Department of Physics

FYS-STK4155 - Applied Data Analysis and Machine Learning

Project 2

Classification and Regression, from linear and logistic regression to neural networks

Authors:
Stig Patey and Jouval Somer

October, 2022

Table of Contents

List of Figures

List of Tables

Abstract

In this project we study regression and classification problems through the development of our own feed-forward neural network (FFNN) code and compare the capabilities of this network with linear regression and logistic regression.

We explore the Franke function for regression and the Wisconsin Breast Cancer (Diagnostic) Data Set (WDBC), for classification. We develop gradient descent methods with associated optimizers and use stochastic gradient descent with the ADAM optimizer to train our neural networks. Finally we use bootstrap methods to achieve confidence in the obtained performance metrics. For the Franke function we achieved a mean MSE of 0.0122 and a mean R2 score of 0.8627, while for the WDBC we achieved a mean accuracy score of 98.0% on 1000 bootstraps, which slightly outperforms the logistic regression which achieved a mean accuracy score of 97.4%.

1 Introduction

The main aim of this project is to study regression and classification problems through the development of our own feed-forward neural network (FFNN) code. Our intent is to verify to what extent FFNN can outperform traditional methods such as linear regression and logistic regression. We will first look at the Franke function for the regression problem before we turn to classification on breast cancer data.

A severe disease for women worldwide is breast cancer. Classification of a tumour, malignant (1) or benign (0) is a binary classification problem, and in this project we will see how our FFNN performs on the Breast Cancer Wisconsin (Diagnostic) Data Set (WDBC). We have looked at the entire data set and also tested principal component analysis (PCA) to reduce the dimensionality. Finally we compare the performance of the FFNN with logistic regression.

Before we discuss the finer details of a neural network, let us contemplate on what sets humans and computers apart. While computers can solve large calculations and follow exact instructions at blazing speed, humans can do a multitude of tasks that a computer has no chance of understanding. The key difference is our ability to learn, how our brains can combine thousands of past experiences into an intuition on how a new task should be solved for the first time.

Long before the computer age humans have desired to create artificial intelligence with human like capabilities. While many were tricked by the automated chess player machine named "The Turk" in the 18th century (Wikipedia 2022), real artificial intelligence had to wait a bit longer. The very concept of neural networks were introduced in 1943 by the neurophysiologist Warren McCulloch and the mathematician Walter Pitts(JL 1943). They presented a simplified computational model of how biological neurons might work together in animal brains to perform complex computations using propositional logic.

Initially the concept of artificial neural networks got inspired by this but today we have departed somewhat from this analogy. As we will see in this project a key feature of neural networks is that they feed information forward from neurons to neurons in layers, and then they learn by sending correction signals based on gradients backwards through the network. After many iterations the model gets better and better, it learns and finally often outperforms earlier methods such as the linear regression methods studied by the authors in an earlier project. (Patey and Somer 2022). A key finding is how well a neural network can approximate functions, as described by the Universal Approximation Theorem.

While the neural network architecture is conceptually simple, it also poses many challenges. As information flow backwards we can get gradients that vanish or explode. This was a key challenge long unsolved, but as we will see, there are methods to mitigate this.

Finally, with all the flexibility of a neural network, a major challenge is to find the optimum settings, the hyperparameters that tune the model. For this we have developed a code that search for hyperparameters for the classification problem. Finally we do not create one FFNN, but several thousand through bootstrap iterations.

2 Theory

We will here discuss neural networks and logistic regression, and the various components needed for these methods. For neural networks we will use the term FFNN and Artificial Neural Network (ANN) interchangeably.

2.1 Neural Networks, an Overview

Consider a linear function $f(x) = ax + b$. This is an affine transformation. Adding several such linear functions is still a linear function. Now let the output from such a linear function be the input of a nonlinear continuous function. Let us call this function the activation function. We can still sum the output of these activation functions, and the sum of continuous functions is still a continuous function. This can be viewed as the composition of functions.

In a neural network this composition takes form as an architecture with several nodes connected. Let us first consider how the input in a single node is transformed to an output in a single neighbour node in a layer immediately forward of it. We input X_i in a node, which is first transformed with a linear transformation where X_i is multiplied with a weight w and then a constant term b, which we call bias, is added. Finally this output which we can call z_i is sent as an input to a nonlinear activation function σ, as depicted below in figure 1. The output of this activation function is then the input to the next node. Note that the nonlinearity of the activation function is a desired property for our design. If all the activation functions were linear then by linearity we could just reduce a neural network back to a simple linear function.

The final layer is called an output layer. For regression we would like the node to output any value on the real line, hence the last layer will typically be a linear identity function. For binary classification we would like the output layer to output probabilities in the domain $(0, 1)$, and for this purpose we typically use a sigmoid activation. We will discuss the particular activation functions in a following section.

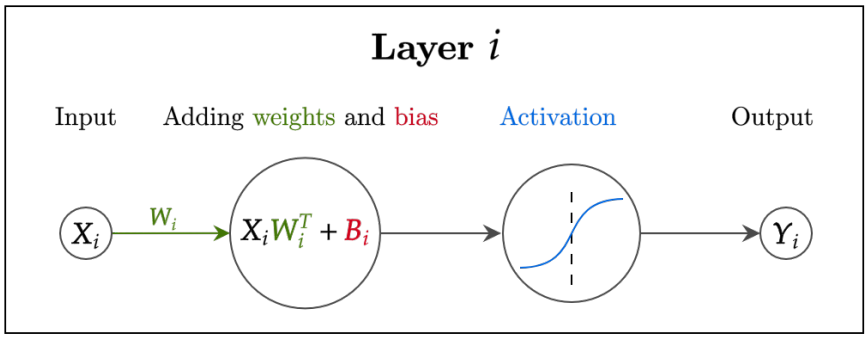

Figure 1: Linear function plus nonlinear activation node to node

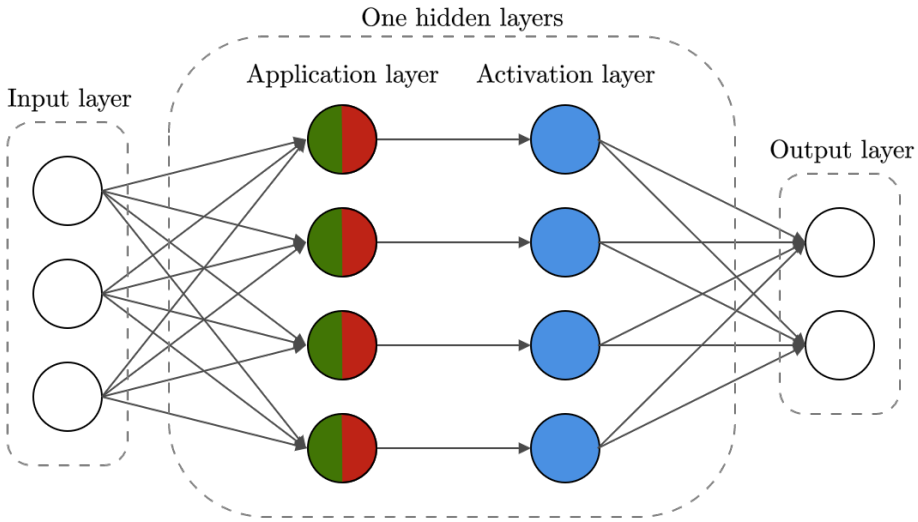

Figure 2: Hidden Layer

Now instead of just one such composite function we instead have several functions that work in parallel. We call the first nodes the input layer nodes. then the final output are the output layer nodes. In between we have what we refer to as a hidden layer. In figure 2, we show only one hidden layer, but we can expand this concept with several hidden layers.

We only consider here fully connected networks, that is all nodes connect forward and backwards to nodes in their immediate neighbour layers, as is seen in the figure 3. There are also other architectures possible, where nodes are added later or bypass a layer.

The sum of each edge between nodes is then the input of a node in the next layer.

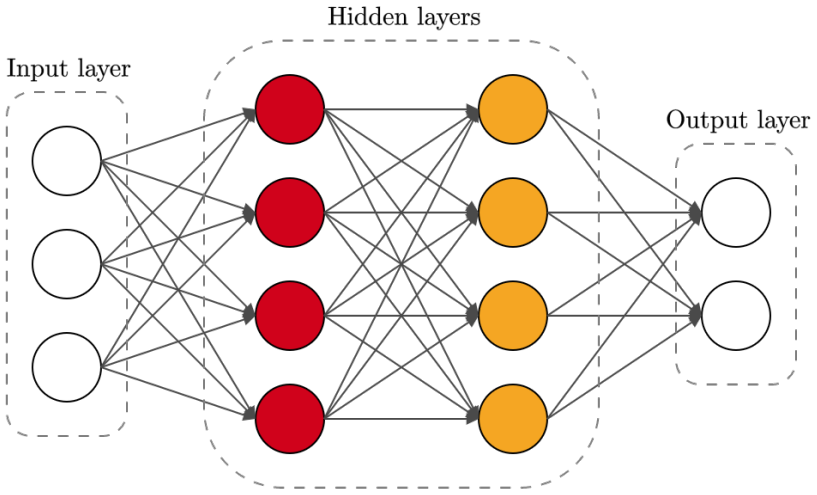

Figure 3: Forward Propagation.

2.1.1 Layers

This conceptually simple design can give rise to a very flexible function which we then call our FFNN.

Up til now we have discussed how the functions work going forward in what is referred to as forward propagation. Next follows a discussion on the information flow in the reverse direction, the back

propagation which is used to update the network in a learning process feeding back information from the output layer backwards. This concept is shown in figure 4.

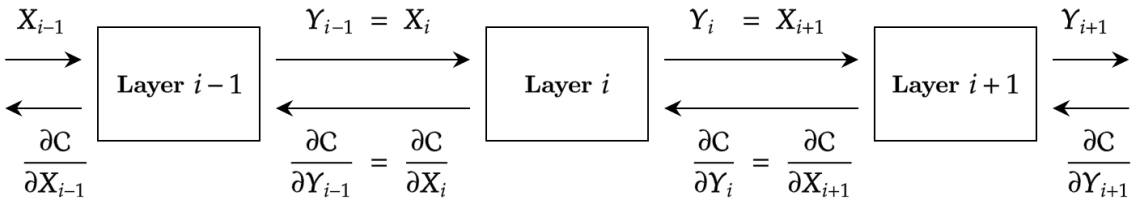

Figure 4: Forward and backward. One layer's output becomes the next layer's input

We will now provide more detail on backpropagation, a concept essential to train the neural network.

2.1.2 Backpropagation

The backpropagation algorithm was in all probability originally invented by Paul Werbos in his 1974 Harvard Ph.D. thesis, and subsequently reinvented in 1982 by David Parker of the Stanford Linear Accelerator Center. Backpropagation was developed into a useable neural network by David Rumelhart and the PDP group, based at UCSD Hecht-Nielsen 1992. In essence the algorithm is a very clever way to find an update for the weights and biases in a cyclic fashion that will bring us closer to the minimum of a selected cost function. The algorithm has three equations. One for acquiring the change of the cost with respect to the weights, one for the change with respect to the biases and one for obtaining the change with respect to the next layer. This is in principle just the chain rule. In matrix-vector form the equations are as follows.

$$\frac{\partial \mathcal{C}}{\partial \boldsymbol{W}} = \boldsymbol{X}^{\boldsymbol{T}} \cdot \frac{\partial \mathcal{C}}{\partial \boldsymbol{Y}} + \lambda \boldsymbol{W} \tag{1}$$

$$\frac{\partial \mathcal{C}}{\partial \boldsymbol{B}} = \frac{\partial \mathcal{C}}{\partial \boldsymbol{Y}} \tag{2}$$

$$\frac{\partial \mathcal{C}}{\partial \boldsymbol{X}_1} = \frac{\partial \mathcal{C}}{\partial \boldsymbol{Y}} \odot a'(\boldsymbol{X}) \qquad \& \qquad \frac{\partial \mathcal{C}}{\partial \boldsymbol{X}_2} = \boldsymbol{W}^{\boldsymbol{T}} \cdot \frac{\partial \mathcal{C}}{\partial \boldsymbol{Y}} \tag{3}$$

$$\frac{\partial \mathcal{C}}{\partial \boldsymbol{X}} = \frac{\partial \mathcal{C}}{\partial \boldsymbol{X}_1} \cdot \frac{\partial \mathcal{C}}{\partial \boldsymbol{X}_2} \tag{4}$$

Here $\odot$ denotes the *Hadamard* product, $\mathcal{C}$ is the cost, $\boldsymbol{X}$ is the input (or the output if one is going backwards), $\boldsymbol{Y}$ is the output (or the input if one is 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). You may notice that there are four equations instead of three. That is because of the way we have set up our code for the NN. A hidden layer is divided into two, an application layer and an activation layer. This is described above in Section 2.1.1.

2.1.3 Activation Functions and Initialization

We have already mentioned the concept of activation functions. As seen in Figure 4 in a neural network information is propagated forward and backwards. Propagation forward makes predictions for the next layer, whereas propagation backwards provides the gradients. In both directions we would like to avoid that the information propagation through the layers vanish or explode. It was shown by (Glorot and Bengio 2010) that the selection of activation functions and appropriate initialization (ref Figure 5) greatly reduce problems with vanishing or exploding gradients.

For this reason we will here discuss activation functions and initialization together.

We start with an overview of the various activation functions in the figure below:

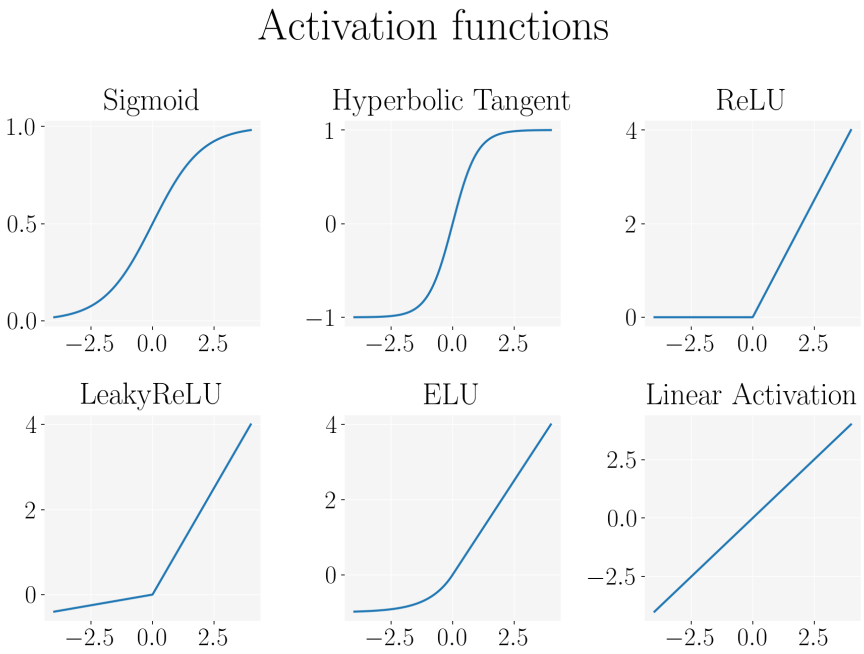

Figure 5: The figure shows various activation functions that were tested in the project. The linear activation function is used for the output layer in regression problems and the Sigmund function can be used in the output layer of classification problems where a continues value between 0 and 1 is desired like a probability.

As seen in Figure 4 in a neural network information is propagated forward and backwards. Propagation forward makes predictions for the next layer, whereas propagation backwards provides the gradients.

In (Figure 5) we see an overview of the most common activation functions.

The Sigmoid function is defined by the function $g(z) = \frac{1}{1+e^{-z}}$, while the Hyperbolic function is defined by the function $g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$.

Sigmoid activation function has been traditionally preferred, as it is also seen in biological neurons. However, the ReLU activation tend to perform better in Artifical Neural Networks (Géron 2017). Sigmoid is used as the activation layer in binary classification, but it can also be used in hidden layers.

The ReLU function is a rectified linear unit defined by the function $g(z) = max(0, z)$. This makes the gradients computationally cheap to compute. The ReLU activation function does not saturate for positive values (as it just returns the input) and therefore it is considered a good choice in deep neural networks.

A problem with ReLu is however 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 2017).

The Leaky ReLU is a RELU modified to avoid zero gradients and the dying ReLu problem, by adding a small slope in the negative part. It is defined by the function $g(z) = max(0, z) - \alpha(min(0, z))$. For α nonnegative and close to zero. This avoids zero gradients.(Binder 2018).

Both ReLu and Leaky ReLU are not differentiable at $z = 0$, but the derivative can still be assigned

a value at 0, typically 1 or 0, so this is thus in practice not a problem.

The exponential linear unit (ELU) is designed to speed up learning in deep neural networks and leads to higher classification accuracy. It is defined by the function $g(z) = max(0, z)+\alpha(min(0, e^z - 1))$. For α nonnegative. (Clevert et al. 2015).

We note that the ELU is smooth, including at $z = 0$. Hence it is everywhere differentiable, unlike the ReLU and Leaky ReLu.

Default values for the hyperparameter is typically 0.01 for Leak ReLu and 1.00 for ELU (Géron 2017). We note by the definition that ELU is lower bounded to $-\alpha$ as $z \to -\infty$. The linear activation function (or the identity function) simply pass the value through the activation layer, it is defined by $g(z) = z$. As mentioned earlier this activation is used for the output layer in a regression. There is no point in using a linear activation function in a hidden layer as then you could just remove that hidden layer and create a simpler model.

The activation functions we have discussed so far are all monotonically increasing. With the exception of ReLu they are even strictly monotonically increasing. Hence the derivatives does not change sign, which is normally favourable. Monotonicity is however not mandatory for an activation function, there exist several non-monotonic activation functions that has interesting properties for certain use cases. (Misra 2019).

Normally the same activation function is used for all nodes in a layer.

As we have seen the activation functions can transform the input data in to many shapes. Consider for example the simple ReLu activation function. This is a piecewise function that output a straight line with a dent at zero. Can this simple function give rise to a nonlinear shape and approximate say a sinus curve? Obviously one ReLu function could not do it, but let one input node be connected with two nodes in a hidden layer that each have a ReLu function. Now let their output go to a single output node. In this 1-2-1 configuration you can flip one ReLu upside down by setting a negative sign on the weight, and you can translate the ReLu functions with the bias. The output node can then form a V-shape at a given interval. We have thus enabled a simple function where we can control that it creates a particular shape.

Adding many more nodes we can combine their ReLu outputs together to approximate functions. An analogy could be the Riemann integral of a sinus curve being made of better and better rectangular approximations. This capability can of course be explored with the other activation functions as well. The ability of a feed forward neural network to approximate a function is stated in the Universal Approximation Theorem. It states that a neural network with one hidden layer can approximate any continuous function for inputs within a specific range, provided it has enough hidden nodes. With two hidden nodes any function can be approximated arbitrarily well, not only continous functions. (Goodfellow et al. 2016). In practice adding an extra layer can reduce computational cost instead of an extreme amount of nodes in one hidden layer.

The selection of activation function is thus an important consideration. In general it can depend on the type of problem studied, whether it is a regression or classification problem, the depth of the network, training time available and so forth. In the end the researcher can try out different activation functions to verify which performs best for the specific case.

After this discussion on the activation functions it is time to turn to the initialization.

With every learning process we must first start somewhere, and in the context of neural networks this "somewhere" means that the weights and biases are initialized at some value.

The very aim of our initialisation is of course to speed up learning and ensure a good end result from the trained network. For this we need information to flow freely through the network in both directions.

If we initialize all the weights and biases at zero, then of course there is nothing to learn! And if all the weights and biases had been initialized at the same value then every node in a layer that takes the same input would send out the same output, and this would contradict the purpose of learning from different nodes in a layer.

So we see that we desire some randomness to ensure diversity in our initialization. This does not mean we can toss a coin to find suitable values. Since the values we compute will pass through activation functions the result can be magnified to the extent that we get exploding or vanishing gradients. For this reason we need, especially for deeper networks, to consider how our initialization work in concert with the selected activation functions.

While random initialization with uniform or normal distribution has been traditionally used, and often works well for simpler networks, it has been shown (Glorot and Bengio 2010) that initialization should ensure the variance of the output of each layer is the equal to the variance of its inputs. Similarly the gradients should have equal variance before and after flowing through a layer in the reverse direction. Such initialization that ensure these properties is called Xavier initialization for Hyperbolic activation, named after Xavier Glorot, and for ReLU a suitable method is He Initialization.

Xavier Initialization is given as the following for Hyperbolic activation with normal distribution, (Glorot and Bengio 2010):

$$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}} \tag{5}$$

He Initialization is given as the following for ReLu activation with normal distribution (Glorot and Bengio 2010):

$$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}} \tag{6}$$

2.2 Training and Minimizing the Error

2.2.1 Cost Functions

In order to train our network we must define a measure for how well it is performing.

For regression we will use the mean sum of squares of errors (MSE). For our binary classification we will use the Binary Cross-Entropy(BCE).

We first define as our MSE cost function:

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

Here y_{d_i} is the i-th data point and $\hat{y}_{NN_i}$ is the networks prediction of this i-th point. The $\boldsymbol{\theta}$ are the weights and and biases for the network. Minimizing the MSE to train our model can be done using solvers based on gradient descent methods which will be described later.

With reference to previous project work by the authors (Patey and Somer 2022), the derivative of the MSE cost function $C(\theta)$ can in general be written in matrix notation as:

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

$$\frac{\partial C(\theta)}{\partial \theta} = -\frac{2}{n}X^\top y + 2X^\top X\theta$$

We can then add a L^2 Parameter regularization term $\frac{\lambda}{2}\boldsymbol{w}^\top\boldsymbol{w}$ to the MSE cost function to regularize the model, where λ is the hyperparameter. Many text books will also refer to this as α for FFNN

regularization. For a FFNN this regularization is added to the weights $\mathbf{w}$ only as regularization of the bias can result in underfitting. The derivative of this term is simply $\lambda\mathbf{w}$.

For simplicity we use the same weight penalty for the entire FFNN, but it can in some applications be desirable to use a separate penalty for each layer(Goodfellow et al. 2016).

We have so far been content with MSE as our cost function, and until around the 1990s MSE was also used to train networks for classification, but it was gradually replaced with cross entropy which improved the performance of the models.

There are several reasons why MSE is not a good cost measure for classification. MSE can take values on the real numberline. For the binary case however we have 0 or 1 as output which follows a Bernoulli distribution and not a Gaussian distribution. Moreover, while MSE is convex for OLS it is not when restricted by the sigmoid function in classification.

The gradient descent method may then not reach the global minima. Cross entropy on the other hand is convex. (Goodfellow et al. 2016).

In this project we will only consider the binary case, the binary cross entropy cost function (BCE).

With BCE we measure how far way our prediction probability (p) is from the true value (y=0 or y=1).

Let us first build up some intuition and find the terms that defines the BCE function. Assume $y = 1$, and define a loss function $L(y, p)$ as :

$$L(y, p) = -y \log(p) = -\log(p)$$

Then clearly by the logarithm we get that $L \to 0$ as $p \to 1$, and $L \to \infty$ as $p \to 0$. Thus an incorrect prediction will be penalized heavily, and when a probability is close to the correct $y = 1$ then we have a small loss value, as desired.

Now assume $y = 0$, but then we can just shift the value by 1 and use the same equation as above substituting in $1 - y$ and $1 - p$, we then get:

$$L(y, p) = -(1 - y) \log(1 - p) = -\log(1 - p)$$

These two terms cannot occur at the same time, due to the binary condition $y = 0$ or $y = 1$. Hence we can sum them together and get:

$$L(y, p) = -y \log(p) - (1 - y) \log(1 - p)$$

At this stage we have looked at the loss from one prediction. But we can sum n loss values together, and this will be our cost function. We here use Hadamard elementwise multiplication and get our Binary Cross Entropy (BCE) cost function as follows:

$$\text{Cost}(\mathbf{Y}, \mathbf{P}) = \frac{1}{n} \sum -\mathbf{Y} \odot \log(\mathbf{P}) - (1 - \mathbf{Y}) \odot \log(1 - \mathbf{P})$$

The derivative of $\log(x)$ is $\frac{1}{x}$. We use this to differentiate $L(y, p)$ w.r,t. p as follows:

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

For the classifier in a binary classification the output layer use the sigmoid activation to find the probability $p = \sigma(z)$. The derivative of the sigmoid function is found by use of the chain rule and quotient rule of derivation as:

$$\frac{\partial p}{\partial z} = p(1 - p)$$

Now by the chain rule we can combine these two derivatives and get:

$$\begin{aligned}\frac{\partial L(y,p)}{\partial z} &= \frac{\partial L(y,p)}{\partial p}\frac{\partial p}{\partial z} = \left(\frac{-y}{p} + \frac{1-y}{1-p}\right)(p(1-p)) \\ &= -y(1-p) + (1-y)p \\ &= p - y\end{aligned}$$

For our cost function C we then get:

$$\frac{\text{Cost}(\mathbf{Y},\mathbf{P})}{\partial z} = \mathbf{p} - \mathbf{y}$$

We have therefore arrived at a simple and computationally stable expression when we propagate backwards with the BCE cost function.

Also for BCE we can add a L^2 Parameter regularization term $\frac{\lambda}{2}\boldsymbol{w}^\top\boldsymbol{w}$ to regularize the model, where λ is the hyperparameter.

2.2.2 Gradient Decent (GD)

Now we have discussed the architecture of a neural network, and the cost function that measures the performance we would like to optimize through training iterations. In this section we will answer how we get there from our starting point, the untrained model initialized at some usually random values. We start with a conceptually simple drawing of a bowl shaped structure that mimics our cost function.

$$\boldsymbol{\theta}^{e+1} = \boldsymbol{\theta}^e - \alpha\nabla_{\boldsymbol{\theta}}\mathcal{C}(\boldsymbol{X},\boldsymbol{\theta}^e)$$

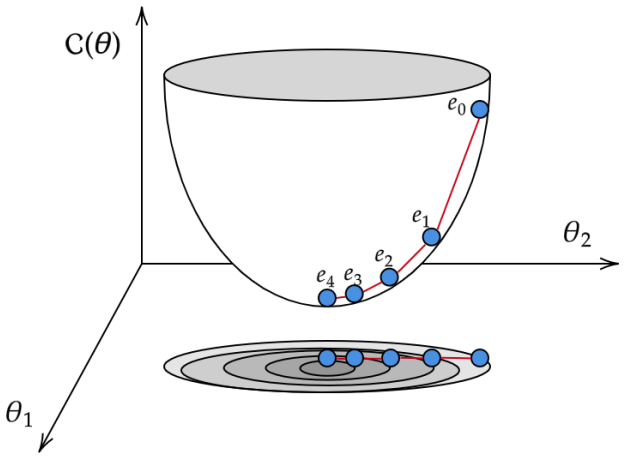

Figure 6: The figure shows the error or cost space as a function of weights. On the $\boldsymbol{\theta}_1\boldsymbol{\theta}_2$ plane there is a 2d projection of the space with contour lines showing the difference in elevation. A similar 2d projection can be seen in Figure 7.

In order to minimize the Cost function the basic idea of Gradient Descent is that we start at some initialization point (which could be a qualified initial guess or a random vector), then we calculate the gradient at that point, and take a small step in the direction of the negative gradient. This step will be calculated as the negative of the gradient times a scalar which is called the *learning*

rate. For a small step we assume that the gradient does not change much and we therefore reach a new point a bit closer to the minimum.

This is similar to the unfortunate situation you would be in, if you had to go down a valley in complete darkness and forgot your flashlight at home! You would take a small step in the steepest direction. Now from this point we repeat this process over and over again in several iterations until we reach a stopping criteria.

One stopping criteria is when the gradient is zero, since then it means we have reached a local minimum. If the cost function is convex then our local minima is also the global minima.

Several problems can arise with this method. First of all, if we take too small steps we may not reach the local minima at all within a maximum given number of iterations. Our stopping criteria is then when the number of iterations reach this limit. Conversely if the step size is too large we may jump across the local minima (the gradient changed sign in the step) and in this case we never reach the minima. Finally the cost function may be non-convex, in which we may get stuck in a local minima, reach a saddle point, or in a flat region where the gradient is zero, but still with the cost function value from the global minima.

Non-convexity is not a problem with OLS. As discussed in (Patey and Somer 2022) OLS is a convex function and Gradient Descent will then get arbitrarily close to the global minimum provided we have enough iterations.

The first problems associated with a proper learning rate can be partly overcome by adding momentum to the gradient algorithm. Momentum provides a memory of the direction we are moving, allowing the steps to move faster in the direction we have accumulated memory of, much like a ball rolling down a slope.

Let $0 \leq \gamma \leq 1$. We can then add moment to our update by:

$$\begin{aligned} \mathbf{v}_e &= \gamma \mathbf{v}_{e-1} + \eta_e \nabla_\theta E\left(\boldsymbol{\theta}_e\right) \\ \boldsymbol{\theta}_{e+1} &= \boldsymbol{\theta}_e - \mathbf{v}_e, \end{aligned}$$

Having some memory of the past directions is beneficial when for example the slope at the current position is extremely small, or even zero, due to a flat region. With momentum we keep moving faster in the direction that has been consistently favourable from the previous steps. Also, assume that there are some irregular formations in a region. Without momentum we would instantly let the direction of the update be based on these formations, possibly creating many unwanted fluctuations. With momentum we would instead be much less affected and, to follow the rolling ball analogy, we would just roll past those cracks in the road and still move in the overall desired direction, and only after a while slow down and turn when the overall shape dictates. For these reasons momentum makes a lot of sense, but for how long should memory of the past be kept? Obviously not forever, that would make it hard to turn. Assume that there are indeed some irregular shapes around our present position and that in order to reach the global minimum we need to turn our update direction fast enough, so as to not bypass them. This takes us to tuning of γ.

We see that $\gamma = 0$ is equivalent to no momentum, and as γ is increased more time steps are memorized, the total time span T considered in the memory is then $T = \frac{1}{1-\gamma}$. If we let $\gamma = 1$ we would keep all time steps in the memory.

Since the terrain can change shape as we take many steps we start to understand that it would be great to update γ based on the present position in the parameter space as well. Ideally we would like to calculate the second derivatives, the Hessian. But that is often impossible or at least computationally expensive. Instead we will use optimizers, which will be discussed later.

Besides momentum, an other element is the importance of feature scaling. If the features are on a very different scale the gradient descent will take a curved path toward the minima, which means it needs much more iterations to converge. In this case feature scaling is important.

2.2.3 Stochastic Gradient Descent (SGD)

The word "stochastic" in stochastic gradient descent refers to the fact that the method has a randomness linked to it. In contrast to GD, SGD does not run through all of the data points for each epoch. Instead it selects only a single sample randomly form the data set and performs the iteration on it. The true gradient $\nabla_{\boldsymbol{\theta}}\mathcal{C}(\boldsymbol{X}, \boldsymbol{\theta}^e)$ is then approximated by the gradient at a single data sample.

$$\boldsymbol{\theta}^{e+1} = \boldsymbol{\theta}^e - \alpha\nabla_{\boldsymbol{\theta}_i}\mathcal{C}(\boldsymbol{X}_i, \boldsymbol{\theta}_i^e)$$

Instead of a stochastic gradient descent with a single sample size a collection of samples are usually used for each epoch. These are called a mini-batches. This is done to potentially achieve a smoother convergence as the gradient is computed over an average of data points. Furthermore, for large scale problems we can now take advantage of parallel and distributed stochastic gradient descent methods.

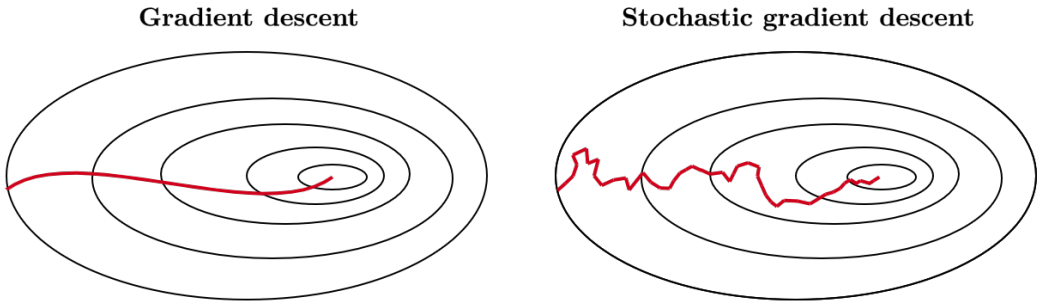

Figure 7: The figures show the convergence path taken by GD and SGD respectively. The path of SGD is noisier than the usual GD path. This is due to the stochastic property of the SGD. Although the path is longer the time to convergence is usually shorter for SGD.

2.2.4 Optimizers

We have discussed the two solver methods, GD and SGD.

We saw that the learning rate must be tuned in order to get the desired results. We could have a fixed learning rate that we tune manually and employ a learning schedule, but in this case we must tune the learning rate to ensure we do not take too large steps in the steepest directions, but that can result in steps in other directions being very small.

Ideally we would like to employ a method to tune the learning rate based on the parameter space we are in. As the terrain of the cost function changes we would like the learning rate to be tuned. We saw the benefit of momentum, and that we would even like to have some knowledge about the second derivatives in order to tune the momentum itself.

This takes us to the optimizer algorithms. We will first look at Adagrad, then RMSprop and finally ADAM.

2.2.4.1 Adagrad

We saw that SGD use the same step size for all parameters when the learning rate is multiplied with the gradient. This is a drawback since there could be interesting features in the other directions. This was the basis for Adagrad to be developed in 2010 (Duchi et al. 2011).

Adagrad stands for adaptive gradient and its algorithm is here referred below from Algorithm 8.4 in (Goodfellow et al. 2016).

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, perhaps 10^{-7}, for numerical stability
Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$
while stopping 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: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L\left(f\left(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right), \boldsymbol{y}^{(i)}\right)$
  Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$
  Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)
  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end while

With Adagrad's we no longer need to manually tune the learning rate.

The main problem with Adagrad is however the way we keep adding the squared gradients in the denominator. When this keeps growing at some point the learning rate will be so small that we make very little progress.

That takes us to the next algorithm that seeks to resolve this particular issue, the RMSprop.

2.2.4.2 RMSprop

RMSprop stands for Root Mean Squared propagation and its algorithm is here referred below from Algorithm 8.5 in (Goodfellow et al. 2016).

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ.
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers.
Initialize accumulation variables $\boldsymbol{r} = 0$
while stopping 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: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L\left(f\left(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right), \boldsymbol{y}^{(i)}\right)$
  Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho\boldsymbol{r} + (1-\rho)\boldsymbol{g} \odot \boldsymbol{g}$
  Compute parameter update: $\Delta\boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. $\left(\frac{1}{\sqrt{\delta + \boldsymbol{r}}}\right.$ applied element-wise)
  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end while

RMSprop now divides the learning rate by an exponentially decaying average of squared gradients. It was developed by Geoff Hinton in 2012 (Hinton et al. 2012).

RMSprop almost always performs much better than AdaGrad. It also generally performs better than Momentum optimization. It was the preferred optimization algorithm of many until Adam optimization came around in 2014 (Diederik P. Kingma and Ba 2014b).

This brings us to Adam in the next section.

2.2.4.3 ADAM

Adam is named from adaptive moments (Diederik P Kingma and Ba 2014a). It is a variant of RMSprop and momentum.

In addition to storing an exponentially decaying average of past squared gradients, Adam also keeps an exponentially decaying average of past gradients, similar to momentum. This can be seen as adding friction to our rolling ball.

The ADAM algorithm is here referred below from Algorithm 8.7 in (Goodfellow et al. 2016).

Algorithm 8.7 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$.
(Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization. (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$
Initialize 1 st and 2 nd moment variables $\boldsymbol{s} = \mathbf{0}, \boldsymbol{r} = \mathbf{0}$
Initialize time step $t = 0$
while stopping 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: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L\left(f\left(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right), \boldsymbol{y}^{(i)}\right)$
  $t \leftarrow t + 1$
  Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1) \boldsymbol{g}$
  Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2) \boldsymbol{g} \odot \boldsymbol{g}$
  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: $\Delta \boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)
  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
end while

Out of the optimizers discussed so far, Adam optimization is very often the best choice. (Géron 2017).

2.3 Logistic Regression

With this discussion of the neural network and the associated solvers and optimizers we will now look at classification with logistic regression, which can also use the same solvers and optimizers.

Logistic regression was developed in the 1940s (Hastie et al. 2015), and has then been widely used for classification. A notable example where logistic regression could have avoided a disaster is the 1986 Challenger accident (Dalal et al. 1989).

In classification the categories could be many but in the simplest case which we will look at it could be the binary categories, true or false, such as the result of a coin flip or a diagnostic test.

For the binary case, let $P(\text{y is true}|X = x)$ denote the probability that y is true given $X = x$. At first glance it could be tempting to try to model this probability as a linear model $y = \beta_0 + \beta_1 x_1 + ...\beta_p x_p$, but it becomes evident that such a model could give probabilities that are negative or greater than one, both violations of true probability measures. We therefore need to find a way to map our explanatory variables to a domain that is limited to $[0, 1]$. Our mapping should preferably also be continuous and differentiable.

One such mapping is the logistic regression which we will explore in the following. Here we model the probability P of a random variable X with:

$$P(\mathbf{X}) = \frac{e^{\beta_0+\beta_1 X_1+\cdots+\beta_p X_p}}{1+e^{\beta_0+\beta_1 X_1+\cdots+\beta_p X_p}} = \frac{\exp(\mathbf{X}\beta)}{1+\exp(\mathbf{X}\beta)} = \frac{1}{1+\exp(-\mathbf{X}\beta)} \tag{7}$$

We see that this probability measure maps to the range $[0, 1]$ since the exponential function maps $(-\infty, \infty)$ to $[0, \infty)$. Furthermore it can be shown that the two mutually exclusive probabilities $P(\text{y is true}|X = x)$ and $P(\text{y is false}|X = x)$ sum to 1.

Now let $z = \mathbf{X}\beta$. z is then a linear function. We can write the probability measure as:

$$P(\mathbf{X}) = \frac{1}{1+\exp(-z)} \tag{8}$$

This we recognize has the same form as the sigmoid function $\sigma(z) = \frac{1}{1+exp(-z)}$ which we also use as the output activation function in FFNN for binary classification.

This can be rewritten as:

$$\frac{P(\mathbf{X})}{1-P(\mathbf{X})} = e^z$$

The left hand side is called the odds. Taking the log on both sides we get:

$$\log(\frac{P(\mathbf{X})}{1-P(\mathbf{X})}) = z = \mathbf{X}\beta$$

The left side is now called the log-odds and from the above we see that with logistic regression $P(\mathbf{X})$ is not linear in $\mathbf{X}$ but the log-odds is linear in $\mathbf{X}$.

The maximum likelihood function $L(\beta)$ is then:

$$L(\beta) = \prod_{i:y_i=1} p(x_i) \prod_{i':y_{i'}=0} (1-p(x_{i'})) = \prod_{i=1}^{n} p_i^{y_i}(1-p_i)$$

Our next step is to take the logarithm on both sides, since it is easier to calculate the derivatives with summation rather than products, and this can safely be done since maximizing the log-likelihood w.r.t. β means we find the same β that maximizes the likelihood, due to the log function being monotonic.

Let $\ell(\beta)$ denote $\log L(\beta)$. Then $\ell(\beta)$ is:

$$\begin{aligned} \ell(\beta) &= \sum_{i=1}^{N} \{y_i \log p(x_i;\beta) + (1-y_i)\log(1-p(x_i;\beta))\} \\ &= \sum_{i=1}^{N} \left\{ y_i \beta^T x_i - \log\left(1+e^{\beta^T x_i}\right)\right\} \end{aligned} \tag{9}$$

For convention we let $C(\beta) = -\ell(\beta)$ denote the cost function that we intend to minimize. This is the log-loss function in statistics, which we can see is the same as the Binary Cross Entropy (BCE) cost we discussed earlier.

$$\mathcal{C}(\boldsymbol{\beta}) = -\sum_{i=1}^{N} \left\{ y_i \beta^T x_i - \log\left(1+e^{\beta^T x_i}\right)\right\}$$

To maximize the log-likelihood, we set its derivatives to zero. These score equations are

$$\frac{\partial \ell(\beta)}{\partial \beta} = \sum_{i=1}^{N} x_i \left(y_i - p\left(x_i; \beta\right)\right) = 0,$$

which are $p+1$ equations nonlinear in β.

For matrix notation, let y denote the vector of y_i values, $\mathbf{X}$ the $N \times (p+1)$ design matrix of x_i values, $\mathbf{p}$ the vector of fitted probabilities with i th element $p\left(x_i; \beta^{\text{old}}\right)$.

Then we have

$$\frac{\partial \ell(\beta)}{\partial \beta} = \mathbf{X}^T(\mathbf{y} - \mathbf{p})$$

3 Method

3.1 Code Design and The Foundation

To build a solid foundation for our NN we started from scratch by first writing our own plain GD method with a fixed LR and then slowly added features to it. First we added momentum and compared convergence with a fixed LR. Secondly we added an adaptive time-based decay for the LR and again compared convergence. An example of the time-based decay is shown below

```
def learning_schedule(epoch, init_LR, decay):
    return init_LR * 1/(1 + decay*epoch)
```

Here init_LR is the initial learning rate, decay is the factor that determines the rapidity of the decay and epoch is the current epoch. We chose decay = 0.01.
Then we implemented a SGD method with a given minibatch size and number of epochs upon verifying it with and with and without momentum, and with a fixed and an adaptive LR. We then implement the Adagrad method in order to tune the learning rate, and did this with and without momentum for plain GD and SGD. Next we add RMSprop and Adam to your library of methods for tuning the learning rate and adding moments.
By plotting a heatmap of the cost we can visually find the optimum regularization parameter λ and polynomial degree of the design matrix. A $\lambda = 0$ will give an OLS or standatd mean square error (MSE) cost anything else will yield a Ridge cost. The optimum λ and polynomial degree will then be returned by the function plotting the heatmap and will be used for the rest of the plots.

3.1.1 Results for Foundation

In order to test our methods, intended to be used later for regression and classification, we first tested their performance making regression on the polynomial $f(x) = 1 + x + x^2$ with added Gaussian noise. We compare here the performance of the various solvers and optimizers over a wide range of iterations and with and without momentum.

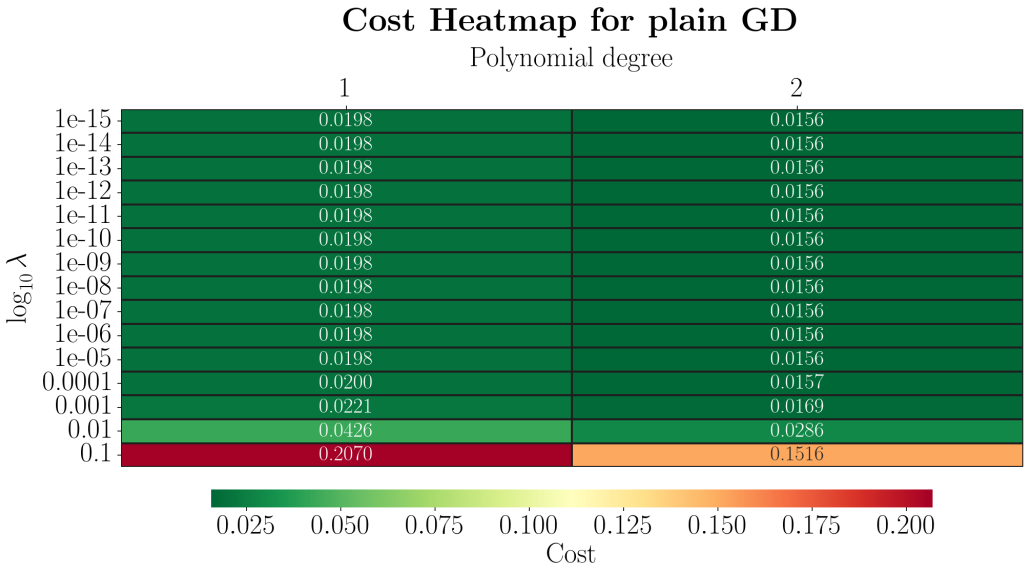

Figure 8: Heatmap of the cost for GD achieved with different values of the regularization parameter λ and polynomial degrees. We see that the lowest cost was achieved with polynomial degree 2, and λ ranging from 10^{-15} and up to 10^{-5} when rounding up to four decimals.

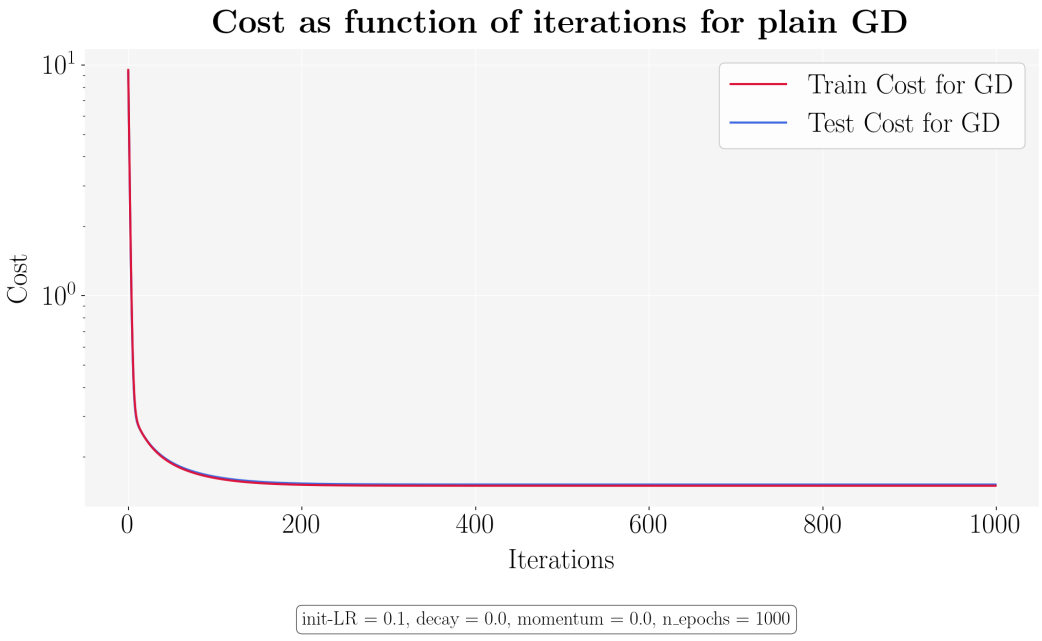

Figure 9: Both the test and train cost seem to converge and there is visually no overfitting.

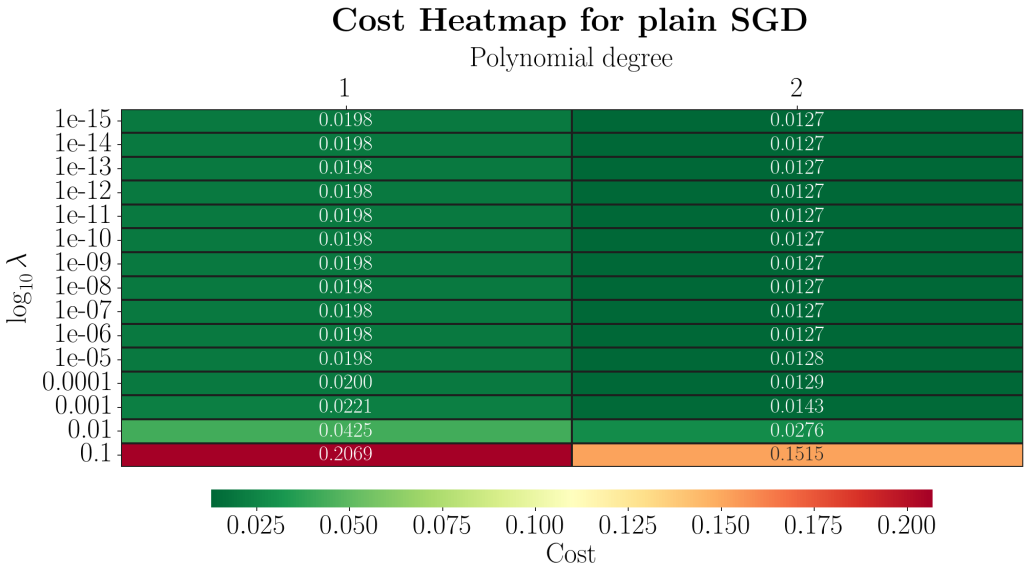

Figure 10: Heatmap of the cost for SGD achieved with different values of the regularization parameter λ and polynomial degrees. We see that the lowest cost was achieved with polynomial degree 2, and λ ranging from 10^{-15} and up to 10^{-6} when rounding up to four decimals.

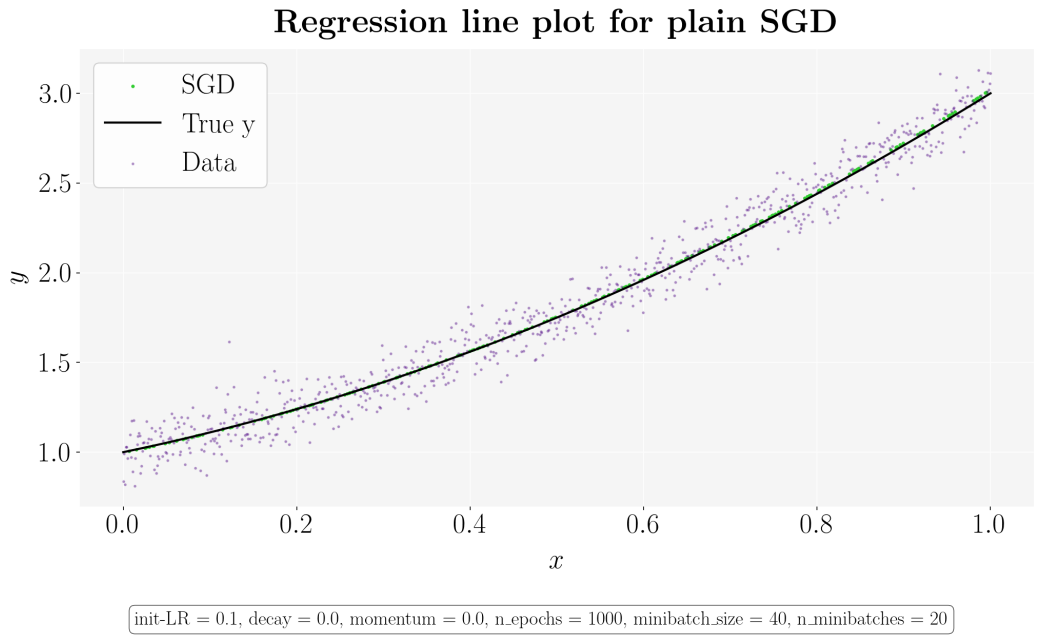

Figure 11: SGD regression on f(x) with 1000 epochs and 20 minibatches.

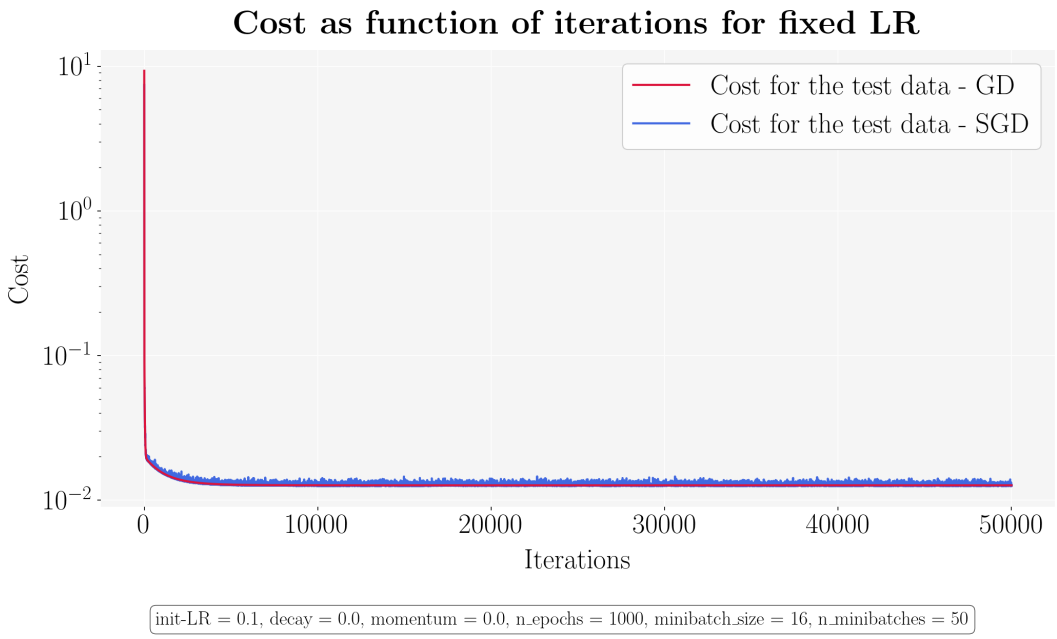

Figure 12: SGD and GD test MSE with 50000 iterations (1000 epochs and 50 minibatches).

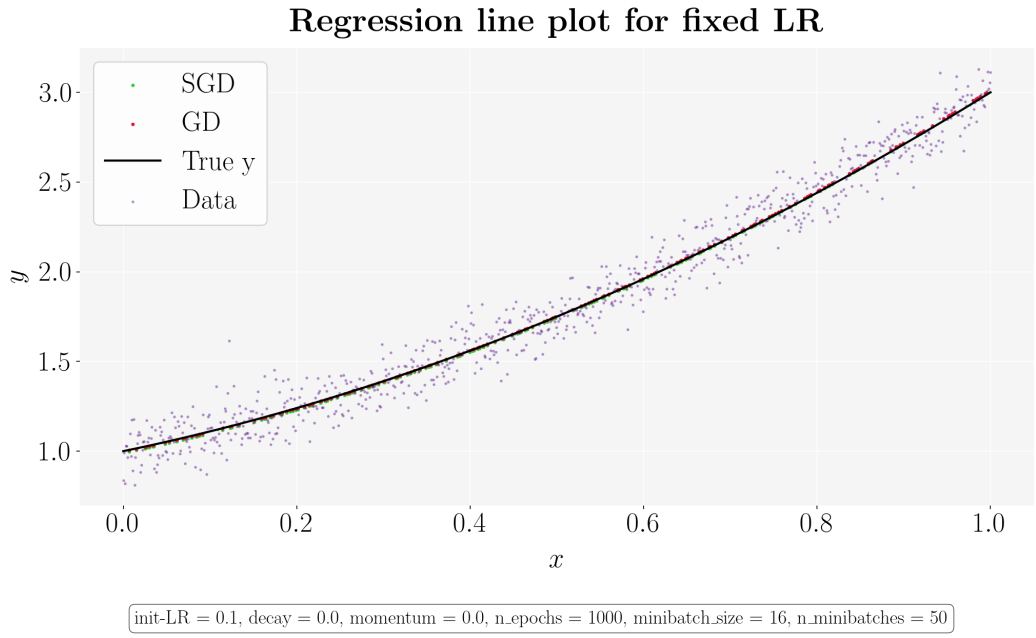

Figure 13: SGD and GD regression on f(x) with a fixed learning rate.

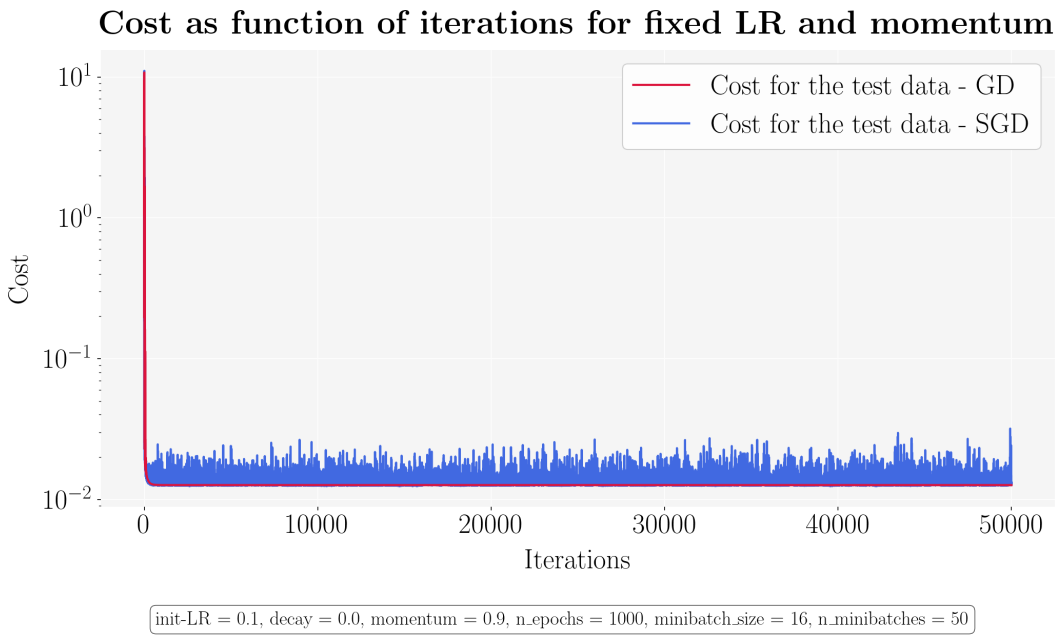

Figure 14: SGD and GD test MSE with 50000 iterations (1000 epochs and 50 minibatches with fixed learning rate and with momentum).

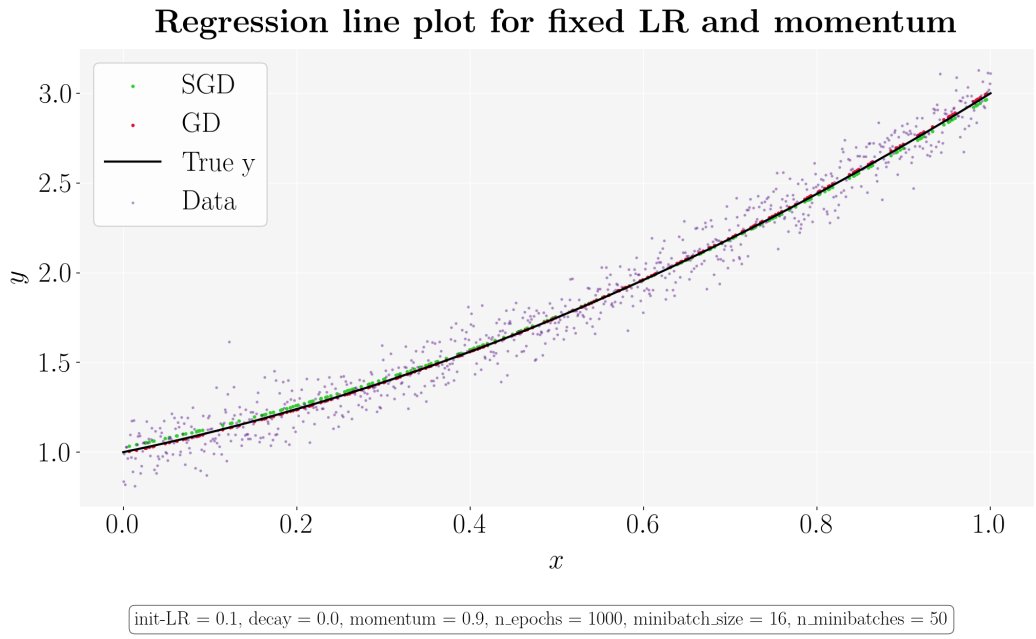

Figure 15: SGD and GD test MSE with 50000 iterations (1000 epochs and 50 minibatches with fixed learning rate and with momentum..

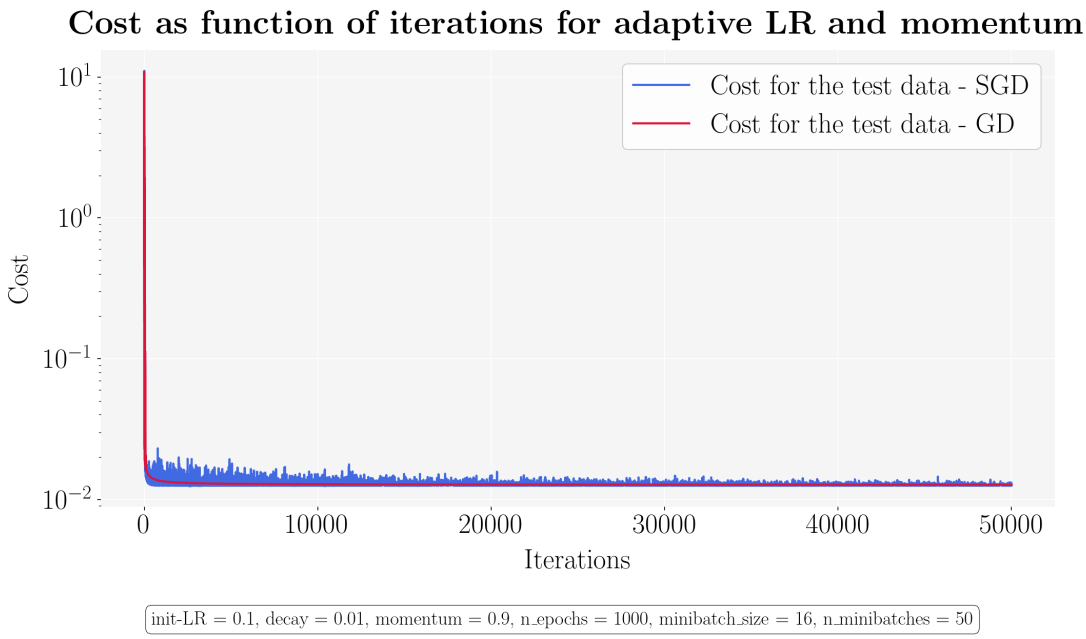

Figure 16: SGD and GD test MSE with 50000 iterations (1000 epochs and 50 minibatches with adapted learning rate and momentum.

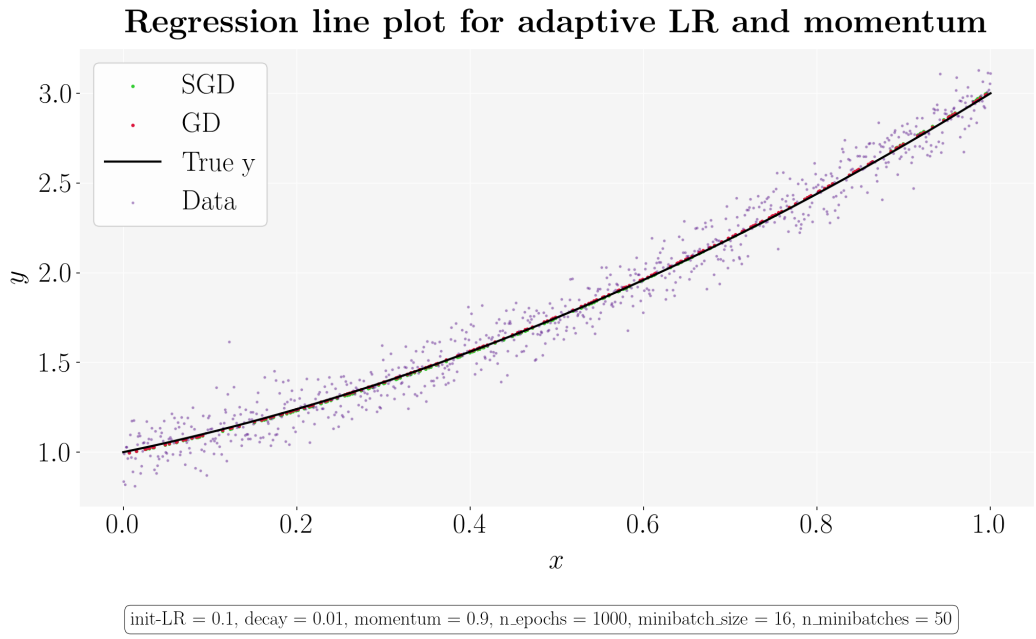

Figure 17: SGD and GD regression with 50000 iterations (1000 epochs and 50 minibatches with adapted learning rate and momentum.

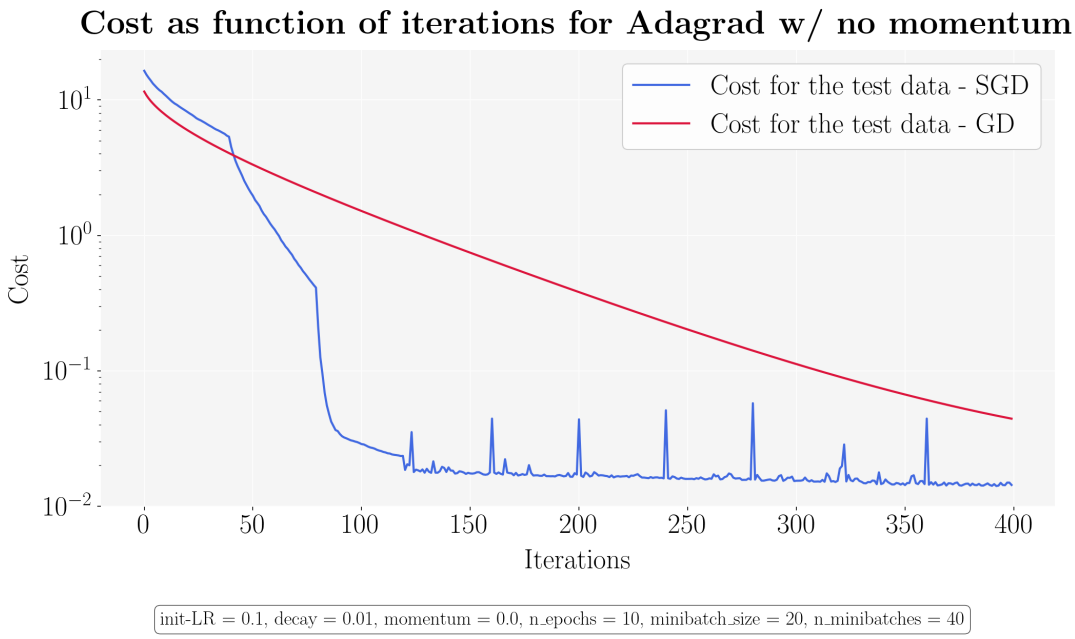

Figure 18: SGD and GD test MSE with 400 iterations with Adagrad and no momentum.

In figure 18 we note that the GD solver converge very slowly. As noted in the theory part, Adagrad adds up the square of the gradients in the denominator and for GD this results in the gradients being very small after some iterations hence it slows down too much. For SGD this is reset for each epoch, hence SGD converges faster. Adagrad is therefore not well suited to be used with GD. We can see how Adagrad would behave with momentum added in the update of gradients by adding a momentum term. We then get a faster convergence as shown in figure 19 below. This can be seen in context with the Adam optimizer which use momentum.

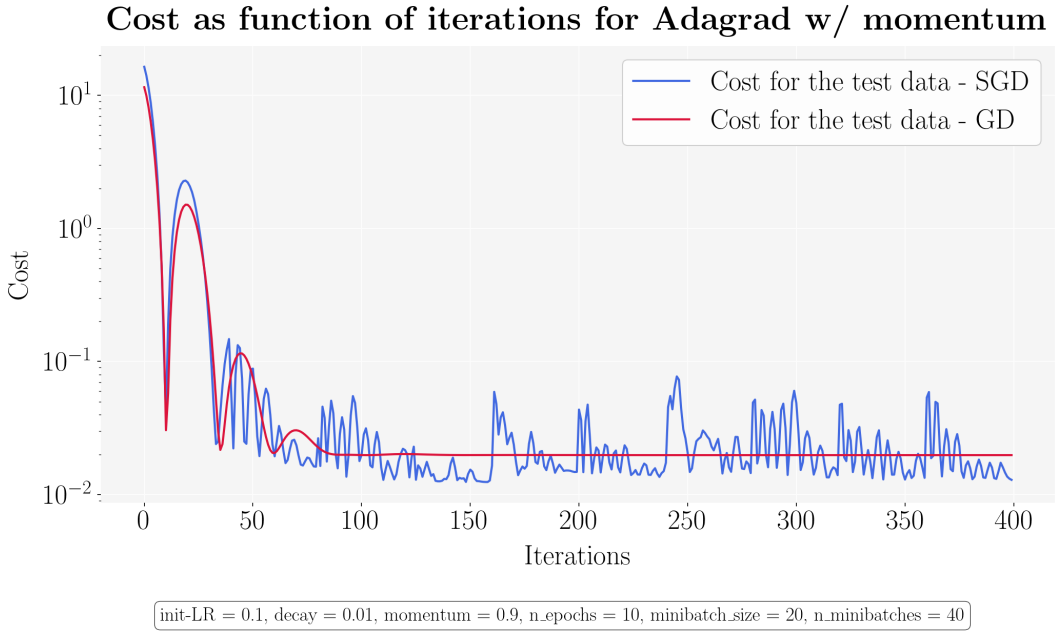

Figure 19: SGD and GD test MSE with 400 iterations with momentum on Adagrad.

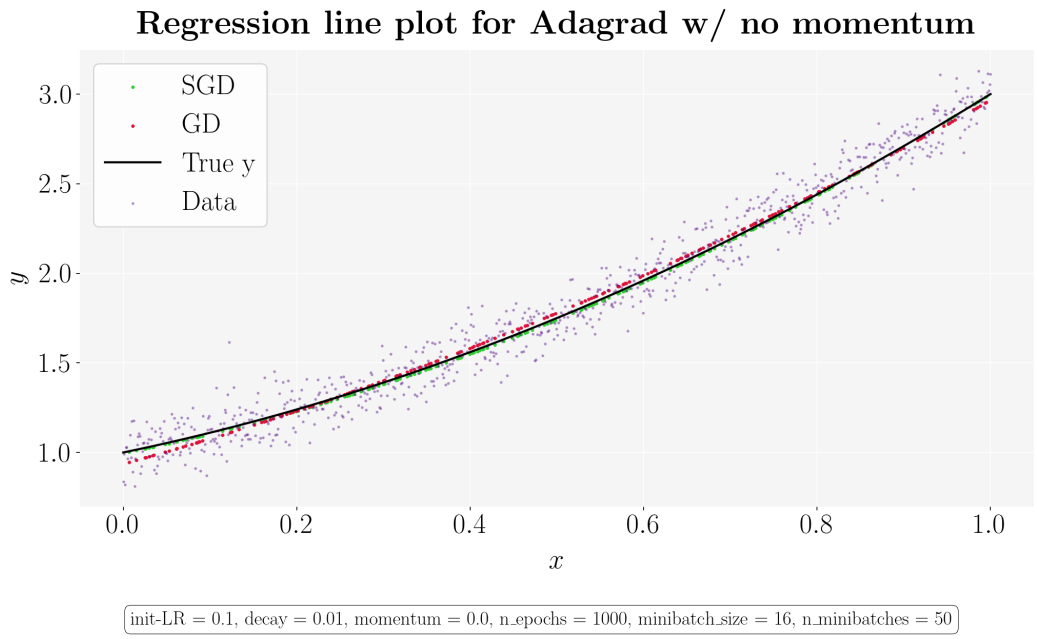

Figure 20: SGD and GD Adagrad regression with 50000 iterations (1000 epochs and 50 minibatches with no momentum.

We see in figure 20 that GD is not able to achieve a good fit after 50000 iterations, as Adagrad without any momentum has made the method slow down too early. SGD achieves a better fit as described above. With momentum Adagrad makes a better fit for GD, seen in figure 21 below. Now with larger fluctuations in the SGD the SGD fit is not as exact as without momentum.

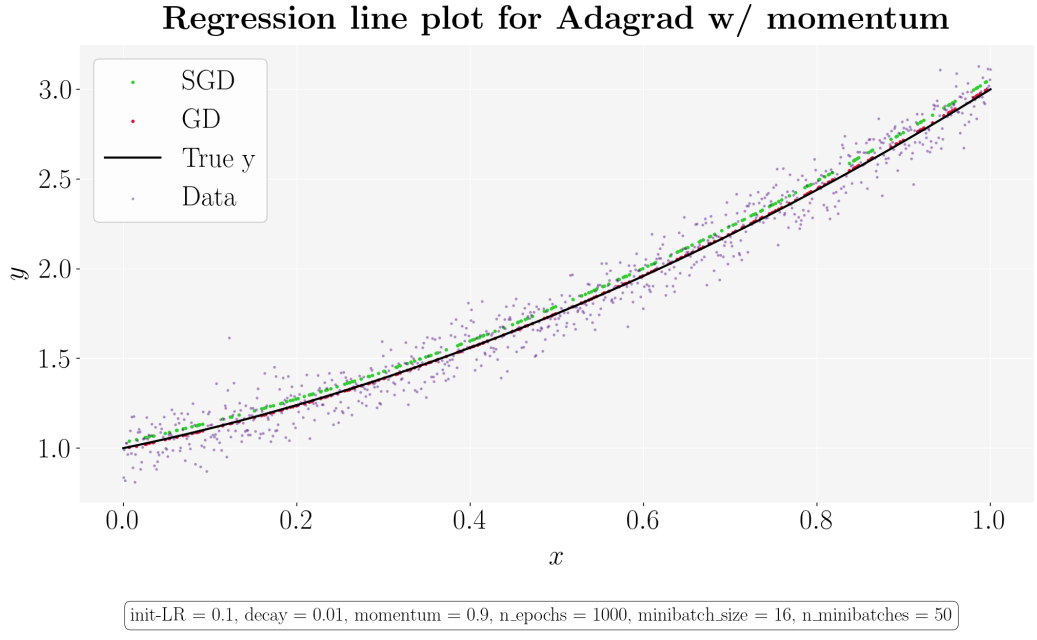

Figure 21: SGD and GD Adagrad regression with 50000 iterations (1000 epochs and 50 minibatches with momentum.

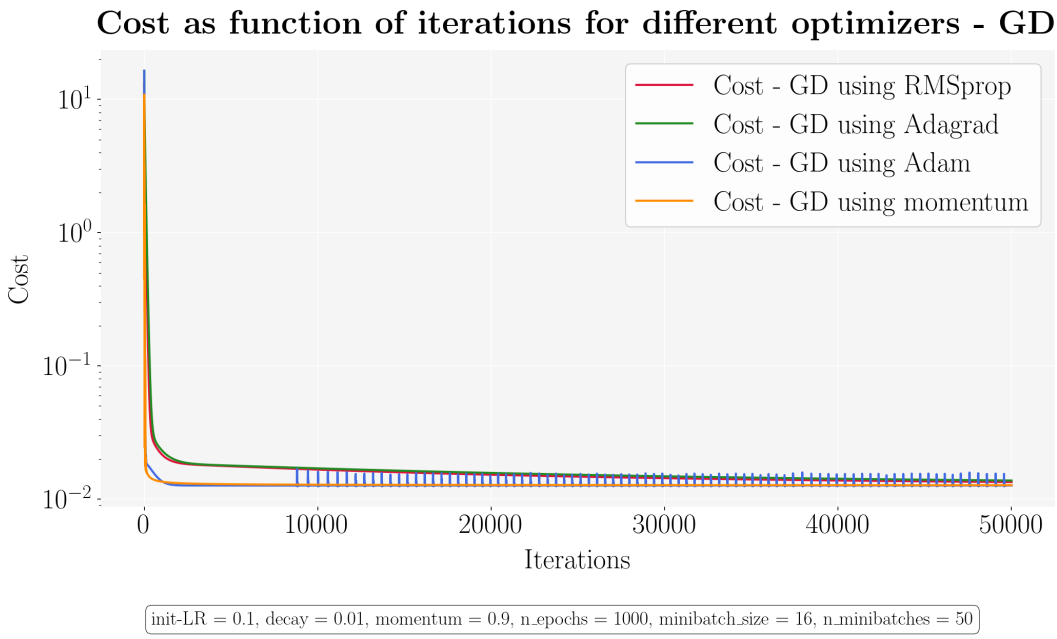

Figure 22: Cost plots for various optimizers - GD.

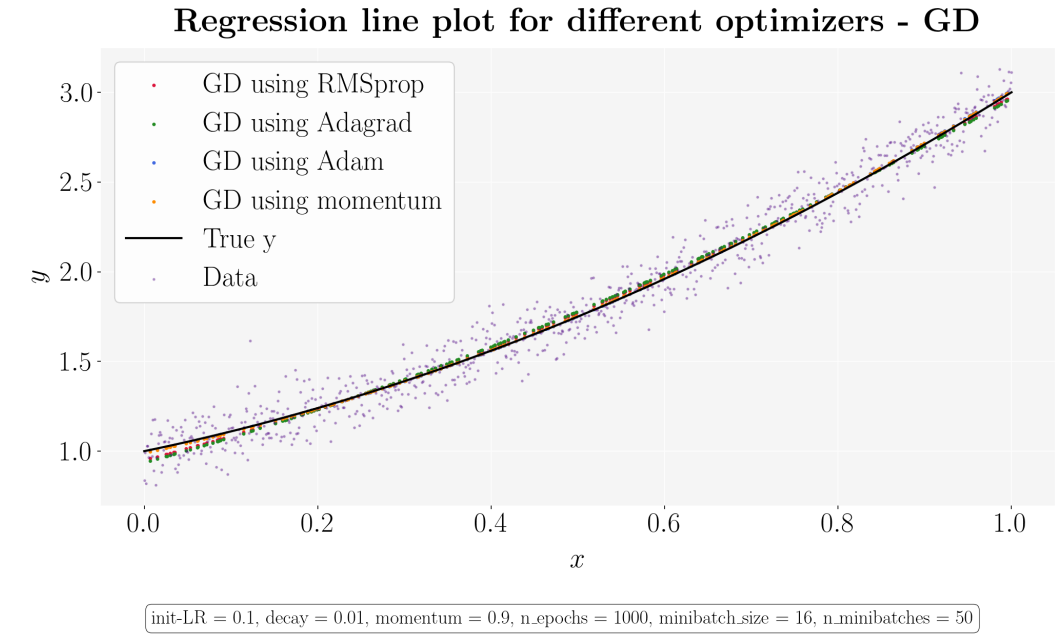

Figure 23: Regression plots for optimizers.

In figure 23 We see that Adagrad and to some extent also RMSprop perform worse since we have not added momentum, compared to GD with moment and ADAM which by design has momentum.

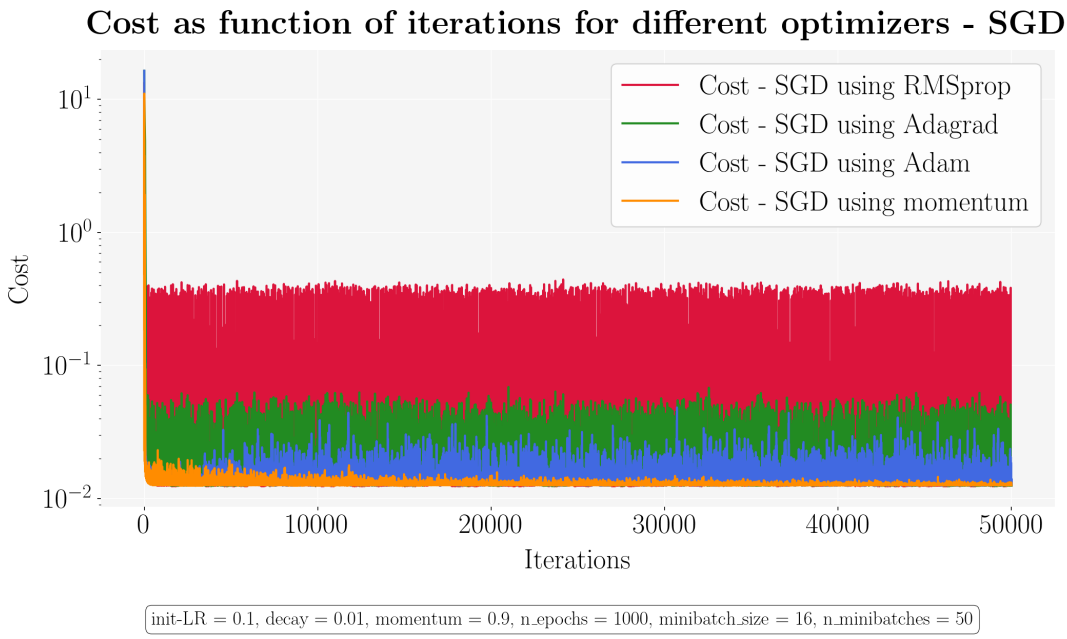

Figure 24: costplot˙OPTIM˙SGD.

We note that due to the stochastic we have fluctuations in the cost for every epoch reset.

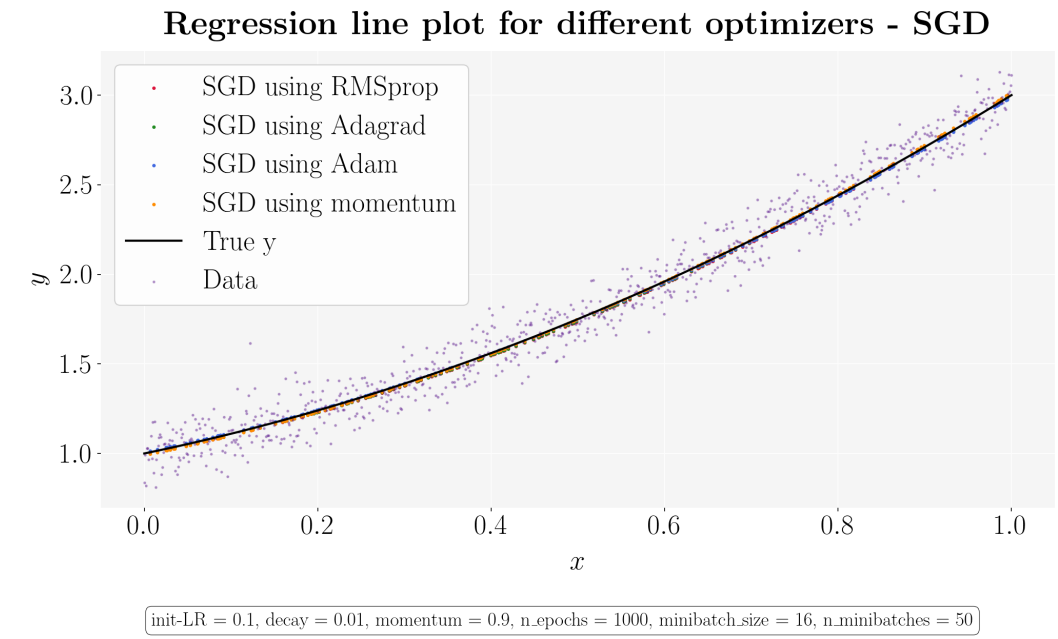

Figure 25: Regression plots for various optimizers - SGD

This concludes our review of the various solver methods and the associated optimizers designed for this project.

3.2 Implementation of a Flexible ANN and Regression Analysis on the Franke Function

Although GD, in general, preformed better than SGD, as to be expected due to that fact that GD uses all the data points thus yielding a better regression line, we opted for SGD in the next part of the project for computational time reasons. Here we will explore the regression prospect of an ANN on the well studied Franke Function (Patey and Somer 2022). In our results from testing GD and SGD on the line $1+x+x^2$ we observe that Adagrad and RMSprop showed inferior performance with regards to convergence time compared with momentum and Adam. We choose therefore to discard those methods and continue with either momentum or Adam. We saw very similar results with momentum and Adam but chose Adam over momentum as it works well in practice and compares favorably to other stochastic optimization methods (Diederik P Kingma and Ba 2014a). Furthermore, since Adam incorporates adaptive learning rates and first and second moments of the gradients, as discussed in the theory section. This would mean that we would not need to run an elaborate hyperparameter search as we would have needed if we had chosen momentum.
Thus, we decided to adapt SGD and Adam as our solver and optimizer for the ANN and our logistic regression.

3.2.1 ANN Code

We coded in python with both functional structures and object oriented structures. The later of which helped reduce the code tremendously. The code is structured in different python files, that import accordingly. This eased the process of creating a flexible NN, as it allowed for a flexible design. We implemented and explored various methods for searching and finding the optimal hyperparameters, both using heatmaps as a form of visual search but also grid searches and other methods as will be stated below. For the grid search we used numpys argwhere function. An example is shown here for finding the lowest cost among several runs with different number of nodes and regularization parameters λs

```
""" Finding the paramaters that give the lowest cost """
index = np.argwhere(MSE_nodes_lambda_SGD == np.min(MSE_nodes_lambda_SGD))
best_n_nodes_cost = n_nodes[index[0,0]]
best_lambda_cost = lambdas[index[0,1]]
print(f"""\nThe lowest cost with SGD was achieved {best_n_nodes_cost} nodes,
      and with lambda = {best_lambda_cost}.\n""")
```

Here index gives the grid coordinates for where the cost was lowest, best_n_nodes_cost extracts the number of nodes that gave the lowest cost, and best_lambda_cost extracts the λ that complimented the number of nodes and together gave the lowest cost. Then the results are printed in the terminal.

The flexible ANN code builds on the fact that a layer is nothing but a function that takes in an $m \times n$ input matrix, applies a $n \times k$ weight matrix, adds a bias term and spits out an $m \times k$ output matrix. Then we add an activation layer that takes in the $m \times k$ matrix and spits out an $m \times k$ matrix. Thus, the ANN with a flexible number of nodes and hidden layers could be initialized as following

```
""" The Neural Network """
ANN = [
        Layer(n_nodes_inputLayer, best_n_nodes_cost),
        LeakyReLU(),
        Layer(best_n_nodes_cost, best_n_nodes_cost),
        LeakyReLU(),
        Layer(best_n_nodes_cost, n_nodes_outputLayer),
        Linear_Activation()
      ]
```

Here the list ANN includes all the layers, activation layers as well. This is based on the structure that can be seen above in Figure 1 and Figure 4. Note that the input and output of the activation layers are the same and thus not specified. This specific ANN has two hidden layers with the LeakyReLU activation function. An example of using the above network on the Franke data could look like this

```
""" Obtaining the data """
X, Y, Z = data_FF(noise=True, step_size=0.02, alpha=0.1, reshape=True)

""" Spliting the data into train and test sets with an 80/20 split respectively """
x_train, x_test, y_train, y_test, z_train, z_test =
      train_test_split(X, Y, Z, test_size=0.2, random_state = seed)

""" Concatenating the x and y inputs into one input """
input_train = np.c_[x_train, y_train]
input_test = np.c_[x_test, y_test]

""" Training the network on the train data with the selected parameters """
train_NN_SGD(ANN, input_train, z_train, input_test, z_test, n_epochs, init_LR, ... )

""" And running the test data forward through the netwark """
z_predict_test = fwd(ANN, test_input)[0]
```

3.3 Regression on the Franke Function

After implementing a flexible ANN with SGD as solver and Adam as optimizer it is time to choose an activation function such as LeakyReLU, ReLU, ELU, Sigmoid or the Hyperbolic tan function and the complexity of the NN, i.e. the number of nodes in each hidden layer and the number of hidden layers.

We first tested the performance manually for one, two, three and four layers with a low epoch in order to asses the best model complexity for continuation. We achieved the worst performance with only one layer, then came the model with four layers and the models with two and three layers achieved the best performances (the lowest cost).

We tested the models with number of nodes equal to $2, 5, 10, 50, 100$ and 150, and λ ranging from 10^{-15} to 10^{1} with 9 values including the end points. The lower node numbers were quickly discarded as they did not preform as well as the others. Thus, we ended up testing for $10, 50, 100$ and 150 nodes.

Simultaneously, we tested the models for different activation functions and achieved the best results, by double performance (half cost), with LeakyReLU, ReLU compared with the other functions. We continued therefore with those two. All the results from this section can be found on github.com/JouvalSomer under the section FYS-STK3155 and Project 2.

The data was not scaled as it was artificially generated and we sat the noise to be 0.1 times a normally distributed noise $\sigma(0, 1)$. Since we were using SGD we saw that it was important to include a relatively large minibatch size in order for the network to get a good overview of the Franke function. We sat therefore the size to be

```
minibatch_size = int(np.floor(x_train.shape[0] * 0.005))
```

i.e. 0.5% of the entire data set. Yielding 200 minibatches of 10 points each.

We will also compare our results with sklearns MLPregressor with a similar design to our network in order to verify our model. An example of a MLPregressor:

```
""" MLPregressor with three hidden layers consisting of 50 nodes each """
regr = MLPRegressor(hidden_layer_sizes=(50, 50, 50),random_state=1,
      learning_rate_init=0.01, max_iter=50, activation='relu',solver='adam')
```

Approach for selecting hyperparameters:

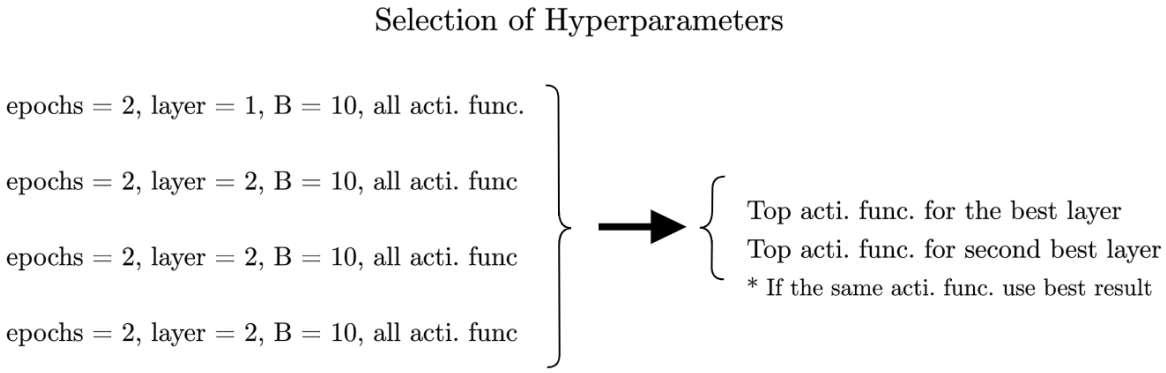

Figure 26: The figure shows the first step in the search for the optimum hyperparameter for the Franke regression.

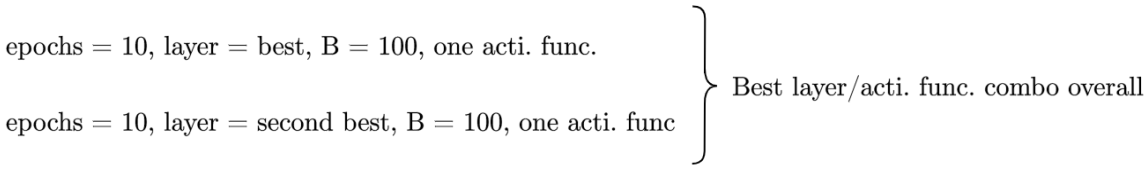

Figure 27: The figure shows the second and last step in the search for the optimum hyperparameter for the Franke regression.

3.4 PCA

The correlation plot in Figure28 shows that many of the features in the Breast Cancer Wisconsin (Diagnostic) Data Set are correlated.

For this reason and for computational efficiencies we evaluated Principal Component Analysis (PCA) to reduce dimensionality (Géron 2017). We therefore tried this with various numbers of pca vectors but got slightly more favourable results on the full feature set.

Following this evaluation we therefore excluded PCA from our further use on the data set.

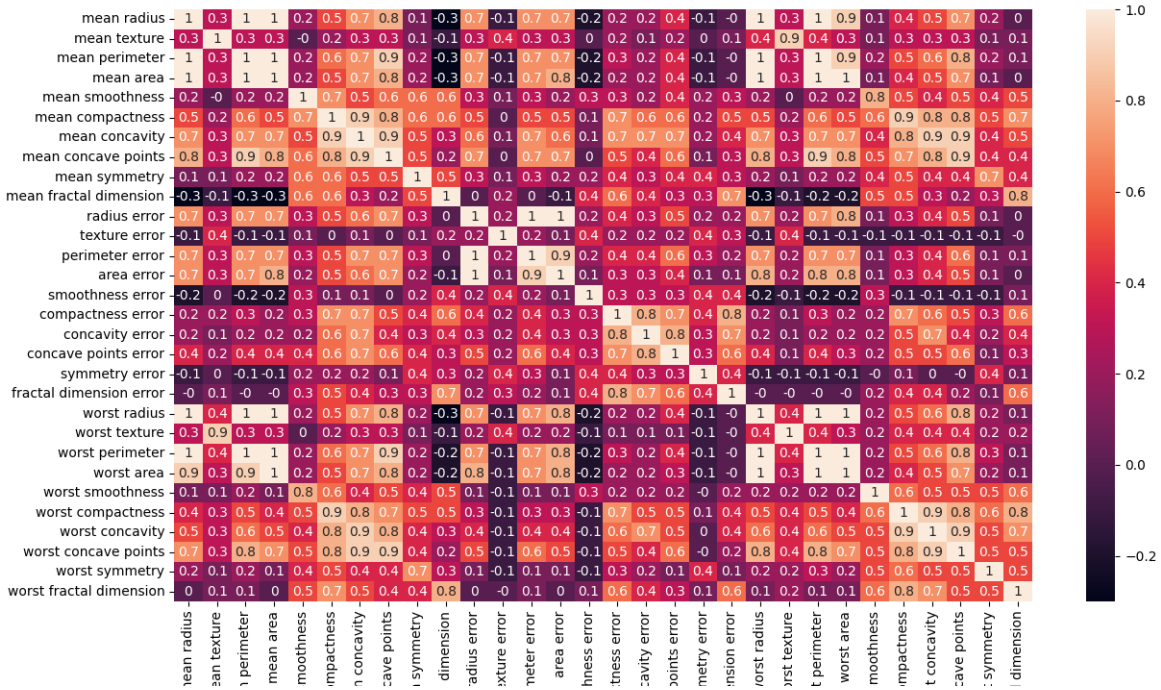

Figure 28: Correlation Matrix

3.5 Logistic Regression

For logistic regression we created a separate python code, in order to compare its performance with the NN, in particular the NN with zero hidden layers and a sigmoid output function. This NN is then identical to a logistic regression.

For the logistic regression we split our data set in a train and a test set, and we scaled the features based on the train set. We then created the design matrix $\mathbf{X}$ which has an intercept column in the first column and then the entire feature set in the other columns. The intercept column is there to have a model that could provide an output $p = e^{\beta_0}$ instead of $p = \frac{1}{2}$ if all the features $x_1, x_2, ...x_p$ are zero.

We then proceeded with hyperparameter optimization. The method is similar to the one described in the next section and the results and the iterative steps are described in section 4.

3.6 Classification of the WDBC using FFNN

In order to use our FFNN as a classifier we changed the output layer to have the sigmoid activation function and we set the cost function to be the Binary Cross Entropy, with associated derivative as described in the theory section.

We set a threshold of 0.5 for the classification. Probabilities higher than 0.5 would be mapped to 1 and below to 0.

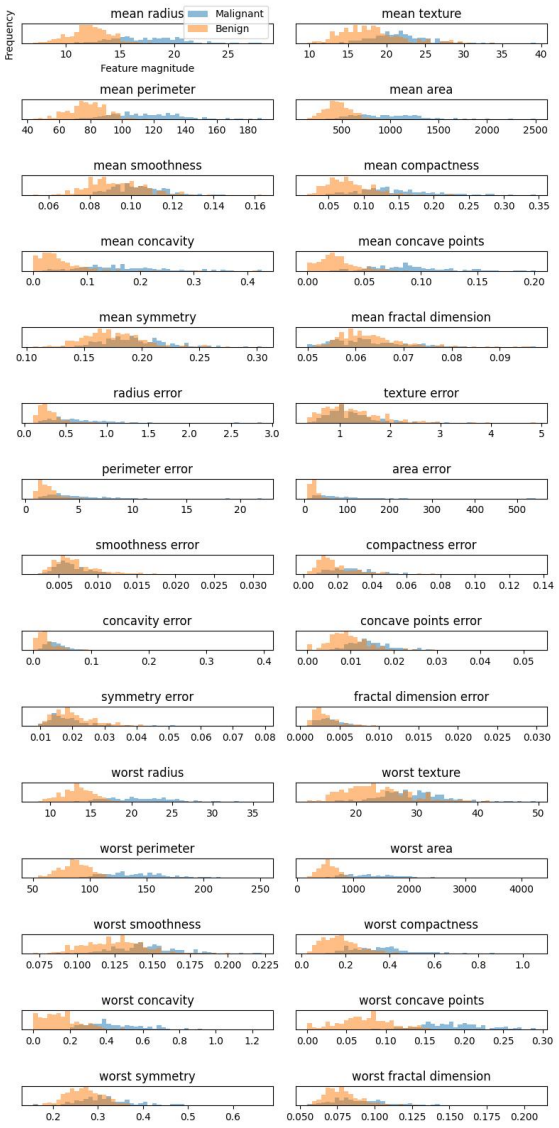

Figure 29: Features in the WDBC data. Blue is malignant, orange benign

As we can see the two categories are partly separated in many of the features. Our aim is to use the information from all of these features combined to make the best possible prediction.

We first split our data set in a train and a test set, and we scale the features based on the train set. We then proceeded with hyperparameter optimization.

3.6.1 Neural Network Hyperparameter Optimization

General: A neural network is a very flexible design, where the learning process and the architecture itself has a lot of hyperparameters. This benefit poses also the challenge of finding the best model. Each hyperparameter defines a dimension, either integer or continuous. The combination of n hyperparameters forms an n-dimensional hyperspace. There are infinite possibilities. Finding the best model is in itself an optimization problem.

In order to select hyperparameters we need to be able to establish when we can have have some confidence that the model has improved. For this reason we need to consider the uncertainty in the statistics from a single optimization of the NN. With few samples the test-train split and other randomized factors can have a large impact on the statistics obtained, for example if several

outliers happen to lie in the test set or not, or how the SGD performed. It would be impossible to compare small incremental performance improvements from a single optimization run. And if we decided to set a fixed random seed, then we could have found a good model for that particular seed that may not generalize with true randomness.

To gain some intuition and narrow our search space we started manually, and followed the advice on hyperparameter search provided in Geron's book (**geron**). We started with just one layer and changed the number of neurons.

We then decided to create a Python code that could search in the hyperspace and rank optimum champion and challenger NN models. For each set of hyperparameters the code would run B bootstraps creating B neural networks and then obtain the statistics from those.

We decided to use accuracy score as our performance metric. With k correctly predicted samples out of a sample size of n in the test set, this score is limited to a finite set of $n+1$ rational numbers, as it comes from the fraction accuracy = $\frac{\mathrm{k}}{\mathrm{n}}$. This limitation is of no concern with larger sample sizes and when taking the means of bootstraps as we did. We also noted that the data set is slightly skewed, but not to the extent that it would cause a concern for our purpose of using accuracy as a metric.

With these concerns in mind we decided to bootstrap the entire process in B bootstrap runs and obtain the mean and variance of cost and accuracy score values from these B bootstrap samples (Efron and Tibshirani 1994).

We then created a code to search in the hyperparameter space and stored the five best values obtained from each search.

For speed we first did a coarse search in a large set of hyperspace values, with only B=10 bootstraps. This gave an idea on the best combinations of hyperparameters.

With this information we then narrowed the search a bit, excluding some hyperparameter values and now searched with B=100 for each search run.

Finally we could see that cost and accuracy stabilized and we could proceed to the final tuning with B=1000 for a narrow search space. The final model was then found, and we repeated runs with B=1000 on that model to verify that we had a statistics with a high level of confidence. The results from this hyperparameter search is available in a file in our GitHub repository.

This search method was a bit naïve in the sense that our search algorithm was not programmed to learn. There are more refined methods available and for projects with even more hyperparameters we could have considered a randomized approach to the search, or dedicated search algorithms (Géron 2017). We describe the results obtained iteratively in section 4.

4 Results

The result section is divided into two. First the results from the regression part of the project followed by the results from the classification part of the project.

4.1 Regression

Results from regression on the Franke Function. Below are eight plots showing first a cost heatmap of the hyperparameter space for number of nodes and the regularization parameter λ for ReLU and Leaky ReLU respectively. These are an average of three runs. Then follows plots of the cost and R2 score for ReLU and Leaky ReLU respectively. These are an average of 100 runs. Lastly you can find a surface plot of the networks approximation of the Franke function with a 2d contour projection of the difference between the NNs aproximation and the actual Franke function. These are also an average of 100 runs.

The complete set of results can be found on GitHub via github.com/JouvalSomer under the section FYS-STK3155 and Project 2.

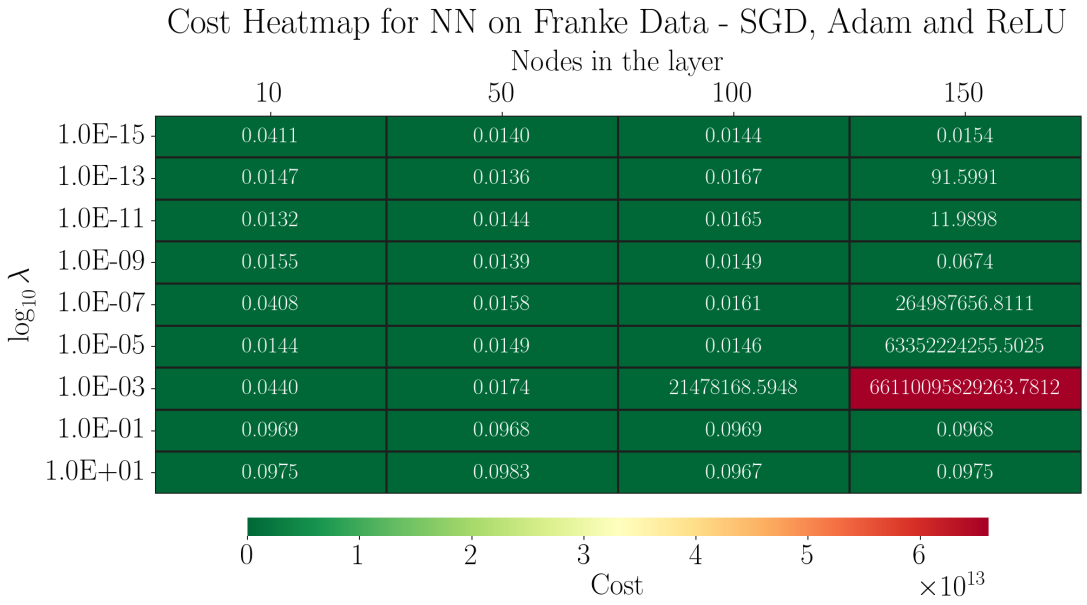

Figure 30: A heatmap of cost as a function of number of nodes and different regularization parameters λ for the the NN with 3 layers and the ReLU activation function. The costs are an avrage of three runs. Some of the values are extremely large and are most likely a result of some computational instability. The lowest cost is found for when using 10 nodes and $\lambda = 10^{-11}$ and is equal to 0.0132. These values for number of nodes and λ was then used in the rest of the runs for ReLU.

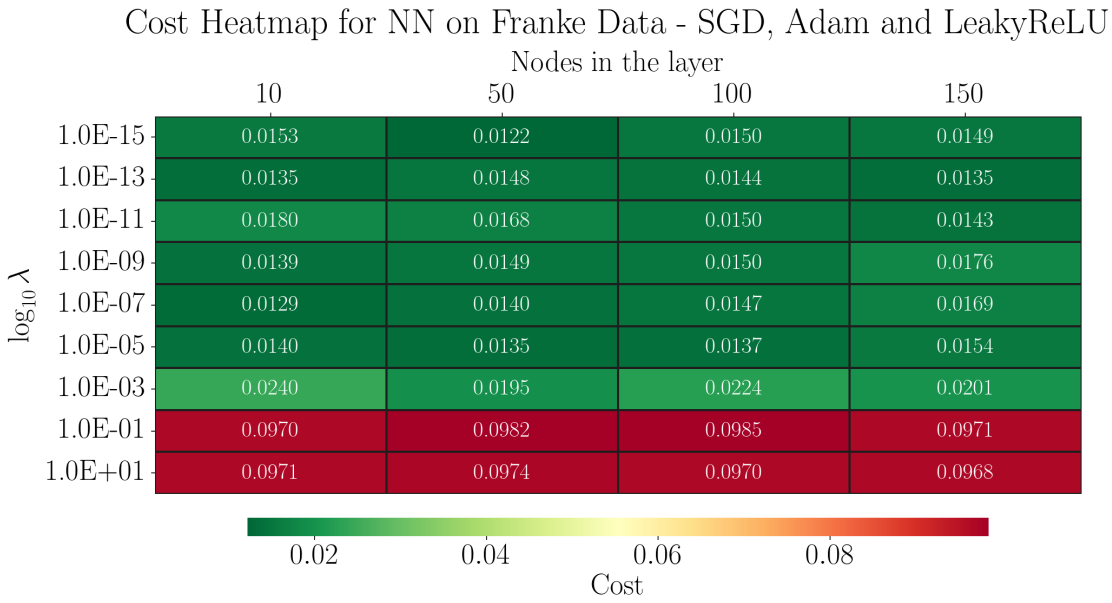

Figure 31: A heatmap of cost as a function of number of nodes and different regularization parameters λ for the the NN with 2 layers and the LeakyReLU activation function. The costs are an avrage of three runs. Some of the values are extremely large and are most likely a result of some computational instability. The lowest cost is found for when using 50 nodes and $\lambda = 10^{-15}$ and is equal to 0.0122. These values for number of nodes and λ was then used in the rest of the runs for LeakyReLU.

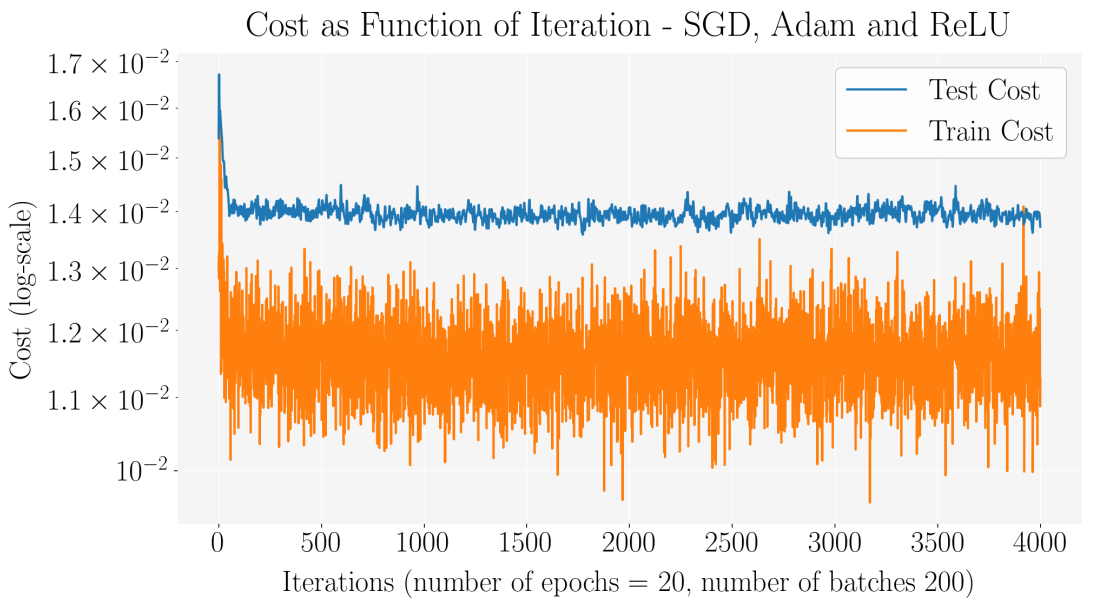

Figure 32: The cost for the train and test data of the NN on the Franke function using 3 layers and the ReLU activation function. The cost is sampled at each update in the SGD thus giving 4000 points for 20 eposhs and 200 minibatches. The test cost is centered approximately $\approx 2 \times 10^{-3}$ higher that the train cost.

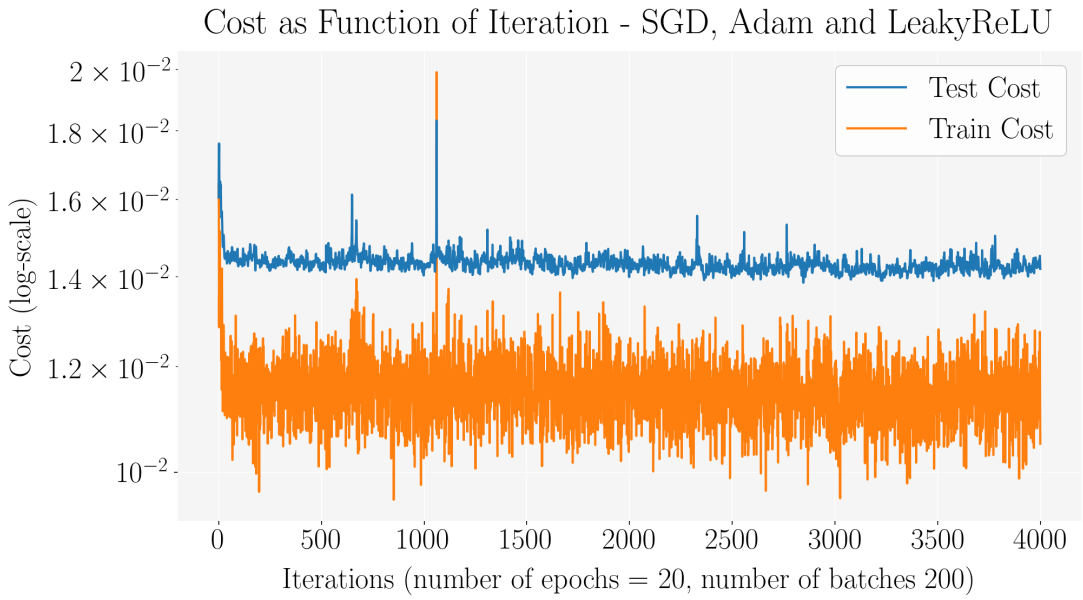

Figure 33: The cost for the train and test data of the NN on the Franke function using 2 layers and the LeakyReLU activation function. The cost is sampled at each update in the SGD thus giving 4000 points for 20 eposhs and 200 minibatches. The test cost is centered approximately $\approx 2.5 \times 10^{-3}$ higher that the train cost.

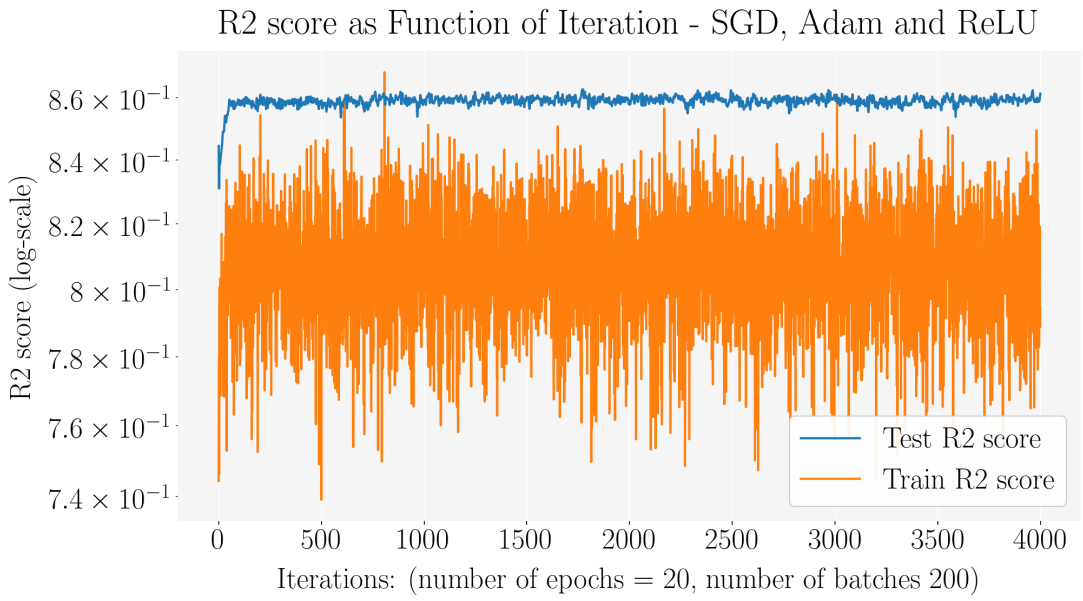

Figure 34: The R2 score for the train and test data of the NN on the Franke function using 3 layers and the ReLU activation function. The R2 score is sampled at each update in the SGD thus giving 4000 points for 20 epochs and 200 minibatches. The test cost is centered approximately $\approx 5 \times 10^{-3}$ higher that the train R2 score.

Figure 35: The R2 score for the train and test data of the NN on the Franke function using 2 layers and the LeakyReLU activation function. The R2 score is sampled at each update in the SGD thus giving 4000 points for 20 eposhs and 200 minibatches. The test cost is centered approximately $\approx 5 \times 10^{-4}$ higher that the train R2 score.

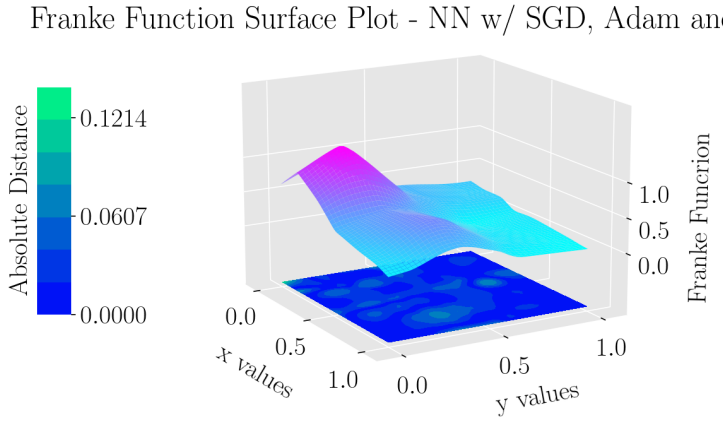

Figure 36: The figure shows a surface plot of the NN-approximation of the Franke function after traning for 20 epochs, using 200 minibatches and the ReLU on 3 hidden layers. The model is an average of 100 runs. In the xy plane you see a projection of the absolute difference between the true Franke function and the NN-approximation. The largest difference is a bit over 0.01214. The smallest difference of 0.0000. Note the colormap of the surface plot is just showing the elevation of the surface and does not have any other metric associated with it.

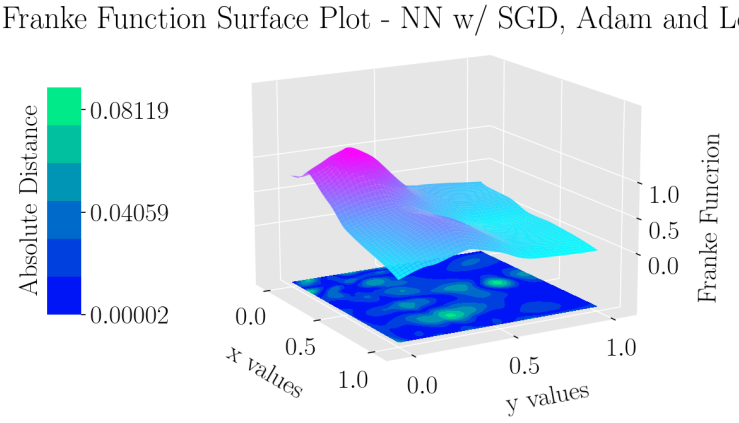

Figure 37: The figure shows a surface plot of the NN-approximation of the Franke function after traning for 20 epochs, using 200 minibatches and the LeakyReLU on 2 hidden layers. The model is an average of 100 runs. On the xy plane you see a projection of the absolute difference between the true Franke function and the NN-approximation. The largest difference is a bit over 0.08119. The smallest difference of 0.0002. Note again that the colormap of the surface plot is just showing the elevation of the surface and does not have any other metric assisted with it.

All in all the NNs regression of the Franke function we were able to achieve a cost of 0.0122 based on the average of three runs. We also got a maximum of absolute difference of 0.08119 and a minimum of the absolute difference of 0.00002 between the true Franke function and the NNs prediction as average of 100 runs. These results were achieved with a model using SGD as solver with minibatch size of 10 giving 200 minibatches per epoch, LeakyReLU as activation layer, Adam as optimizer, 2 hidden layers with 50 nodes each and a regularization parameter $\lambda = 10^{-15}$. Essensially a MSE. The highest R2 score was achieved with ReLU and was 0.8627. These results were achieved with the same model as above with the only discrepancies being 2 layers, 10 nodes with λ = 1e-11.

The sklearn MLPregressor got an R2 score that converged to around 0.86.

4.2 Classification

Results from classification of the Breast Cancer Wisconsin (Diagnostic) Data Set is divided in two subsections, one for the FFNN classification and the second with the logistic regression.

4.2.1 Neural Network classification

The final model achieved a mean accuracy of 0.980 with 1000 Bootstraps with the following hyperparameters:

One hidden layer with 20 nodes, SGD with ADAM with 25 epochs, 30 minibatch size, lambda 0.000875. test size: 0.20. Activation function: Sigmoid. PCA: Not used.

We verified our FFNN by comparing the performance with SkLearn, which achieved similar performance, at 0.970 with 100 Bootstraps. We did not test Sklearn at 1000 bootstraps due to time issues and associated convergence warnings.

The hyperparameter search space was narrowed down in an iterative way. First we started with a wide space. We typically selected 5-10 values within each of the intervals looped over all of them in a nested for loop. The results are provided in the table below.

For each setting of the parameters we created B bootstrap neural networks. We reviewed the mean accuracy score and the associated standard deviation with our search algorithm.

We tuned in several iterations in this wide space, gradually adjusting parameter ranges. After several runs we had achieved a level of confidence on which parameters we could narrow down, and then we performed the same process multiple times in a medium sized search space with these new parameter ranges set.

Finally we could proceed to a final search in a narrow search space. In this region the mean results from a bootstrap run were almost similar. We had to increase to B=1000 to gain confidence in which changes could incrementally improve the model further.

See table 1 for hyperparameters below, and also the selected B bootstrap for each search space. The search in each search space was also done multiple times before we had confidence in the set of hyperparameters for a narrower space.

In the table activation functions are abbreviated as: Sigmoid (S), Hyperbolic (H), ReLu (R), LeakyReLu (L), ELU (E)

Hyperparameter	Wide Search	Medium Search	Narrow Search	Final Model
Hidden Layers (L)	0,1,2,3	1,2	1	1
Nodes in each L (N)	1-50	10-30	18-22	20
Epochs	10-1000	20-100	20-40	25
Minibatch size	1-50	15-25	18-22	30
Lambda (λ)	0-10	0-0.1	0-0.01	0.00087
Activation in HL	S,H,R,L,E	S,H	S	S
Test size	0.05 - 0.20	0.20	0.20	0.20
PCA (Y/N) components	Y/N 1-30	Y/N 3-20	N	N
Bootstraps (B)	10,50,100	100,1000	1000	1000

Table 1: Search Space for the FFNN Classification problem

Example run report:

RESULT FROM THE LAST RUN IN THE BOOTSTRAP:

Correctly scored 112 out of 114. Incorrectly scored 2 out of 114. Scored positive by the NN: 71 Scored negative by the NN: 43 Correctly scored positive by NN out of total positive: 69/69 Correctly scored negative by NN out of total negative: 43/45

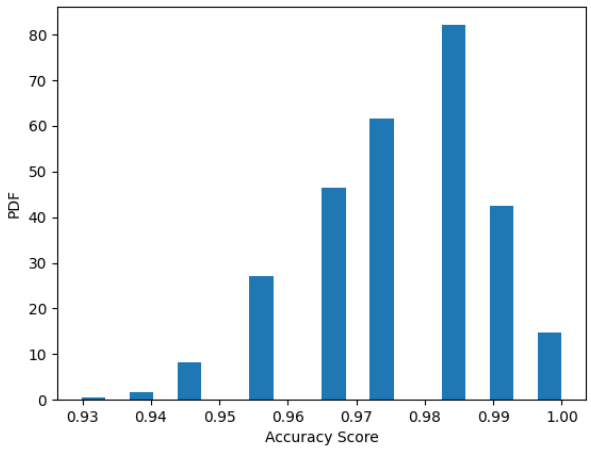

Figure 38: FFNN Regression - Accuracy score histogram from a run of 1000 FFNN bootstraps

4.2.2 Logistic Regression

With a neural network with no hidden layers and the sigmoid function in the activation layer, we end up with a logistic regression. The bias in the neural network is then the intercept β_0 in the logistic regression. Finally we have the same sigmoid function in both cases.

Nevertheless we created a separate code for logistic regression and compared with the neural network configured as a logistic regression and the Sklearn logistic regression. We let all three methods have 1000 bootstrap runs in order to compare the results.

We did a hyperparameter search similar to the one discussed for neural networks, first on the logistic regression code, then on the same for a neural network as logistic regression. The search results are provided in our Github repository.

The final logistic model achieved a mean accuracy of 0.974 with 1000 Bootstraps with the following hyperparameters:

SGD with ADAM with 500 epochs, 30 minibatch size, lambda 0.1. Test size: 0.20. Activation function: Sigmoid. PCA: Not used.

The final neural network model with no hidden layers has a mean accuracy of 0.975 with 1000 Bootstraps with:

SGD with ADAM with 40 epochs, 15 minibatch size, lambda 0. Test size: 0.20. Activation function: Sigmoid. PCA: Not used.

SkLearn achieved similar performance. We used Sklearn 1000 times in a bootstrap and achieved a mean accuracy score of 0.977.

The accuracy varies from each bootstrap run. The Accuracy score was within a 95% interval of = [0.93 0.99].

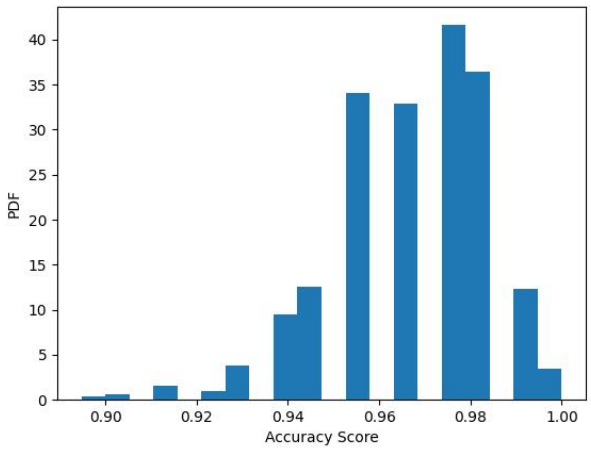

Figure 39: Logistic Regression - Accuracy score histogram from a run of 1000 bootstraps

5 Discussion

5.1 Regression

5.1.1 Hyperparameters λ and Number of Nodes in Each Hidden Layer

There is not much difference in the model complexity when it comes to the number of nodes. At least between 10 and 150 nodes as one can see in Figure 26. The lowest cost for ReLU as achived with 50 nodes and $\lambda = 10^{-15}$

5.1.2 Cost and R2 score

A way to test overfitting of a neural network is to compare the difference in cost for the train and test data set. If the train is much lower one could be overfitting the train data set. In our case the differences in train and test were relatively, only $\approx 2 \times 10^3$ and $\approx 2.5 \times 10^3$ higher that the train cost calculated form the center of the train and test films. These distances are of the same scale as the with of the train fluctuations and are therefore acceptable. The cost seems to have reached a convergence and is probably why we did not get much better results for higher values of epochs. One thing that we found strange and that could be interesting to explore further is the fact that the test R2 score is higher than the train score. We used scikit-learns r2˙score and have not had any issues with it prior so this was not expected.

5.1.3 Activation Functions

From the results for the different layers with only 2 epochs and 200 minibatches one could quickly see that the activation functions Sigmoid, hyperbolic tangent and ELU were not very good for this regression problem. They attained a cost that was at least twice that of what ReLU and LeakyReLU achieved. The gradients of the Sigmoid and hyperbolic tangent may have vanished due to their asymptotic nature, but why ELU performed so poorly compared with say ReLU is hard to say as it is essentially a smoothed out ReLU with a negative x side equal to $-\alpha$ or -1 in our case.

5.1.4 GD vs. SGD

For our case it was obvious to use the SGD method as we were limited by computational power thus benefiting greatly form the stochastic estimation of the gradient that the SGD does. When that is said we did take an average of 100 runs. But at the same time finding the optimum hyperparameters would have taken a lot longer with GD, so this way we could chose to use an extensive amount of data points just when we needed a good result and not when searching for the best model.

5.1.5 Comparison

When comparing the results for cost and R2 score with those obtained in project 1 we got both a higher R2 score and a lower cost with the NN compared with the analytical matrix inversion method OLS, Ridge and LASSO. (Se Table 2, section 4.4.2 in Patey and Somer 2022) This Ridge with cross-validation from project 1 got the lowest cost with 0.2072 while we here got 0.0122 with $\lambda = 10^{-15}$ essentially an OLS MSE, and the higest R2 score from project 1 was 0.7769 whilest the NN got 0.8627. A factor that should be taken into consideration here is that fact that the data from project one was scaled and the data for this project was not. On the other hand the Gaussian noise for this was scaled twice as much for this project compared with the latter.

5.1.6 Further improvements

There is a lot more exploring to be done in the search for the optimum model. We were held back due to computational time. With a powerful computer exploring a larger minibatch size or even GD would be interesting as the network would have more data to work with for each epoch. In addition one could explore a grater range of model complexity to get a firmer grasp on how the hyperparameters space looks like for such a regression problem. As detailed in the theory section, according to the universal approximation theorem for neural networks Goodfellow et al. 2016 one would in theory be able to approximate the smooth Franke function within an arbitrarily small number ϵ using a single hidden layer.

5.2 Classification

In order to evaluate our the performance of our NN classifier we decided to use the accuracy score as our quantitative measure of its performance. This measures the proportion of test examples for which the model produces the correct output (Goodfellow et al. 2016).

With our search for optimum hyperparameters we noted that a wide range of possible combinations had similar performance. While the best performers used only one hidden layer, some models with two hidden layers had almost the same performance. We were able to get a mean accuracy score of 0.980 on 1000 bootstraps. This performance is comparable to other studies on neural networks on the Wisconsin breast cancer dataset (Prakash and Visakha 2020). Compared to a logistic regression we noted that the NN with one layer got slightly better result, but the logistic regression model also performed very well and it could be of importance for the end user that it is a simpler model that is easier to interpret. This can also have implications for the patient, as for certain health care, such as breast cancer diagnosis discussed here, there are also concerns in Europe that the black box nature of a neural network may not meet the intent of certain legislation regarding transparency in communication of health care decisions (Mourby et al. 2021). This could favour the logistic regression model.

6 Conclusion

In this project we studied regression and classification problems through the development of our own feed-forward neural network (FFNN) code. We compared the performance of a FFNN with traditional methods such as linear regression and logistic regression.

For the NNs regression of the Franke function we were able to achieve a MSE of 0.0122 based on the average of three runs. We also got a maximum of absolute difference of 0.08119 and a minumum of the absolute difference of 0.00002 between the true Franke function and the NNs prediction as average of 100 runs.

The NN out performed the analytical methods from last project by 0.0122 against 0.2072 for the MSE cost and 0.8627 to 0.7769 for the R2 score.

For classification we had high expectations on the performance of a neural network, and even though we managed an accuracy score of 98.0% the logistic regression achieved 97.4%. For the end user the practicalities and interpretability of the logistic regression model may make it the chosen model, even considering the lower accuracy.

Despite our assumptions that PCA decomposition could improve a model, we saw that it did not on the WBDC data set. For classification we got the best results for the FFNN with the traditional sigmoid function in the hidden layer, a priori we had assumed that we could have achieved better performance with more recent activation functions. It could be that a further search in the hyperparameter space could have yielded better results. We noted that hyperparameter search is an interesting optimization problem that could warrant further studies.

Bibliography

Binder, Harald (2018). *COMPUTER AGE STATISTICAL INFERENCE B. Efron T. Hastie (2016). New York, NY: Cambridge University Press. 475 pages, ISBN 978-1-107-14989-2.*

Clevert, Djork-Arné, Thomas Unterthiner and Sepp Hochreiter (2015). 'Fast and accurate deep network learning by exponential linear units (elus)'. In: *arXiv preprint arXiv:1511.07289*.

Dalal, Siddhartha R, Edward B Fowlkes and Bruce Hoadley (1989). 'Risk analysis of the space shuttle: pre-Challenger prediction of failure'. In: *Journal of the American Statistical Association* 84.408, pp. 945–957.

Duchi, John, Elad Hazan and Yoram Singer (2011). 'Adaptive subgradient methods for online learning and stochastic optimization.' In: *Journal of machine learning research* 12.7.

Efron, Bradley and Robert J Tibshirani (1994). *An introduction to the bootstrap*. CRC press.

Géron, Aurélien (2017). 'Hands-on machine learning with scikit-learn and tensorflow: Concepts'. In: *Tools, and Techniques to build intelligent systems*.

Glorot, Xavier and Yoshua Bengio (2010). 'Understanding the difficulty of training deep feedforward neural networks'. In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, pp. 249–256.

Goodfellow, Ian, Yoshua Bengio and Aaron Courville (2016). *Deep learning*. MIT press.

Hastie, Trevor, Robert Tibshirani and Martin Wainwright (2015). 'Statistical learning with sparsity'. In: *Monographs on statistics and applied probability* 143, p. 143.

Hecht-Nielsen, Robert (1992). 'Theory of the backpropagation neural network'. In: *Neural networks for perception*. Elsevier, pp. 65–93.

Hinton, Geoffrey, Nitish Srivastava and Kevin Swersky (2012). 'Neural networks for machine learning lecture 6a overview of mini-batch gradient descent'. In: *Cited on* 14.8, p. 2.

JL, McCulloch (1943). 'A logical calculus of ideas immanent in nervous activity'. In: *Bull. of Math. Biophysics* 5, pp. 115–133.

Kingma, Diederik P and Jimmy Ba (2014a). 'Adam: A method for stochastic optimization'. In: *arXiv preprint arXiv:1412.6980*.

— (2014b). *Adam: A Method for Stochastic Optimization*. DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980.

Misra, Diganta (2019). 'Mish: A self regularized non-monotonic neural activation function'. In: *arXiv preprint arXiv:1908.08681* 4.2, pp. 10–48550.

Mourby, Miranda, Katharina Ó Cathaoir and Catherine Bjerre Collin (2021). 'Transparency of machine-learning in healthcare: The GDPR & European health law'. In: *Computer Law & Security Review* 43, p. 105611.

Patey, Stig and Jouval Somer (Oct. 2022). *A linear regression model for terrain data*. URL: https://github.com/JouvalSomer.

Prakash, Sidharth S and K Visakha (2020). 'Breast cancer malignancy prediction using deep learning neural networks'. In: *2020 Second International Conference on Inventive Research in Computing Applications (ICIRCA)*. IEEE, pp. 88–92.

Wikipedia (2022). *Mechanical Turk — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Mechanical%20Turk&oldid=1121128526. [Online; accessed 11-November-2022].

Appendix

A Github Repository

The code and associated test runs for this project is available at: The source files, test results and other files can be found at the GitHub: https://github.com/JouvalSomer/FYS-STK3155