Classification and Regression

Femtehjell, Hoel, Otterlei and Steeneveldt

October 2023

UiO : University of Oslo

Contents

List of Figures

Abstract

In this project, our focus was centered on classification and regression problems and how to solve them with neural networks. Specifically, we have looked at feed forward neural networks and how the optimization works. The optimization was done through the means of the gradient descent methods. To further optimize, we have tested different variations of the gradient descent method, such as momentum, AdaGrad, RMSProp and ADAM. All methods tested for both stochastic and deterministic versions. Further, we have applied these methods to a simple univarite polynomial, Franke's function and the Wisconsin Breast Cancer Data. Testing the regression methods on the univariate polynomial and Franke function and the classification methods on the Wisconsin Breast Cancer data set. Across the board, Adam was the top performer.

1 Introduction

The concept of self operating machines or 'Automatons' has captured human imagination throughout history, stretching back as far as the ancient Greeks. This fascination has endured throughout the centuries, influencing Leonardo DaVinci's vision of a mechanical knight to *the Mechanical Turk*, the infamous 18th-century fraudulent chess-playing machine.

Fast forward to the 1940's, Warren McCulloch and Walter Pitts came up with a mathematical model to describe the neuron cell in the brain. They called this model threshold logic (McCulloch & Pitts, 1943). This marked the genesis of what we now recognize as artificial neural networks (ANNs).

ANNs, inspired by the intricate workings of the human brain, are composed of interconnected neurons or nodes. These nodes, assembled into multiple layers, function to discern patterns and process data in a way that emulates the human neural activity.

A particular type of ANNs is the Feed-Forward Neural Networks (FFNNs). In FFNNs, the information moves in only one direction—from the input layer, through the hidden layers (if any), and to the output layer. The term *feed-forward* refers to this unidirectional flow of data, with no cycles or loops in the network. Essentially, FFNNs create a straightforward pathway for data.

The objective of this project is to investigate the application of FFNNs and other conventional methodologies in addressing regression and classification issues. Apart from FFNNs, we deploy linear regression techniques like OLS and Ridge to anticipate a bivariate polynomial. For tackling classification tasks, we employ logistic regression along with FFNNs.

Our study begins with a detailed exposition of the theoretical concepts underpinning our implementations, which is followed by a description of the implementations themselves. Subsequently, we present the results and engage in a discussion to interpret the findings.

2 Theory

2.1 The Cost function

In machine learning, the primary goal is to make accurate predictions based on known input data. The nature of these predictions can vary widely depending on the task at hand. For instance, a machine learning algorithm may predict whether an email is spam or not, an application of binary classification. In other cases, the algorithm might be used to anticipate the price of a house given certain features like its size, location and age, illustrating a regression task.

Alternatively, it could predict whether a given image depicts a cat, a dog, a horse, or some other animal, a scenario referred to as multi-class classification. Similarly, algorithms can also forecast sequences of data, such as the words that come next in a sentence, generally considered as sequence prediction problems. In all these scenarios, the machine learning model uses patterns recognized from training data to make these predictions. The measure of accuracy from these predictions subsequently determines the subsequent course of action, i.e., optimizing the model's performance, which is where the concept of a cost function comes in.

The cost function serves as a measure for a model's performance on a specific data set. It is what defines the multidimensional *cost surface* we wish to minimize with respect to our model parameters. The choice of cost function is thus very important. The optimal cost function is dependent on the specific task at hand – whether it be regression, binary classification, or multi-class classification.

During regression analysis, where the aim is to predict a single response variable based on one or more independent variables. We assume our observed response y can be modeled as

$$y = f(\boldsymbol{x}) + \epsilon$$

$\mathbf{x}$ are our dependent variables and ϵ is the stochastic noise inherent in y. We may assume $\epsilon \sim N(0, \sigma^2)$. We consider n samples of y and assume that there are p characteristics which define each of the samples, such that $y_i = f(\boldsymbol{x}_i) + \varepsilon_i$ where $\boldsymbol{x}_i = \big[x_{i,0}, x_{i,1}, \ldots, x_{i,p-1}\big]$.

We gather this information in a matrix $\mathbf{X}$, called the design matrix, such that

$$\mathbf{X} = \begin{bmatrix} x_{0,0} & x_{0,1} & \cdots & x_{0,p-1} \\ x_{1,0} & x_{1,1} & \cdots & x_{1,p-1} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,0} & x_{n,1} & \cdots & x_{n,p-1} \end{bmatrix}.$$

It is common practice to apply the *Mean Squared Error* (MSE), defined by

$$\frac{1}{n} \sum_{i=0}^{n-1} (y_i - \hat{y}_i)^2,$$

where $\hat{y}_i$ are our predictions, in order to measure the performance of our fitting. Additionally we often include regularization, in order to avoid overfitting and improve generalization.

2.1.1 OLS and Ridge

By assuming that there is a linear relationship between our samples y and the characteristics, we can apply *Ordinary Least Squares* (OLS) in order to find the parameters $\boldsymbol{\beta} = \begin{bmatrix}\beta_0, & \beta_1, & \ldots, & \beta_{p-1}\end{bmatrix}^T$ which minimizes the MSE. Our predictions are then calculated by $\hat{y}_i = \boldsymbol{x}_i \cdot \boldsymbol{\beta}$. In matrix notation this can be simplified to

$$\hat{\boldsymbol{y}} = \begin{bmatrix} x_{0,0} & x_{0,1} & \ldots & x_{0,p-1} \\ x_{1,0} & x_{1,1} & \ldots & x_{1,p-1} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,0} & x_{n,1} & \ldots & x_{n,p-1} \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_{p-1} \end{bmatrix} + \begin{bmatrix} \epsilon_0 \\ \epsilon_1 \\ \vdots \\ \epsilon_{p-1} \end{bmatrix} = \mathbf{X}\boldsymbol{\beta} + \epsilon,$$

where the $\mathbf{X}$ is called the design matrix and $\boldsymbol{\beta}$ is called weights. In this new notation, calculating the MSE can be written

$$\frac{1}{n}\left(\boldsymbol{y} - \hat{\boldsymbol{y}}\right)^T\left(\boldsymbol{y} - \hat{\boldsymbol{y}}\right)^T = \frac{1}{n}\left(\boldsymbol{y} - \mathbf{X}\boldsymbol{\beta}\right)^T\left(\boldsymbol{y} - \mathbf{X}\boldsymbol{\beta}\right)^T.$$

Minimizing this, we seek $\hat{\boldsymbol{\beta}}$ which through derivation can we found analytically as

$$\hat{\boldsymbol{\beta}} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\boldsymbol{y}$$

whenever $\mathbf{X}^T\mathbf{X}$ is invertible.

If we add $L2$ regularization to this, i.e. by determining our cost through

$$\frac{1}{n}\left(\boldsymbol{y} - \hat{\boldsymbol{y}}\right)^T\left(\boldsymbol{y} - \hat{\boldsymbol{y}}\right)^T + \lambda\|\boldsymbol{\beta}\|_2^2,$$

we punish our prediction for assigning undue weight to individual characteristics, in effect motivating it to assign weights where they have the greatest impact. Note that $\|\boldsymbol{\beta}\|_2$ is the ℓ_2 norm, defined by

$$\|\boldsymbol{x}\|_2 = \sqrt{\sum_{x_i \in \boldsymbol{x}} x_i^2},$$

hence the name. This is commonly referred to as Ridge regression, where the analytical solution $\hat{\boldsymbol{\beta}}$ can be found as

$$\hat{\boldsymbol{\beta}} = \left(\mathbf{X}^T\mathbf{X} + \lambda I_p\right)^{-1}\mathbf{X}^T\boldsymbol{y},$$

again only whenever $\mathbf{X}^T\mathbf{X} + \lambda I$ is invertible. λ is referred to as a regularization- or hyperparameter, and determines how strict we are with preventing high values in $\boldsymbol{\beta}$.

2.1.2 Binary Classification

When we use MSE as a loss function, we're primarily asking the model, "How far is your prediction from the actual value?" This question is appropriate within regression, where we're attempting to predict continuous values. In classification we are however more interested in getting discrete labels correct, i.e. whether a prediction falls into class 0 or class 1, rather than how far the predicted probability is from the actual value.

Let's consider a binary classification problem, where we have a target which belongs to class 1, which our model predicts into class 0. This is a complete misclassification, however, the squared error is only 1. We therefore need to consider a different measure of how far off our predictions are. We therefore apply *Binary Cross-entropy* (BCE), sometimes referred to as *log loss*.

To find the BCE loss we evaluate the distance between our predicted probability value p with the actual value $y \in \{0, 1\}$. The loss function is given by

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

If $y = 1$, we simply get $L(1, p) = -\log(p)$. Clearly $L \to 0$ as $p \to 1$ and $L \to \infty$ as $p \to 0$. If however $y = 0$, we get $L(0, p) = -log(1 - p)$. We then see the reverse behaviour, where $L \to 0$ as $p \to 0$ and $L \to \infty$ as $p \to 1$. This measure of the loss is thus much more appropriate in a binary classification problem.

The BCE cost function is then simply defined as the mean over n evaluations of the loss,

$$C(\boldsymbol{y}, \boldsymbol{p}) = \frac{1}{n} \sum_{i=0}^{n-1} L(y_i, p_i).$$

Minimizing this function is the purpose of a binary classifier.

2.1.3 Logistic Regression

We obtain logistic regression as we attempt to predict the probability of labeling in K classes through a function in x (Hastie et al., 2009, p. 119). As we are dealing with probabilities, we require that all the predicted probabilities sum to 1, and that they all lie in $[0, 1]$.

For a given observation $\boldsymbol{x}_i$, we thus seek a function which will give us the probability of getting a value of y_i. In the binary case, where $y_i \in \{0, 1\}$, we apply the *sigmoid* function, sometimes referred to as the logit function, defined by

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

Note that $1 - p(t) = p(-t)$, satisfying our requirements.

We assume that the probability of an observation with features $\boldsymbol{x}_i = \begin{bmatrix} x_{i,0}, & x_{i,1}, & \ldots, & x_{i,p-1} \end{bmatrix}^T$ landing in category $\boldsymbol{y}$, either 0 or 1, can be modeled by $f(\mathbf{x_i} \cdot \boldsymbol{\beta})$, where

$f : \mathbb{R} \to (0, 1)$. In a binary case, we thus get

$$p\left(y_i = 1|\boldsymbol{x}_i, \boldsymbol{\beta}\right) = \frac{e^{\boldsymbol{x}_i\boldsymbol{\beta}}}{1 + e^{\boldsymbol{x}_i\boldsymbol{\beta}}}$$
$$p\left(y_i = 0|\boldsymbol{x}_i, \boldsymbol{\beta}\right) = 1 - p\left(y_i = 1|\boldsymbol{x}_i, \boldsymbol{\beta}\right)$$

To fit this model we seek to find the parameters $\boldsymbol{\beta}$ which maximizes the likelihood function, which for a data set $\mathcal{D} = \{y_i, \boldsymbol{x}_i\}$ is defined as

$$L\left(\mathcal{D}|\boldsymbol{\beta}\right) = \prod_{i=0}^{n-1} p\left(y_i = 1|\boldsymbol{x}_i, \boldsymbol{\beta}\right)^{y_i} p\left(y_i = 0|\boldsymbol{x}_i, \boldsymbol{\beta}\right)^{1-y_i}.$$

Maximizing this equation is what we call the method of maximum likelihood. To simplify, we take the log of the equation and get the *log-likelihood* ℓ and also multiply with -1 such that it becomes a minimization problem instead of a maximization problem. We then get, writing $p(\boldsymbol{x}_i) = p\left(y_i = 1|\boldsymbol{x}_i, \boldsymbol{\beta}\right)$,

$$\begin{aligned}
\ell(\boldsymbol{\beta}) &= -\sum_{i=1}^{n} y_i \log p(\boldsymbol{x}_i) + (1 - y_i) \log(1 - p(\boldsymbol{x}_i)) \\
&= -\sum_{i=1}^{n} y_i \log p(\boldsymbol{x}_i) + \log(1 - p(\boldsymbol{x}_i)) - y_i \log(1 - p(\boldsymbol{x}_i)) \\
&= -\sum_{i=1}^{n} y_i \boldsymbol{x}_i\boldsymbol{\beta} + \log(1 - p(\boldsymbol{x}_i)) \\
&= -\sum_{i=1}^{n} y_i \boldsymbol{x_i}\boldsymbol{\beta} + \log\left(1 - \frac{e^{\boldsymbol{x_i}\boldsymbol{\beta}}}{1 + e^{\boldsymbol{x_i}\boldsymbol{\beta}}}\right) \\
&= -\sum_{i=1}^{n} y_i \boldsymbol{x_i}\boldsymbol{\beta} + \log\left(\frac{1}{1 + e^{\boldsymbol{x_i}\boldsymbol{\beta}}}\right) \\
&= -\sum_{i=1}^{n} y_i \boldsymbol{x_i}\boldsymbol{\beta} - \log\left(1 + e^{\boldsymbol{x_i}\boldsymbol{\beta}}\right)
\end{aligned}$$

Note that the first equation is just the binary cross entropy cost without $\frac{1}{n}$. The same cost function used for binary classification, even when there is not a linear relationship between the features and the observed categories. The minima with respect to $\boldsymbol{\beta}$ is found at $\frac{\partial \ell}{\partial \boldsymbol{\beta}} = 0$

$$\begin{aligned}
\frac{\partial \ell}{\partial \boldsymbol{\beta}} &= -\sum_{i=1}^{n} y_i \boldsymbol{x_i} - \frac{\boldsymbol{x_i} e^{\boldsymbol{x_i}\boldsymbol{\beta}}}{1 + e^{\boldsymbol{x_i}\boldsymbol{\beta}}} \\
&= -\sum_{i=1}^{n} \boldsymbol{x_i}\left(y_i - p(\boldsymbol{x_i})\right) = \mathbf{0}
\end{aligned}$$

This gives p non-linear equations. This can also be expressed using matrix notation

$$\frac{\partial \ell}{\partial \boldsymbol{\beta}} = \boldsymbol{X}(\boldsymbol{y} - \boldsymbol{p}) = \mathbf{0}$$

Here $\mathbf{0}$ denotes the $n \times n$ zero matrix, $\mathbf{X}$ is our design matrix, containing our n $\boldsymbol{x_i}$s as column vectors, $\boldsymbol{y}$ consists of our observed categories and $\boldsymbol{p}$ our predicted category($\boldsymbol{p_i} = p(\boldsymbol{x_i})$)

2.1.4 Accuracy score

The accuracy score, although not suited as a cost function for optimization, is a commonly used metric for evaluating classification models. It represents the proportion of correct predictions out of all predictions made, defined as

$$\text{Accuracy Score} = \frac{\text{\# of Correct Predictions}}{\text{Total of Predictions}}.$$

While it provides a straightforward and intuitive measure of overall model performance, the accuracy score can be misleading in scenarios where classes are imbalanced. In such cases, other metrics like precision or the F1 score may provide a more complete picture of a model's performance. These metrics fall outside the scope of this project, and are thus excluded. Nevertheless, the accuracy score serves as a great start point for model evaluation and comparison.

2.2 Optimization

As we've observed, the cost function is at the core of solving machine learning problems, which fundamentally involve optimization. That is we wish to find the minimum of a function. In a perfect scenario we would like to find this analytically, however for various reasons, whether they be computational or purely mathematical in nature, this is often not possible. Therefore we opt for iterative methods instead.

2.2.1 Gradient descent

Let $f : \mathbf{X}^n \to \mathbf{Y}$ be the function we wish to minimize, where $\mathbf{X}^n \subseteq \mathbb{R}^n, \mathbf{Y} \subseteq \mathbb{R}$. Let $\boldsymbol{w}^* \in \mathbf{X}^n$ be an extrema of a differentiable function f, it then satisfies $\nabla f(\boldsymbol{w}^*) = 0$. Let $\boldsymbol{\xi} \in \mathbf{X}^n$ be a local minima. We then approximate ξ by the iteration

$$\begin{aligned} \boldsymbol{w}_{k+1} =& \boldsymbol{w}_k - \eta_k \nabla f \boldsymbol{w}_k) \\ =& \boldsymbol{w}_k - \eta_k \boldsymbol{g}_k, \end{aligned} \tag{1}$$

denoting the gradient $\nabla f(\boldsymbol{w}_k)$ as $\boldsymbol{g}_k$.

The scalar in front of the gradient term, here η_k, is called the learning rate. If we let it be constant we get gradient descent. For each iteration we move our approximation of the minima in the opposite direction of the gradient; the direction of maximum growth.

This is also known as simultaneous relaxation and under certain conditions on the Hessian it can be shown that $\boldsymbol{w}_k$ converges to $\boldsymbol{\xi}$ given that $\boldsymbol{w}_0$ is close enough to $\boldsymbol{\xi}$ (Süli & Mayers, 2003, p. 117–118). For non-differentiable functions we may still obtain convergence by clever choice of η_k and numerical approximations of the gradient. Figure 2.1 shows a simple example of 30 iterations performed on $f(w) = w^2$ with $\eta = 0.01$.

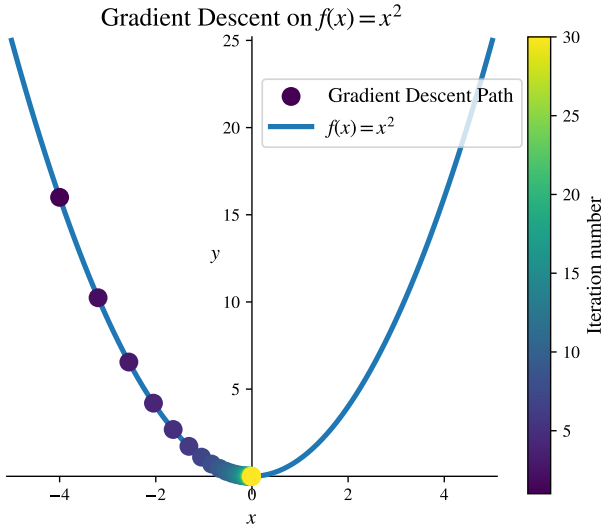


Figure 2.1: Gradient Descent on the function $f(x) = x^2$, starting from $x_0 = -4$ with $\eta = 0.01$, ran for 30 iterations.

For this simple example the iteration converged quickly, reaching an absolute value less than 0.001 after only 27 iterations. Gradient descent is not always this efficient.

The two natural weaknesses of gradient descent are neatly explained in Watt et al. (2020, p. 65–71). The first stems from the gradient always being perpendicular to the contour line. This is easily proved through parameterization.

Lemma 2.1. *Let $w(t)$ be a parametrization of a contour surface for a differentiable function $f : X^n \to Y$. Then $\nabla f \perp w(t)$.*

Proof.

$$f(w(t)) = C \Rightarrow \frac{d}{dt} f(w(t)) = 0 \Rightarrow \nabla f(w(t)) \cdot \frac{\partial w(t)}{\partial t} = 0$$

The result follows from $\frac{\partial w(t)}{\partial t}$ being parallel to the contour □

This may lead to *zig-zagging* behavior for ill conditioned problems, as seen in Figure 2.2.

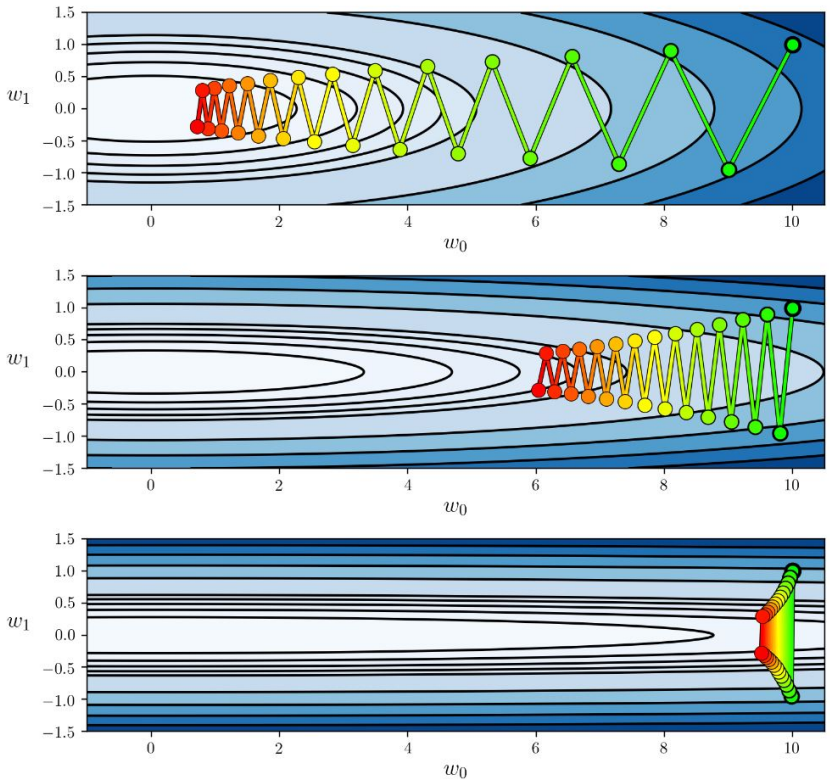


Figure 2.2: Figure 3.13 (Watt et al., 2020, p. 68) illustrating the zig-zagging behavior of gradient descent.

The other challenge of gradient descent is its slow crawling behaviour in flatter regions of a function such as saddle points. This is nicely illustrated by 50 iterations with $\eta = 0.1$ on the function

$$g(w) = \max\{0, 1 + (3w - 2.3)^3\}^2 + \max\{0, 1 + (-3w + 0.7)^3\}^2,$$

as seen in Figure 2.3.

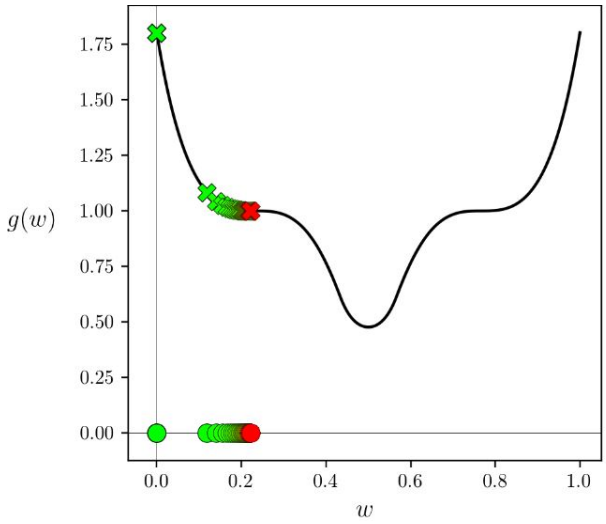


Figure 2.3: Figure 3.14 from (Watt et al., 2020, p. 70) illustrating the slow crawling behaviour of gradient descent.

2.2.2 Stochastic gradient descent

Recall that the cost function is defined as a measurement of the distance between our predictions and the observed values. A large dataset can therefore quickly turn computationally expensive, if we were to compute the cost function and corresponding gradient for everything all at once. To combat this, we often apply *Stochastic gradient descent* (SGD).

We first introduce batch gradient descent, where we split out dataset into *batches* of size M. For each iteration, we compute the gradient for each batch individually, updating our estimate for each. Instead of computing the cost and corresponding gradient for all data points each time as in gradient descent, we thus only calculate it for a smaller subset, having the effect of requiring less iterations.

In stochastic gradient descent, we apply the same method, differing by drawing a sample of M random points from our data set with replacement. The cost surface achieved by this is truly a different surface from the one achieved by using all the data points, but the number of features and thus the dimentionality, is the same. For this method to work we are banking on there being enough overlap between the samples such that the gradient is expected to point in the same direction when calculated for all data points.

It is common practice to organize our algorithm in a double loop. If we have n samples in total, the inner loop iterates over $\lfloor \frac{M}{n} \rfloor$. That is the number of batches that fit in our dataset. The outer loop iterates over the number of epochs.

Algorithm 1: Batch Gradient Descent

Data: Dataset D=$x_1, ..., x_n$, Learning rate η, Batch Size M, No. of Epochs E
Result: Model Parameters w
for *i in range(E)* **do**
    batches = create_batches(D, M);
    for *j in range(n//M)* **do**
        batch = batches[j];
        gradient = compute_gradient(batch, w);
        w = w - $\eta \cdot$ gradient;

Let $\boldsymbol{x}_i$ be the ith batch for $i = 1, \ldots, m$. We can rewrite (1) as

$$\boldsymbol{w}_{j+1} = \boldsymbol{w}_j - \eta_j \nabla f_i(\boldsymbol{x}_i)$$

where $j = 1, ..., m \cdot \#epochs$. Notice that the gradient is indexed as it is defined from the data points in the batch.

As stated in the previous subsection, gradient descent finds the local minima $f(\boldsymbol{\xi})$ by exploration. This is beneficial for convex functions, or if the local minima is already known to be the global minima. For non-convex functions, or in the case where the global minima is not known (which is the case most of the time), SGD introduces more exploration. This means that instead of getting stuck in a local minima, by randomness, there is a higher probability of finding the global minima. SGD also have an added bonus that it is dependent on the batch size rather than the data set size. Thus the learning rate can still converge, even if the data set is large.

It is important to keep in mind that for the following descent methods we can define their corresponding stochastic counterparts in the same way as we did for gradient descent. And that gradient descent and stochastic gradient descent is sometimes used to refer to the family of optimizers and their stochastic counterparts.

2.2.3 Momentum based gradient descent

Momentum is an added parameter meant to address these issues of vanilla gradient descent. The iteration is defined as

$$\begin{aligned} \boldsymbol{w}_{k+1} &= \boldsymbol{w}_k + \boldsymbol{v}_{t+1}, \\ \boldsymbol{v}_{t+1} &= \rho \boldsymbol{v}_t - \eta \boldsymbol{g}_k, \qquad \boldsymbol{v}_0 = 0 \end{aligned} \tag{2}$$

$\boldsymbol{v}$ is called the velocity term and ρ is the momentum parameter. ρ can be thought of as the velocity's resistance to change while the learning rate η characterizes the influence of the gradient. An intuitive analogy is to think of our descent algorithm as a particle moving on the surface we wish to minimize. The gradient is the force acting on our particle, η is its amplifier, and ρ is the mass of the

particle. This way our particle might be able to roll past the saddle point in Figure 2.3 and obtain a less "zig-zaggy" path than in Figure 2.4.

Here we see momentum gradient descent performed on the same ill conditioned problem as the first panel of Figure 2.2 with ρ equal to $0, 0.2, 0.7$ respectively.

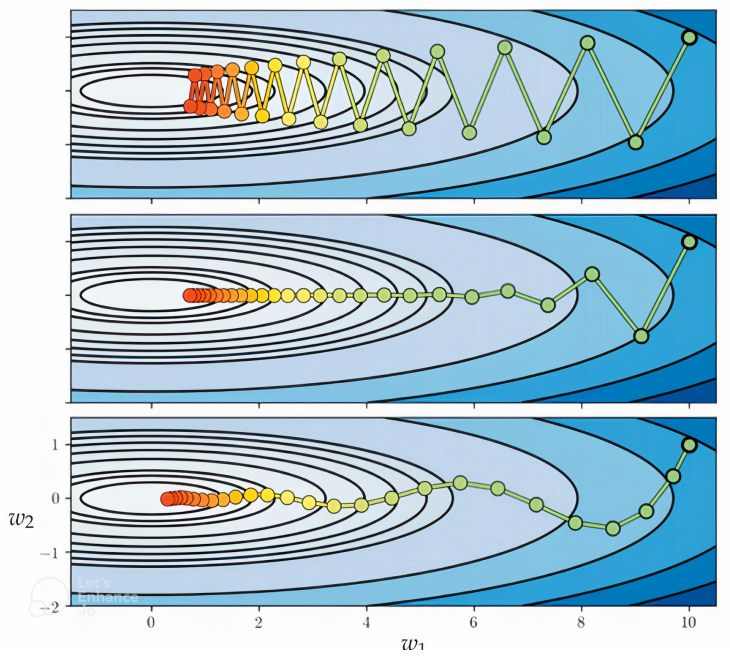


Figure 2.4: Figure A3 from Watt et al. (2020, p. 478). The zig-zagging behavior of gradient descent can be ameliorated using the momentum-accelerated gradient descent

2.2.4 AdaGrad

The Adaptive gradient optimizer, or AdaGrad was introduced by Duchi et al. (2011) and its update rule is given by

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \eta \operatorname{diag}(G_k)^{-1/2} \odot \boldsymbol{g}_k$$

where $\odot$ is the Hadamard product and G_k is the sum of the outerproducts of the previous gradients

$$G_k = \sum_{i=0}^{k} \boldsymbol{g}_i \boldsymbol{g}_i^T.$$

We thus have that $\operatorname{diag}(G_k)^{(n)}$ (the n^{th} component of $\operatorname{diag}(G_k)$) is the sum

of the squares of the previous gradients in direction n

$$\operatorname{diag}(G_k)^{(n)} = \sum_{i=0}^{k} \left(g_k^{(n)}\right)^2.$$

Normally a small numerical stabiliser is added to G_k to avoid division by zero, giving us

$$\boldsymbol{w}_{k+1}^{(n)} = \boldsymbol{w}_k^{(n)} - \frac{\eta}{\sqrt{\delta + \operatorname{diag}(G_k)^{(n)}}} \odot g_k^{(n)}.$$

As $\operatorname{diag}(G_k)^{(n)}$ increases, the step size in direction n decreases, which reduces the emphasis on that particular direction as it is traversed. This is the core concept of AdaGrad's design. AdaGrad is most effective when two conditions are met: first, when we need to move the same distance in each dimension, and second, when we have an appropriate choice of η. Opting for a too small η could result in the algorithm converging very slowly or coming to a halt due to the decaying step size. Conversely, selecting a too large η could lead to sluggish convergence. Thus, selecting an appropriate value of η is vital to ensuring efficient functioning of the algorithm.

2.2.5 RMSprop

RMSprop, root mean square propagation, was first introduced in 2012, in lecture 6 of the Coursera course "Neural Networks for Machine Learning" by Geoff Hinton (Hinton, 2012). It's update rule is defined as

$$\begin{aligned} \boldsymbol{v}_{k+1} &= \rho \boldsymbol{v}_k + (1-\rho)\boldsymbol{g}_k^{\circ 2} \\ w_{k+1}^{(n)} &= w_k^{(n)} - \frac{\eta}{\sqrt{\delta + v_{k+1}^{(n)}}} g_k^{(n)} \end{aligned}$$

where $^{\circ 2}$ is the Hadamard power defined by $\boldsymbol{x}^{\circ 2} = \boldsymbol{x} \odot \boldsymbol{x}$.

We have chosen to only give the element wise update rule, as the vector notation for the final equation is not very pretty for RMSprop. δ is a numerical stabilizer, $0 \leq \rho \leq 1$ and $\boldsymbol{v}_0$ is usually initialised to the zero vector.

Like AdaGrad, the update keeps some sort of memory of previous gradients. Where ρ can be thought of as the strength of the memory, i.e. how much of the previous iterations should impact the next. This allows for some of the same benefits as AdaGrads decaying learning rate.

2.2.6 ADAM

Adaptive moment estimation, most commonly known as ADAM, is an optimization algorithm which combines some of the aspects from RMSprop with

momentum. It's update is given by

$$\begin{aligned}
\boldsymbol{m}_{k+1} =& \frac{\rho_1 \boldsymbol{m}_k + (1-\rho_1)\boldsymbol{g}_k}{\left(1-\rho_1^{k+1}\right)} \\
\boldsymbol{v}_{k+1} =& \frac{\rho_2 \boldsymbol{v}_k + (1-\rho_2)\boldsymbol{g}_k^{\circ 2}}{\left(1-\rho_2^{k+1}\right)} \\
w_{k+1}^{(n)} =& w_k^{(n)} - \frac{\eta}{\sqrt{\delta + v_{k+1}^{(n)}}} m_{k+1}^n,
\end{aligned}$$

where $0 \leq \rho_1, \rho_2 < 1$.

Notice that the denominator of $\boldsymbol{m}_{k+1}$ and $\boldsymbol{v}_{k+1}$ starts somewhere in $(0, 1]$ and then increases to 1 as $k \to \infty$. This gives an amplification of $\boldsymbol{m}_k$ and $\boldsymbol{v}_k$, which subsides for larger k. This is meant to alleviate a bias towards zero as $\boldsymbol{m}_0, \boldsymbol{v}_0 = 0$ (Kingma & Ba, 2014, Section 3). . Optimization algorithms can be used to minimize any specific function. However, the latter three are all designed for minimizing a particular set of functions; the cost functions of Artificial Neural Networks (ANNs). These cost functions measure the error made by ANNs in predicting observed values from input data. How these predictions are made and adjusted is the topic of section 2.4

2.2.7 Time-based decay

Time-based decay removes the need for the user to fix the learning rate η themselves. Here, the rate is adjusted automatically based on the number of iterations computed, and two hyperparameters t_0 and t_1. It is defined simply by

$$\eta = \frac{t_0}{e \cdot M + m + t_1}, \tag{3}$$

where e is the number of epochs passed, M is the size of batches, and m is the number of minibatches computed in the current epoch. Thus the earlier iterations are given higher priority, as $\eta \to 0$ as $e \to \infty$, with $\eta = \frac{t_0}{t_1}$ for the first iteration.

2.3 Neural Networks

Neural networks are computational systems inspired by the networks of neurons within the brain. In the brain, when a neuron receives an electrical impulse above a certain threshold, it sends an electrical signal to the neurons connected to it, causing a chain reaction. In order to mimic this, an artificial neural network is set up with a number of layers, each with a set of artificial neurons. The first layer is commonly called the input layer, which is followed by a number of hidden layers, culminating in the final layer called the output layer.

The *feed-forward* neural network (FFNN), was one of the first, and perhaps simplest, artificial neural network. Each node is connected to nodes in the next

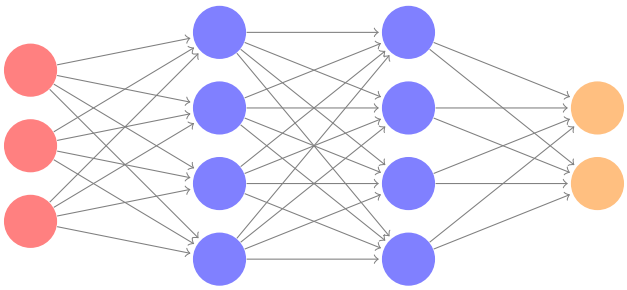


Figure 2.5: Illustration of a fully connected feed-forward neural network with an input layer (red), two hidden layers (blue) and an output layer (orange).

layer, with a weight $w \in \mathbb{R}$ attached to each edge, representing the strength of the connection between them. Additionally, we attach a bias $b \in \mathbb{R}$ to the node itself, also refereed to as the offset. Henceforth, we describe the *fully connected* feed-forward neural network, where the nodes are connected to *all* nodes in the next layer, illustrated in Figure 2.5.

Information in this network moves from left to right. The value of a node is dependent on the weights and values of the previous layer, as well as the bias. Let $N_l \in \mathbb{N}$ be the number of nodes in layer l with the input layer being $l = 0$. We then calculate

$$z_i^1 = \sum_{j=0}^{N_0-1} w_{ij}^1 x_j + b_i^1 \qquad \text{for } i = 0, 1, \ldots, N_1 - 1,$$

with w_{ij}^l being the weight connecting node i in layer l and node j in layer $l-1$, b_i^l the bias of node i and x_j the j^{th} input. We call z the activation value. Let $f^l : \mathbb{R} \to \mathbb{R}$ be a function, called the activation function for the l^{th} layer. We calculate the output of the node by $a_i^l = f^l(z_i^l)$. For the following layers, we calculate a *feed-forward pass*

$$a_i^l = f^l(z_i^l) = f^l\left(\sum_{j=0}^{N_l-1} w_{ij}^l a_j^{l-1} + b_i^l\right) \qquad \text{for } i = 0, 1, \ldots, N_l - 1,$$

illustrated in Figure 2.6.

We gather the weights connecting the l^{th} and $(l-1)^{th}$ layers in a $N_l \times N_{l-1}$ matrix $\mathbf{W}_l$, and the biases and outputs in $N_l \times 1$ column vectors $\mathbf{b}_l, \mathbf{y}_l$ as follows.

$$\mathbf{W}_l = \begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,N_{l-1}} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,N_{l-1}} \\ \vdots & \vdots & \ddots & \vdots \\ w_{N_l,0} & w_{N_l,1} & \cdots & w_{N_l,N_{l-1}} \end{bmatrix} \qquad \mathbf{b}_l = \begin{bmatrix} b_0^l \\ b_1^l \\ \vdots \\ b_{N_l}^l \end{bmatrix} \qquad \mathbf{y}_l = \begin{bmatrix} y_0^l \\ y_1^l \\ \vdots \\ y_{N_l}^l \end{bmatrix}$$

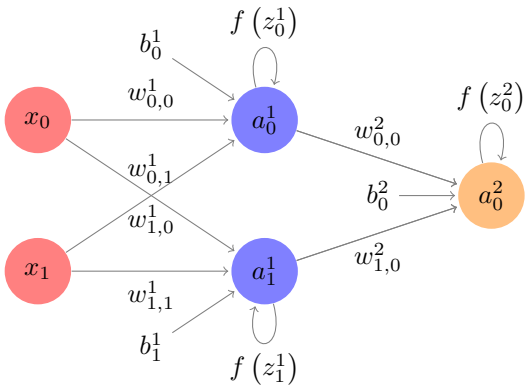


Figure 2.6: Illustration of a feed-forward pass in a neural network with two inputs (red), two hidden nodes (blue), and one output node (orange).

We can then simplify the summation to get $\mathbf{z}_l = \mathbf{W}_l \mathbf{y}_{l-1} + \mathbf{b}_l$. We then extend the function $f^l : \mathbb{R} \to \mathbb{R}$ to the function $F^l : \mathbb{R}^{N_l \times 1} \to \mathbb{R}^{N_l \times 1}$ by defining

$$F^l(\boldsymbol{x}) = \left[f^l\left(x_0\right),\ f^l\left(x_1\right),\ \ldots,\ f^l\left(x_{N_l-1}\right)\right]^T,$$

such that the function operates elementwise. For simplicity, this function F^l will also be notated f^l.

We define the realization of our neural network as the function $\mathcal{N} : \mathbb{R}^{N_0 \times 1} \to \mathbb{R}^{N_h \times 1}$, with h being the output layer as

$$\mathcal{N}(\boldsymbol{x}) = f^h\left(\mathbf{W}_h f^{h-1}\left(\mathbf{W}_{h-1} f^{h-2}\left(\ldots f^1\left(\mathbf{W}_1 \boldsymbol{x} + \mathbf{b}_1\right)\right) + \mathbf{b}_{h-1}\right) + \mathbf{b}_h\right).$$

Defining the function $\mathcal{A}^l : \mathbb{R}^{N_{l-1} \times 1} \to \mathbb{R}^{N_l}$ as the function

$$\mathcal{A}^l(\boldsymbol{x}) = \mathbf{W}_l \boldsymbol{x} + \mathbf{b}_l,$$

we can simplify our notation by noting that $\mathcal{N}$ is a composition of functions

$$\begin{aligned}\mathcal{N} &= f^h \circ \mathcal{A}^h \circ f^{h-1} \circ \mathcal{A}^{h-1} \circ \ldots \circ f^1 \circ \mathcal{A}^1 \\ &= \bigcirc_{i=1}^{h} f^i \circ \mathcal{A}^i.\end{aligned}$$

The Universal Approximation Theorem states that a neural network with even a single hidden layer, can approximate a continuous function on a compact subset arbitrarily well with an arbitrary number of hidden nodes given a fitting choice for the activation function (Cybenko, 1989). The proof of this theorem falls outside the scope of this project, but we highly recommend i.e. Guliyev and Ismailov (2015) or Cybenko (1989) if you have a passing interest in analysis. As we will see, our choice of activation function matters greatly.

2.3.1 Activation functions

For the sake of brevity, we restrict our self to one activation function for the hidden layers, and one for the output layer. To motivate our choice of activation function, we begin with the definition of a linear operator from real analysis (Lindstrøm, 2017, p. 150).

Definition 2.3.1. Assume that V and W are two vector spaces over $\mathbb{R}$. A function $A : V \to W$ is called a *linear operator* (or a *linear map*) if it satisfies:

(i) $A(\alpha \boldsymbol{u}) = \alpha A(\boldsymbol{u})$ for all $\alpha \in \mathbb{R}$ and $\boldsymbol{u} \in V$.

(ii) $A(\boldsymbol{u} + \boldsymbol{v}) = A(\boldsymbol{u}) + A(\boldsymbol{v})$ for all $\boldsymbol{u}, \boldsymbol{v} \in V$.

Theorem 2.2 (Composition of linear operators)**.** *Assume that U, V, W are vector spaces over $\mathbb{R}$ and that $A : U \to V$, $B : V \to W$ are linear operators. Then $B \circ A$ is a linear operator.*

Proof. Let $\boldsymbol{u}, \boldsymbol{v} \in V$, $\alpha, \beta \in \mathbb{R}$. Then

$$B(A(\alpha \boldsymbol{u} + \beta \boldsymbol{v})) = B(\alpha A(\boldsymbol{u}) + \beta A(\boldsymbol{v})) = \alpha B(A(\boldsymbol{u})) + \beta B(A(\boldsymbol{v})),$$

showing that $B \circ A$ is indeed a linear operator. □

Due to Theorem 2.2, if we were to choose a linear activation function for the hidden layer, $\mathcal{N}$ would just be a composition of linear operators, and therefore be linear itself. We would then not get any benefit from adding multiple hidden layers, as it could just be represented as one linear operation. This greatly restricts the hypothesis space, which is why we opt for non-linear activation functions (Chollet, 2018, p. 72).

A common choice, and the one first presented in Cybenko (1989), is a so called sigmoidal function σ. A sigmoidal function is a function $\sigma : \mathbb{R} \to \mathbb{R}$ which satisfies

$$\lim_{t \to +\infty} \sigma(t) = 1 \qquad \text{and} \qquad \lim_{t \to -\infty} \sigma(t) = 0.$$

A simple function which satisfies this, is the $(0, 1)$-clip function, defined by

$$f(x) = \begin{cases} 1 & \text{for } x \geq 1 \\ x & \text{for } x \in (0, 1) \\ 0 & \text{for } x \leq 0 \end{cases}.$$

The most common choice however is perhaps the standard logistic function (Jentzen et al., 2023), often just called the sigmoid function, defined by

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

Both of these functions are clearly sigmoidal, illustrated in the plot in Figure 2.7.

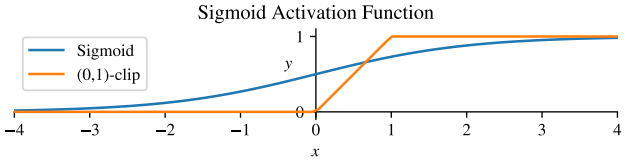


Figure 2.7: A plot of the sigmoid and $(0, 1)$-clip activation functions on $[-4, 4]$.

Another commonly used activation function is the rectified linear unit (ReLU), defined by

$$f(x) = \max(0, x) = \frac{x + |x|}{2} = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}.$$

However, this function suffers from what some call dying ReLUs, where nodes in effect die by outputting constant 0. To combat this, we introduce the leaky rectified linear unit (LReLU), defined by

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ \gamma x & \text{if } x < 0 \end{cases}.$$

for some positive $\gamma \in \mathbb{R}$. These functions are illustrated in Figure 2.8.

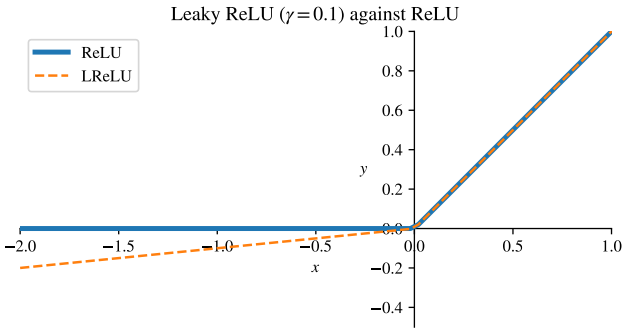


Figure 2.8: A plot of leaky rectified linear unit (LReLU) activation function on $[-2, 1]$, with $\gamma = 0.1$, against ReLU.

Two other important functions that are particularly used on the output layer for multi-class classification problems, are the arg max and $softmax$ functions.

The arg max function delivers the index or position associated with the highest value from a numeric input vector. If each node in the output layer denotes the models predicted probability of an input being classified in the corresponding class, it becomes clear that the arg max function essentially provides the predicted class or classification. This function is however not differentiable and therefore is not suited for training. Which will be clear by section 2.3.2.

The $softmax$ function is used to convert any vector to a valid discrete probability distribution. It is given by

$$softmax(\mathbf{z}) = \begin{bmatrix} \frac{e^{z_1}}{\sum_i e^{z_i}} \\ \vdots \\ \frac{e^{z_n}}{\sum_i e^{z_i}} \end{bmatrix} = \mathbf{a}$$

It keeps the sorted order of the input vector. That is if $z_i < z_j$ then $a_i < a_j$. It also satisfies

$$\sum_{i=1}^{n} a_i = 1$$

2.3.2 Minimizing the cost function: Backpropagation

Backpropagation serves as the heart of model optimization. As the name implies, it involves a backwards pass through our Artificial Neural Network (ANN). To understand its operation, we need to consider a model, referred to here as $\mathcal{N}$. After computing the output for a given input variable, $\boldsymbol{x}$, denoted by $\mathcal{N}(\boldsymbol{x})$, it becomes inherently natural to assess the performance of our model. Specifically, we want to compute the loss emerging from deviations between the model's output and the expected values

$$L(\mathcal{N}(\boldsymbol{x}), \mathbf{y})$$

The mean over n such losses gives us the cost function

$$C(\mathcal{N}, \boldsymbol{x}^*, \mathbf{y}^*) = \frac{1}{n} \sum_{i=0}^{n} L(\mathcal{N}(\boldsymbol{x}^{(i)}), \mathbf{y}^{(i)})$$

Here we let $\mathbf{x}^*$ and $\mathbf{y}^*$ denote the matrices containing the column vectors $\mathbf{x}^{(i)}$ and $\mathbf{y}^{(i)}$. These matrices are commonly referred to as our labeled data or our training data. This cost function is our optimization problem, the function we wish to minimize.

$$\arg\min_{\mathcal{N}} C(\mathcal{N}, \boldsymbol{x}^*, \mathbf{y}^*)$$

$\mathcal{N}$ consists of three components: our weights, biases and activation functions. Minimizing with respect to the activation functions is not necessary, as by the Universal Approximation Theorem, there are many that will suffice. By minimizing with respect to the weights and biases we get the back propagation algorithm.

Letting η and μ encompass all the different learning rates by our methods

in section 2.2 we get the general iteration on layer l

$$\hat{w_{i,j}^l} = w_{i,j}^l - \eta \frac{\partial C}{\partial w_{i,j}^l}$$
$$\hat{b_i^l} = b_i^l - \mu \frac{\partial C}{\partial b_i^l}$$

Here we are taking some notational liberty in the use of partial derivatives. As we'll see these are functions of the weights, the pre-activated and the activated values of the network, and are of course evaluated at the current values for a iteration.

We return to a_i^l begin the value of node i in layer l and z_i^l the value before activation. Then for some arbitrary layer l we get, by the chain rule

$$\frac{\partial C}{\partial w_{i,j}^l} = \frac{\partial C}{\partial a_i^l} \frac{\partial f^l}{\partial z_i^l} \frac{\partial z_i^l}{\partial w_{i,j}^l} \tag{4}$$
$$\frac{\partial C}{\partial b_i^l} = \frac{\partial C}{\partial a_i^l} \frac{\partial f^l}{\partial z_i^l} \frac{\partial z_i^l}{\partial b_i^l} \tag{5}$$

The third factors of these products simplifies to

$$\frac{\partial z_i^l}{\partial w_{i,j}^l} = \frac{\partial}{\partial w_{i,j}^l} \sum_{k=0}^{N_l - 1} w_{i,k}^L a_k^{l-1} + b_i^l = a_j^{l-1} \tag{6}$$
$$\frac{\partial z_i^l}{\partial b_i^l} = 1 \tag{7}$$

The second, $\frac{\partial f^l}{\partial z_i^l}$ can simplified to $f^{l'}(z_i^l)$ whenever the activation function f^l is a function of just one variable.

To see how $\frac{\partial C}{\partial a_i^l}$ expands. Consider Figure 2.9. A change in a_i^l will trickle down to the next layer. For a single edge we get

$$\frac{\partial C}{\partial a_i^{l+1}} \frac{\partial a_j^{l+1}}{\partial a_i^l} = \frac{\partial C}{\partial a_i^{l+1}} \frac{\partial f^{l+1}}{\partial z_j^{l+1}} w_{i,j}^{l+1}$$

Summation over all the edges gives us $\frac{\partial C}{\partial a_i^l}$

$$\frac{\partial C}{\partial a_i^l} = \sum_{i=0}^{N_{l+1}-1} \frac{\partial C}{\partial a_i^{l+1}} \frac{\partial f^{l+1}}{\partial z_j^{l+1}} w_{i,j}^{l+1} \tag{8}$$

If l is the final layer in the network $\frac{\partial C}{\partial a_i^l}$ is the partial derivative with respect to the multivariate function C - no more chain rule with respect to other nodes.

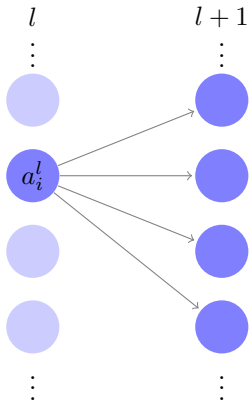


Figure 2.9: Illustration of Connections from one Neuron in a Neural Network

Combining equation (4) with (6) and (5) with (7). We get

$$\frac{\partial C}{\partial w_{i,j}^l} = \frac{\partial C}{\partial a_i^l} \frac{\partial f^l}{\partial z_i^l} a_j^{l-1} \tag{9}$$

$$\frac{\partial C}{\partial b_i^l} = \frac{\partial C}{\partial a_i^l} \frac{\partial f^l}{\partial z_i^l} \tag{10}$$

Equations (8) through (10) form the backbone of the backpropagation algorithm. As we observe, equation (8), which is integral to both (9) and (10), relies on the partial derivatives of the proceeding layer. After computing the forward pass and storing all the relevant network values, we can initiate the backpropagation process, moving from right to left, one layer at the time. This systematic approach allows us to calculate all the gradients necessary for our optimization algorithms.

2.3.3 Automatic Differentiation

Automatic Differentiation (AD), also known as algorithmic differentiation or computational differentiation, refers to a set of techniques to numerically evaluate partial derivatives. It differs from symbolic differentiation and numerical differentiation.

At a high level, automatic differentiation leverages the principle that any computational calculation, regardless of its complexity, carries out a series of basic arithmetic tasks (like addition, subtraction, multiplication, and division) and fundamental functions (such as exp, log, sin, cos, etc.). Every one of these functions are stored as separate variables. For instance our computation might encounter the expression $sin(a + b)$, where a, b are two of the input variables. This could be stored as h_1. Furthermore $a + b$ could be stored as h_2. This gives

$$\frac{\partial h1}{\partial a} = \frac{\partial h1}{\partial h2}\frac{\partial h2}{\partial a}$$
$$\frac{\partial h1}{\partial b} = \frac{\partial h1}{\partial h2}\frac{\partial h2}{\partial b}$$

Just as we did for backpropagation!

This way we can find all the partial derivatives with respect to any of the input variables by repeatedly applying the chain rule (Walther, 2007) . This method is not only very precise but also efficient. It does however, require added storage, in the worst case growing proportionally with number of operations in the function. (Baydin et al., 2018)

The use of automatic differentiation techniques is most relevant for the calculation of $\frac{\partial C}{\partial a_i^L}$ for the final layer. As a single pass of reverse accumulation AD (the process briefly explained above) allows us to calculate the partial derivatives with respect to the entire layer a^L

2.3.4 Derivative of Binary Cross Entropy with respect to weights

Recall the Loss and BCE cost function given by

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

$$C(\mathbf{y}, \mathbf{p}) = \frac{1}{n}\sum_{i=1}^{n} L(y_i, p_i)$$

Let's look at the loss function first.

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

For binary classification it's common to use the sigmoid as our activation function for the final layer to obtain a valid probability. Giving $\sigma(z) = p$ as the derivative is given by

$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$

We get

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

Thus for the cost function with respect to the final layer of weights we get

$$\begin{aligned}\frac{C}{\partial \mathbf{w^L}} &= \frac{\partial C}{\partial \mathbf{p}} \frac{\partial \mathbf{p}}{\partial \mathbf{w^L}} \odot \mathbf{w^L} \\ &= \frac{1}{n} (\mathbf{p} - \mathbf{y}) \odot \mathbf{w^L}\end{aligned}$$

This analytical expression is very convenient for backpropagation.

3 Method

In order to verify that our methods work, we began with testing our gradient descent code with the polynomial

$$f(x) = 4 - 3x + 2x^2. \tag{11}$$

With this, we can more easily visualize and understand the difference between the various methods. We generated our data with 100 values for $\boldsymbol{x}_i \in N(0,1)$, and added noise $\boldsymbol{\varepsilon}_i \in N(0,1)$ to our values such that $\boldsymbol{y} = f(\boldsymbol{x}) + \boldsymbol{\varepsilon}$ seen in Figure 3.1. Utilizing polynomial regression, our design matrix was then

$$X = \begin{bmatrix} 1 & x_0 & x_0^2 \\ 1 & x_1 & x_1^2 \\ & \vdots & \\ 1 & x_{100} & x_{100}^2 \end{bmatrix}, \tag{12}$$

seeking to find $\boldsymbol{\theta} = [\theta_0, \theta_1, \theta_2]^T$ minimizing our cost function. We utilized both OLS and Ridge regression, applying both the analytical derivative and automatic differentiation.

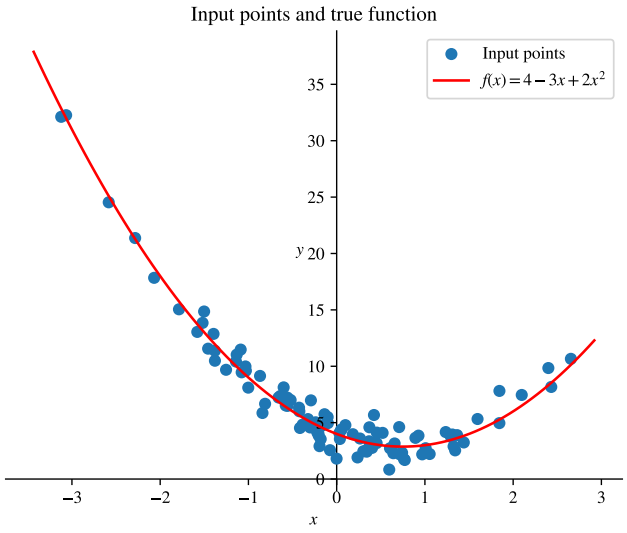


Figure 3.1: Values used for verifying gradient descent on $f(x) = 4 - 3x + 2x^2$.

With a better understanding of how gradient descent with the various methods works, we moved on to regression with a neural network. Here, we utilized

Franke's function, defined by

$$f(x,y) = \frac{3}{4}\exp\left(-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}\right) + \frac{3}{4}\exp\left(-\frac{(9x+1)^2}{49} - \frac{(9y+1)}{10}\right) + \frac{1}{2}\exp\left(-\frac{(9x-7)^2}{4} - \frac{(9y-3)^2}{4}\right) - \frac{1}{5}\exp\left(-(9x-4)^2 - (9y-7)^2\right).$$

for $x, y \in [0,1]$. A major advantage with regression through the neural network is that we do not have to transform our inputs to a polynomial as in (12), instead allowing the network to learn the behaviour through the hidden layers. This has the advantage of not coaxing the network into fitting a polynomial. We therefore have only two input nodes for x and y, with one output node outputting the predicted height z. We used 100 evenly spaced points in $[0,1]$ for our values of x and y, transforming our inputs to a mesh grid giving 100×100 points.

For classification, we used the original Wisconsin Breast Cancer Dataset (Wolberg, 1992), attempting to predict from measurements of tumours whether they are malignant or benign. The dataset consists of 9 features, being labeled either 2 for benign or 4 for malignant. We transform the labels with $f(x) = x/2 - 1$ such that the labels are binary. There are 699 data points, however 16 of them are missing some data and are thus excluded from our analysis.

There are several different ways to initialize the weights and biases of ANNs; nonetheless we chose random initialization. Our weights are sampled from a normal distribution with mean zero and standard deviation 1. Conversely our weights are drawn from a narrower distribution, with the same mean, but a standard deviation of 0.01.

3.1 The code

Our original implementation primarily used `NumPy` for the linear algebra and `autograd` for automatic differentiation. Due to computational concerns, it was then rewritten to primarily rely on `JAX` (Bradbury et al., 2018), which is a Google developed library combining `XLA` and `autograd` designed for high-performance computing. This has the advantage of being a lot faster, however leads to more verbose code. As an example, consider the sigmoid function implemented in `JAX` and `NumPy`.

```
from jax import lax, jit
import numpy as np

@jit
def sigmoid_jax(X):
    return lax.reciprocal(lax.add(1.0, lax.exp(-X)))

def sigmoid_numpy(X): # NumPy equivalent
    return 1 / (1 + np.exp(-X))
```

Note that we apply the `@jit` decorator to enable just-in-time compilation, which is a leading factor in the speedup. The sigmoid function is also a built-in function in `JAX` under the name `lax.logistic`, however we chose to use our own implementation to keep everything in the same style.

As `jit` works poorly with classes, requiring methods to be completely functional in order to assure correct functionality, we chose to extract the most costly computations into their own functions, using pure python primarily for higher level bookkeeping. One of the trickier parts of porting to `JAX` was that it has no built in method for elementwise differentiation. Thus to differentiate the hidden functions, which are performed elementwise, we used the following code.

```
from jax import grad, vmap
from autograd import elementwise_grad

def derivate_jax(func):
    return vmap(vmap(grad(func)))

def derivate_autograd(func): # autograd equivalent
    return elementwise_grad(func)
```

As we are working with column vectors in $\mathbb{R}^{n\times 1}$, we have to apply two vectorizing maps. If `f`: $\mathbb{R}^{n\times 1} \to \mathbb{R}^{n\times 1}$, then `vmap(f)`: $\mathbb{R}^1 \to \mathbb{R}^1$. Lastly, note that `JAX` treats vectors of dimension `(1,)` differently from scalars, so we have to apply a final vectorizing map to achieve our desired result of `vmap(vmap(f))`:$\mathbb{R} \to \mathbb{R}$ as `grad` only works on scalar-output functions.

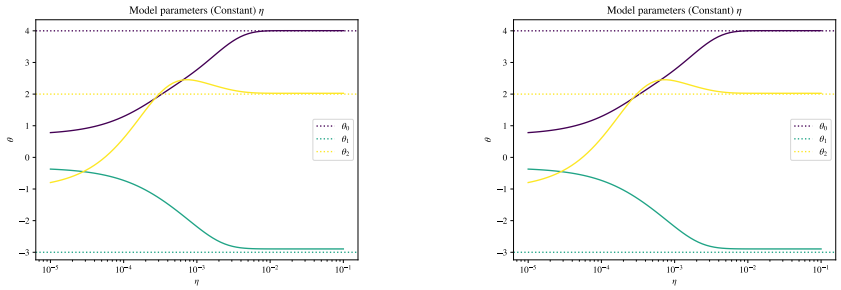
To be able to structure our code better, we developed a simple package which can be installed with `pip3 install .` in the Project2 directory. Note that there seems to be a bug in `JAX` where anything smaller than 10^{-7} is rounded to 0 when using `jit`.

4 Results

We generated a large number of plots, however we found it excessive to add everything here. The remaining figures are available on GitHub.

4.1 Analysis of univariate polynomial

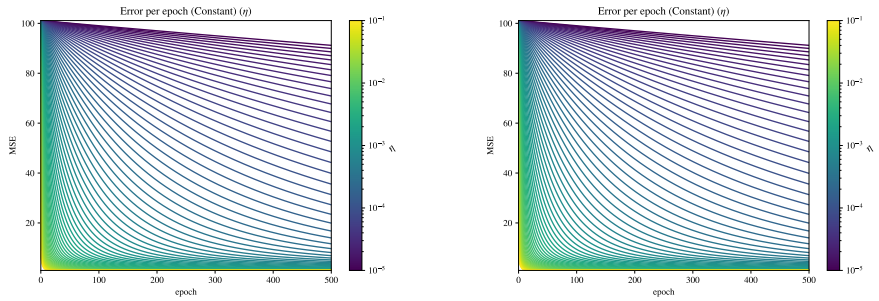
We began with the very simple case of attempting to fit (11) using Gradient Descent with a constant learning rate, with OLS as our cost function.



(a) Analytical derivative of OLS.

(b) Automatic differentiation of OLS.

Figure 4.1: Model parameters θ for varying degrees of a constant learning rate η, using ordinary least squares with Gradient Descent. Run over 500 epochs.



(a) Analytical derivative of OLS.

(b) Automatic differentiation of OLS.

Figure 4.2: MSE of Gradient Descent with OLS over epochs for varying values of η.

As we see in Figure 4.1 and Figure 4.2, we get identical results using automatic differentiation and the analytical expression. We found the same results across the board, and the results using the analytic expressions will therefore be excluded from this section for the sake of brevity. In Figure 4.3 we see how

Gradient Descent fits the target polynomial, predicting it almost perfectly for higher learning rates.

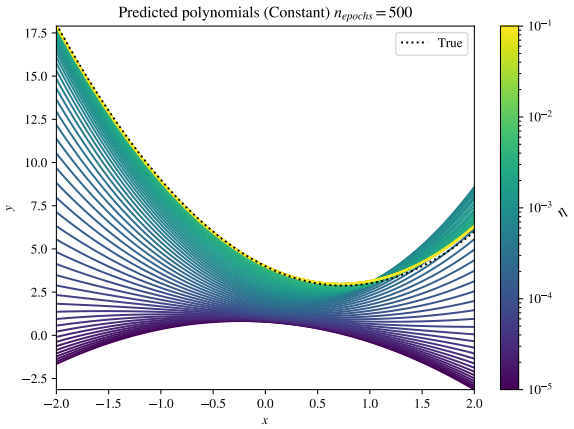
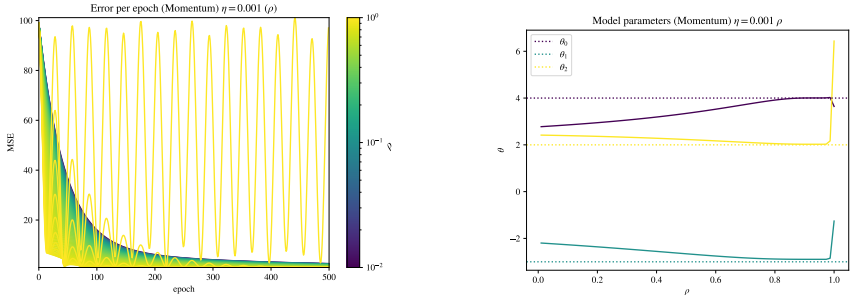


Figure 4.3: Predicted polynomials for the resulting parameter values $\boldsymbol{\theta}$ from Gradient descent with constant learning rate after 500 epochs.

Introducing momentum, as in (2) to our model can greatly improve results. We begin by seeing how the behaviour in Figure 2.4 is seen in our case.

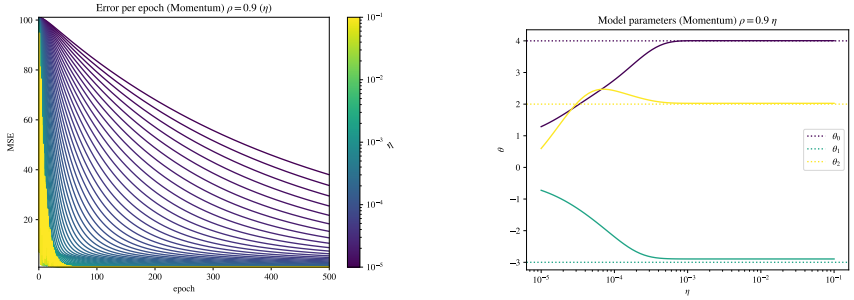


(a) Error for varying ρ.

(b) Parameters θ for varying ρ.

Figure 4.4: Gradient descent with varying momentum values ρ over 500 epochs with a learning rate of $\eta = 10^{-3}$.

As we see in Figure 4.4, we get greatly improved convergence results with higher values of ρ, although we have to be careful as we approach $\rho = 1$ as it can become quite unstable. Due to this, we settle for a value of $\rho = 0.9$ for the rest of our exploration.



(a) Error for varying η.

(b) Parameters θ for varying η.

Figure 4.5: Gradient descent with momentum $\rho = 0.9$ over 500 epochs with varying η.

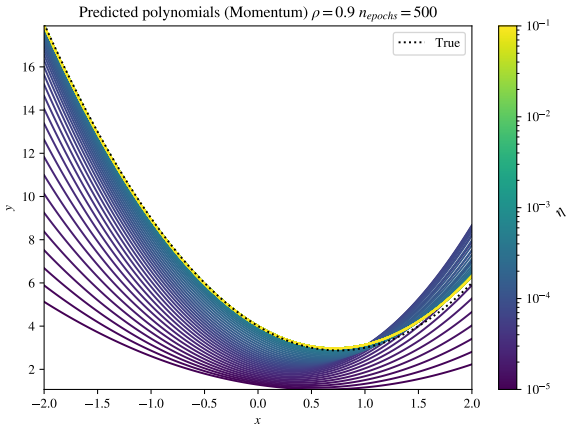


Figure 4.6: Predicted polynomials for the resulting parameter values $\boldsymbol{\theta}$ from Gradient Descent with momentum after 500 epochs.

In Figure 4.5 and Figure 4.6 we now see convergence for much smaller values of η. In Figure 4.7 we see what a great effect our choice of ρ has, both converging faster and becoming more unstable for higher values. Note the extremely skewed range of ρ, generated with arctan to highlight the details for higher values.

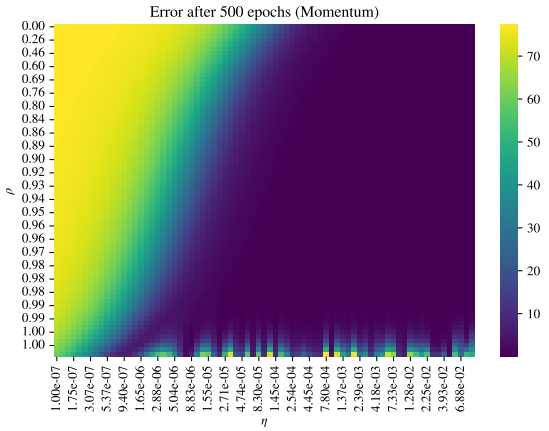
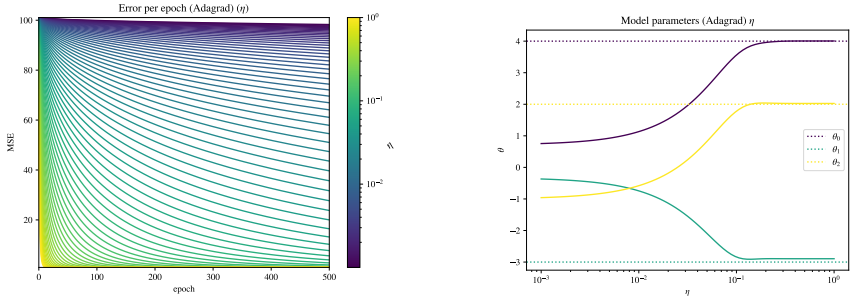


Figure 4.7: Heatmap of error after 500 epochs for varying momentum values ρ and learning rates η.



(a) Error for varying η.

(b) Parameters θ for varying η.

Figure 4.8: Gradient descent with Adagrad over 500 epochs with varying η.

In Figure 4.8 we see that Adagrad is considerably less sensitive to our choice of η, and it is interesting to note that we see the expected behaviour of Adagrad adjusting each parameter, removing the peak for θ_2 seen previously, and being much more uniform. This is seen in Figure 4.9, and note that it does not *overshoot* the target.

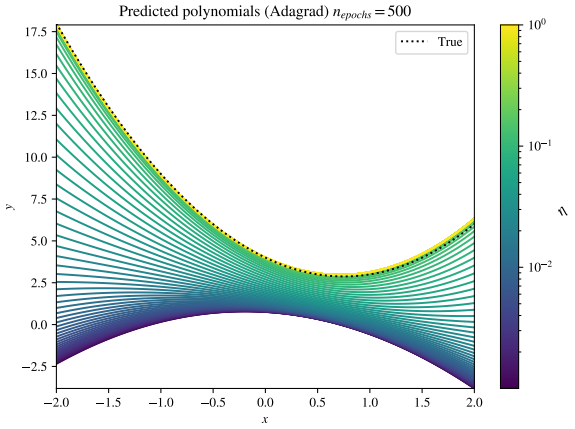
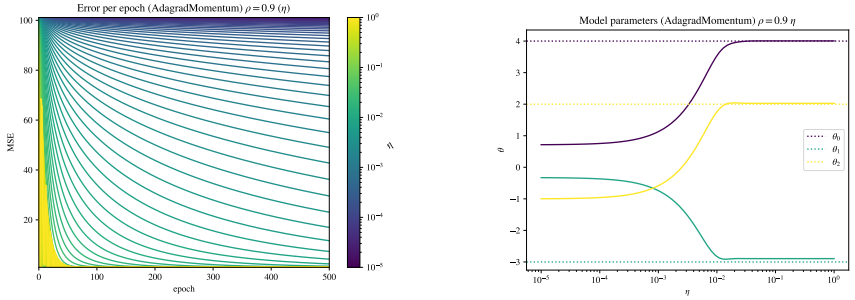


Figure 4.9: Predicted polynomials for the resulting parameter values $\boldsymbol{\theta}$ using Adagrad after 500 epochs.

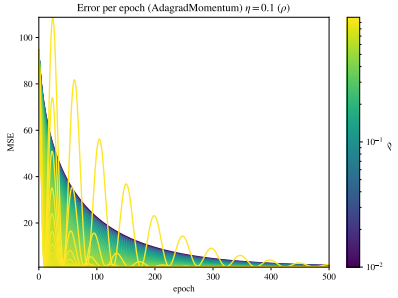
Introducing momentum leads to the same tendencies seen previously, seen in Figure 4.10, Figure 4.11 and Figure 4.12.



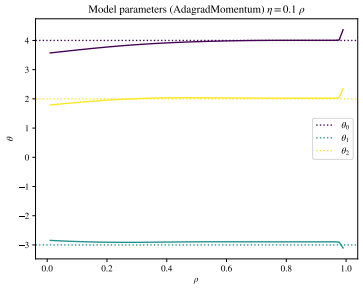
(a) Error for varying η.

(b) Parameters θ for varying η.

Figure 4.10: Gradient descent with Adagrad Momentum over 500 epochs with varying η, $\rho = 0.9$.

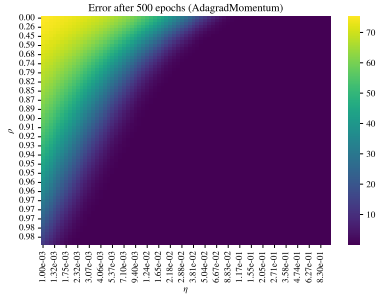


(a) Error for varying ρ.

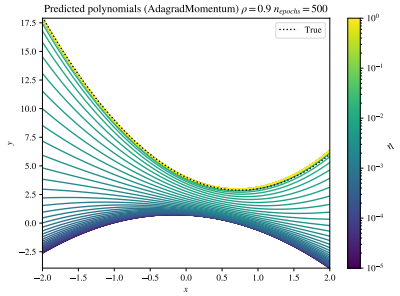


(b) Parameters θ for varying ρ.

Figure 4.11: Gradient descent with Adagrad Momentum over 500 epochs with varying ρ, $\eta = 0.1$.

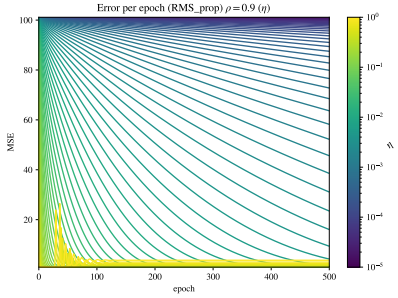


(a) Heatmap of η vs ρ.

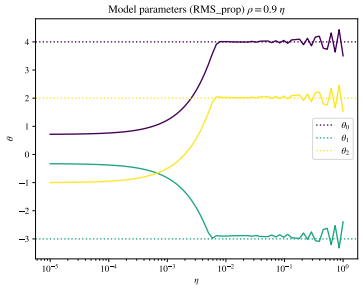


(b) Predicted polynomials for varying η, $\rho = 0.9$

Figure 4.12: Heatmap and predictions from Gradient descent with Adagrad Momentum over 500 epochs.

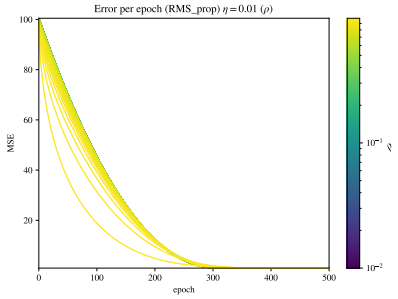


(a) Error for varying η.

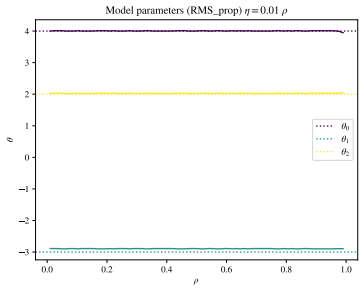


(b) Parameters θ for varying ρ.

Figure 4.13: Gradient descent with RMSprop over 500 epochs with varying η, $\rho = 0.9$.



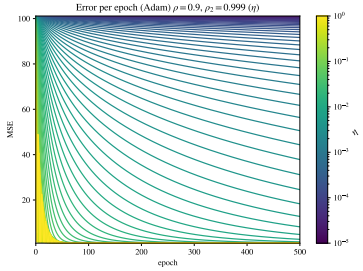
(a) Error for varying ρ.



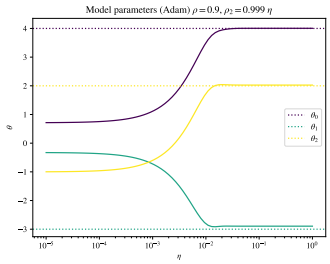
(b) Parameters θ for varying ρ.

Figure 4.14: Gradient descent with RMSprop over 500 epochs with varying ρ, $\eta = 0.01$.

RMSprop gave rather stable results across the varying values of ρ, perhaps due to the simple nature of our chosen test function, as seen in Figure 4.14. The final scheduler applied to deterministic Gradient Descent was Adam, seen in Figure 4.15. Varying ρ_1 in Figure 4.16 gave similar results to what we've seen previously. Varying ρ_2 in Figure 4.17 gave highly chaotic results, and such we will henceforth be sticking with $\rho_2 = 0.999$.

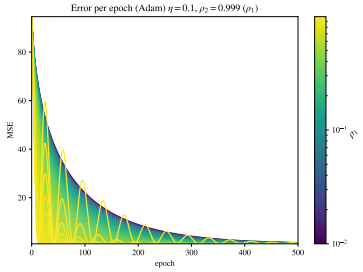


(a) Error for varying η.

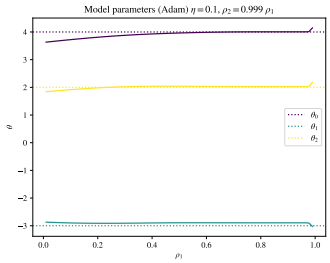


(b) Parameters θ for varying η.

Figure 4.15: Gradient descent with Adam over 500 epochs with varying η, $\rho_1 = 0.9$ and $\rho_2 = 0.999$.

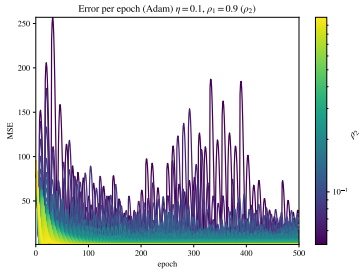


(a) Error for varying ρ_1.

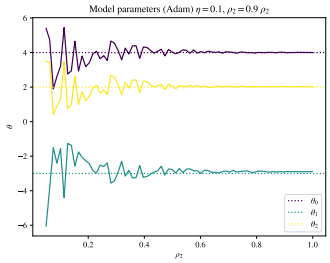


(b) Parameters θ for varying ρ_1.

Figure 4.16: Gradient descent with Adam over 500 epochs with varying ρ_1, $\eta = 0.1$ and $\rho_2 = 0.999$.



(a) Error for varying ρ_1.



(b) Parameters θ for varying ρ_2.

Figure 4.17: Gradient descent with Adam over 500 epochs with varying ρ_2, $\eta = 0.1$ and $\rho_1 = 0.9$.

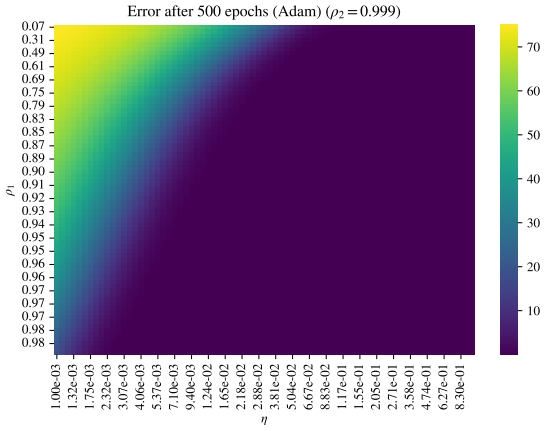
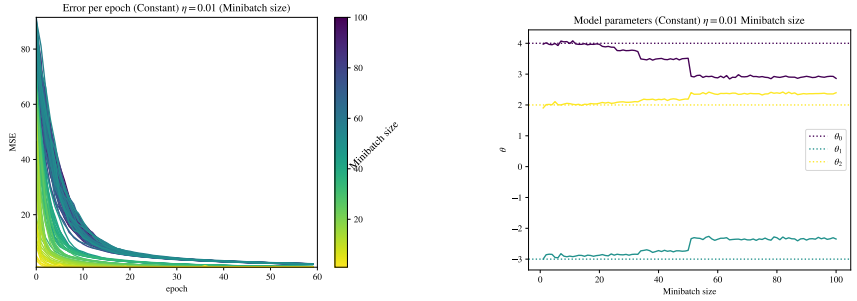


Figure 4.18: Heatmap of ρ_1 vs. η for Gradient Descent with Adam after 500 epochs, with $\rho_2 = 0.999$.

Fixing ρ_2 as 0.999 in Figure 4.18, we see the equivalent results to what we've seen previously. Notably, a number of these methods were developed for use with Stochastic Gradient Descent, described in algorithm 1.



(a) Error for varying minibatch sizes.

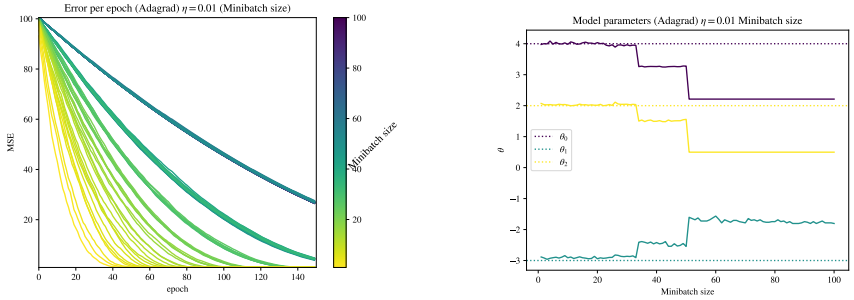
(b) Parameters θ for varying minibatch size.

Figure 4.19: Stochastic Gradient Descent for varying number of sizes of minibatches over 60 epochs.

The stochastic noise is evident in Figure 4.19. Interestingly, we observe a decrease in accuracy for larger minibatch sizes, perhaps resulting from the sampling of batches being with replacement, as the minibatch size approaches the total number of points 100.

algorithm 1 further explains the larger gaps in (a) and the step-like pattern in (b). The number of iterations in its inner loop is determined by $\lfloor \frac{n}{M} \rfloor = \lfloor \frac{100}{M} \rfloor$. This causes abrupt jumps when decreasing the batch size grants the algorithm

another iteration, i.e. when $\left\lfloor \frac{100}{M-1} \right\rfloor = \left\lfloor \frac{100}{M} \right\rfloor + 1$. This is shown more clearly in Figure 4.20.

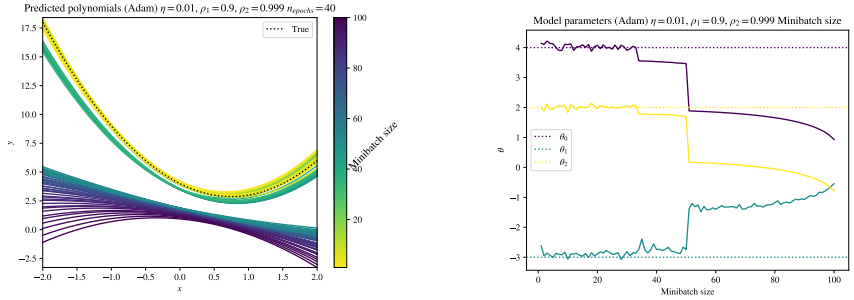


(a) Error for varying minibatch sizes

(b) Parameters θ for batchsize

Figure 4.20: Stochastic Gradient Descent with Adagrad over 150 epochs with $\eta = 0.1$

These gaps are again different for ADAM in Figure 4.21. These differences stem from the resetting of the cumulative memory parameters for each epoch. Remember from subsubsection 2.2.6 how the amplification of step size is largest at the beginning of the iteration. This causes the steeper steps in Figure 4.21 (b), greatly affecting the convergence rate seen in Figure 4.22



(a) Error for varying minibatch sizes

(b) Parameters θ for batchsize

Figure 4.21: Stochastic Gradient Descent with Adam over 40 epochs with $\rho_2 = 0.999$, $\rho_1 = 0.9$ and $\eta = 0.01$

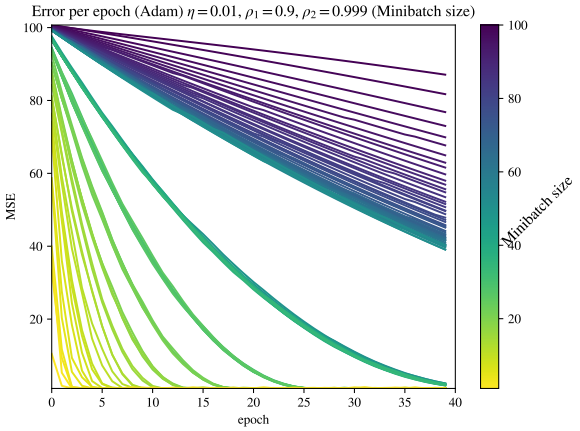
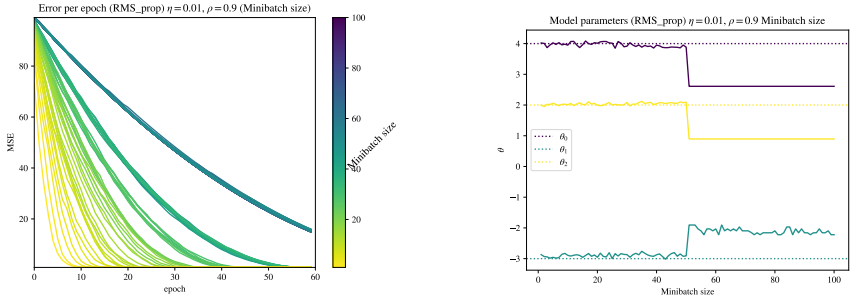


Figure 4.22: Error for Stochastic Gradient Descent with Adam after 40 epochs, with $\rho_2 = 0.999$ and $\eta = 0.01$.

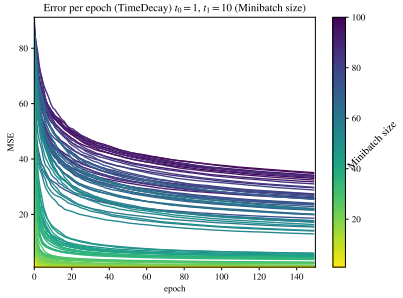


(a) Error for varying minibatch sizes

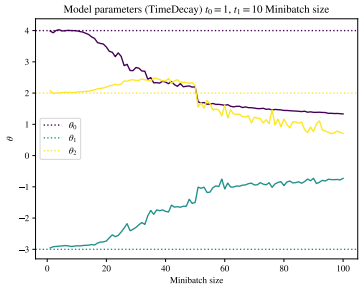
(b) Parameters θ for batchsize

Figure 4.23: Stochastic Gradient Descent with RMSprop over 60 epochs with $\rho = 0.9$ and $\eta = 0.1$

With RMSprop we see the steepest steps, approximately converging with just two batches as seen in Figure 4.23. We see interesting behaviour with time decay in Figure 4.24, starting out with rapid convergence, but quickly stabilizing at various degrees of success. This is not suprising, giving the simplistic nature of the scheduler. It has less significant jumps, being also dependent on the size of the batches.



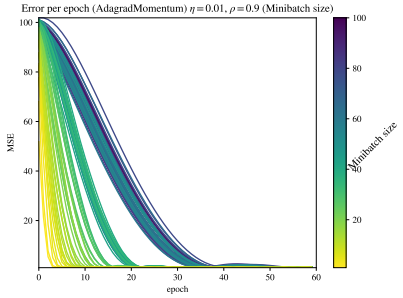
(a) Error for varying minibatch sizes



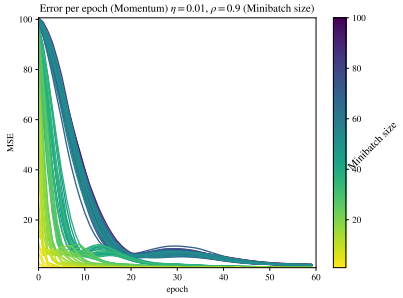
(b) Parameters θ for batchsize

Figure 4.24: Stochastic Gradient Descent with Time decay over 150 epochs with $t_0 = 1$ and $t_1 = 10$

Perhaps the most interesting graph, is seen in Figure 4.25 with Momentum and Adagrad Momentum. Here we can clearly see the similarities between the approaches. Note that while the standard Momentum approach converges faster, it jumps up for a bit after first bottoming out, a behaviour we hardly see with Adagrad.



(a) Adagrad Momentum



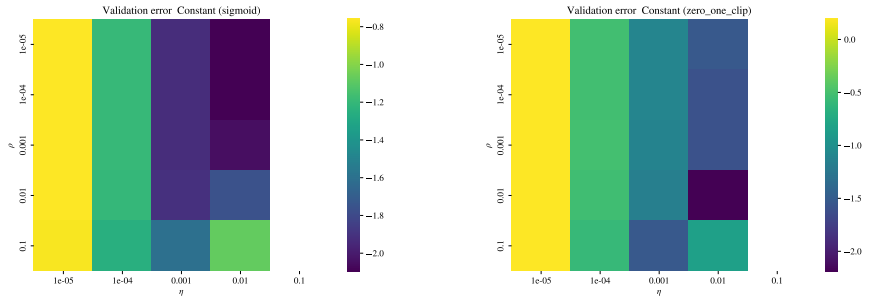
(b) Momentum

Figure 4.25: Error for varying size of minibatches from Stochastic Gradient Descent with Momentum and Adagrad Momentum over 60 epochs with $\eta = 0.01$ and $\rho = 0.9$

4.2 Franke's function

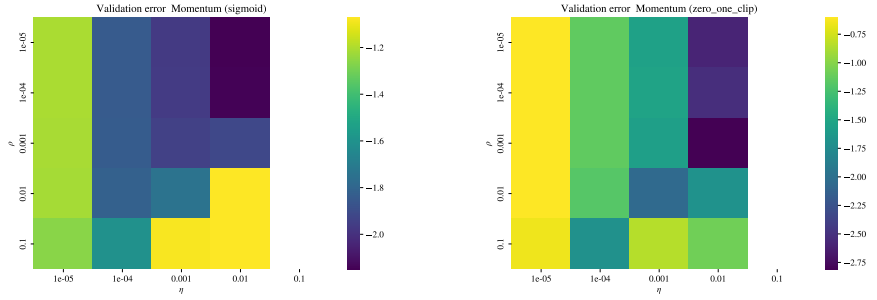
With a better understanding of how drastic our choices of hyperparameters can affect our final results, we begin by exploring Franke's function through the use of a neural network. There are a lot of moving parts, with varying learning rates, regularization terms, activation function, etc. In order to get any results, we fix the hyperparameters for the schedulers from what has previously been shown to work, namely $\rho/\rho_1 = 0.9$, $\rho_2 = 0.999$. We also restrict ourselves to 100 epochs with a minibatch size of 40. For the sake of consistency, the neural networks all consisted of three hidden layers of width respectively 75, 75 and 100. Due to computational constraints, we only applied a 80/20 train-validation split to see how well the networks perform on unseen data.

Note that due to a typo, the y-axis of the heatmaps are labeled ρ when they should be λ. What follows are heatmaps of the final validation error for varying combinations of scheduler and hidden function, plotted in log-scale.



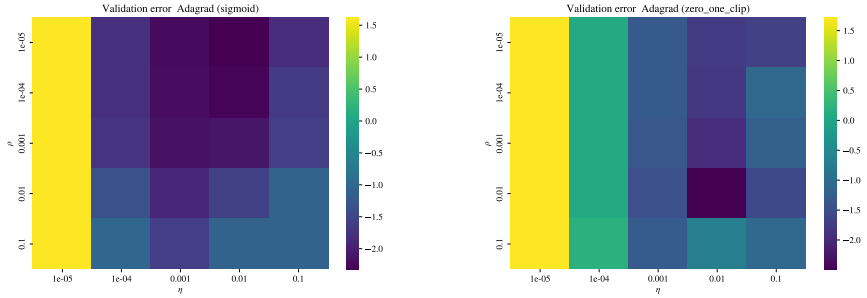
(a) Sigmoid (b) (0,1)-clip

Figure 4.26: Heatmap of η vs. λ for constant scheduler



(a) Sigmoid (b) (0,1)-clip

Figure 4.27: Heatmap of η vs. λ for momentum scheduler



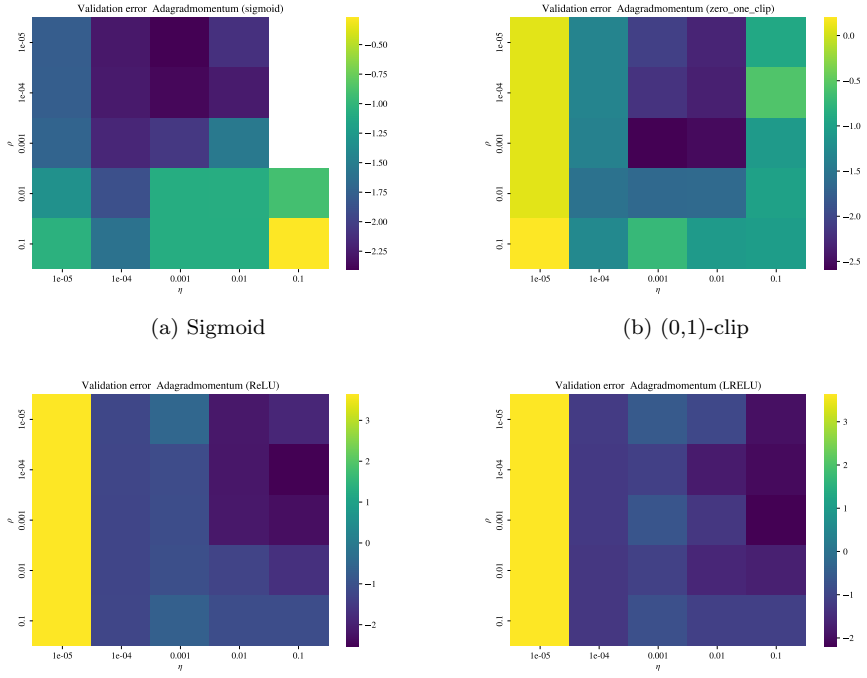
(a) Sigmoid

(b) (0,1)-clip

(c) ReLU

(d) LReLU

Figure 4.28: Heatmap of η vs. λ for Adagrad scheduler



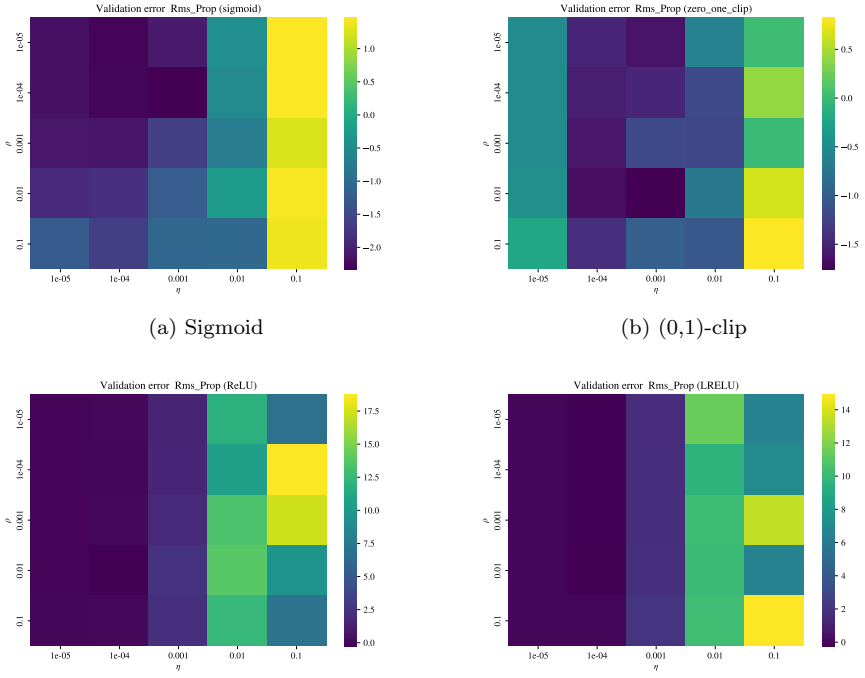
(a) Sigmoid

(b) (0,1)-clip

(c) ReLU

(d) LReLU

Figure 4.29: Heatmap of η vs. λ for AdagradMomentum scheduler



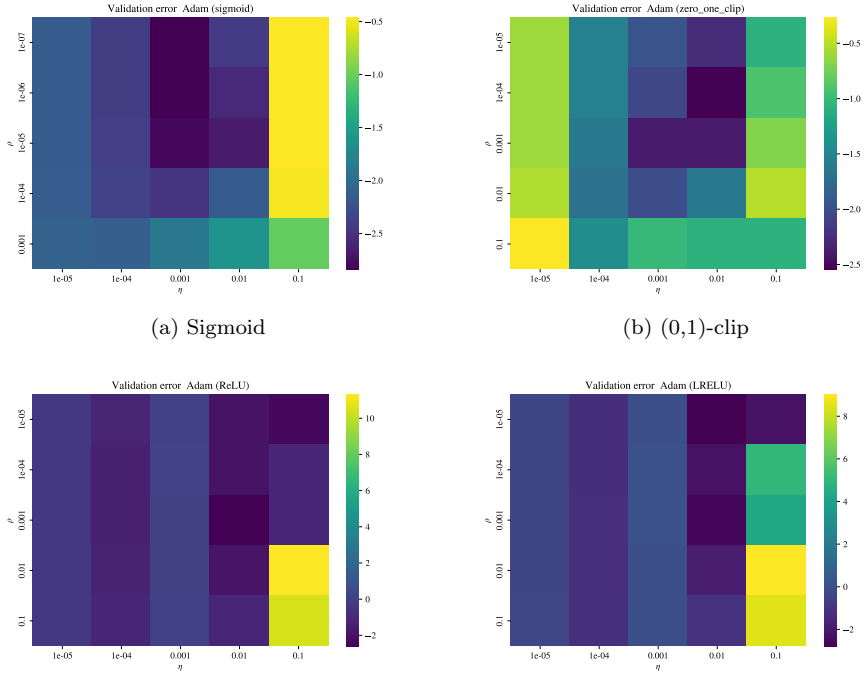
(a) Sigmoid

(b) (0,1)-clip

(c) ReLU

(d) LReLU

Figure 4.30: Heatmap of η vs. λ for RMSprop scheduler



(a) Sigmoid

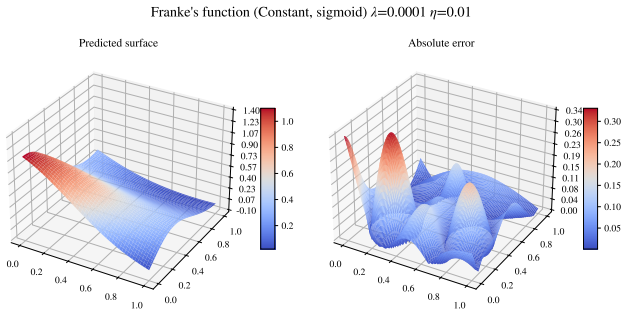
(b) (0,1)-clip

(c) ReLU

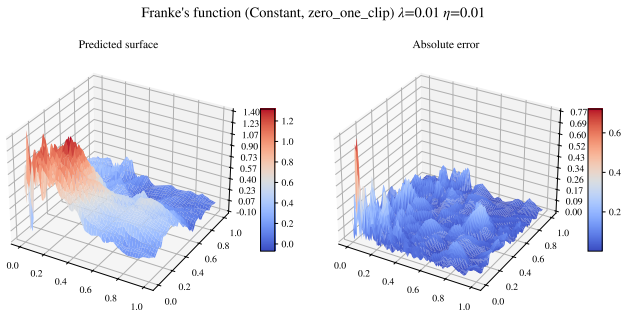
(d) LReLU

Figure 4.31: Heatmap of η vs. λ for Adam scheduler

Selecting the top performers from each, we get



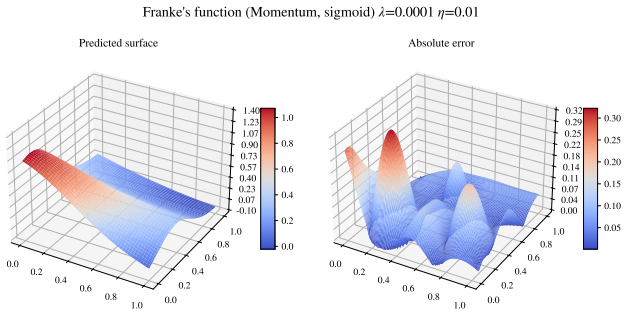
(a) Sigmoid



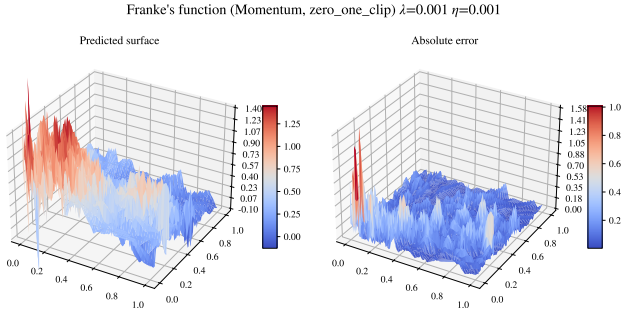
(b) (0,1)-clip

Figure 4.32: Predicted surface and absolute error with constant scheduler.

In Figure 4.32 we see drastically different surfaces visually, however they had comparable validation error as seen in Figure 4.26.



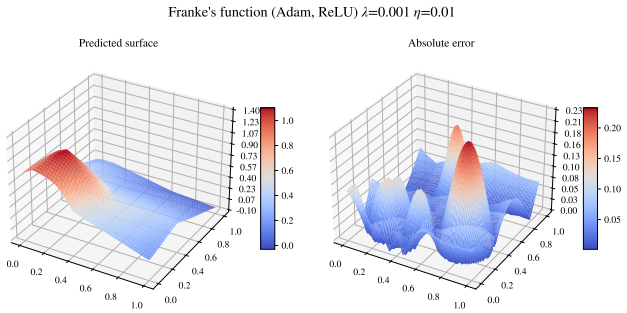
(a) Sigmoid



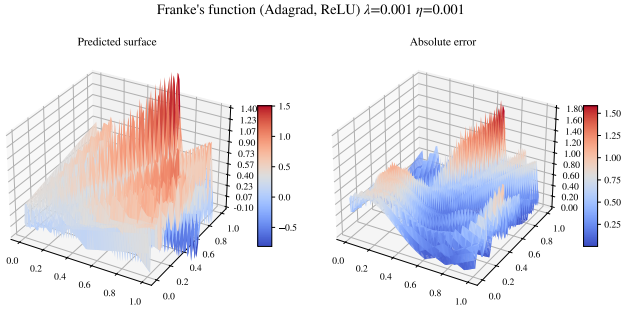
(b) (0,1)-clip

Figure 4.33: Predicted surface and absolute error with momentum scheduler.

Selecting the top performers from Figure 4.27 generated Figure 4.33. Note that (b) supposedly had better results, which might be due to it having slimmer "peaks", rather than (a)'s more continuous errors.



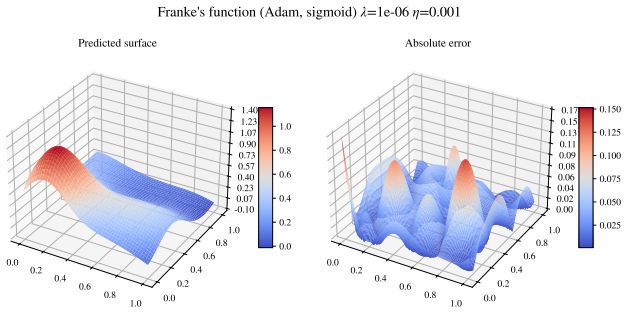
(a) Adam



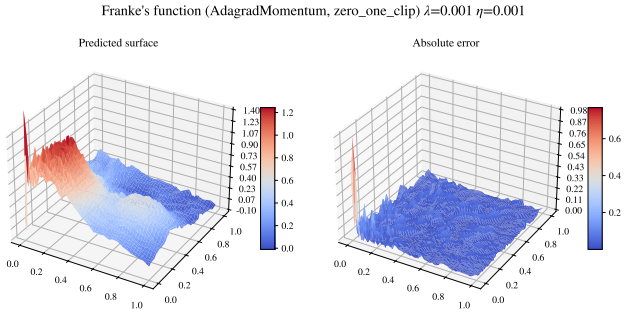
(b) Adagrad

Figure 4.34: Predicted surfaces using ReLU, with Adam and Adagrad.

We had wildly inconsistent results using ReLU and LReLU, illustrated in Figure 4.34, the reason for this is at the time of writing unknown to the authors.



(a) Adam with sigmoid



(b) AdagradMomentum with (0,1)-clip

Figure 4.35: Top performing predictions of Franke's function.

Our top performers were found with the combination of the Adam scheduler and sigmoid activation functions, and Adagrad Momentum with (0,1)-clip, illustrated in Figure 4.35. Keep in mind that these results were generated with a comparatively small neural network, run over a limited number of iterations.

4.3 Tumor classification

When exploring the optimal parameters, we ran into the same issues with ReLU and LReLU as seen previously, illustrated here in Figure 4.36. They are therefore excluded from this section.

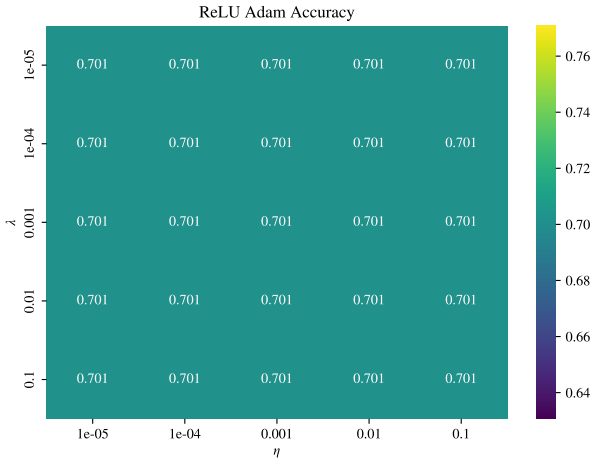
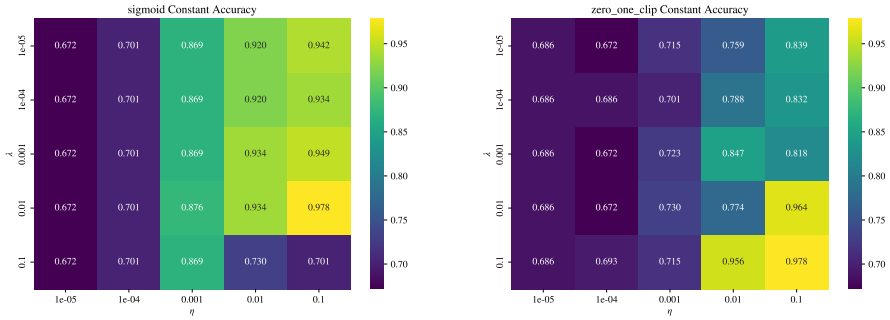


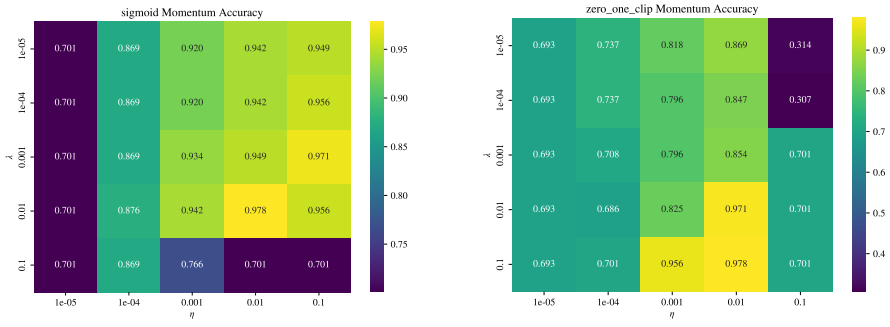
Figure 4.36: Accuracy when using ReLU as the hidden activation function.



(a) Sigmoid

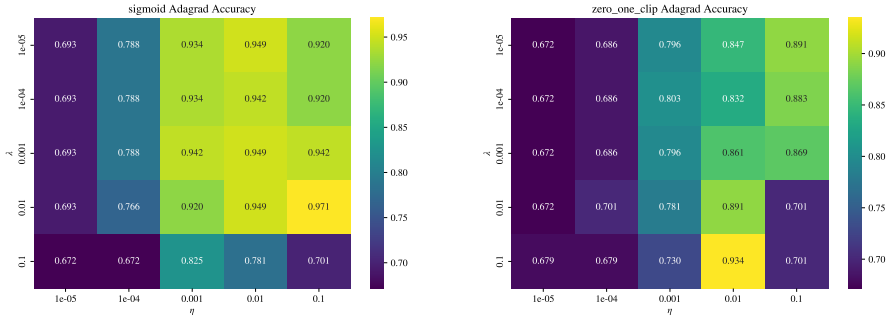
(b) (0,1)-clip

Figure 4.37: Heatmap of η vs. λ for constant scheduler.



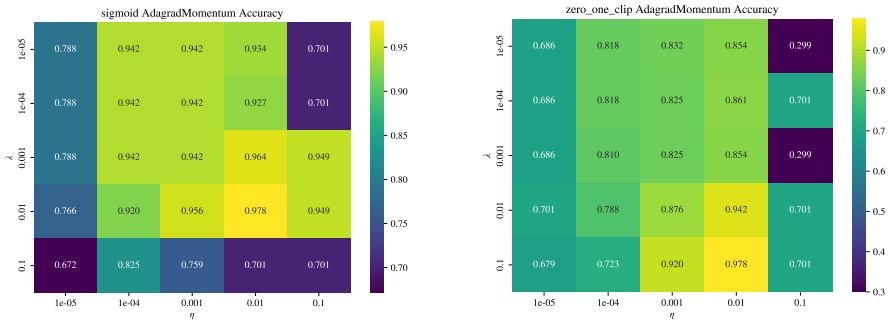
(a) Sigmoid (b) (0,1)-clip

Figure 4.38: Heatmap of η vs. λ for Momentum scheduler.



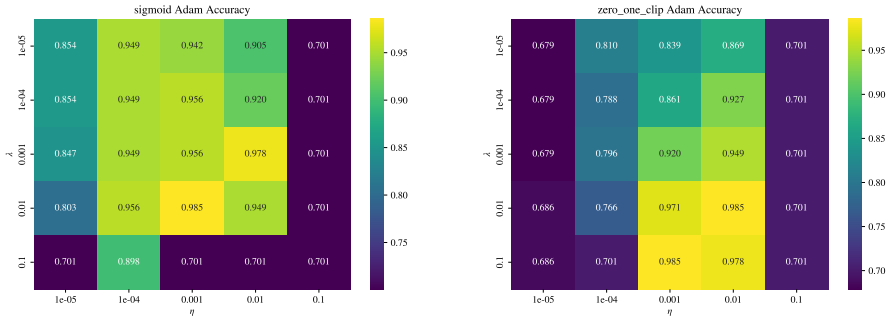
(a) Sigmoid (b) (0,1)-clip

Figure 4.39: Heatmap of η vs. λ for Adagrad scheduler.



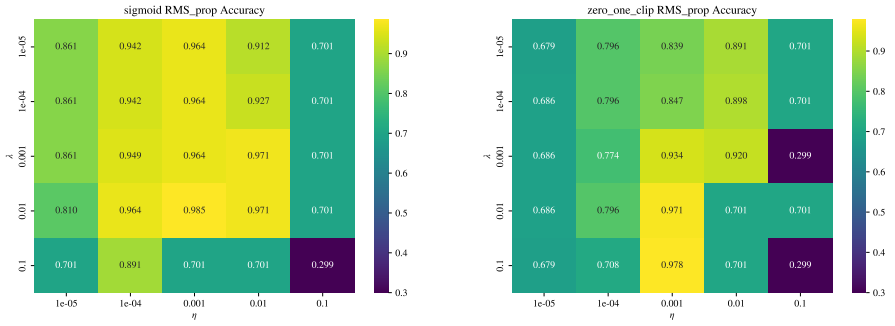
(a) Sigmoid (b) (0,1)-clip

Figure 4.40: Heatmap of η vs. λ for AdagradMomentum scheduler.



(a) Sigmoid (b) (0,1)-clip

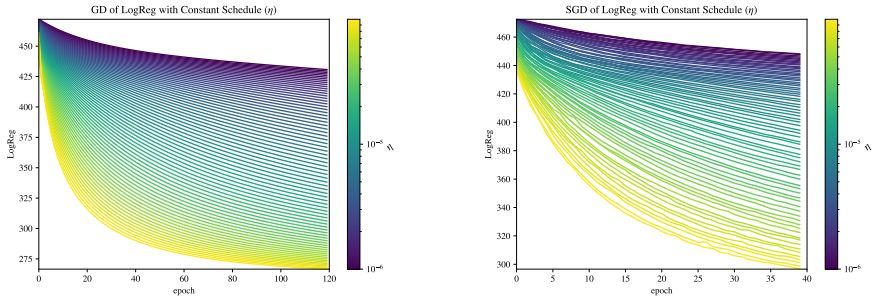
Figure 4.41: Heatmap of η vs. λ for Adam scheduler.



(a) Sigmoid (b) (0,1)-clip

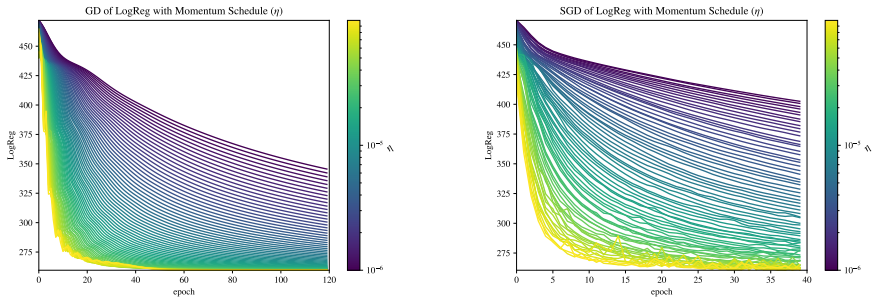
Figure 4.42: Heatmap of η vs. λ for RMSprop scheduler.

Figures 4.37 to 4.42 illustrate the validation accuracy for the different schedulers and hidden activation functions. Note that higher values are considered better for accuracy. We got a peak accuracy of 0.985 with Adam and RMSprop.



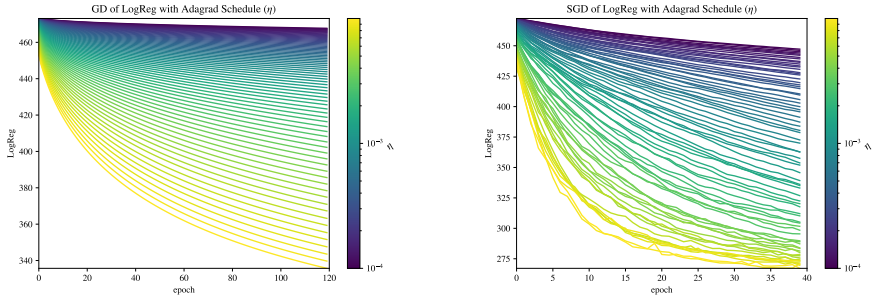
(a) GD (b) SGD

Figure 4.43: Logistic regression with Constant scheduler.



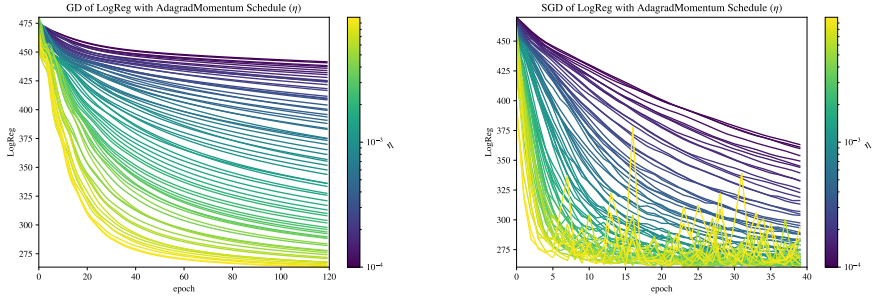
(a) GD (b) SGD

Figure 4.44: Logistic regression with Momentum scheduler.



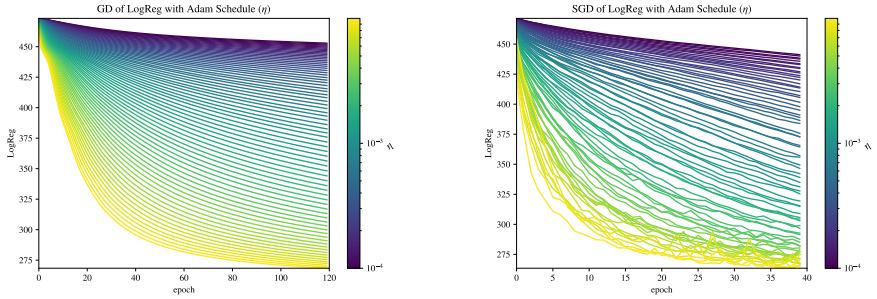
(a) GD (b) SGD

Figure 4.45: Logistic regression with Adagrad scheduler.



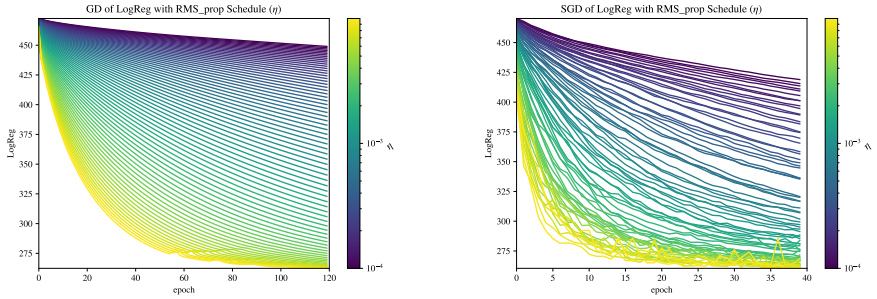
(a) GD (b) SGD

Figure 4.46: Logistic regression with AdagradMomentum scheduler.



(a) GD (b) SGD

Figure 4.47: Logistic regression with Adam scheduler.



(a) GD (b) SGD

Figure 4.48: Logistic regression with RMSprop scheduler.

Figures 4.43 to 4.48 show the results from logistic regression with varying learning parameters. Although not included in this report due to time constraints, our best accuracy was found with Adam with an accuracy of 0.86%.

5 Discussion

5.1 Univariate Polynomial Analysis

In our analysis, all deterministic optimizers, given 500 epochs and a fair choice of parameters, consistently converged to correct θ values. These parameters included η for Gradient Descent, ρ for Momentum, AdagradMomentum and RMSprop, and ρ_1 and ρ_2 for Adam. This ease of convergence most likely stems from the simplicity of the problem.

In addition, methods such as Adagrad, Adam, RMS Prop and other momentum-based strategies showed less sensitivity to the choice of η. They converged quickly even with high η values. This feature hints at their superior flexibility, suggesting they could provide faster convergence for more complex descent problems.

For a more direct comparison of optimization methods' performances, errors could have been plotted on the same graph. This would provide immediate insights into each method's speed and stability of convergence. However, the unique sensitivities of each method to parameter choice would still require individual analysis.

Our examination of stochastic techniques showed similar results for smaller batch sizes. However, these methods displayed instability when the batch size was not significantly smaller than the number of training samples. Given the number of samples and the simplicity of the problem, these methods were not expected to yield any additional performance.

5.2 Franke's function

In our model, validation was conducted through evaluating the performance on unseen data. More robust validation methods such as bootstrap or cross-validation could have been employed for a more comprehensive analysis, but these were not implemented due to time constraints.

For our final layer activation, i.e., z-prediction, we employed the identity function to avoid biasing our output to a specific range. For the cost function we employed the L2 regularization term on the MSE to form the Ridge cost function. Our computational limitations encouraged us to opt for the stochastic variants of all our schedulers, as the deterministic counterparts were tediously slow when working with 10 000 points of data.

Throughout the analysis, we found that more advanced schedulers like Adam, AdagradMomentum, and RMSprop notably expanded the range of acceptable η and λ values (here, 'acceptable' is defined as yielding the lowest range of test errors observed in the magnitude of 10^{-3}). ADAM in particular gave low test errors for almost all our λ and η values as seen in Figure 4.31

When it comes to the activation of inner layers, the heatmaps gave a somewhat misleading representation of the error. As ReLU and LReLU seemed to allow for more variation in λ and η as seen in Figure 4.28. Furthermore, ReLU

typically outperformed LReLU across all schedulers, as illustrated in Figure 4.31 and Figure 4.29

Our 3D plots however, gave a more nuanced picture of the error. Also allowing us to see how the error is distributed. Notably for our best results seen in Figure 4.35. The total error of the two methods were about the same, although the distribution was completely different.

5.3 Tumor classification

In our analysis, the final evaluation of our trained models was based on their accuracy score over a test set. While this method is straightforward, it may not perfectly mirror real-world applications. Certain misclassifications may have more serious implications than others, meaning not all errors should be weighted equally. For instance, in a traffic light recognition system, classifying a red light as green could have far more severe consequences than misclassifying a green light as red. Therefore, for a comprehensive and realistic evaluation in future studies, a cost-sensitive error metric could be employed.

Nevertheless, we can observe that our FFNNs were able to achieve a remarkably high accuracy score, exceeding 0.985 with both ADAM and RMSprop optimizers (as shown in Figure 4.42 and Figure 4.41 respectively). This performance greatly surpassed that of our logistic regression models, which achieved only a score of 0.86. This leads us to believe that the relationship between the features and the tumour class is a more complex one that requires the capabilities of a neural network to accurately mode

6 Conclusion

This project highlighted the complexity of more advanced machine learning, as there are a large number of moving parts. Illustrating, and getting working results, proved difficult, particularly when restrained on computing power. We were perhaps a bit ambitious with the scope of our theory section, although we believe that the understanding we have gained of how Neural Networks operate will prove valuable in the future, having no previous knowledge before this project. We were faced with a number of different challenges, from how to visualize our results to attempting to get bearable performance out of our code. Further exploration possible from this project is how modelling using larger models, and utilizing methods like cross-validation and bootstrapping, can bring more accurate and informative results. Instead of focusing on the performance of our methods, we chose to focus on how the different methods differ from each other.

7 References

References

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., & Siskind, J. M. (2018). Automatic differentiation in machine learning: A survey.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., & Zhang, Q. (2018). *JAX: Composable transformations of Python+NumPy programs* (Version 0.3.13). http://github.com/google/jax

Chollet, F. (2018). *Deep learning with python* (1st edition.). Manning Publications Co.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, *2*, 303–314. https://api.semanticscholar.org/CorpusID:3958369

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, *12*(7).

Guliyev, N. J., & Ismailov, V. E. (2015). A single hidden layer feedforward network with only one neuron in the hidden layer can approximate any univariate function. *Neural Computation*, *28*, 1289–1304. https://api.semanticscholar.org/CorpusID:5791099

Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The elements of statistical learning: Data mining, inference and prediction*. Springer Verlag, Berlin.

Hinton, G. (2012). Neural networks for machine learning- lecture 6.

Jentzen, A., Kuckuck, B., & von Wurstemberger, P. (2023). Mathematical introduction to deep learning: Methods, implementations, and theory.

Kingma, D. P., & Ba, J. (2014). *Adam: A method for stochastic optimization* [arXiv preprint arXiv:1412.6980].

Lindstrøm, T. L. (2017). *Spaces: An introduction to real analysis* (Vol. 29). American Mathematical Soc.

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, *5*, 115–133.

Süli, E., & Mayers, D. F. (2003). *An introduction to numerical analysis*. Cambridge University Press. https://doi.org/10.1017/CBO9780511801181

Walther, A. (2007). Automatic differentiation of explicit runge-kutta methods for optimal control. *Computational Optimization and Applications*, *36*, 83–108.

Watt, J., Borhani, R., & Katsaggelos, A. (2020). *Machine learning refined: Foundations, algorithms, and applications (2nd ed.)* Cambridge University Press. https://doi.org/10.1017/9781108690935

Wolberg, W. (1992). Breast Cancer Wisconsin (Original). https://doi.org/10.24432/C5HP4Z

8 Appendix

The code is publicly available on github.com/augustfe/FYSSTK, written in python.