

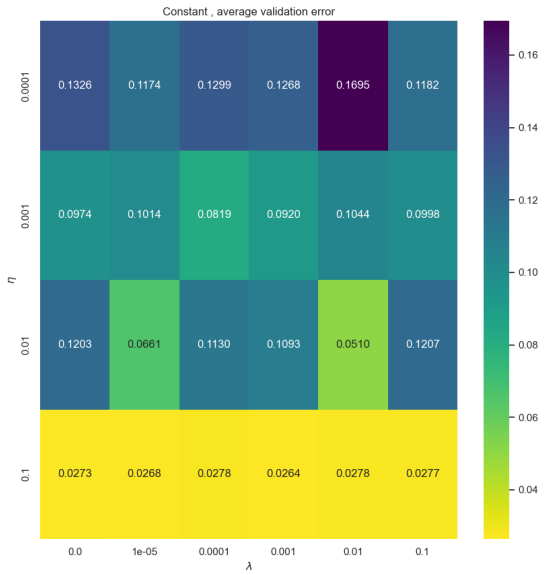

Figure 21: Heatmap of the normal Stochastic Gradient Descent from the Scikit-Learn library

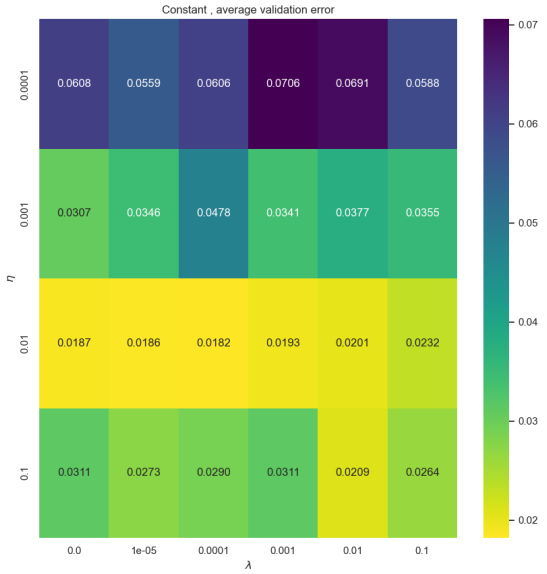

Figure 22: Heatmap of the Stochastic Gradient Descent with Momentum added from the Scikit-Learn library

Although the error values are not exactly the same, we can see that the pattern observed in our gradient descent implementation **matches the one on Scikit-Learn's**. Indeed, for the Constant GD part, the error decreases with the learning rate.

The same can be said with ADAM, in the sense that the validation error decreases with the learning, but increases back up at $\eta = 0.1$; a **clear indication of overfitting**.

Overall, it seems that our implementation properly matches what Scikit-Learn computes, hence does the **correct computation**s.

5.2 Finding the most suitable batch size

Here, we need to get an overview of what batch size to use when fitting our model. For this, we will Introduce a **single hidden layer with nodes** based on our input. This yields the following graph:

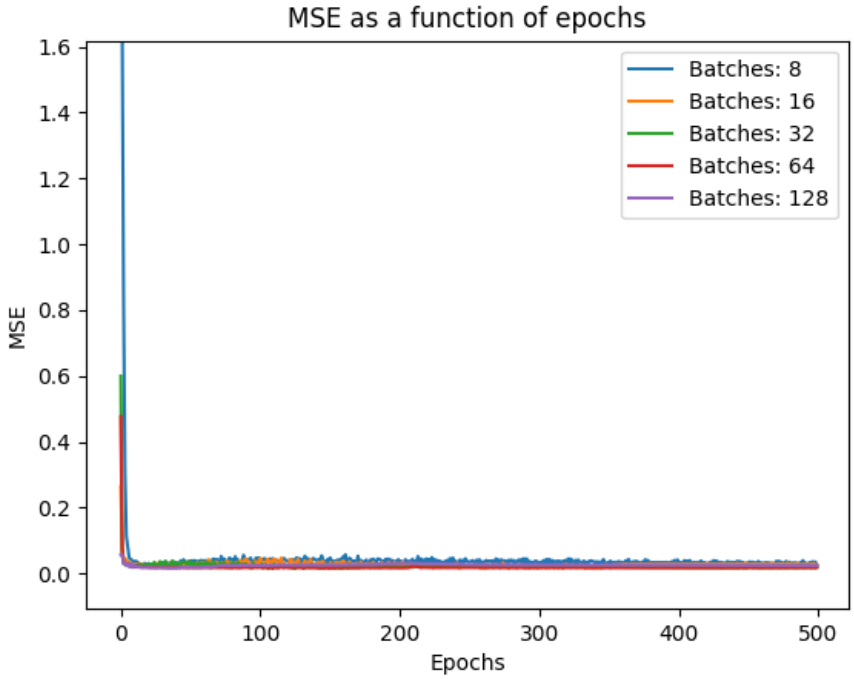

Figure 23: Mean Squared Error on ADAM Scheduler, given multiple batch sizes.

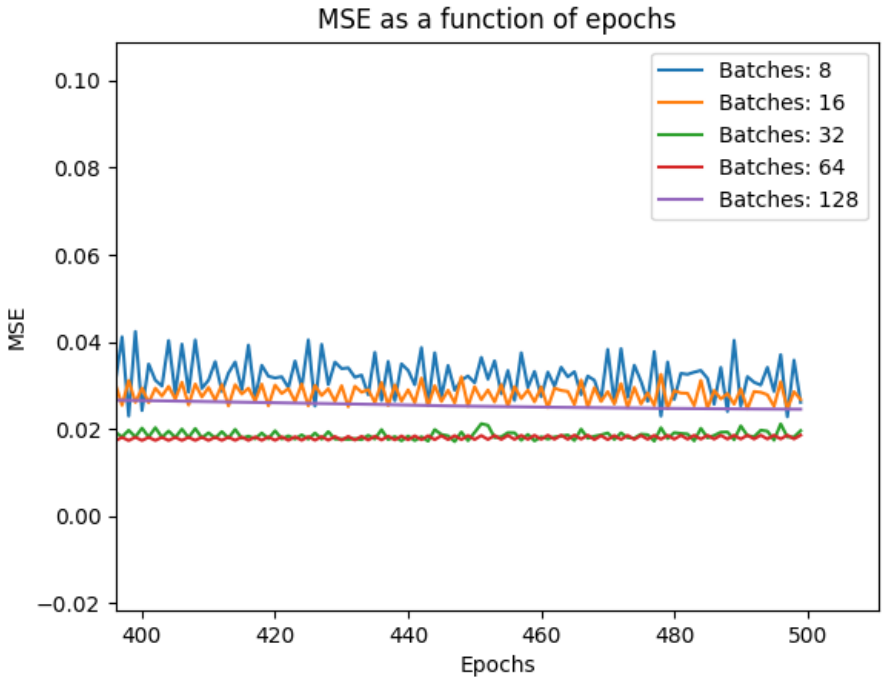

Figure 24: The same MSE, but zoomed on the bigger epoch sizes.

It seems the best performance is achieved at **32-64 batches**. Even though using 64 batches is slightly better, with it comes **increased running time** which will be a detriment when running more complex architectures in the coming tasks. Therefore, **we will stick with 32 for now**. To arrive at these batch sizes and hyperparameters, we used *500 epochs*, because we think it's a decent amount to use in order to get a better picture of each scheduler's behavior moving forward, as we aim to introduce hidden layers and in doing so increase the overall complexity of our architecture.

5.3 Selecting the best amount of Neurons in Hidden Layers

Now that all schedulers have been studied, we now have to implement them in a FFNN, and as aforementioned, this is done by introducing **hidden layers** to our architecture, as that is one of the key features of a feed-forward neural network. In doing so, it is important that we optimize our hyperparameters in tandem with both the number of hidden layers, as well as the number of nodes within each layer.
We start off by finding the optimal parameters for each individual scheduler. The figures 25 and 26 include heatmaps of our best performing schedulers using a **single hidden layer** with $X.shape[1] + \frac{1}{2}$ (where X.shape[1] is the amount of features in X) nodes and the **sigmoid activation function** :

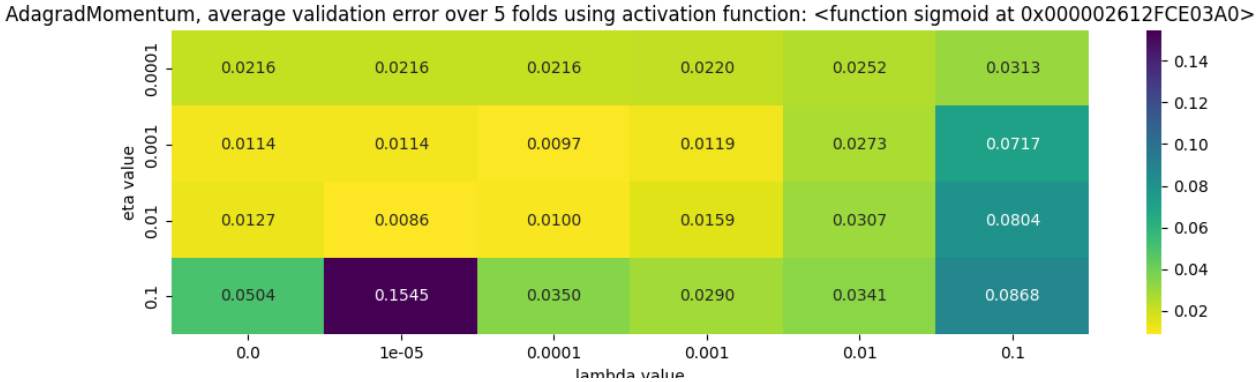

Figure 25: Heatmap of the Adagrad scheduler with Momentum with a hidden layer applied.

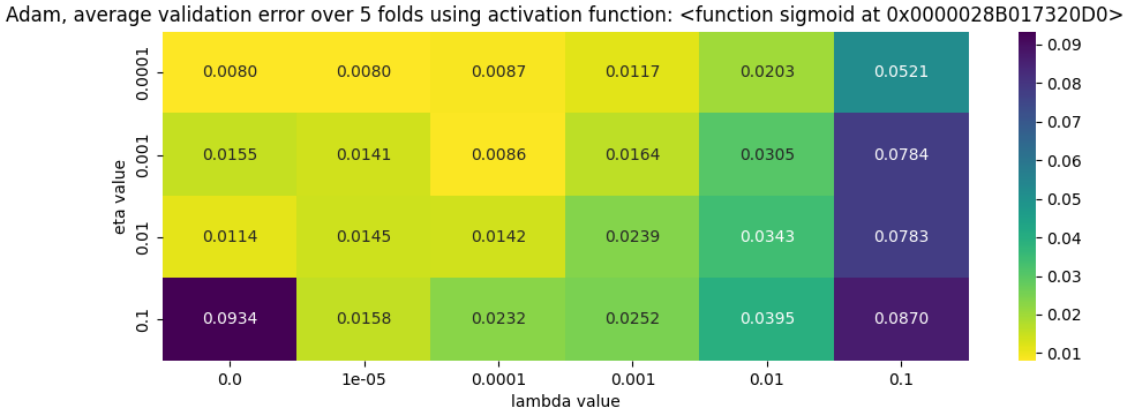

Figure 26: Heatmap of the ADAM scheduler with Momentum with a hidden layer applied.

In doing so, it is immediately apparent that Adam is the superior gradient descent scheduler with hidden layers in the picture. We will thus proceed with Adam at a learning rate of $\eta = 0.0001$ and a lambda of $\lambda = 10^{-5}$.
Now that these parameters have been fixed, we use them to optimize the amount of nodes in a single layer, with possible values being power of twos up to 128, and including nodes based on input features. After this, we try n hidden layers with the best performing amount of nodes up to 4. These computations can be found in the file *optimize_nodes.py* and a more detailed description can be found here 7.2

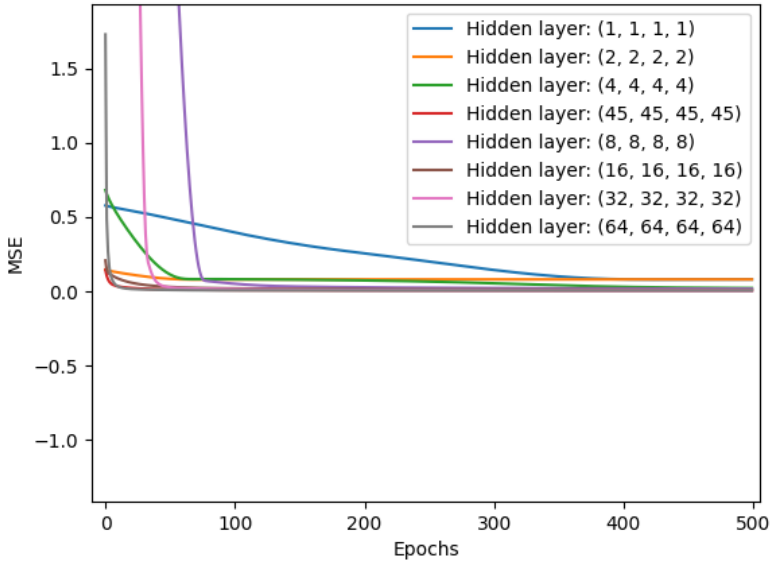

Figure 27: MSE of ADAM given the number of neurons per hidden layers

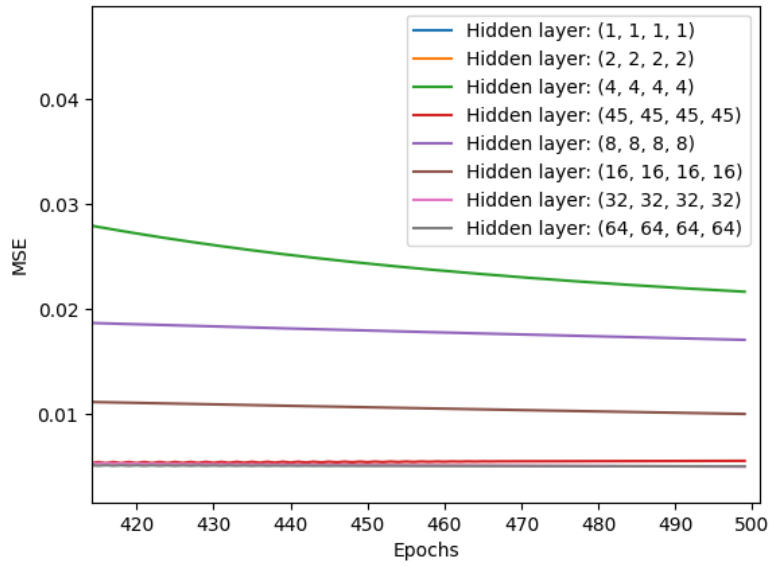

Figure 28: MSE of ADAM given the number of neurons per hidden layers, zoomed on the bigger epoch sizes.

We arrive at the conclusion that **a hidden layer of (64, 64, 64, 64) nodes is optimal**. As for the output layer, using the **identity function** seems like a reasonable choice as this is a *regression case*. By that we mean that the predicted numerical values without any transformations is **what we want as output**.
Comparing our implementation with the SciKit-Learn implementation **MLPRegressor** gives the following graph :

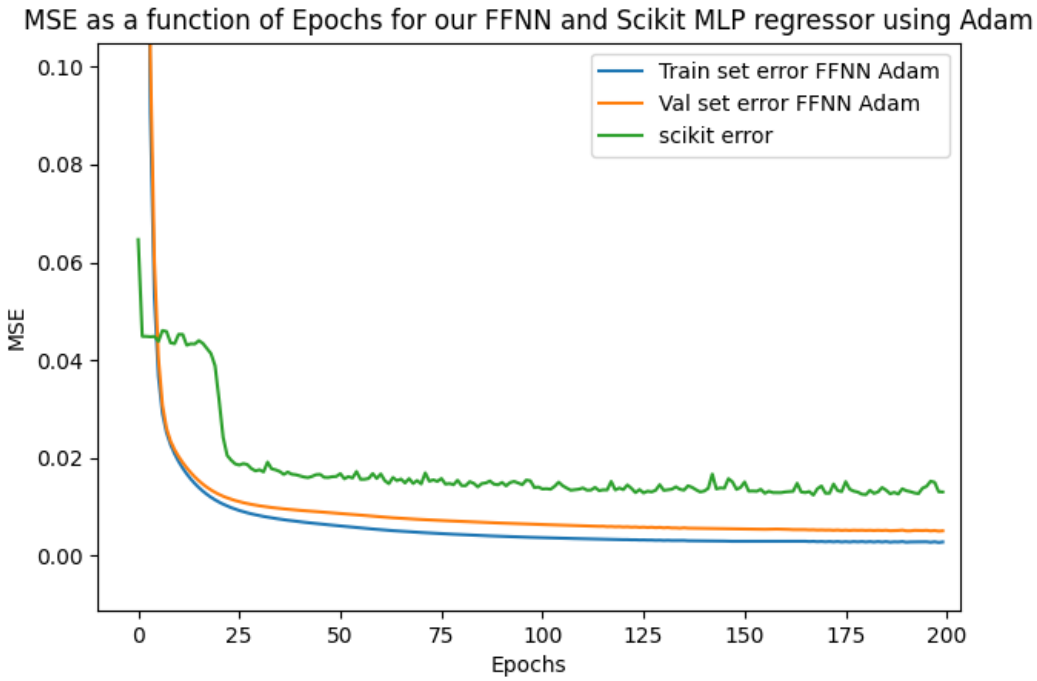

Figure 29: MSE given the number of neurons per hidden layers, this time compared with Scikit-Learn's implementation.

The MSE values do not match, but we can nonetheless observe the same conclusion as with our implementation. Indeed, we can see that the **MSE error decreases as the epochs size increases, while converging at around 50 epochs**. The Scikit-Learn implementation seems to vary a little bit more, as its plot is not as smooth, but that can be explained by the fact that we implemented **cross-validation** alongside our computations.

5.4 Selecting the best Activation Function

Now we will repeat the process from the subsection 5.3, but now also with the activation functions **RELU** and **Leaky RELU** in the hidden layer and their respective optimal hidden layers and parameters. Plotting them gives the following graph:

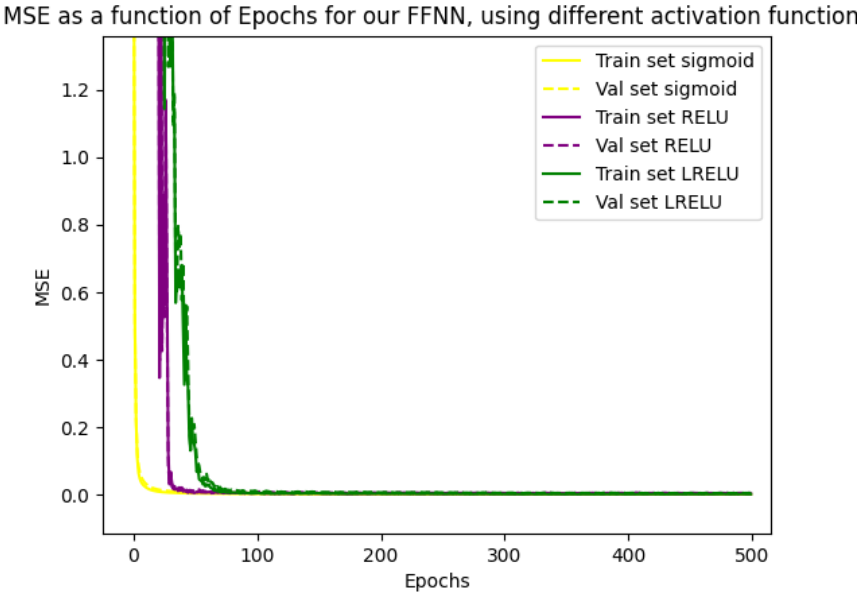

Figure 30: MSE of ADAM given the activation function

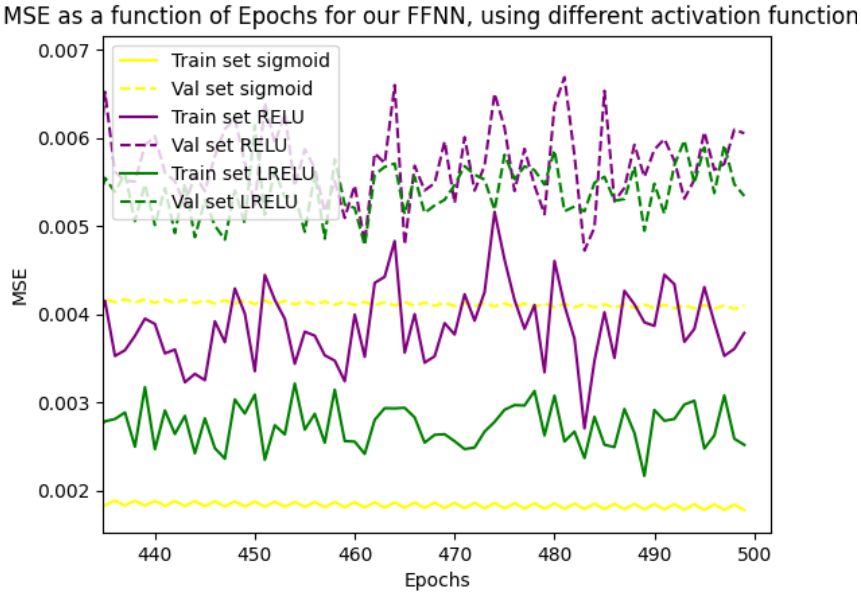

Figure 31: MSE of ADAM given the activation function, zoomed on the bigger epoch sizes.

Here it is evident that the **sigmoid is the best performing activation function for the hidden layer**. The reason for this could be that the Franke function we use has *smooth transitions* and *regions close to 1 or 0* which are patterns the **sigmoid is good at capturing**. The effects of the noise we added to it likely helps the sigmoid as well as it is **more "smooth" than both LRELU and RELU**, which are more rigid and therefore more sensitive to noise. (see figure 7)

We now want to compute the R2 score of the network with sigmoid as the activation function and the identity function as the output function. We rewrote the R2 score such that the best values are the ones close to 0 to use it as a cost function :

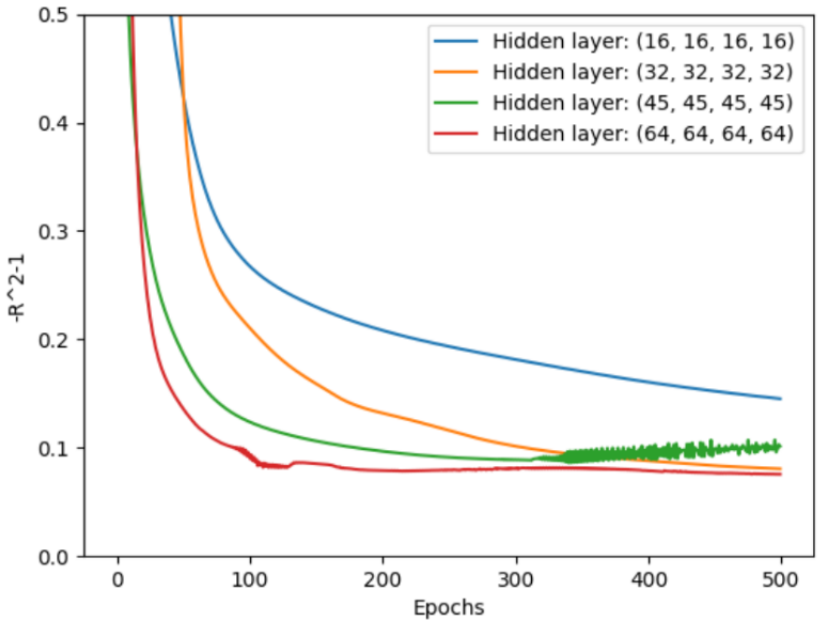

Figure 32: R^2 Score of ADAM using the **sigmoid function** as an activation function and given multiple sizes of hidden layers.

As the amount of nodes increases the R^2 score becomes better. A hidden layer of (64, 64, 64, 64), which we found to be optimal with MSE in Figure 28, gives a R^2 score around 0.9. As the R^2 score needs to be as close to 1 as possible, this score is quite good.

5.5 Selecting the best Output Function

Coming back to the sigmoid as the activation function in the hidden layers, and Adam with $\eta = 0.0001$ and $\lambda = 10^{-5}$, as this gave a very low MSE before. This time however, we test if using sigmoid as the output function instead of the identity function makes a difference:

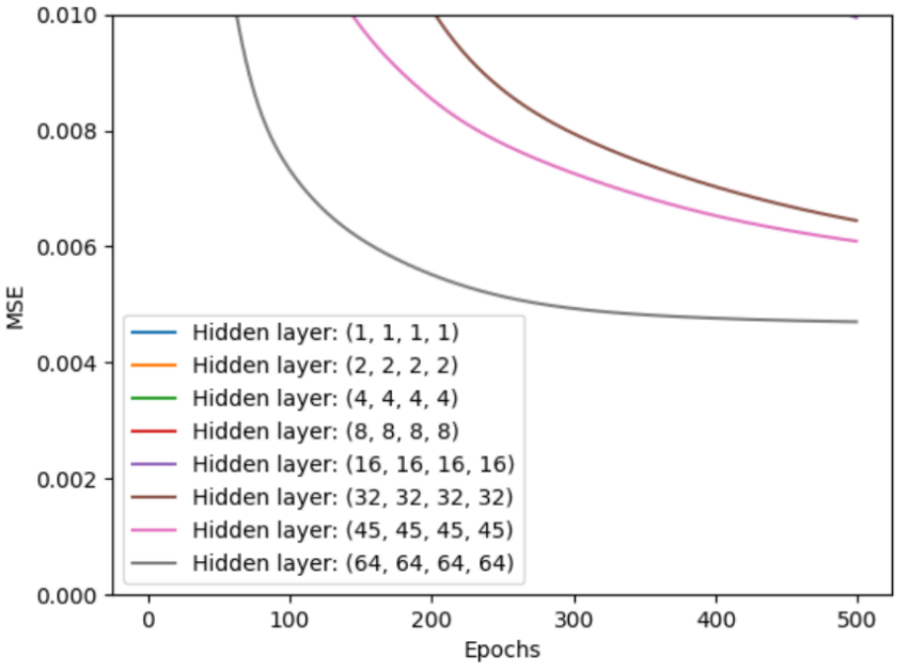

Figure 33: MSE score of ADAM, this time using sigmoid as an output function.

The MSE error when using the sigmoid function as the output function gives a MSE around 0.005. Comparing this Figure 33 with Figure 28 when we previously used the identity function as output function, we realize that **their MSE error are about similar**, hence it is best to leave the output function as identity, as it allows us to do less computations for pretty much the same results.

5.6 Comparing with Ridge Regression

In our first project, we concluded that **Ridge regression was the best linear regression method to use** after comparing it to OLS and lasso regression. To recall, Ridge regression on the Franke function with noise gave us the following results :

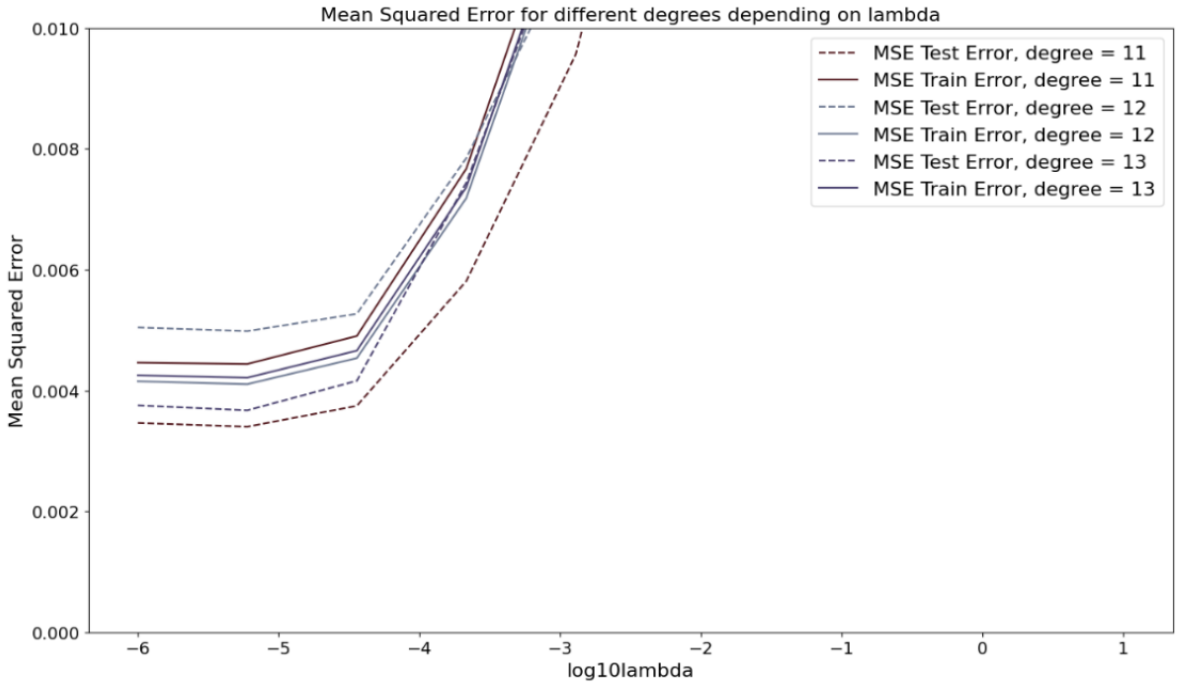

Figure 34: MSE score of Ridge Regression on Franke's Function. Since we are still using the same dataset, we can therefore use values that we have previously computed.

Focusing on Figure 34, we choose to look at degree 13, as it's the **only degree of which its train error is below its test error**. As these errors are in between 0.004 and 0.006, we can conclude that using our neural network with sigmoid as the activation function is a better model, since it gives errors between 0.002 and 0.004 (see figure 31), which are lower errors.

5.7 Performance on Wisconsin's Breast Cancer Dataset

Now what we've discussed and optimized everything we wanted on our Franke's Function regression problem, let's now reapply the same procedure on *Wisconsin's Breast Cancer Dataset*. As mentionned in 4.2, this is a **binary classification problem** that we are gonna look for now.

To start off, we will employ our **ADAM sigmoid implementation** on the classification case in the form of the The Wisconsin Breast Cancer Dataset. By implementing functionality for confusion matrices in our FFNN class and using the **ADAM method with the optimized parameters for Franke's Function** (more details on 7.3, we can get the following output:

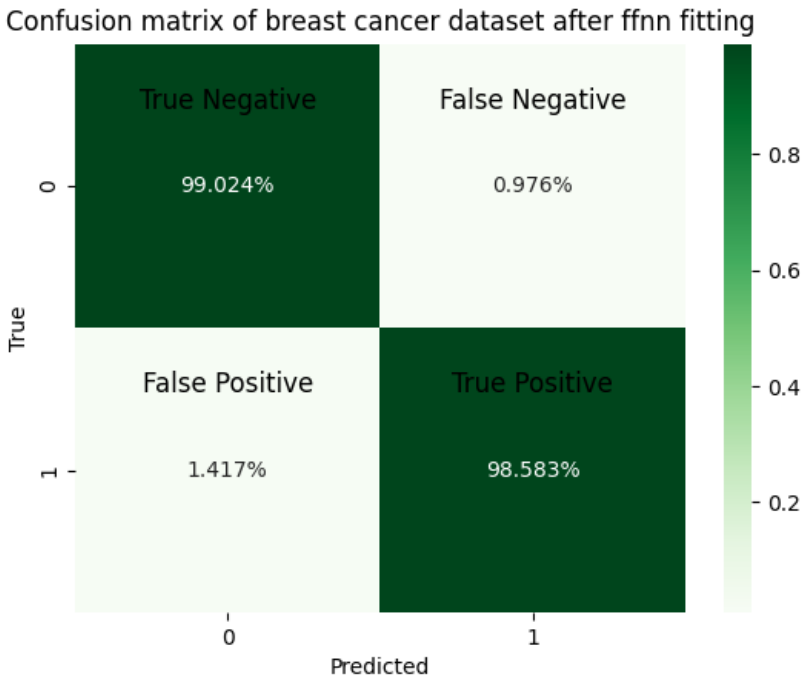

Figure 35: Confusion matrix our Adam scheduler on Wisconsin's Dataset. With Franke's Function parameters, we achieved an **accuracy of 98.80%**

This is a very good result **for parameters that are optimized for regression on a noised Franke Function**, with an overall accuracy of 98.80%. Given our task, we still want to try to optimize it further (recall that our goal is to detect breast cancer, so we want to **maximize the precision**).
Optimizing individual activation functions for the classification case for both hidden layer sizes and nodes, and hyperparameters lambda and learning rate (refer to the "figs/cancer optimization" folder in the github repo for more in-depth results) yields the following confusion matrices:

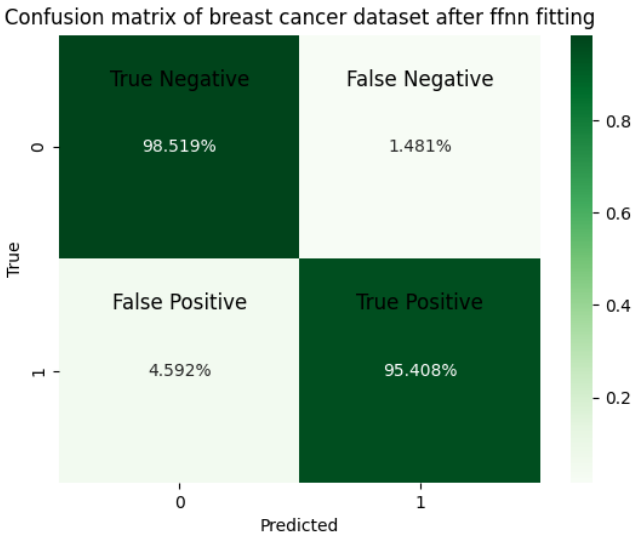

Figure 36: Confusion matrix of our Adam scheduler on Wisconsin's Dataset. Using sigmoid with a learning rate of 0.01, a lambda of 0.0001 and a hidden layer of (64, 64, 64, 64), we achieved an **accuracy of 96.96%**, lower than our previous implementation 35

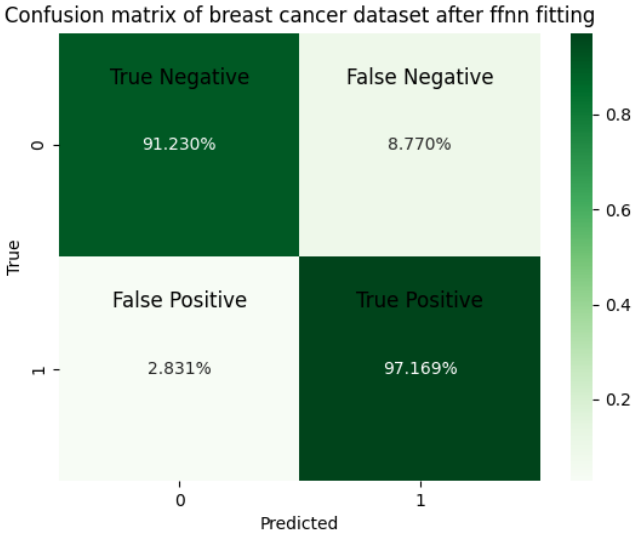

Figure 37: Confusion matrix of our Adam scheduler on Wisconsin's Dataset. Using RELU with a learning rate of 0.001, a lambda of 10^{-5} and a hidden layer of (16, 16, 16, 16), we achieved an **accuracy of 94.20%**, again, lower than both previous implementations

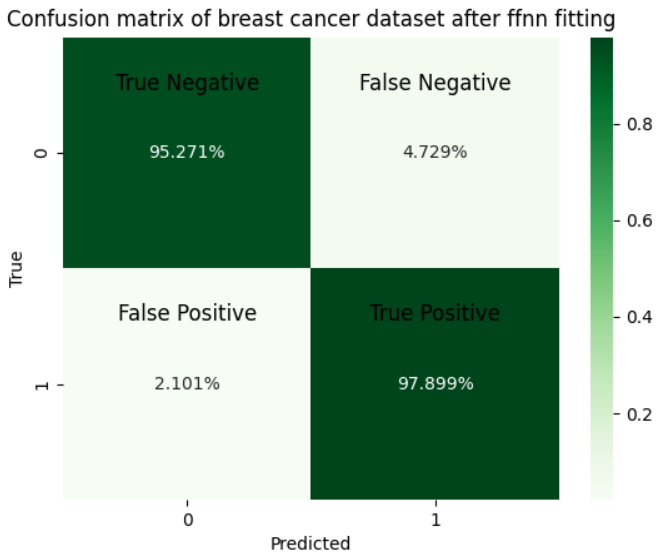

Figure 38: Confusion matrix of our Adam scheduler on Wisconsin's Dataset. Using LRELU with a learning rate of 0.001, a lambda of 0.0001 and a hidden layer of (8, 8, 8, 8), we achieved an **accuracy of 96.59%**, the best of the three tries, but still underperforming compared to 35

It is obvious **none of these performed better than our original parameters** obtained through optimizing for the Franke function regression case. Therefore we will try to optimize the architecture around the optimal hyperparameters from the regression case.
We will try different sizes in the hidden layer around the value 64, as it is our **best performer thus far**, as well as deciding 4 hidden layers is optimal, as it has given the **best results so far for the classification case**. The sizes we try for each hidden layer are in the list [50, 64, 78, 92], and now we try different values for each layer at the same time to broaden our search. Interestingly, the neural network still gives the best results with **all layers having the same number of nodes**, and we ultimately get the **hidden layer [78, 78, 78, 78] as the best performer**. These results are obtained in the python script *"optimize_classification_final.py"*, with the description in 7.4.

The resulting confusion matrix is shown below:

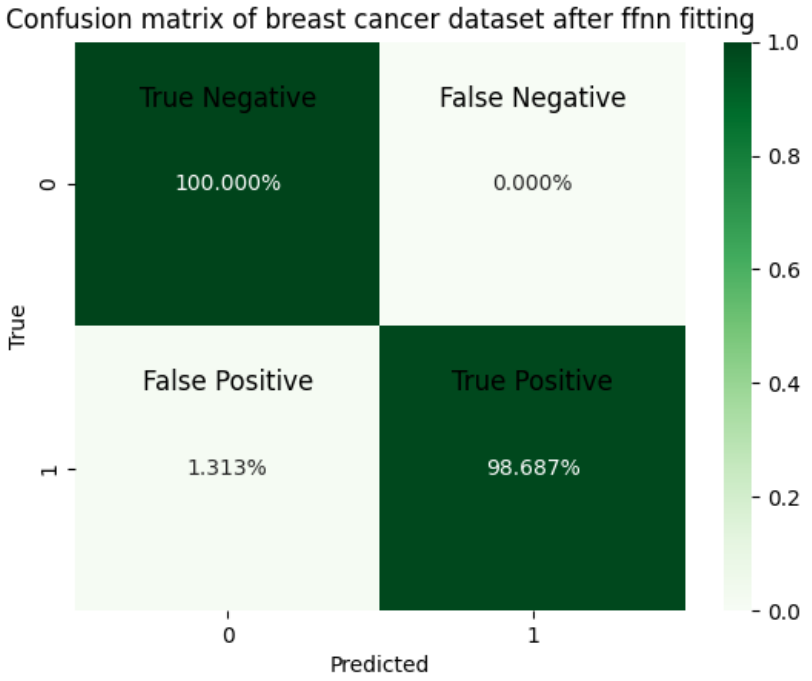

Figure 39: Confusion matrix our Adam scheduler on Wisconsin's Dataset. Using $\eta = 0.0001$, $\lambda = 10^{-5}$ and [78, 78, 78, 78] as hidden layer, we achieved an accuracy of 99,35%

We are pleased to achieve a **100% true negative rate** (handy when diagnosing cancer), and an overall accuracy of 99,35%. The Neural Network likely **performs better with symmetric hidden layers** because features in breast cancer dataset are reflected across a symmetry axis. By this, we mean that the underlying features of both benign or malignant samples have some **degree of symmetry in their patterns**. When the network's architecture is also symmetric it is more susceptible to generalizing these symmetrical patterns.

Comparing our results with Scikit-Learn's library, we get the following confusion matrix :

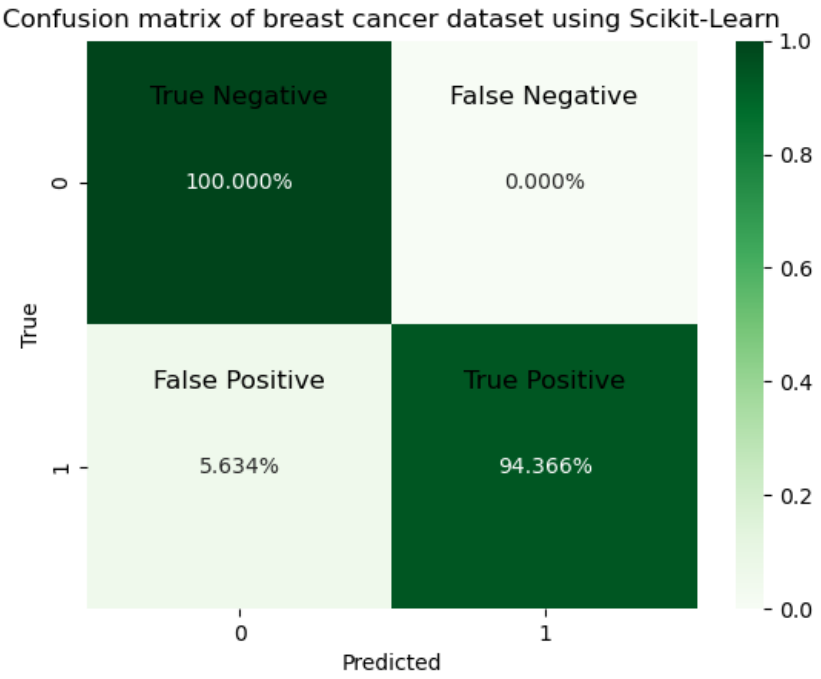

Figure 40: Confusion matrix our Adam scheduler on Wisconsin's Dataset. Using the same parameters as in 39 but implementing them on **Scikit-Learn's MLPClassifier**, we achieved an accuracy of 97.18%

The results are slightly worse than our FFNN implementation, but the overall conclusion remains the same : a **remarkable 97.81%** accuracy and most importantly a **100% precision**. This allows us to conclude that our FFNN is reliable to properly fit and predict Wisconsin's dataset with these specific parameters.

5.8 Comparing with Logistic Regression

Lastly, we will compare our Neural Network model with the *Logistic Regression* method, as we have mentioned that it performs very well on binary classification cases. 3.12 The Logistic Regression can easily be obtained from our FFNN, as it is simply defined by a **gradient descent with batches**, and a **Logistic Regression cost function**. Since we are looking on a brand new scheduler, it is necessary to find once more the best suitable hyperparameters, hence we will plot their heatmap the same way as we did in 5.1. Also, because we are studying a *classification problem*, we will now display in the heatmap the **accuracy score**, rather than the error score, as it is easier to compute.
Tweaking some parameters in our model, we obtain the following results :

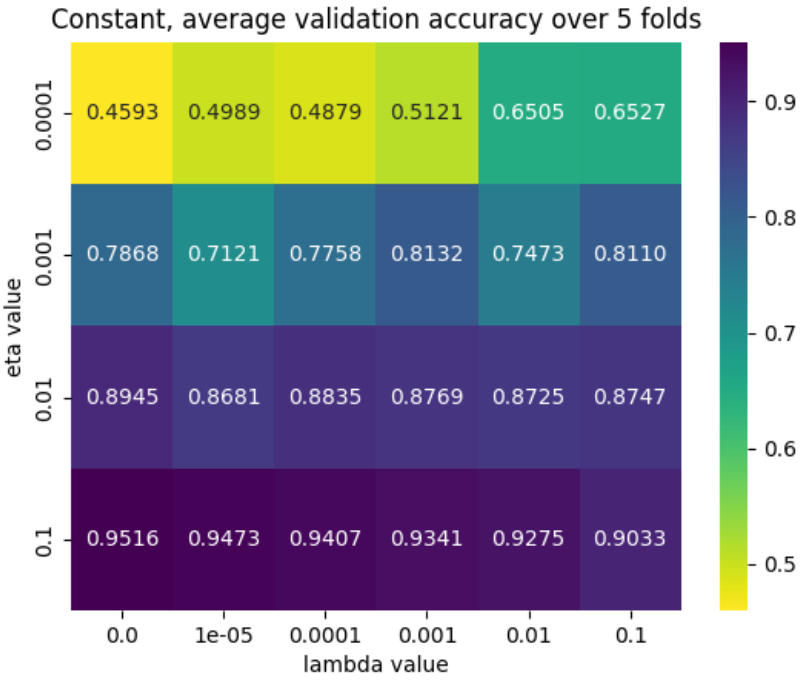

Figure 41: Heatmap of our **Logistic Regression scheduler**, implemented with our FFNN class.

And as we can see, the highest accuracy is achieved with $\eta = 0.1$ and $\lambda = 0$, for an accuracy of 95.16%. Is it however **not as good as the results we have previously obtained**, showing that our model with proper parameters regularization can **surpass even the Logistic Regression**. This can also be due to the fact that, although the output is binary, there are a lot of variables to study from, transforming this study case into a **deep learning problem**. In this situation, neural networks should be giving better results than Logistic Regression (as mentioned in 3.12).

Scikit-Learn also has a library to implement Logistic Regression that gives the best possible accuracy score, no matter the parameters (since it will find and optimize them itself). Looking at the score of the LogisticRegression() class, we obtain an **accuracy of 96.49%**, which is pretty close to what we have observed in our logistic regression implementation (95.16%).

We can confirm that it optimized the parameters itself by looking at the **heatmap of the MLPClassifier** implemented by Scikit-Learn :

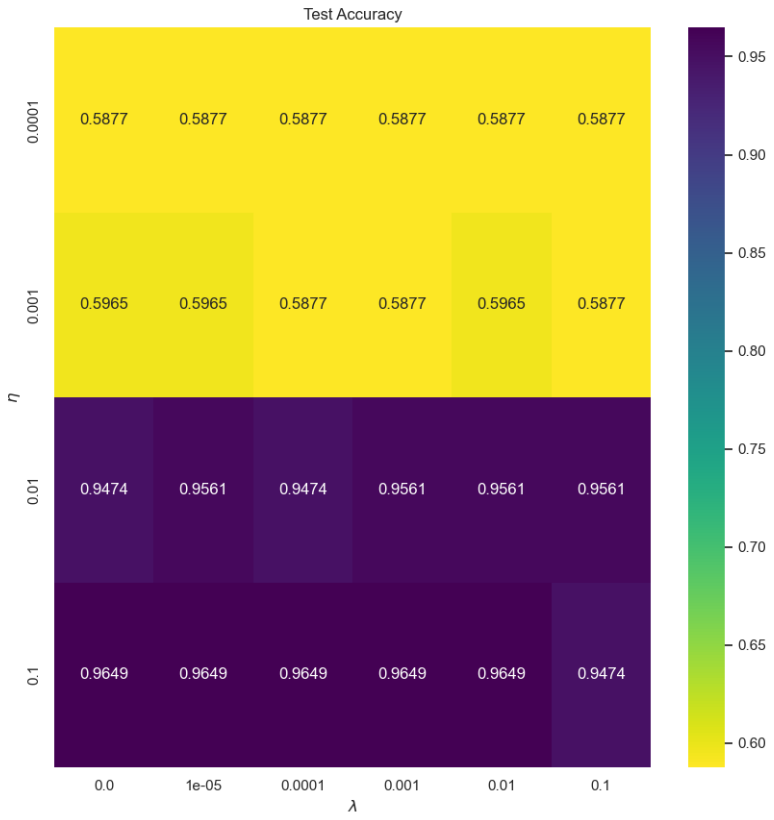

Figure 42: Heatmap of our **Logistic Regression scheduler**, implemented with Scikit-Learn's MLPClassifier model.

If we truly want to observe the behavior of our logistic model, we can instead look at **Tensorflow's implementation**, which allows us to set a lot more variables than Scikit-Learn. Running it with the same parameters as in our FFNN, the heatmap results in :

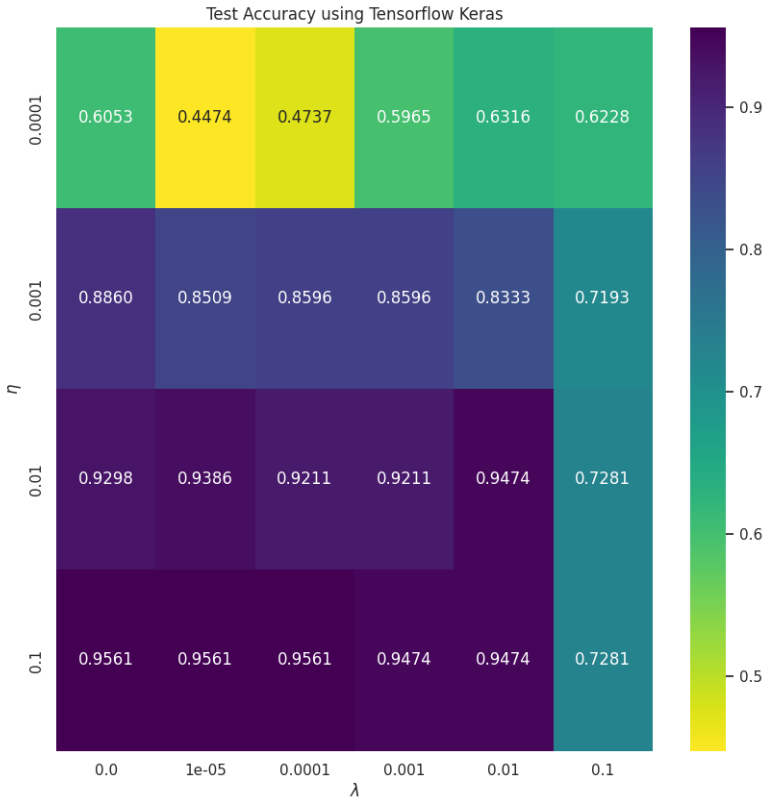

Figure 43: Heatmap of our **Logistic Regression scheduler**, implemented with Tensorflow's Keras Sequential model.

And once again, we can see that the best accuracy is found at the same η and λ as we have observed in our FFNN, with a **similar accuracy as well**, thus validating our implementation.

6 Conclusion

In this report, we have studied a classification and regression cases, with the goal of achieving the best results with the help of various supervised learning algorithms. The algorithms were optimized for both hyperparameters and architectures alike for every case. For both cases, the FFNN was exceptional at helping us produce the desired results. FFNNs have a flexible architecture that can be optimized for a variety of cases, meaning it **can understand complex relationships** in input data.
FFNNs proved effective in both regression and classification tasks, with hyperparameters and architecture that could be tuned to handle both. The cons of having many aspects that need careful tuning, is that it is a **time consuming process** that **does not always lead to improvement**. It is also a **computationally expensive algorithm**, especially as the size of the dataset increases and the architecture becomes more complex.
We used ridge regression in our previous project, which proved **more effective runtime-wise but not results-wise**. Ridge regression is a stable algorithm that introduces a regularization parameter (λ), that helps deal with the problems non-regularization regression algorithms such as OLS encounter with multicollinearity. What holds it back in comparison with FFNNs is the fact that it **assumes a linear relationship** between features and the target variable.
Logistic regression is also an algorithm we have employed. Logistic regression is **simpler and more efficient** than an FFNN due its less complex architecture, making it well suited for binary classification tasks, though it **may not generalize nonlinear relationships** in the data compared to FFNNs.
For both the regression and classification case, the FFNN with Adam gradient descent and sigmoid activation function for the hidden layer and proper tuning proved the best algorithm for obtaining the best results. Its ability to model non-linear decision boundaries allowed it to succeed above the rest.
We could push our results even further by exploring the multitude of activation and cost functions that exist nowadays, but we cannot tell if they will prove to be more useful than those mentioned in this project, and they might only do so at the cost of time computation.

7 Appendix

As for our previous project, all of our codes are available on our GitHub, but the most important implementations are explained below.

7.1 Cross Validation

We added Cross Validation in our model in order to detect and prevent overfitting. As soon as we select a scheduler, the model will apply a K-fold Cross Validation and compute the calculations, making them slightly more accurate. This code is found within the FFNN implementation :

```
for fold in range(folds):
    start_index = fold * batch_size
    end_index = (fold + 1) * batch_size if fold < folds - 1 else None
    X_val_fold = X[start_index:end_index, :]
    t_val_fold = t[start_index:end_index, :]

    X_train_fold = np.delete(X, range(start_index, start_index + X_val_fold.shape[0]), axis=0)
    t_train_fold = np.delete(t, range(start_index, start_index + t_val_fold.shape[0]), axis=0)

    # Scale the data
    X_train_fold_scaled = min_max_scaler.fit_transform(X_train_fold)
```

```
    X_val_fold_scaled = min_max_scaler.transform(X_val_fold)

    cv.append((X_train_fold_scaled, t_train_fold, X_val_fold_scaled, t_val_fold))
```

7.2 Optimizing the number of nodes in a hidden layer

We have implemented a code that allows to know the best amount of nodes given a certain amount of layers (that we have also optimized).

```
n_layers_scores, layer = optimize_n_hidden_layers(X_train, z_train, folds, scheduler, batches, epochs, la


for i in range(len(n_layers_scores)):
    lab = f"Hidden layer: {layer[i]}"
    plt.plot(n_layers_scores[i]["val_errors"], label = lab)
plt.legend()
plt.xlabel("Epochs")
plt.ylabel("MSE")
plt.show()

#getting the length for next optimization
min_error = float('inf')
best_layer_length = None

for i in range(len(n_layers_scores)):
    current_error = min(n_layers_scores[i]["val_errors"])

    if current_error < min_error:
        min_error = current_error
        best_layer_length = len(layer[i])

#Try powers of two in nodes
nodes_to_try = np.power(2, np.arange(7))
#add the number of nodes we initially tested with
nodes_to_try = np.insert(nodes_to_try, 6, nodes_in_layer)

n_nodes_scores, node_layer = optimize_n_nodes(X_train, z_train, folds, scheduler, batches, epochs, lam, b

for i in range(len(n_nodes_scores)):
    lab = f"Hidden layer: {node_layer[i]}"
    plt.plot(n_nodes_scores[i]["val_errors"], label = lab)
plt.legend()
plt.xlabel("Epochs")
plt.ylabel("MSE")
plt.show()
```

7.3 Implementing Confusion Matrix

Because the classification problem only takes values of 0 or 1, we need to implement another way in our FFNN to compute the error (here the misclassification error). This is the goal of the following code :

```
cv_scores = None
        if self.classification:
            confusion_matrix = np.zeros((2, 2))
        else:
            [...]
        for X_train, t_train, X_val, t_val in cv:
            self.reset_weights()
            scaled_batches = max(int(batches / (X.shape[0] / X_train.shape[0])), 1)
            scores = self.fit(X_train, t_train, scheduler, scaled_batches, epochs, lam, X_val, t_val)

            if self.classification:
                confusion = self.calc_confusion(t_val, self.predict(X_val))
                confusion_matrix += confusion / folds
            if cv_scores is None:
                cv_scores = {key: value / folds for key, value in scores.items()}
            else:
                cv_scores = {key: cv_scores[key] + value / folds for key, value in scores.items()}

        if self.classification:
            cv_scores["confusion_matrix"] = confusion_matrix

        return cv_scores
```

7.4 Optimizing the classification parameters

Like for the number of neurons, we want to find the best hidden layer configuration for our classification problem, this way we get the best possible results. That is implemented with the following code :

```
for i in range(len(starting_layer)):
curr_scores = np.zeros(len(try_nodes))

for j, nodes in enumerate(try_nodes):
    starting_layer[i] = nodes
    hidden_layer = tuple(starting_layer)

    ffnn = FFNN((X.shape[1], *hidden_layer, 1), seed=seed, cost_func=CostLogReg, output_func=sigmoid, hid

    scores = ffnn.cross_validation(X, z, folds, adam, batches, epochs, lam)
    acc = np.max(scores["val_accs"])

    curr_scores[j] = acc
    print(f"Current hidden layer {hidden_layer}")
    if acc > best_accuracy:
        best_accuracy = acc
```

```
        best_nodes = nodes

starting_layer[i] = best_nodes
```

References

[1] abhishekaslk. *Epoch in Machine Learning*. Accessed 10 Nov 2023. June 2023. URL: https://www.geeksforgeeks.org/epoch-in-machine-learning/.

[2] Sam Anan. *Who created the sigmoid activation function, and how was it developed?* Accessed 13 Nov 2023. 2020. URL: https://www.quora.com/Who-created-the-sigmoid-activation-function-and-how-was-it-developed.

[3] Pragati Baheti. *3 Types of Neural Networks Activation Functions*. Accessed 13 Nov 2023. May 2020. URL: https://www.v7labs.com/blog/neural-networks-activation-functions#3-types-of-neural-networks-activation-functions.

[4] Pragati Baheti. *Activation Functions in Neural Networks [12 Types and Use Cases]*. Accessed 10 Nov 2023. May 2021. URL: https://www.geeksforgeeks.org/epoch-in-machine-learning/.

[5] Jason Brownlee. *A Gentle Introduction to Cross-Entropy for Machine Learning*. Accessed 16 Nov 2023. 2022. URL: https://machinelearningmastery.com/cross-entropy-for-machine-learning/.

[6] Stefania Cristina. *The Chain Rule of Calculus for Univariate and Multivariate Functions*. Accessed 15 Nov 2023. 2022. URL: https://machinelearningmastery.com/the-chain-rule-of-calculus-for-univariate-and-multivariate-functions/.

[7] Niklas Donges. *Gradient Descent in Machine Learning: A Basic Introduction*. Accessed 13 Nov 2023. May 2023. URL: https://builtin.com/data-science/gradient-descent.

[8] Manu Garg et al. "Email Spam Detection Using Logistic Regression". In: *4UG Scholar Computer Science and Engineering Meerut Institute of Engineering and Technology* (2022). Accessed 17 Nov 2023. DOI: 10.47750/pnr.2022.13.S10.245.

[9] Veena Ghorakavi. *Neural Networks — A beginners guide*. Accessed 10 Nov 2023. Apr. 2023. URL: https://www.geeksforgeeks.org/neural-networks-a-beginners-guide/.

[10] harkiran78. *Artificial Neural Networks and its Applications*. June 2023. URL: https://www.geeksforgeeks.org/artificial-neural-networks-and-its-applications/.

[11] Morten Hjorth-Jensen. *ADAM Optimizer*. Accessed 13 Nov 2023. Oct. 2023. URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week39.html?highlight=momentum#adam-optimizer.

[12] Morten Hjorth-Jensen. *Deriving the back propagation code for a multilayer perceptron model*. Accessed 10 Nov 2023. Oct. 2023. URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapter9.html?highlight=neural%20network#deriving-the-back-propagation-code-for-a-multilayer-perceptron-model.

[13] Morten Hjorth-Jensen. *Feed-forward neural networks*. Accessed 10 Nov 2023. Oct. 2023. URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapter9.html?highlight=neural%20network#feed-forward-neural-networks.

[14] Morten Hjorth-Jensen. *Gradient Descent Methods and Neural Networks*. Accessed 10 Nov 2023. Oct. 2023. URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week40.html#neural-network-types.

[15] Morten Hjorth-Jensen. *Momentum based GD*. Accessed 13 Nov 2023. Oct. 2023. URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week39.html?highlight=momentum#momentum-based-gd.

[16] Morten Hjorth-Jensen. *RMS prop*. Accessed 13 Nov 2023. Oct. 2023. URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week39.html?highlight=momentum#rms-prop.

[17] Morten Hjorth-Jensen. *Second moment of the gradient*. Accessed 10 Nov 2023. Oct. 2023. URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week40.html#second-moment-of-the-gradient.

[18] Morten Hjorth-Jensen. *Steepest Descent*. Accessed 13 Nov 2023. Oct. 2023. URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week39.html#steepest-descent.

[19] Morten Hjorth-Jensen. *Stochastic Gradient Descent (SGD)*. Accessed 13 Nov 2023. Oct. 2023. URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week39.html#stochastic-gradient-descent-sgd.

[20] Morten Hjorth-Jensen. *The Neural Network*. Accessed 17 Nov 2023. Oct. 2023. URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week43.html#the-neural-network.

[21] Courtlin Holt-Nguyen. *When to Use Logistic Regression vs. Deep Learning for Business Problems*. Accessed 14 Nov 2023. Mar. 2023. URL: https://medium.com/accelerated-analyst/when-to-use-logistic-regression-vs-deep-learning-for-business-problems-589359d49600.

[22] IBM. *What are neural networks?* Accessed 17 Nov 2023. 2023. URL: https://www.ibm.com/topics/neural-networks.

[23] Lili Jiang. *A Visual Explanation of Gradient Descent Methods (Momentum, AdaGrad, RMSProp, Adam)*. Accessed 13 Nov 2023. June 2020. URL: https://towardsdatascience.com/a-visual-explanation-of-gradient-descent-methods-momentum-adagrad-rmsprop-adam-f898b102325c.

[24] Mbali Kalirane. *Gradient Descent vs. Backpropagation: What's the Difference?* Accessed 14 Nov 2023. Apr. 2023. URL: https://www.analyticsvidhya.com/blog/2023/01/gradient-descent-vs-backpropagation-whats-the-difference/.

[25] Kolamanvitha. *Introduction to Optimization*. Accessed 11 Nov 2023. May 2021. URL: https://medium.com/mlearning-ai/lets-say-we-are-left-in-the-middle-of-the-mountain-ranges-blind-folded-and-need-to-find-the-89b42d733cc7.

[26] MathsAtHome. *The Chain Rule Made Easy: Examples and Solutions*. Accessed 15 Nov 2023. 2023. URL: https://mathsathome.com/chain-rule-differentiation/.

[27] Mahsa Mir. *House Prices Prediction Using Deep Learning*. Accessed 17 Nov 2023. 2020. URL: https://towardsdatascience.com/house-prices-prediction-using-deep-learning-dea265cc3154.

[28] Sarang Narkhede. *Understanding Confusion Matrix*. Accessed 10 Nov 2023. May 2018. URL: https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62.

[29] Ayush Pant. *Introduction to Logistic Regression*. Accessed 16 Nov 2023. 2019. URL: https://towardsdatascience.com/introduction-to-logistic-regression-66248243c148.

[30] Rukshan Pramoditha. *Overview of a Neural Network's Learning Process*. Accessed 10 Nov 2023. Feb. 2022. URL: https://medium.com/data-science-365/overview-of-a-neural-networks-learning-process-61690a502fa.

[31] Pytorch. *AUTOGRAD MECHANICS*. Accessed 10 Nov 2023. 2023. URL: https://pytorch.org/docs/stable/notes/autograd.html.

[32] Debneil Sara Roy. "Forecasting The Air Temperature at a Weather Station Using Deep Neural Networks". In: *ScienceDirect* (2020). Accessed 17 Nov 2023. DOI: https://doi.org/10.1016/j.procs.2020.11.005.

[33] Simplilearn. *What is Epoch in Machine Learning?* Accessed 10 Nov 2023. Nov. 2023. URL: https://www.simplilearn.com/tutorials/machine-learning-tutorial/what-is-epoch-in-machine-learning.

[34] W. Street, W. Wolberg, and O. Mangasarian. *Breast Cancer Wisconsin (Diagnostic)*. Accessed 14 Nov 2023. 1993. URL: https://archive.ics.uci.edu/dataset/17/breast+cancer+wisconsin+diagnostic.

[35] Sonja Surjanovic and Derek Bingham. *Virtual Library of Simulation Experiments : Test functions and datasets*. Accessed 14 Nov 2023. 2013. URL: https://www.sfu.ca/~ssurjano/franke2d.html.

[36] Great Learning Team. *Activation Functions in Neural Networks Explained*. Accessed 10 Nov 2023. Dec. 2022. URL: https://www.mygreatlearning.com/blog/activation-functions/.

[37] Ayush Thakur. *ReLU vs. Sigmoid Function in Deep Neural Networks*. Accessed 13 Nov 2023. May 2022. URL: https://wandb.ai/ayush-thakur/dl-question-bank/reports/ReLU-vs-Sigmoid-Function-in-Deep-Neural-Networks--VmlldzoyMDk0MzI.

[38] Sakshi Tiwari. *Activation functions in Neural Networks*. Accessed 10 Nov 2023. Feb. 2023. URL: https://www.geeksforgeeks.org/activation-functions-neural-networks/.

[39] Turing. *Understanding Feed Forward Neural Networks With Maths and Statistics*. Accessed 14 Nov 2023. 2023. URL: https://www.turing.com/kb/mathematical-formulation-of-feed-forward-neural-network.

[40] Chi-Feng Wang. *Automatic Differentiation, Explained*. Accessed 13 Nov 2023. Mar. 2019. URL: https://towardsdatascience.com/automatic-differentiation-explained-b4ba8e60c2ad.

[41] Wikipedia. *Automatic Differentiation*. Accessed 13 Nov 2023. Nov. 2023. URL: https://en.wikipedia.org/wiki/Automatic_differentiation.

[42] Hafidz Zulkifli. *Deep learning - Computation and optimization*. Accessed 10 Nov 2023. Jan. 2017. URL: https://jhui.github.io/2017/01/05/Deep-learning-computation-and-optimization/.

[43] Hafidz Zulkifli. *Understanding Learning Rates and How It Improves Performance in Deep Learning*. Accessed 10 Nov 2023. Jan. 2018. URL: https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10.