Exploring Gradient Descent Methods for Neural Network Applications: A Case Study on Breast Cancer Prediction

Iris Bore, Frida Lien & Emma Storberg

November 6, 2024

Abstract

Neural networks are highly topical at the moment, both in popular science and academic circles, with a wide range of uses across many disciplines. One area where neural networks can make a big impact is in diagnostic aid for medical professionals. However, there are challenges related to the implementation of neural networks, both in terms of their creation and interpretation. In this project, we build a neural network from scratch to predict instances of breast cancer in the Wisconsin Breast Cancer (Diagnostic) Dataset [1]. This is done step by step, beginning by implementing a variety of gradient descent techniques as standalone methods for determining optimal parameters in linear regression. These methods are thereafter implemented for use in the training of our neural networks. First, we contrasted our neural networks with linear regression techniques for prediction of numerical values from a second-degree polynomial, followed by a binary classification task, which we compare to logistic regression. The performance of our gradient descent methods alone aligned decently with the theory, with the momentum-based methods in particular performing well. We were able to train a network to predict polynomial data with an MSE of 2.5 after 10 epochs, which is worse than linear regression (especially OLS, which was almost indistinguishable from the exact solution). When it came to classification, however, we saw strange behavior of our own code, with the network performing well on smaller multiclass datasets, but not being able to predict breast cancer data. We continued our analysis with a neural network implemented with PyTorch instead, and found that the best model (one hidden layer with 41 nodes, activated by Leaky ReLU) could predict with 96% accuracy. We conclude that neural networks are a useful tool for some problems, but not all. We must be wary of their drawbacks and shortcomings before choosing to use them in real-world (especially medical) contexts.

Contents

1 Introduction

Neural networks are a highly relevant machine learning technique in today's digital era, with a vast range of uses in various fields, from healthcare to finance and beyond. They are increasingly employed in medical diagnostics to analyze complex data patterns and assist in early disease detection, with striking accuracy in preliminary testing [2]. This report will explore some of the fundamental techniques used in the creation and training of neural networks, aiming to produce models based on real medical data.

When we make predictions, it is important to have a way of assessing their accuracy. This can be done with a cost function, as covered in detail in our previous report *Linear Regression and Basic Resampling Techniques for Modeling 2D Datasets* [3]. We utilized the fact that we can minimize the cost of some choice of model parameters by finding where the derivative of the cost function is 0, and we will make use of the same observation to determine which model parameters to use in our neural network. The key difference from our previous approach is this: Rather than determining the optimal model parameters explicitly using derived analytical expressions, we will instead use a numerical method to iterate stepwise closer to the minimum of the cost function. This stepping method is known as *gradient descent*.

In this report, we first consider this process in isolation, and thereafter compare the results of these numerical techniques to their analytical counterparts. Once we confirm that our gradient descent methods work as intended, we will implement them as a way of training a neural network.

Our previous work used various linear regression methods for predicting 2D datasets. Taking a step back from this, we will train the neural network on a dataset generated from a simple second-degree polynomial as an initial test of its numerical prediction capabilities. Afterwards, we will explore another common usage of neural networks, namely classification tasks, and compare their performance to that of logistic regression. For this purpose, we will consider the Wisconsin Breast Cancer (Diagnostic) Dataset [1], aiming to predict the presence of cancer in each patient with the highest degree of accuracy possible. Finally, we will evaluate how the different methods of regression and classification fare in comparison to one another across various gradient descent techniques.

2 Method

2.1 Gradient Descent

In our previous report [3], we looked at elementary linear regression techniques for prediction. The cost functions we considered were largely used simply as a step in deriving the analytical expressions for optimal model parameters $\hat{\boldsymbol{\beta}}$. This allowed us to find $\hat{\boldsymbol{\beta}}$ explicitly and without iteration. In contrast, gradient descent methods are numerical methods that approximate the optimal model parameters by moving closer to them step by step.

The method for this first part of the experiment will be to use gradient descent methods for OLS and ridge regression. We have chosen this as an first test of our gradient descent implementation for a few reasons.

First, the analytical solutions for the optimal model parameters provide a great reference point for the performance of the numerical methods, which will be useful to assess before we continue. Additionally, due to the cost functions of OLS and ridge regression both being convex, we will not end up in local minima [4], which would prevent us from finding the model parameters for which the cost function is minimized.

This report employs a few different variations of gradient descent. In the most basic case, typically denoted *plain gradient descent*, we simply wish to leverage the core concept that a function $F : \mathbb{R}^n \to \mathbb{R}^n$ decreases fastest in the direction of the negative gradient $-\nabla F(\boldsymbol{x})$. Hjorth-Jensen [5] introduces the following expression:

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t - \eta_t \nabla F(\boldsymbol{x}_t), t \geq 0$$

for some scalar $\eta_t > 0$, a parameter we call the *learning rate* in the context of machine learning.

This expression shows how we can, through many iterations, nudge an initial starting vector $\boldsymbol{x}_0$ towards the point where F is at its minimum, which is exactly what we are trying to do for some cost function C when training models. It can be shown that for small enough η_t, $F(\boldsymbol{x}_{t+1}) \leq F(\boldsymbol{x}_t)$ [5], or in other words, that we will move towards a function minimum.

Empirically, the value of η_t seems to greatly affect the output produced, which makes the learning rate infamously difficult to tune properly [6]. In essence, it works like a step size, telling us how far we are allowed to move in the direction of the gradient we have determined. Though we can set it to a constant value, more often than not, η_t will vary based on the iteration t (hence the subscript). Given the random initialization of our starting point, we can assume we are positioned quite far away from the cost function minimum when we begin iterating, and as such, we may want to take large steps in the direction of the minimum. But as we move closer, it may be useful to decrease the step size to more approach the minimum with precision, and avoid missing it entirely. As a crude solution, we can define a learning rate η_t that decays linearly up to some iteration τ, after which it is commonly left constant [6]:

$$\eta_t = (1-\kappa)\eta_0 + \kappa\eta_\tau$$

with $\kappa = \frac{t}{\tau}$, and η_τ typically set to 1% of η_0 [6]. The challenge, therefore, is setting the initial value η_0, which we will try to determine in our experiments. Other ways of tuning η_t, so-called *adaptive* learning rates, will be explored in detail later on.

2.1.1 Stochastic Gradient Descent

The idea of moving step-wise from some initial input $\boldsymbol{x}_0$ to the minimum of the cost function is the utilized across all of the gradient descent methods we will consider, with only a few modifications to address issues that arise along the way when using only this naive approach. For instance, the typical scale of the datasets used in many predictive algorithms today (both in terms of data points and features) is so large that plain gradient descent quickly becomes computationally infeasible, even for relatively simple problems [7]. To combat this, we can utilize *stochastic gradient descent*. The key idea here is that instead of computing the gradient of every single data point in every iteration, we can introduce some stochasticity by randomly picking one point every time, which will reduce the number of calculations by many orders of magnitude [7].

The justification for stochastic gradient descent as a strategy hinges on the underlying idea that we can almost always decompose the cost function as a sum over all the training samples [6], which in turn means its gradient is expressed as a sum over gradients in the individual data points:

$$\nabla_\theta C(\boldsymbol{\theta}) = \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} c_i(\boldsymbol{x}_i, \boldsymbol{\theta})$$

where n is the number of data points, and $\boldsymbol{\theta}$ is a vector containing all the model parameters we optimize with respect to[1].

Note that the formal definition of stochastic gradient descent says to choose only one point per iteration, but in practice, it is common to choose a subset of the data for which to calculate the gradients [5]. This is known as a *mini-batch*, and utilizing mini-batches to calculate more than one gradient per iteration will, perhaps unsurprisingly, lead to faster convergence towards the analytical gradients than with only a single point at a time [5]. Mini-batches are especially useful when our data is clustered [7], as we can construct the mini-batches in such a way that we choose one representative from each grouping of data points per iteration, with the aim of facilitating quick and accurate convergence towards the true model parameters.

Thus, the definition of stochastic gradient descent (with mini-batches B_k, represented here as disjoint sets with the indices of the data points they contain) is shown below:

$$\nabla_\theta C(\boldsymbol{\theta}) = \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} c_i(\boldsymbol{x}_i, \boldsymbol{\theta}) \rightarrow \sum_{i\in B_k}^{n} \nabla_{\boldsymbol{\theta}} c_i(\boldsymbol{x}_i, \boldsymbol{\theta}),$$

[1]In linear regression for polynomial approximation, $\boldsymbol{\theta}$ will be the coefficients of the polynomial, which we referred to as $\boldsymbol{\beta}$ in our previous report [3].

resulting in a gradient descent step that looks like this:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \sum_{i \in B_k}^{n} \nabla_{\boldsymbol{\theta}} c_i(\boldsymbol{x}_i, \boldsymbol{\theta}_t).$$

We pick k at random with equal probability from $[1, \frac{n}{M}]$, where M is the number of mini-batches. An iteration over M is called an *epoch* [5]. In our implementation, we will follow the common approach (and in particular the examples highlighted by Hjorth-Jensen [5]), such that we first choose a number of epochs, and for each epoch we iterate over the number of mini-batches. We have also chosen to randomly sample our mini-batches without replacement, as this seems to be the most common approach in the literature, perhaps owing to its "superior empirical performance", as described by Raschka [8].

2.1.2 Gradient Descent with Momentum

In practice, we almost always use stochastic gradient descent with an additional *momentum* term, which is functionally the memory of the direction we moved in parameter space in previous iterations [5]. This can be a useful technique to mitigate two common problems. First of all, only taking into account the current gradient means that at saddle points, the gradient will be inconsequentially small, leading to small or no weight updates, and the entire learning process will stagnate [9]. This is known as the *vanishing gradient problem*, which is the first issue the addition of a momentum term aims to rectify.

The second challenge we face is that the movement through parameter space by gradient descent is very jittery, even when we "smooth out" the path by considering the average mini-batch gradient [9]. The main idea to grasp here is that in order to tackle both of the issues we have looked at, we would benefit from incorporating information about the gradient in a surrounding area, and in particular, the path we have followed in previous steps. The algorithm *gradient descent with momentum* achieves exactly this; its implementation is described by the equations below:

$$\begin{aligned}
\boldsymbol{v}_t &= \gamma \boldsymbol{v}_{t-1} + \eta_t \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i \in B_k}^{n} c_i(\boldsymbol{x}_i, \boldsymbol{\theta}) \right) \\
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \boldsymbol{v}_t
\end{aligned}$$

where $\boldsymbol{v}_t$ is a momentum term, and $\gamma \in [0, 1]$ is a momentum parameter that controls the rate at which contributions from previous gradients exponentially decay. Common values of γ in practice include 0.5, 0.9, and 0.99 [6], but it can also vary over time, just as with the learning rate η_t. Typical use involves initially setting it to a small value that is raised later on. However, adapting γ is generally considered less important than shrinking η_t over time [6], so we will only explore the effect of the latter in this report.

2.1.3 Adaptive Learning Rate Algorithms

At this point, we are ready to take a more sophisticated approach to the learning rate of the model. Earlier, our strategy was to take larger steps at first, when we are likely far away from a minimum, and make the step size smaller over time. Another intuitive idea we may want to incorporate is taking large steps in shallow, flat directions, and small steps in narrow or steep directions.

The double derivative, known as the *Hessian* for vector functions, contains exactly the information we need to keep track of curvature and adjust η_t accordingly. However, calculating it can be extremely computationally costly [5]. We will consider three algorithms that accomplish a similar effect without enormous computational overhead: *AdaGrad* [6], *Root Mean Squared Propagation* (RMSProp) [6] and *Adam* [10]. As such, we will set aside our crude process for shrinking η_t with the number of iterations t in favor of these methods with adaptive learning rates, meaning they can take into account the landscape when determining the step size to use. In the methods discussed from here on, we will consider η_t a global constant, which we will simply denote as η. It is typically chosen to be a value of 10^{-3} [5].

Beginning with the AdaGrad algorithm, we are able to individually adapt the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all previously computed squared gradient values [6]. This is shown in the expressions below:

$$
\begin{aligned}
\boldsymbol{g} &= \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i \in B_k}^{n} c_i(\boldsymbol{x}_i, \boldsymbol{\theta}) \right) \\
\boldsymbol{r}_t &= \boldsymbol{r}_{t-1} + \boldsymbol{g} \odot \boldsymbol{g} \\
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta \frac{1}{\delta + \sqrt{\boldsymbol{r}_t}} \odot \boldsymbol{g}
\end{aligned}
$$

where δ is a small constant we apply for numerical stability (typically around 10^{-7}) [6]. The binary operator $\odot$ can be understood as element-wise multiplication; division and the square root are also applied element-wise. Also, we call $\boldsymbol{r}_t$ the *gradient accumulation variable*, which we initialize to 0 [6].

The effect of using these expressions is that the parameters with the largest partial derivatives (i.e. in regions of steep descent) have a correspondingly rapid decrease in their learning rate (meaning we take small steps). The parameters with small partial derivatives, on the other hand, have a smaller decrease in their learning rate. This achieves greater progress in the more gradually sloped directions of parameter space, as desired [6]. In addition, as the sum of the squared gradients grows with the iterations, dividing by its square root will shrink the step size of the model over time, similar to our previous approach with η_t, but with the additional benefit of also adapting to the landscape we are in.

While these properties make AdaGrad beneficial to us in theory, it is not always so in practice [6]. For example, AdaGrad is designed for rapid convergence of convex functions, and may thus run into problems with non-convex landscapes. As it takes into account all past values of the squared gradient (captured in $\boldsymbol{r}_t$), it may have made the learning rate too small by the time it reaches a convex region where it should converge (described as a "convex bowl" by Goodfellow, Bengio, and Courville [6]).

The convex bowl exemplifies a case where our second algorithm, Root Mean Squared Propagation (RMSProp), fares better. Its update rules are:

$$
\begin{aligned}
\boldsymbol{g} &= \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i \in B_k}^{n} c_i(\boldsymbol{x}_i, \boldsymbol{\theta}) \right) \\
\boldsymbol{r}_t &= \rho \boldsymbol{r}_{t-1} + (1-\rho) \boldsymbol{g} \odot \boldsymbol{g} \\
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta \frac{1}{\delta + \sqrt{\boldsymbol{r}_t}} \odot \boldsymbol{g}
\end{aligned}
$$

where we have now brought in a decay rate ρ (usually a value of about 0.9, according to Hjorth-Jensen [5]). As we can see, the introduction of ρ allows RMSProp to use exponentially smaller contributions from past calculations, as opposed to the whole history. This functionally discards information provided from gradients in the extreme past, leading to rapid convergence once a convex bowl is found. These advantages over AdaGrad are part of what makes RMSProp such a widely-used algorithm today [6], despite the introduction of an additional hyperparameter ρ to tune. The regularization constant δ will typically be around the scale of 10^{-6} to 10^{-8} in this case [6] [5].

The AdaGrad and RMSProp algorithms are without a momentum component in the forms shown above, but we can easily add a momentum term to them and enjoy many of the same benefits as when we added a momentum term to basic stochastic gradient descent. We simply combine the equations by replacing the gradient in the momentum expression with the rescaled gradient as in the adaptive learning rate algorithm in question [6]. We show RMSProp with a momentum term added below:

$$
\begin{aligned}
\boldsymbol{g} &= \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i \in B_k}^{n} c_i(\boldsymbol{x}_i, \boldsymbol{\theta}) \right) \\
\boldsymbol{r}_t &= \rho \boldsymbol{r}_{t-1} + (1-\rho) \boldsymbol{g} \odot \boldsymbol{g} \\
\boldsymbol{v}_t &= \gamma \boldsymbol{v}_{t-1} + \eta \frac{1}{\delta + \sqrt{\boldsymbol{r}_t}} \odot \boldsymbol{g} \\
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \boldsymbol{v}_t
\end{aligned}
$$

The final algorithm of adaptive learning rate we will use is Adam. Unlike AdaGrad and RMSProp, Adam always makes use of a momentum component, and in some ways it can be seen as a variant of RMSProp with momentum [5], as we saw earlier. We will use the gradient accumulation variable $\boldsymbol{r}_t$ as previously, but this time, it is more natural to view it as the *second moment* of the gradient. One new addition is that we now also require the *first moment* of the gradient, written as $\boldsymbol{s}_t$. The first and second moments, as well as the time step t, are all initialized to 0. The update steps in Adam are displayed below:

$$\begin{aligned}
\boldsymbol{g} &= \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i \in B_k}^{n} c_i(\boldsymbol{x}_i, \boldsymbol{\theta}) \right) \\
t &= t + 1 \\
\boldsymbol{s}_t &= \rho_1 \boldsymbol{s}_{t-1} + (1 - \rho_1)\boldsymbol{g} \\
\boldsymbol{r}_t &= \rho_2 \boldsymbol{r}_{t-1} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g} \\
\hat{\boldsymbol{s}}_t &= \frac{\boldsymbol{s}_t}{1 - \rho_1^t} \\
\hat{\boldsymbol{r}}_t &= \frac{\boldsymbol{r}_t}{1 - \rho_2^t} \\
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta \frac{\hat{\boldsymbol{s}}_t}{\delta + \sqrt{\hat{\boldsymbol{r}}_t}} \odot \boldsymbol{g}
\end{aligned}$$

where we now have two parameters ρ_1 and ρ_2 that determine the memory lifetime of $\boldsymbol{s}_t$ and $\boldsymbol{r}_t$ respectively. Kingma and Ba [10] propose $\rho_1 = 0.9$ and $\rho_2 = 0.999$ as good default values for these parameters (as well as setting $\delta = 10^{-8}$), so we will run our experiments with these values as a starting point.

Despite their similarities, there are a few important differences between Adam and RMSProp with momentum that may affect performance. First, in Adam, we directly incorporate momentum as an estimate of the first moment of the gradient $\boldsymbol{s}_t$, weighted exponentially to an extent determined by ρ_1. RMSProp with momentum also only rescales $\boldsymbol{r}_t$ and not $\boldsymbol{s}_t$, while Adam rescales both, as well as applies bias corrections to the first and second moment estimators to account for their initialization at the origin. Since RMSProp with momentum does not have a correction factor, it can produce high-bias estimates early in the training process [6].

Adam is regarded as a fairly robust method, which makes it among one of the most widely-used methods of gradient descent today [6]. Other popular choices include stochastic gradient descent, stochastic gradient descent with momentum, RMSProp and RMSProp with momentum, according to Goodfellow, Bengio, and Courville [6]. They surmise that different choices of method may well have more to do with users' familiarity with the method than any theoretical benefit the method offers, as the main concern in practice becomes easing the notoriously arduous burden of tuning hyperparameters. As we proceed with our experiments, we will evaluate whether the emphasis on simplifying the hyperparameter tuning process resonates with our findings as well.

2.2 Neural Networks

In the next stage of experimentation, we wish to expand our implementation of gradient descent methods and use them in neural networks.

A *neural network* is a computational model inspired by the human brain, wherein data is processed through a series of nodes (or neurons) organized in layers. The first of these layers is known as the *input layer*, and it is here the initial data is received. The data will typically have a set of features, each of which is given its own node in the input layer. Next come the hidden layers, which, importantly, perform non-linear transformations of the input data. Finally, an output layer will produce the final output (prediction) of the network, depending on the specific problem and how many parameters we wish to predict.

Each node in a neural network performs a dot product of its input $\boldsymbol{x}$ with its associated weights $\boldsymbol{w}$, adds a bias $\boldsymbol{b}$, and applies some non-linear activation function α. Mathematically, we express this as:

$$z = \sum_{i=1}^{n} w_i x_i + b$$

$$a = \alpha(z)$$

such that a is the output at that node.

When we consider the operations done on an entire layer at once, we can instead consider a vector of nodes $\boldsymbol{a}$, a weight matrix $\mathbf{W}$ and a vector $\boldsymbol{b}$ containing the biases. Thus, we can express the nodes $\boldsymbol{a}_n$ of the nth layer with the following expressions:

$$\boldsymbol{z}_n = \alpha(\mathbf{W}\boldsymbol{a}_{n-1} + \boldsymbol{b})$$

$$\boldsymbol{a}_n = \alpha(\boldsymbol{z}_n)$$

This means that in a 2-layer network, the final output $\boldsymbol{a}_2$ can be expressed through the output of the previous layer $\boldsymbol{a}_1$ by combining the expressions above:

$$\boldsymbol{a}_2 = \alpha_2(\underbrace{\mathbf{W}_2\boldsymbol{a}_1 + \boldsymbol{b}_2}_{\boldsymbol{z}_2})$$

where α_2 is the specific activation function of that layer, and $\boldsymbol{a}_1$ is the output of the previous layer. We can also expand this expression by writing it in terms of the inputs $\boldsymbol{x}$, such that the full calculation from input to output is given by:

$$\boldsymbol{a}_2 = \alpha_2(\underbrace{\mathbf{W}_2\alpha_1(\overbrace{\mathbf{W}_1\boldsymbol{x} + \boldsymbol{b}_1}^{\boldsymbol{z}_1}) + \boldsymbol{b}_2}_{\boldsymbol{z}_2})$$

This is simply the same equation as before, with $\boldsymbol{a}_1$ written in the form $\alpha_1(\mathbf{W}_1\boldsymbol{x} + \boldsymbol{b}_1)$, representing the weights and bias of the first layer applied to the input data, then activated by the activation function α_1.

Calculating outputs based on the original input in this way is the known as *feeding forward*, or *forward propagation*. This is the first of two major algorithms we must understand to be able to train and make predictions with a neural network. Feeding forward passes input data through all the layers of the network to obtain an output, using the weights, biases and activation functions as described above. The outputs of each layer are computed sequentially, starting from the input and ending at the output layer, feeding the input forward through each layer, hence the name. In short, forward propagation is the process by which the model returns an output.

The expressions above also make clear the significance of having non-linear activation functions: Any linear operations applied as activation functions as we move from layer to layer would allow us to rearrange the expression such that we form a single weight matrix with a bias added — effectively reducing the network to one layer. Only with non-linear activation functions can we take advantage of the sequential computation of the layers and the additional capabilities this grants the neural network in terms of computational power and pattern recognition.

2.2.1 Activation Functions

In this report, we are primarily considering three different activation functions: the sigmoid function (denoted by σ) [11], and the ReLU[2] and Leaky ReLU (ReLU_L) functions [12]. They are defined as follows:

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

$$\text{ReLU}(z) = \max(0, z)$$

$$\text{ReLU}_\text{L}(z) = \begin{cases} x & \text{if } x \geq 0 \\ 0.01x & \text{if } x < 0 \end{cases}$$

As we have touched upon, these activation functions play a role in the calculation of an output, which they can impact in a variety of ways to help produce predictions that correctly represent the datasets and the task in question. For instance, the sigmoid function maps input values to $(0, 1)$, so it can better represent probabilities than ReLU and Leaky ReLU can.

For numerical predictions on the other hand, it makes more sense to apply a function like ReLU to the output layer. ReLU has many advantages related to computational speed and convergence rate, since it will always have a gradient of either or 0 or 1, which can be calculated quickly [13] (we will soon revisit why this is significant). ReLU's tendency to set many activations to 0 leads to a sparsity in its outputs, which can also aid in avoiding *overfitting*, a concept we have previously discussed in detail [3].

[2]Rectified Linear Unit

Still, depending on the dataset, this quality may be a disadvantage. Unlike sigmoid, ReLU has completely unbounded outputs, which can lead to instability [13]. Additionally, datasets with many negative values will have many nodes evaluate to zero with the use of ReLU. This is sometimes referred to as having "dead neurons" [13], which will halt the training process by setting gradients to 0, hindering the computational ability of the network. Leaky ReLU can offer a compromise in cases like these: Instead of removing negative values entirely, it will scale them by a factor of 10^{-2}, allowing a small positive gradient to mitigate the vanishing gradient problem [12].

2.2.2 Training the Network

Up to this point, we have explored the inner structure of a neural network, and seen how an output can be produced through a series of both linear and non-linear operations. In practice, we typically initialize the weight matrices with random numbers (usually drawn from some probability distribution; we will use Xavier initialization [14]), and bias vectors with some small value (for instance 0.01) [15]. We will follow this approach, but we are missing a crucial element: Although we can now compute an output by feeding forward an input, we have no reason to think this prediction will be any good with the weights and biases set as described.

To quantify the "goodness" of the prediction, we return to the concept of a cost function, which, as we know, measures how well the neural network's output matches the target values [3]. We would like to minimize this value, and when we did linear regression, we did so by minimizing the cost for some choice of model parameters $\hat{\boldsymbol{\beta}}$. This time, we would like to minimize the cost as a function of the weights and biases $(\mathbf{W}, \boldsymbol{b})$ instead. The minimization process is what we call *training* the neural network.

Backpropagation is the second algorithm we need in order to do this. Specifically, we backpropagate to minimize the cost by adjusting the weights and biases of each layer iteratively. It works by calculating the gradient of the cost with respect to each weight using the chain rule. Recall the expression for the output of a 2-layer neural network introduced earlier:

$$\boldsymbol{a}_2 = \alpha_2(\underbrace{\mathbf{W}_2\alpha_1(\overbrace{\mathbf{W}_1\boldsymbol{x} + \boldsymbol{b}_1}^{\boldsymbol{z}_1}) + \boldsymbol{b}_2}_{\boldsymbol{z}_2})$$

The cost is expressed as:

$$C(\boldsymbol{\theta}) = \mathrm{MSE}(\boldsymbol{a}_2)$$

where $\boldsymbol{\theta}$ is a vector containing all weights and biases in the 2-layer network, and $\boldsymbol{a}_2$ is its output, as described above. In this case, we use the MSE as the cost, although, as we will see, it can be given by some other function.

We can find the gradient of C with respect to the weights of the second (outermost) layer $\mathbf{W}_2$ by the chain rule:

$$\frac{\partial C}{\partial \mathbf{W}_2} = \frac{\partial C}{\partial \boldsymbol{a}_2}\frac{\partial \boldsymbol{a}_2}{\partial \boldsymbol{z}_2}\frac{\partial \boldsymbol{z}_2}{\partial \mathbf{W}_2},$$

while the expression for the gradient of the weights of the first layer looks like this:

$$\frac{\partial C}{\partial \mathbf{W}_1} = \frac{\partial C}{\partial \boldsymbol{a}_2}\frac{\partial \boldsymbol{a}_2}{\partial \boldsymbol{z}_2}\frac{\partial \boldsymbol{z}_2}{\partial \boldsymbol{a}_1}\frac{\partial \boldsymbol{a}_1}{\partial \boldsymbol{z}_1}\frac{\partial \boldsymbol{z}_1}{\partial \mathbf{W}_1},$$

with analogous calculations of the bias gradients $\frac{\partial C}{\partial \boldsymbol{b}_2}$ and $\frac{\partial C}{\partial \boldsymbol{b}_1}$.

The important observation we make here is that many of the factors used to find these gradients are reused in the gradients of both layers. Backpropagation takes advantage of this to calculate the gradients of every single weight and bias efficiently in one backwards traversal of the network, starting with the gradient at the output node. Intuitively, we can view backpropagation as combining the effect of each operation on the intermediary outputs to see how changes to some specific weights or biases impact the final cost, which, again, is what we aim to minimize.

In short, backpropagation works by first calculating the cost at the output layer using the cost function C, and propagating the cost backwards through the network, layer by layer, progressively computing the gradients of all operations we do along the way. Once the gradients are calculated, we can use them to incrementally move the cost towards a minimum by updating the weights and biases appropriately, which will in turn update the final output that we started with. The training process involves iteratively performing forward propagation to find an output, computing the cost in the output layer, backpropagating this cost, and updating the weights and biases accordingly in the direction of a lower cost. We continue doing this until the network's performance on the testing dataset meets some condition, such as a gradient that is lower than some tolerance (in our case 10^{-8}), or simply some upper limit to the number of iterations we are willing to do (1000 for us).

2.2.3 Classification Tasks and Logistic Regression

One common usage of neural networks is for classification tasks [16]. By this, we mean problems in which data points need to be placed in discrete categories, such as the focus of this report, the Wisconsin Breast Cancer Dataset [1]. Our goal is to correctly distinguish between the patients in the dataset with and without breast cancer.

The dataset consists of labeled data from 569 patients, with 30 features to aid us in our binary classification task. The target distribution is fairly balanced, with 212 patients (about 37%) having cancer. This dataset is commonly used in training and classification tasks performed by simple neural networks like ours because it is well-structured, with only positive numerical features and no missing values. Thus, it requires no significant pre-processing on our part. As in our previous report, we will do a train-test-split of the data and scale it using the `StandardScaler` method from Scikit-Learn [17].

Since we are now using our neural network for another type of task, some changes should be made to its implementation to reflect this. One difference is the output of the network and how we might interpret it. Compared to what we have seen previously, a more natural output structure is to have output nodes corresponding to each of the categories, and let each node return the probability of some input being identified as that category. The category associated with the highest probability will then be the classification of the data point input, thus categorizing the dataset.

Additionally, sigmoid is the best-suited activation for the final layer, as we would like to interpret the output of each node as a probability. It will only return values between 0 and 1, supporting this interpretation, with a value close to 1 meaning the associated category is a likely choice for the input, and values close to 0 meaning it is not.

Another change is the cost function itself. In the classification case, it makes more sense to consider something like *cross-entropy* rather than the MSE. The expression for the cross-entropy is shown below:

$$C(\boldsymbol{\theta}) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m} y_{ij}\log(\hat{y}_{ij})$$

where $\boldsymbol{y}$ is the true label of the data points (one-hot encoded) and $\hat{\boldsymbol{y}}$ is the output predicted by the model. We interpret i as the index of the data points, while j indexes the different categories. Binary classification (as we are doing in this case, with patients who either do or do not have cancer) is a special case of the above function where $m = 2$, which is given by the expression below:

$$C(\boldsymbol{\theta}) = -\frac{1}{n}\sum_{i=1}^{n}[y_i\log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)]$$

There are a multitude of reasons why cross-entropy is normally preferred over the MSE for determining cost in classification tasks [18], but the main intuition we would like to highlight is this: For numerical predictions, the prediction error can be arbitrarily large, which a cost function like MSE will take into account and be able to penalize hard. In classification, however, the largest possible error we can have is 1, as all predictions are the probabilities of a data point belonging to a certain category. In other words, the distinction lies in the domain of the error values. The inclusion of the negative logarithm in the cost function enables us to punish confidently incorrect predictions much harder than MSE does, while still taking into account the closeness of a prediction [19]. These traits are favorable to us in effectively (and efficiently) training our models to perform classifications.

In light of this, we see an additional benefit to our dataset containing a high cancer rate: If we were to sample from a full population, where only a relatively small percentage have breast cancer, the network could be tempted to only predict the majority class, thus never identifying the patients with cancer. Essentially, with a cost function that harshly punishes confidently incorrect guesses and with low chances of any given person having cancer (regardless of other input data), the model will predict accordingly. Because we are interested in using neural network for disease prediction, we must differentiate between different types of errors the network makes, more so than in the strictly numerical case for polynomial prediction.

In this case specifically, we are interpreting our output classes as negative or positive results – cancer or no cancer[3]. This means that when our model misclassifies a data point, we may want to know not only whether or not the prediction was correct, but also if not, what is the nature of the error? Are we misdiagnosing a healthy person as having cancer, or are we allowing a sick person to go unnoticed? These are distinct problems that are rectifiable in different ways, and that constitute an important metric to consider before deploying models of this kind in the real world. If we need to make sure the latter does not occur (for instance when modeling data from the full population), we need to be able to isolate those cases, for instance to enable them to be penalized harshly when we train, so that the model avoids them.

[3]In this dataset, a value of 1 means the patient is cancer-free.

A *confusion matrix* is a handy tool for visually separating predictions into four categories: true negative (predicted cancer, and the patient did indeed have cancer), true positive (predicted no cancer, and no cancer was present), false negative (no cancer, but the model predicted cancer anyway) and false positive (cancer, but the model did not identify it). This is one way we will evaluate the performance of the models in the experiments carried out in this report.

Another test of the neural network's performance is comparing its classification to that of a *logistic regression* model, which is given by the expression below.

$$p(x) = \frac{1}{1 + e^{-x}}$$

Logistic regression[4] is an example of a *soft classifier*, meaning it will return a value between 0 and 1 that we interpret as the probability that a data point x belongs to one of the binary categories $y \in \{0, 1\}$ [20]. For us, outputs close to 0 represent a low likelihood of the patient having cancer, and outputs close to 1 represent a high likelihood.

3 Results

This section documents the results of various experiments we did. Since the final aim of the project is to predict breast cancer data accurately, we systematically tested all necessary components to perform this task, such that we ensure correctness (and reproducibility) of our final results. All plots show the performance of the model on testing data only.

3.1 Linear Regression with Gradient Descent Methods

In the first set of experiments, we are using a dataset in the form of a polynomial $f(x) = 5x^2 + 7x + 3$ over the interval $x \in [0, 1)$. We began with the most simple method, plain gradient descent, which we used

[4]Logistic regression is described by a function we have already seen, namely the sigmoid function. We will implement it by initializing a neural network with no hidden layers, activated by sigmoid.

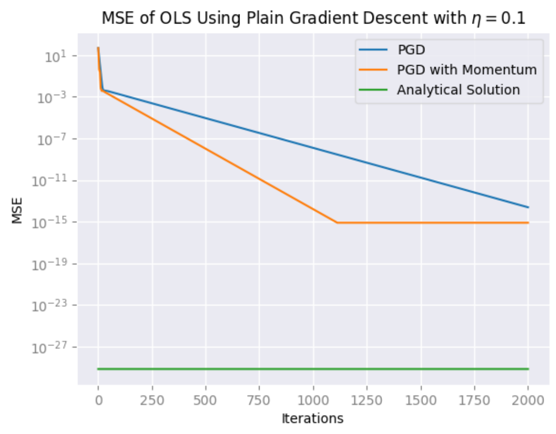

Figure 1: Plain gradient descent (PGD) with and without momentum compared to the analytical solution, with a constant learning rate of $\eta = 0.1$ and a momentum parameter $\gamma = 0.5$.

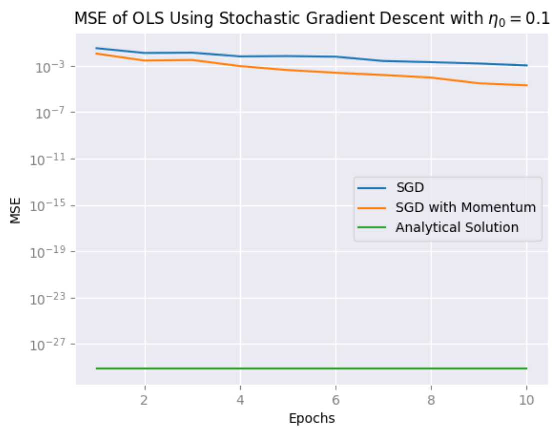

Figure 2: Stochastic gradient descent (SGD) with and without momentum compared to the analytical solution, with a linearly decaying learning rate beginning at $\eta_0 = 0.1$ and a momentum parameter $\gamma = 0.5$. The size of the mini-batches is also fixed at 4.

with a constant learning rate. We tested the values $\eta \in \{0.1, 0.01, 0.001, 0.0001\}$ to find the best possible value to fixate for our experiments[5], and based on this test, we saw that we had the best results when $\eta = 0.1$. With this learning rate, we studied convergence speed towards the minimum when using plain gradient descent with and with momentum. The plot of this (and the comparison to the analytical solution) is in figure 1, up to 10 epochs [21][22].

Next, we looked at our stochastic methods, with and without momentum. In this case we tried a linearly decaying learning rate to see if we would have any major improvements over plain gradient descent. In this case, the linearly decaying learning rate used was:

$$\eta_t = (1 - \frac{t}{5})0.1 + \frac{t}{5}0.001$$

where we found the value $\eta_0 = 0.1$ after a few tests, as well as consulting the literature. We also kept the mini-batch size fixed at 4, which we found produced decent values in this case. The comparison of the MSE with and without momentum as a function of epochs is displayed in figure 2. We also found our that when timed, stochastic methods gave us results quicker than the plain versions, with minimal detriment to the MSE. Therefore, from this point on, we will continue using our stochastic methods for computational efficiency.

Next, we looked at the methods with adaptive learning rates. All methods were initialized with the same global learning rate $\eta = 0.1$ to begin with, as this was the best result in both of our previous tests. We wanted to see how the different methods of adaptive learning rate converged, and if this behavior was consistent with the theory. The development of the MSE over epochs is shown in figure 3. We see faster convergence in methods with momentum.

As part of our testing process, we compared the performance of our gradients to that of the automatic differentiation library Autograd [23]. Since our tests pass[6], we expected our methods to give similar results as these pre-existing ones, which they did.

[5]The plots showing our testing can be found on our GitHub in the directory `figures/all_plots`.

[6]All code (including tests) can be found on our GitHub: `https://github.com/emmastorberg/FYS-STK4155_Project2/`

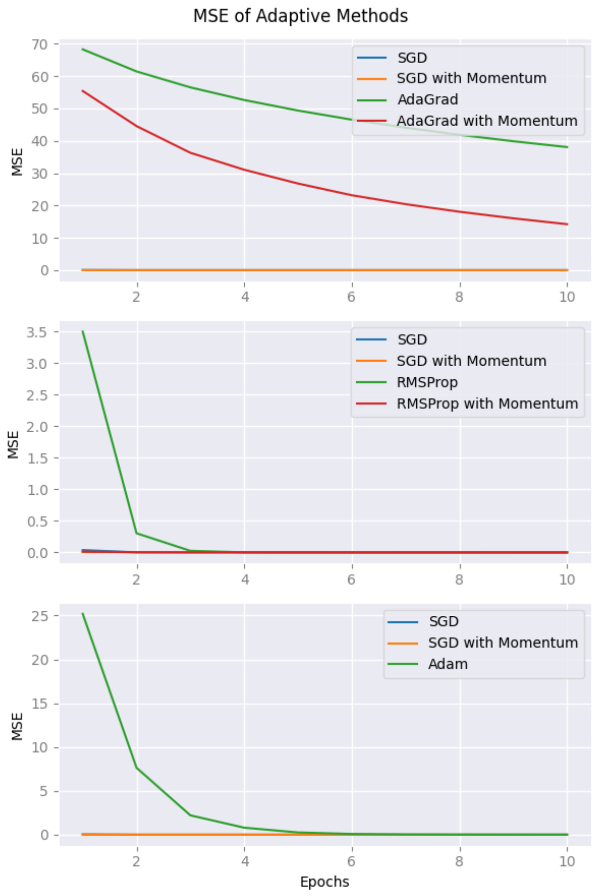

Figure 3: MSE as a function of epochs for the different emthods of adaptive learnign rate. For all methods, mini-batch size is fixed at 4, and momentum terms where applicable are scaled by $\gamma = 0.5$. The learning rate of SGD decays linearly, as in figure 2.

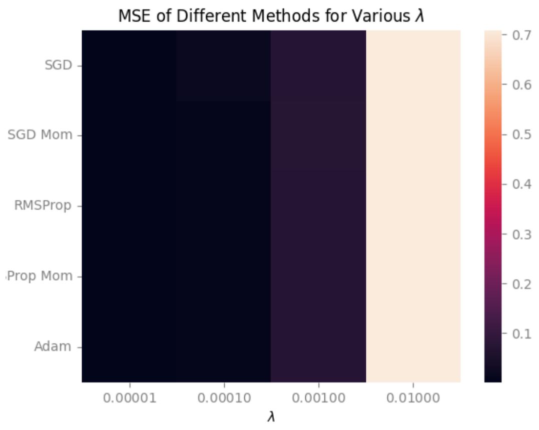

Figure 4: Grid search table of five possible methods compared for different values of the hyperparameter λ. In all cases, the global learning rate η is set to 0.1, the mini-batch size is fixed at 4, and momentum terms are scaled by $\gamma = 0.5$. "Mom" signifies a method with momentum added.

Up to this point, we have only used OLS. When we use ridge regression instead, there is the additional hyperparameter λ to consider. We performed a grid search to determine what combination of λ and gradient descent method to choose, and tested $\lambda \in \{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}\}$ against five of the seven methods we tested for OLS in figure 3 (AdaGrad is excluded because of overall poor performance, as we also see in the figure 3). The grid search table can be seen in figure 4, where it is clear that smaller values of λ do best, so we will use $\lambda = 0.00001$.

It is difficult to visually determine exactly which method is best, and upon reading off the values numerically, we also found minimal differences in the darkest regions. This is also consistent with the plots, where we see all methods except AdaGrad seem to converge nicely. Therefore, we determine that we will use Adam as we continue to use gradient descent in a neural network, as this seems in line with the literature on the subject.

3.2 Numerical Prediction with Neural Network

We saw that the optimal gradient descent methods for linear regression were RMSProp (with and without momentum) and Adam for OLS, and we chose Adam with $\lambda = 0.00001$ as the best-performing combination for ridge regression.

There is an enormous number of possible neural networks we can create even with only the relatively few gradient descent methods and activation functions at our disposal, in addition to the limitation of computation time. We eliminated a few degrees of freedom here by sticking to Adam as our gradient descent method (based on its performance in the experiments up to this point), and with ReLU as the activation function for the outer layer, guided by the theoretical intuition of what values it can produce (only positive values; the range of our polynomial $f(x) = 5x^2 + 7x + 3$ is positive real numbers only).

To construct the network, we tried to sequentially eliminate degrees of freedom by fixating certain parameters, and thereafter exploring slight modifications to these. We started by using the same number of nodes per hidden layer, and performing a grid search to find the best combination. The results of this are shown in figure 5, with ReLU activating all hidden layers as well as the output layer. As we can see, there are many options that have low MSEs (dark color). We will choose a network with four nodes and four hidden layers for simplicity, which had an MSE of 0.09854. Also of note is that the lightest sections with high MSE are from networks that predict zero everywhere, which is an unexpected finding.

Next, we tried varying what activation functions to use on the hidden layers. The plot of the resulting MSEs is shown in figure 6. It looks like ReLU is (slightly) better for activating the hidden layers in this case, so we continued working with the same neural network with four hidden layers, four nodes per layer and ReLU activating everything. This became the basis for further modifications in hopes of improving performance.

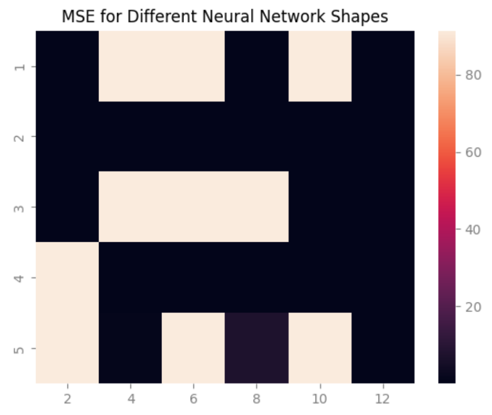

Figure 5: Grid search table showing the MSE of neural networks with varying number of hidden layers (y-axis), with a fixed number of nodes per layer (x-axis) and ReLU on all layers.

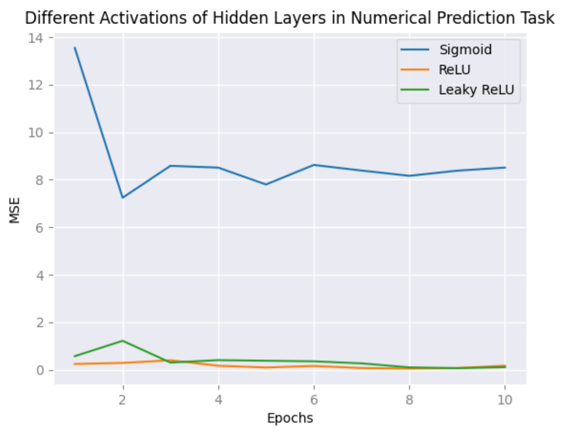

Figure 6: MSE of a neural network of four hidden layers and four nodes per layer activated by different functions in the hidden layers (ReLU was always used on the output layer).

We attempted some slight variations to the hidden layers, the nodes in them and the activation functions at each stage, without significant improvement, so we stuck to the network we determined previously. Its specifications and hyperparameters can be seen in table 1 below:

Hyperparameter	**Value**
Number of data points	$n = 100$
Scaling of data	`StandardScaler`
Mini-batch number (epochs)	$M = 4$
Mini-batch size	20 (test data)
Optimizer	Adam
Initial learning rate	$\eta = 0.1$
Decay rate of first moment	$\rho_1 = 0.9$
Decay rate of second moment	$\rho_2 = 0.99$
Numerical stability constant	$\delta = 10^{-8}$
Cost function	MSE
Input layer	1 node, ReLU
Hidden layer 1	4 nodes, ReLU
Hidden layer 2	4 nodes, ReLU
Hidden layer 3	4 nodes, ReLU
Hidden layer 4	4 nodes, ReLU
Output layer	1 node, ReLU

Table 1: The chosen hyperparameters and other relevant values for best performance in numerical prediction task.

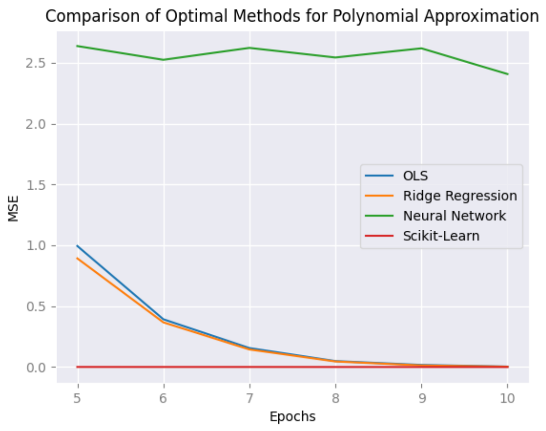

Figure 7: MSE as a function of epochs (shown from 5 to 10 epochs) for our best versions of OLS, Ridge and neural network, all with the optimizer Adam. They are also compared to Scikit-Learn's ridge regression by SGD with $\eta = 0.1$.

The neural network described above was able to predict with an MSE around 2.5 on the testing data, as we see in figure 7. This is overall a decent performance, but we also see that our linear regression methods (as well as ridge regression with Scikit-Learn) perform much better, with less tuning required. We also show the predictions made by each method in figure 8. As we can see, most of the linear regression methods excel and are barely distinguishable from the exact solution, while the neural networks sticks out from the rest.

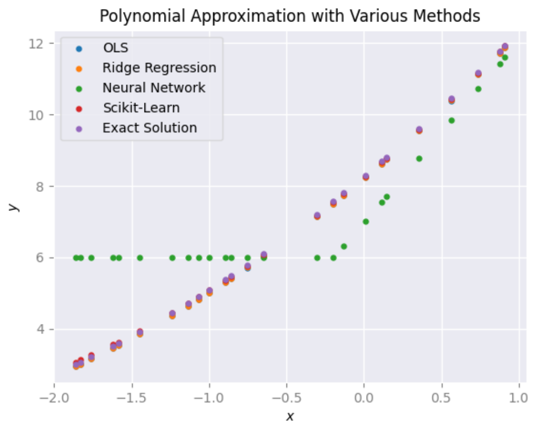

Figure 8: Comparison of predictions with optimal versions of all methods shown on scaled input data.

3.3 Classification with Neural Network

Moving on to the classification task, we once again faced the issue of an extreme number of possibilities to test in order to construct the optimal network. We limited our scope to only the optimizer Adam with $\rho_1 = 0.9, \rho_2 = 0.99$ and $\delta = 10^{-8}$ as before.

3.3.1 First Attempt with Our Own Code

Similar to the procedure in the creation of a network for numerical prediction, to make the optimal network for binary classification, we first employed a grid search of a fixed number of nodes per layer, and thereafter we altered the activation function on the hidden layers to see what gave the best results. In this process, we uncovered an error in the implementation. It looked as though the network predicted the average of the target vector, which gave an accuracy approximately equal to the frequency of target value 1 in the output array (since we are working with binary classification for this dataset).

The source of the error, as well as its solution, remains elusive. We studied the implementation and behavior of our network extensively. Unit testing also passes for all of our methods[7].

We initially scaled our data with `StandardScaler`, which we thought might be the issue, but changing the scaling to `MinMaxScaler` (also from Scikit-Learn [17]) offered only an incremental change (from an accuracy of approximately 0.62 to 0.63). All gradient descent methods had also passed rigorous unit testing, and we also knew from the experiments in the previous section on the numerical prediction task that our neural network code was capable of producing reasonable results from a theoretical standpoint.

Next, we tested our network on different datasets, including the iris dataset [24], which is a common and beginner-friendly dataset used for testing classification networks.

[7]See our GitHub: `https://github.com/emmastorberg/FYS-STK4155_Project2`

Our implementation[8] performs well on the iris data when scaled with `StandardScaler`, though not with `MinMaxScaler`. This showed that it was able to do some classification tasks, so we hoped the implementation was not the problem, but rather that the choice of parameters happened to lead us to a local minimum, which is why the accuracy stopped improving at an early iteration. However, an identical instantiation with PyTorch, an optimized tensor library for deep learning [26], creates a network that is able to classify the data points without issue, so the instantiation and choice of parameters is most likely not the problem.

These confusing findings left us with an unclear path towards analyzing our code implementation on breast cancer data, as was the goal of this project. We found that through systematic testing and trial and error, we could achieve accuracy as high as 1 on the iris data when testing the whole dataset[9]. However, seeing as this is a multiclass rather than binary classification problem, continuing to work with a network implemented with our own code on the iris dataset will leave us unable to compare our results to a logistic regression, nor will we be exploring the application of machine learning methods in the medical field, as was our initial motivation.

Therefore we ultimately made the decision to continue working with the breast cancer data using PyTorch to explore the optimal instantiation of a network for such a dataset. The resulting neural network will also be compared to a logistic regression.

3.3.2 Continued Exploration with PyTorch

Our search for the optimal set of hyperparameters for the task was in part systematic, and in part educated guesswork and trial and error. The first systematic test was a grid search, where we fixed the number of nodes in each layer. Throughout this task, we al-

[8]We activate the output layer with another activation function better for multiclass classification, known as the *softmax* function [25]. Its definition is given in appendix A.

[9]There is a risk of overfitting in this case, but it goes to show that the neural network is still making sensible predictions, which it does not do at all with the breast cancer data.

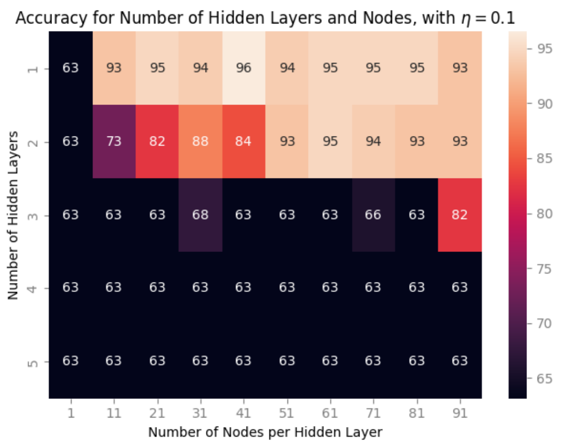

Figure 9: Grid search table showing the accuracy of neural networks with varying number of hidden layers, with a fixed number of nodes per layer and Leaky ReLU on all hidden layers, and sigmoid on the output layer.

ways used sigmoid as the activation function. A grid search table showing the best combination of number of layers and nodes per layer (with Leaky ReLU on all layers) is shown in figure 9. As we can see, the optimal result seems to be with fewer hidden layers, as overall one or two hidden layers performs the best.

Using this as a starting point for further exploration, we continued looking into if a different set of activation functions for the hidden layers would have any effect, but Leaky ReLU performed the best compared to the other activation functions, so we did not change the activation; the plot of this is in appendix B, figure 12.

As the networks we tested have such high accuracy scores (96% accuracy for a network with one hidden layer and 41 nodes in that layer, as we see in figure 9), we consider this good for the time being. Values of our chosen neural network are shown in table 2. However, though the network has high accuracy in its predictions, we must consider what type of errors it makes before determining this is a suitable network for diagnosing cancer. The confusion matrix is shown in figure 10 on the next page.

Hyperparameter	**Value**
Number of data points	$n = 569$
Scaling of data	`StandardScaler`
Learning rate	$\eta = 0.1$
Cost function	Cross-entropy
Input layer	30 nodes, Leaky ReLU
Hidden layer	41 nodes, Leaky ReLU
Output layer	1 node, sigmoid

Table 2: Table showing our chosen values for the optimal neural network in classification task. Otherwise, PyTorch defaults for SGD are used, meaning for instance a constant learning rate instead of adaptive methods [26].

3.4 Classification with Logistic Regression

As discussed, an alternative to a neural network requiring far less tuning is logistic regression. This corresponds to a network with no hidden layers, activated by the sigmoid function. The resulting confusion matrix is shown in figure 11 on the next page.

As we can see, the performance of logistic regression alone is worse compared to a neural network, with overall more errors, and more false positives than false negatives, unlike the neural network. Since a negative in this case means the patient has cancer, the higher rate of false positives is problematic, as it tells us the model is more likely to ignore a patient with cancer than give a cancer diagnosis to a patient without it. We also experienced how it requires much less time to set up and tune, which should be taken into account in holistically evaluating these methods.

4 Discussion

In general, our results aligned quite well with the theory, with a few hiccups in the classification part. In the first part of the project, we found that inclusion of a momentum term did lead to faster convergence, though it was difficult to see significant differences in smoothness. As expected, stochastic gradient descent was a quicker method to compute than plain, which makes sense.

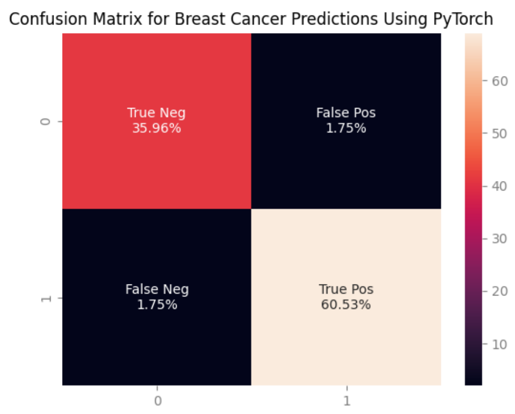

Figure 10: Confusion matrix for our choice of neural network instantiated with PyTorch. Note that the dataset is such that a negative value, i.e. 0, means the patient has cancer.

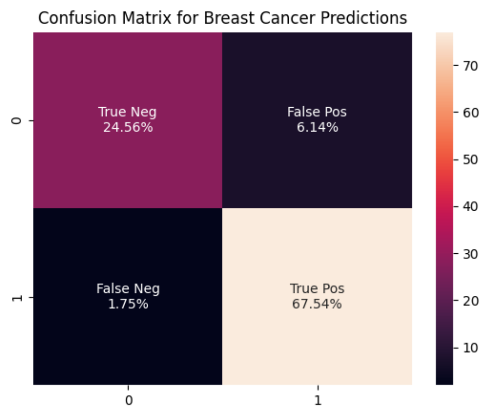

Figure 11: Confusion matrix from logistic regression

Our gradient descent methods were also less accurate than using the analytical solutions. This is a finding that is consistent with the theory, as they are numerical approximations of the gradient.

We also saw that dedicated linear regression methods worked better to predict values generated from a second-degree polynomial dataset than a neural network. This makes sense: Neural networks are universal approximators, and can be utilized for a range of tasks, but we expect inaccuracy compared to methods capable of given exact solutions (such as OLS in this case). The neural networks also required far more tuning, which was a time-consuming process that was almost circumvented entirely with use of linear regression methods (especially OLS).

As for the classification task, this is where we ran into strange issues. Our network worked for multiclass classification with the iris dataset, but struggled to predict cancer data at all, even with parameters that the analogous PyTorch methods performed well with. This suggests an error in our implementation that became very difficult to rectify in the more complex neural network code. Though we suspect this is the result of a minor error, we also believe it represents a drawback with use of neural networks on a large scale: When they produce strange results, it can be hard to decipher what is going wrong.

The problems in our code made us resort to PyTorch to complete our analysis and exploration of neural networks, in pursuit of a suitable model for the breast cancer data set. As we saw in figure 10, the model did not produce significantly more false negatives than false positives, which we view as a success in this case. Indeed, our goal is to minimize both types of errors; a false negative means unnecessary distress for the patient and their family, and wasted hospital resources on additional tests. A false positive, however, could be fatal if cancer goes untreated. Therefore, if the confusion matrix shows a high false positive rate, our loss function should be modified to penalize false positives more heavily.

Fortunately, this was not necessary for us to do with our neural network, but we would like to highlight it as one of our main considerations in this project. We did observe this phenomenon in the logistic regression however, which is another point in favor of the neural network we created for this specific case.

Another important aspect to evaluate is the explainability of the model used. When predicting breast cancer, we often want practitioners to use the model as decision support, not as the single source of truth. This is less straightforward with our neural network compared to for instance a logistic regression model, where we examined feature importance. Although prediction and understanding are related, accurate predictions can still come from flawed models, making it difficult for practitioners to interact with the model. These are considerations that must be taken before deploying neural networks in real-life contexts.

In a potential expansion of this project, it would be natural to tackle the errors we faced and see if we can create a neural network with our own code implementation, as was our original plan, and perhaps in the process, optimize our methods for faster runtimes. We are left with the impression that we have barely scratched the surface of the modifications that are possible for further tailoring a neural network to a specific task or dataset. It might therefore be rewarding to study the effect of varying other parameters than the ones we focused on in this report for specific tasks, or perhaps exploring the explainability aspect in sensitive use-cases, like in medicine.

5 Conclusion

The project has given us a solid theoretical and practical introduction to the underlying methods of neural networks. However, we also faced issues along the way with complicated implementations.

We aimed to create a neural network completely from scratch, including implementing the gradient descent methods that are used in the training process. Through experiments with each subsequent part that was implemented, we were able to produce a network that could predict numerical values based on data generated from a polynomial, initialized with four hidden layers and four nodes in each of these layers. It used Adam to do gradient descent in its backpropagation, and did reasonably well with an MSE of around 2.5 after 10 epochs, but not as well as basic linear regression techniques.

Following those initial tests, we constructed a network that could predict breast cancer data. From a theoretical and empirical standpoint, this should be a task a neural network is well-suited to do, and with the PyTorch implementation we saw solid performance with a neural network with one hidden layer and 41 nodes in that layer, activated by Leaky ReLU in its hidden layers and sigmoid on its output.

Thus, we are left with an important question: Are neural networks always worth all the tuning they require? Based on our findings, the answer is clearly *no*. Our first use-case, numerical prediction, showed that OLS (implemented with a variety of gradient descent methods) was better suited than our neural network to predict polynomial data. The classification task let us see both sides: While we experienced first-hand how numerical instability and the implementation of the complex algorithms involved in machine learning can lead to hidden errors, we also saw the power of neural networks when we implemented PyTorch and were able to predict breast cancer with 96% accuracy. The neural network outperformed its counterpart logistic regression overall and was less likely to overlook cancer in patient, at the expense of a degree of explainability and computational time.

In summary, we achieved our overarching goal of exploring the creation and usages of neural networks. Despite issues with our implementation, we saw how they can be used to predict medical data, and in this way function as a diagnostic tool. In the process, we gained understanding of the nuances in neural networks, such as the cases in which a neural network may not be the best choice. It is important to tailor the methods used to the specific situation, and weigh the advantages against the risks that are presented.

6 References

[1] *load_breast_cancer*. Scikit-Learn Developers. URL: https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_breast_cancer.html.

[2] Dan Zeltzer et al. "Diagnostic Accuracy of Artificial Intelligence in Virtual Primary Care". In: *Mayo Clinic Proceedings: Digital Health* 1.4 (2023), pp. 480–489. ISSN: 2949-7612. DOI: https://doi.org/10.1016/j.mcpdig.2023.08.002. URL: https://www.sciencedirect.com/science/article/pii/S2949761223000706.

[3] Emma Storberg & Frida Lien. "Linear Regression and Basic Resampling Techniques for Modeling 2D Datasets". Project 1 in FYS-STK4155, fall 2024 semester. URL: https://github.com/emmastorberg/FYS-STK4155_Project1/blob/main/FYS_STK4155_Project1_FinalReport.pdf.

[4] Morten Hjorth-Jensen. *Optimization, the central part of any Machine Learning algorithm.* URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapteroptimization.html#revisiting-our-linear-regression-solvers.

[5] Morten Hjorth-Jensen. *Optimization and Gradient Methods.* URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week39.html.

[6] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* Cambridge, MA, USA: MIT Press, 2016. Chap. 8. URL: http://www.deeplearningbook.org.

[7] StatQuest with John Starmer. *Stochastic Gradient Descent, Clearly Explained!!!* YouTube. May 2019. URL: https://www.youtube.com/watch?v=vMh0zPT0tLI.

[8] Sebastian Raschka. *Machine Learning FAQ.* URL: https://sebastianraschka.com/faq/docs/sgd-methods.html.

[9] Rauf Bhat. *Gradient Descent With Momentum.* Towards Data Science. URL: https://towardsdatascience.com/gradient-descent-with-momentum-59420f626c8f.

[10] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980 (2014). URL: https://api.semanticscholar.org/CorpusID:6628106.

[11] *Sigmoid function.* Wikipedia. URL: https://en.wikipedia.org/wiki/Sigmoid_function.

[12] *Rectifier (neural networks).* Wikipedia. URL: https://en.wikipedia.org/wiki/Rectifier_(neural_networks).

[13] Serkan Kızılırmak. *Rectified Linear Unit (ReLU) Function: Understanding the Basics.* URL: https://medium.com/@serkankizilirmak/rectified-linear-unit-relu-function-in-machine-learning-understanding-the-basics-3770bb31c2a8.

[14] *What Is Xavier Initialization?* 365 DataScience. URL: https://365datascience.com/tutorials/machine-learning-tutorials/what-is-xavier-initialization/.

[15] Morten Hjorth-Jensen. *Constructing a Neural Network code with examples.* URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week42.html.

[16] *Deep Learning Models For Classification : A Comprehensive Guide.* Metana. Aug. 2013. URL: https://metana.io/blog/deep-learning-models-for-classification-a-comprehensive-guide/.

[17] *Compare the effect of different scalers on data with outliers.* Scikit-Learn Developers. URL: https://scikit-learn.org/stable/auto_examples/preprocessing/plot_all_scaling.html.

[18] DataMListic. *Why We Don't Use the Mean Squared Error (MSE) Loss in Classification.* YouTube. June 2023. URL: https://www.youtube.com/watch?v=bNwI3IUOKyg.

[19] James McCaffrey. *Why You Should Use Cross-Entropy Error Instead Of Classification Error Or Mean Squared Error For Neural Network Classifier Training.* URL: https://jamesmccaffrey.wordpress.

com/2013/11/05/why-you-should-use-cross-entropy-error-instead-of-classification-error-or-mean-squared-error-for-neural-network-classifier-training/.

[20] Morten Hjorth-Jensen. *Logistic Regression.* URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapter4.html.

[21] Mugdha Mangesh Shinde. *Implementing Basic Neural Network and Predicting Breast Cancer.* URL: https://medium.com/@shindemugdha28/implementing-basic-neural-network-and-predicting-breast-cancer-30d25f73bb74.

[22] Muhammad Umar Nasir et al. “Breast Cancer Prediction Empowered with Fine-Tuning”. In: *Computational Intelligence and Neuroscience* 2022.1 (2022), p. 5918686. DOI: https://doi.org/10.1155/2022/5918686. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1155/2022/5918686. URL: https://onlinelibrary.wiley.com/doi/abs/10.1155/2022/5918686.

[23] Dougal Maclaurin et al. *Autograd.* URL: https://github.com/HIPS/autograd?tab=readme-ov-file.

[24] *load_iris.* Scikit-Learn. URL: https://scikit-learn.org/1.5/modules/generated/sklearn.datasets.load_iris.html.

[25] *Softmax function.* Wikipedia. URL: https://en.wikipedia.org/wiki/Softmax_function.

[26] *PyTorch documentation.* PyTorch. URL: https://pytorch.org/docs/stable/index.html.

A Softmax Function

The softmax function takes a vector $\boldsymbol{z} = (z_1, ..., z_K) \in \mathbb{R}^K$ and maps it to $(0,1)^K$, such that the ith entry of the resulting vector is given by:

$$\text{Softmax}(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}.$$

We interpret this as "[applying] the standard exponential function to each element z_i of the input vector $\boldsymbol{z}$ (consisting of K real numbers), and [normalizing] these values by dividing by the sum of all these exponentials" [25]. It works well for predictions representing the likelihood of more than two different options, since the normalization ensures the entries in the output layer all add up to 1 [25].

B Varying Activation Functions of Classification Neural Network

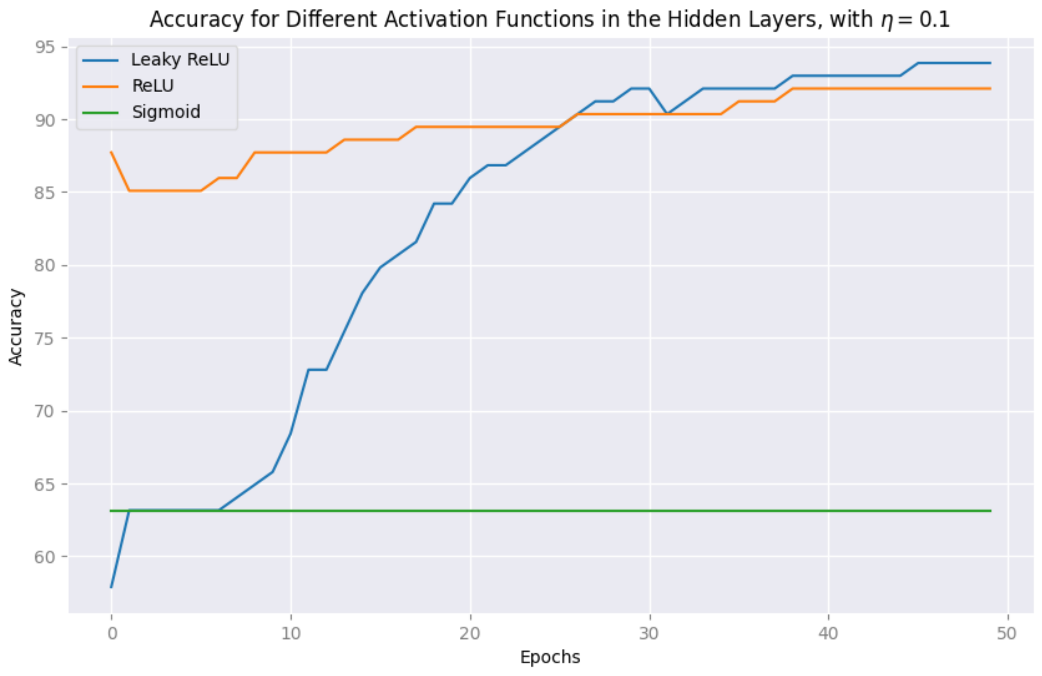

Figure 12: Accuracy of a neural network (instantiated with PyTorch) with one hidden layer of 41 nodes activated by different activation functions on the hidden layer.