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

Written by R.S

Abstract

This article delves into the intricacies of Feedforward Neural Networks (FFNNs) in the context of construction prediction and classification. The primary objective is look at the behavior of FFNNs, shedding light on their application in construction-related tasks. The study closely examines the theoretical underpinnings of FFNNs, focusing on fundamental components such as input layers, hidden layers, and activation functions.

The study involves comparative simulations, with traditional methods like linear regression for prediction and logistic regression for classification serving as benchmarks. Through these simulations, the study explores how FFNNs operate in these scenarios, comparing them to ordinary techniques.

By studying FFNNs' behavior through practical simulations, this study provides insights into their suitability for construction prediction and classification tasks.

Contents

Introduction

This study delves into FFNNs in the context of construction prediction and classification, aiming to look at their intricacies when applied to construction data. At the core of FFNNs lies a series of computations facilitated by interconnected layers – the input layer receiving initial data, hidden layers performing computations using weighted connections, and an output layer generating predictions. The application of activation functions at hidden layers introduces non-linearity, enabling FFNNs to capture complex data patterns that elude traditional linear methods.

Central to the training process is the concept of the Forward and Backward pass. During the forward pass, input data traverses through the network, generating predictions. In the backward pass, error gradients are calculated from the loss function, and these gradients are propagated backward through the network. The Backpropagation, rooted in the chain rule, enables the efficient computation of gradients.

Building on the foundation of gradient descent methods, this study explores various theoretical concepts in addition to a small simulation study bought for a prediction task and a classification task. FFNNs are benchmarked against ridge and linear regression, providing a comparative perspective to evaluate their performance in prediction tasks.The FFNN is assessed for prediction purposes, employing the $SRTM\_data\_Norway\_1.tif$ dataset and the FFNN are assessed for classification purposes, employing the Wisconsin Breast Cancer dataset. The simulation outcomes are examined, with the aim of getting a feel for the FFNN strength and limitations in comparison to traditional methods.

The code of our simulation study can be found in this link: https://github.com/gituser1234566/some_coding/tree/main/project2fys-stk4155

This text has largely been motivated by [GBC16],[HTF01] and discussions using ChatGPT 3.5.

In the simulation study we frequently use the FFNN from [Hjo23]

Introduction to FFNN

Universal Approximation Properties and Depth

In recent years, neural networks have garnered widespread acclaim as versatile tools for a myriad of tasks. Their appeal lies in their capability to discern intricate patterns within vast datasets. This ability is formally underpinned by the universal approximation theorem, stating that a feedforward network with a linear output layer and at least one hidden layer featuring any bounded non-linear activation function (such as the logistic sigmoid) can accurately approximate any Borel measurable function from one finite-dimensional space to another. This holds true with any desired non-zero error, contingent upon having a sufficient number of hidden units. Furthermore, the derivatives of the feedforward network can approximate the derivatives of the function with arbitrary precision. This universality extends to approximating any function mapping from any finite-dimensional discrete space to another.

While the original theorems were framed with activation functions saturating for both negative and positive arguments, universal approximation theorems now encompass a broader class of activation functions, including the widely-used rectified linear unit (ReLU) which we will come back to.

To illustrate, consider the XOR function, a binary operator returning 0 or 1 given two inputs (0 or 1):

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

Treating this as a regression problem, we employ Mean Squared Error (MSE) as the loss function. However, in practice, MSE is not the optimal choice for modeling binary data.

$$J(\theta) = \frac{1}{4} \sum_{x \in X} (f(x) - f(x; \theta))^2 \tag{2.1}$$

Consider the linear model:

$$f(x; w, b) = x^T w + b \tag{2.2}$$

A linear model applied directly cannot effectively implement XOR. However, by transforming the output through a non-linear function, we can overcome this limitation.

Due to the linear function's constraints, the minimization problem solution is $w = 0$ and $b = \frac{1}{2}$. All weights are set to 0, and b is adjusted to minimize $J(\theta)$.

A linear model applied directly cannot effectively implement XOR. When $x_1 = 0$, the model's output has to increase as x_2 increases. When $x_1 = 1$, the model's output has to decrease as x_2 increases. Our linear model is a linear combination of the weights of x_1 and x_2. The weights of x_2 are only dependent on x_2, therefore x_1 has no influence on them and vice versa. If we, however, transform the output by wrapping a non-linear function around it, we can change the placement of the points so that they can be modeled linearly.

Can, we consider adding an additional layer?:

$$f^{(2)}(x; W, c) \tag{2.3}$$

Values of these hidden units serve as the input for a second layer, the output layer. Although the output layer remains a linear regression model, it is now applied to h rather than x. The network consists of two functions, with $f^{(1)}$ becoming the input of $f^{(2)}$, $h = f^{(1)}(x; W, c)$, and $y = f^{(2)}(h; w, b)$, resulting in the complete model $f(x; W, c, w, b) = f^{(2)}(f^{(1)}(x))$.

Choosing both $f^{(1)}$ and $f^{(2)}$ to be linear presents the same problem for b_2, the bias of the second layer. Therefore, a non-linear function is essential to describe the relationship between input and output. The general setup of a neural network achieves this through an affine transformation, incorporating a linear transformation of the input with a parameter matrix and an affine part that shifts the parameter input multiplication with a vector called the bias. The result is then passed through a non-linear function, known as an activation function. This process can be expressed as $h = g(W^T x + c)$, which in our case for this example will be the rectified linear function (ReLU).

The two-step process for estimating the XOR function is defined as:

$$f(x; W, c, w, b) = w^T \max(0, W^T x + c) + b \tag{2.4}$$

With:

$$W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \quad c = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \quad w = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \quad b = 0 \tag{2.5}$$

For this example, X is defined as:

$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \tag{2.6}$$

First, we multiply X with W:

$$XW = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix} \tag{2.7}$$

Next, we add c:

$$XW = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \tag{2.8}$$

Multiplying with the weight vector:

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \tag{2.9}$$

This example provides a foundational understanding of a neural network. In real-world scenarios, networks are more intricate and not as easily analyzed. The iterative process of applying a non-linear function to an affine transformation is extended by repeatedly applying such transformations to the same input x, creating a network of transformations that expand and contract dimensions. Each node or transformation, potentially composed of other transformations, is known as an artificial neuron:

$$y = f\left(\sum_{i=1}^{n} w_i x_i\right) = f(u) \tag{2.10}$$

A set of neurons and layers transition between each layer in a graph represents an activation function applied to the input of the previous layer, multiplied by weight, and adding bias. This iterative network building starts with input data represented by a design matrix. Feedforward neural networks (FFNN) are fully connected, with every input variable being a parent of every hidden node in

the first hidden layer. Each node in the second hidden layer has all the nodes in the first hidden layer as parents, and so forth. For all nodes in the first layer, we write:

$$y_i^{(1)} = f(z_i^{(1)}) = f\left(\sum_{j=1}^{M} w_{i,j}^{(1)} x_j + b_i^{(1)}\right) \tag{2.11}$$

where $y_i^{(1)}$ is the output of all nodes in layer 1 for input feature i.

This process continues, and every layer can be expressed as:

$$y_i^{(l)} = f^{(l)}\left(\sum_{j=1}^{N_{l-1}} w_{i,j}^{(l)} y_j^{(l-1)} + b_i^{(l)}\right) \tag{2.12}$$

More explicitly:

$$y_i^{(l+1)} = f^{(l+1)}\left(\sum_{j=1}^{N_l} w_{i,j}^{(l)}\left(...f\left(\sum_{n=1}^{N_0} w_{i,j}^{(1)} x_{m,n} + b_m^{(1)}\right)...\right) + b_i^{(l)}\right) \tag{2.13}$$

In vector form, the output of hidden layer 2 for m features becomes:

$$\begin{bmatrix} w_{11} & w_{12} & \dots & w_{1m} \\ w_{21} & w_{22} & \dots & w_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \dots & w_{mm} \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ \vdots \\ h_{1m} \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

This calculation is performed in the fitting procedure of the network to the function we are trying to approximate and is referred to as the forward step.

Activation Functions in Neural Networks

Considerable research has focused on identifying optimal activation functions for neural networks. Here are some commonly used ones:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{2.14}$$

$$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \tag{2.15}$$

$$\text{ReLU}(x) = \max(0,x) \tag{2.16}$$

$$\text{Leaky ReLU}(x,\alpha) = \begin{cases} x, & \text{if } x > 0 \\ \alpha x, & \text{otherwise} \end{cases} \tag{2.17}$$

$$\text{ELU}(x,\alpha) = \begin{cases} x, & \text{if } x > 0 \\ \alpha(e^{x}-1), & \text{otherwise} \end{cases} \tag{2.18}$$

$$\text{Softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}, \quad \text{for } i = 1,2,\ldots,K \tag{2.19}$$

2.1 Backpropagation

Backpropagation Example for Prediction

The next step to understand when it comes to fitting a neural network is the backward-step utilizing backpropagation.This step starts after the forward pass. Backpropagation is a critical step in minimizing the cost function of a neural network. For illustration, we focus on the widely-used Mean Squared Error (MSE) cost function, scaled by $\frac{1}{2}$ for simplicity:

$$\frac{1}{2}\sum_{i=1}^{n}(y_i - t_i)^2,$$

In the context of neural networks, the loss is typically defined in statistical terms to align with the model's goal of approximating stochastic functions of the input. In a frequentist setting with point estimates, the loss takes the form of maximum likelihood:

$$J(\theta) = -E_{x,y\sim\hat{p_{data}}}[\log p_{model}(y|x)]$$

Notably, using this loss function leads to the same minimization as the negative expected log likelihood of i.i.d. normal distribution input data points.

Here, t_i is the target, y_i is the estimated target, x is the input, and z_j^l is the activation of neuron j in the l-th layer, defined as:

$$z_j^l = \sum_{i=1}^{M_{l-1}} w_{i,j}^l a_i^{l-1} + b_j^l,$$

As discussed earlier, this can be expressed in matrix form:

$$z^l = (W^l)^T a^{l-1} + b^l$$

For this example, let's choose the sigmoid function:

$$a_j^l = f(z_j^l) = \frac{1}{1 + \exp -z_j^l}$$

In neural networks, the parameters are weights and biases. The objective is to refine estimations to align with the target. This involves finding the minimum of the cost function. To do this, we take the derivative of the cost function with respect to the parameters, using the chain rule. For the chosen cost function, the derivative concerning parameters in the last layer is:

$$\frac{dC(W^L)}{dw_{j,k}^L} = (a_j^L - t_j)\frac{da_j^L}{dw_{j,k}^L} \tag{2.20}$$

where:

$$\frac{da_j^L}{dw_{j,k}^L} = \frac{da_j^L}{dz_j^L}\frac{dz_j^L}{dw_{j,k}^L} \tag{2.21}$$

$$\frac{da_j^L}{dw_{j,k}^L} = (a_j^L - t_j)a_j^L(1 - a_j^L)a_k^{L-1} \tag{2.22}$$

We introduce δ^L as:

$$\delta_j^L = \frac{dC}{dz_j^L} = \frac{dC(W^L)}{da_j^L}\frac{da_j^L}{dz_j^L} \tag{2.23}$$

δ^L can also be expressed as:

$$\delta_j^L = \frac{dC}{dz_j^L} = \frac{dC(W^L)}{db_j^L}\frac{db_j^L}{dz_j^L} \tag{2.24}$$

The versatile δ_j^l for any l facilitates computation:

$$\delta_j^l = \sum_k \delta_k^{l+1}\frac{dz^{l+1}}{dz_j^l} \tag{2.25}$$

Now, with δ_j^l, we can compute any $\frac{dC(W^L)}{dw_{j,k}^L}$:

$$\frac{dC}{dw_{j,k}^l} = \delta_j^l \frac{dz_j^l}{dw_{j,k}^l} \tag{2.26}$$

As we progress in our quest for minimization through gradient methods, the parameter update equations come into focus:

$$w_{j,k}^l = w_{j,k}^l - \eta \delta_j^l \frac{dz_j^l}{dw_{j,k}^l} \tag{2.27}$$

$$b_j^l = b_j^l - \eta \delta_j^l \tag{2.28}$$

The choice of activation function plays a crucial role in ensuring the smooth execution of the backward step, where parameters are perturbed to their newly updated evaluations. This step involves calculating the direction, step size, and the update itself through gradient methods.

Backpropagation traverses from the output layer to the input layer, propagating the error gradient. However, challenges like the vanishing and exploding gradient problems can arise. The vanishing gradient problem occurs when the gradient diminishes as it traverses layers, causing parameters in lower levels to remain unchanged, leading to convergence issues. Conversely, the exploding gradient problem involves excessively large updates, resulting in algorithm divergence.

The sigmoid function, often associated with biological neurons, exhibited unfavorable behaviors. A pivotal paper by Glorot and Bengio in 2010 revealed issues with the logistic sigmoid, emphasizing problems with weight initialization. Sigmoid's mean of 0.5 and saturation at 0 or 1, especially with large inputs, lead to vanishing gradients, which can hinder effective backpropagation.

To address this, Glorot and Bengio proposed an initialization technique to ensure proper signal flow in both forward and backward directions. This involved equalizing the variance of output and input at each layer.

what about the ReLU?. ReLU has its own challenges, such as "dying ReLUs," where some neurons cease to contribute during training. To overcome this, the ELU and Leaky ReLU were introduced. Guidelines for choosing activation functions suggest that ELU outperforms Leaky ReLU, which, in turn, outperforms ReLU. ReLU performs better than hyperbolic tangent, which, in turn, performs better than the logistic function.

Prediction and Cost Computation in Neural Networks

In the context of prediction within neural networks, the evaluation often involves utilizing the output of the final hidden layer during the forward pass to assess how closely the model approximates the target. When dealing with categorical

tasks, where assigning probability values to each class is crucial, an additional layer is introduced after the final layer. This layer serves to fractionate the output, with each fraction representing the probability of the input belonging to a specific class. The determination of each fraction's size is contingent upon class weights. The softmax function is commonly employed for this purpose, expressed as:

$$P(y_{i,c} = 1|x_i, \theta) = \frac{exp((a_i^{hidden})^T w_c)}{\sum_{c'=0}^{C-1} exp((a_i^{hidden})^T w_{c'})} \tag{2.29}$$

From a statistical standpoint, the cost function is defined in terms of likelihood, specifically:

$$P(D|\theta) = \prod_{i=1}^{n} \prod_{c=0}^{C-1} [P(y_{i,c} = 1)]^{y_{i,c}} \tag{2.30}$$

To express the cost function, denoted as $C(\theta)$, the negative logarithm of the likelihood is computed:

$$C(\theta) = -\log P(D|\theta) \tag{2.31}$$

In binary classification scenarios, the cost function(CrossEntropy) commonly takes the form:

$$C(\theta) = -\sum_{i=1}^{n} (y_i \log p(y_i|x_i, \beta)) + (1 - y_i) \log 1 - p(y_i|x_i, \beta) \tag{2.32}$$

For binary classification task, the activation output a_i^l is computed by:

$$a_i^L = \frac{exp(z_i^l)}{1 + exp(z_i^l)} \tag{2.33}$$

Where z_i^l is given by:

$$z_i^l = \sum_j w_{i,j}^l a_j^{l-1} + b_i^l \tag{2.34}$$

Reformulating the cost function $C(W)$ with the newly defined variables, we have:

$$C(W) = -\sum_{i=1}^{n} (t_i \log a_i^L) + (1 - t_i) \log 1 - a_i^L \tag{2.35}$$

The derivative of the cost function with respect to the final layer becomes:

$$\frac{dC(W)}{da_i^L} = \frac{a_i^L - t_i}{a_i^L(1 - a_i^L)} \tag{2.36}$$

Furthermore, when differentiating $f(z_i^l)$ with respect to $w_{j,k}^l$, we find:

$$\frac{df(z_i^l)}{dw_{j,k}^l} = \frac{df(z_i^l)}{dz_i^l}\frac{dz_j^l}{dw_{j,k}^l} \tag{2.37}$$

$$\frac{df(z_i^l)}{dw_{j,k}^l} = \frac{df(z_i^l)}{dz_i^l}a_k^{l-1} \tag{2.38}$$

For more than two classes we use:

$$f(z_i^l) = \frac{exp(z_i^l)}{\sum_{m=1}^K exp(z_m^l)} \tag{2.39}$$

And the derivative $\frac{df(z_i^l)}{dz_j^l}$ expressed as:

$$\frac{df(z_i^l)}{dz_j^l} = f(z_j^l)(\delta_{i,j} - f(z_j^l)) \tag{2.40}$$

2.2 Limitations of Supervised Learning in Deep Neural Networks

Deep Neural Networks (DNNs) are best suited in settings with a lot of data. However, this becomes a limitation in scenarios where data is scarce or datasets are small, typically comprising hundreds to a few thousand samples. In such cases, alternative methods leveraging hand-engineered features may outperform DNNs.

2.3 Logistic Regression in Classification

Our simulation study will compare the classification from a FFNN with logistic regression.In ordinary linear regression, the response is assumed to be given by:

$$y = f(x) + \epsilon, \tag{2.41}$$

where ϵ follows a normal distribution. Logistic regression, on the other hand, models the response as:

$$y = p_i + \epsilon, \tag{2.42}$$

with ϵ being multinomially distributed for more than two classes and binomially distributed in binary classification.

One notable advantage of logistic regression in classification lies in its mapping from the whole real line to the interval between [0,1], yielding intuitive probabilistic outputs. This characteristic provides a meaningful interpretation, especially in binary classification scenarios.

Linear regression, when applied to binary classification, tends to produce high outputs for values that are very negative or very positive. However, interpreting these outputs in proximity to the mean or handling outliers becomes challenging. Additionally, in scenarios with more than three classes, logistic regression addresses the issue of masking, where the classes cannot be effectively separated, leading to challenges in distinguishing some classes from others.

The regression part of multinomial logistic regression is revealed in its setup:

$$\frac{p(Y=1|X=x)}{p(Y=K|X=x)} = \beta_{0,1} + \beta_1^T x, \tag{2.43}$$

$$\frac{p(Y=K-1|X=x)}{p(Y=K|X=x)} = \beta_{K-1,1} + \beta_{K-1}^T x. \tag{2.44}$$

Class probabilities are expressed as:

$$p(Y=k|X=x) = \frac{e^{\beta_{0,k}+\beta_k^T x}}{1+\sum_{k'=1}^{K-1} e^{\beta_{0,k'}+\beta_{k'}^T x}} \tag{2.45}$$

$$p(Y=k|X=x) = \frac{1}{1+\sum_{k'=1}^{K-1} e^{\beta_{0,k'}+\beta_{k'}^T x}}. \tag{2.46}$$

2.4 Softmax Function for Classification Tasks

To understand the softmax for our setting,we start by looking at the negative of the cost function for classification,i.e the log likelihood.

$$\log P(y=i;z) = \log softmax(z)$$

We want to maximize this expression.The expression can further be broken down as:

$$\log softmax(z) = z_i - log \sum_j exp(z_j)$$

When maximizing the log-likelihood,the first term encourages z_i to be large large as possible,while the second term encourages all of z to be small as possible.When it comes to the second term,$\log \sum_j exp(z)$,this becomes approximately as $max_j z_j$.The intuition behind this approximation is that $\exp z_k$ is insignificant for any z_k that is noticeably less than $max_j z_j$.This implies that the negative

log-likelihood cost function always strongly penalizes the most active incorrect prediction.If the correct answer already has the largest input to the softmax,then $-z_i + log \sum_j exp(z_j) \approx max_j z_j = z_i$ will be close to zero. This example will then contribute little to overall training cost,which will be dominated by other examples that are not yet correctly classified. [GBC16][chapter 6,page 181-182]

Unregularized maximum likelihood drives the model to learn parameters that make the softmax predict the fraction of counts of each outcome observed in the training set:

$$\text{softmax}(z(x;\theta))_i \approx \frac{\sum_{j=1}^{m} 1_{y^{(j)}=i, x^{(j)}=x}}{\sum_{j=1}^{m} 1_{x^{(j)}=x}}.$$

While the softmax function is effective, it shares a saturation issue with the sigmoid function, reaching 1 or 0 when inputs are extreme.

2.5 Gradient Descent

Bought for neural networks and multinomial logistic regression we do not have analytical solutions of the minimizer of the cost function.We therefore have to resort to numerical gradient methods to find approximations to the minimum. We begin our exploration of gradient descent algorithms by deriving the vanilla second-order gradient method.

Objective: Our goal is to minimize a twice-differentiable function $f(\mathbf{x})$.

Second-Order Taylor Expansion: The second-order Taylor expansion of f around a point $\mathbf{x}_k$ is given by:

$$f(\mathbf{x}) \approx f(\mathbf{x}_k) + \nabla f(\mathbf{x}_k)^T(\mathbf{x} - \mathbf{x}_k) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_k)^T \mathbf{H}_f(\mathbf{x}_k)(\mathbf{x} - \mathbf{x}_k)$$

where:

- $\nabla f(\mathbf{x}_k)$ is the gradient of f at $\mathbf{x}_k$,
- $\mathbf{H}_f(\mathbf{x}_k)$ is the Hessian matrix of f at $\mathbf{x}_k$.

Minimization: To minimize $f(\mathbf{x})$, we set the gradient of the expansion to zero:

$$\nabla f(\mathbf{x}_k) + \mathbf{H}_f(\mathbf{x}_k)(\mathbf{x} - \mathbf{x}_k) = 0$$

Solving for x:

$$\mathbf{H}_f(\mathbf{x}_k)(\mathbf{x} - \mathbf{x}_k) = -\nabla f(\mathbf{x}_k)$$

Update Rule: Let $\Delta\mathbf{x} = \mathbf{x} - \mathbf{x}_k$, then:

$$\mathbf{H}_f(\mathbf{x}_k)\Delta\mathbf{x} = -\nabla f(\mathbf{x}_k)$$

Solving for $\Delta\mathbf{x}$, we get:

$$\Delta\mathbf{x} = -(\mathbf{H}_f(\mathbf{x}_k))^{-1}\nabla f(\mathbf{x}_k)$$

Iteration Update: Finally, the iteration update rule is given by:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta\mathbf{x} = \mathbf{x}_k - (\mathbf{H}_f(\mathbf{x}_k))^{-1}\nabla f(\mathbf{x}_k)$$

Convex Functions

Next, let's consider the notion of convex functions.

Definition 2.5.1 (Convex Function)**.** Let $X \subset \mathbb{R}^n$ be a convex set. Assuming the function is continuous, $f : X \to \mathbb{R}$ is convex if $f(tx_1 + (1-t)x_2)) < tf(x_1) + (1-t)f(x_2)$ for all $x_1, x_2 \in X$ and for all $t \in [0,1]$. If $\leq$ is replaced by strict inequality in the definition, we demand $x_1 \neq x_2$ and $t \in (0,1)$, then f is strictly convex. For a single-variable function, convexity means that if you draw a straight line connecting $f(x_1)$ and $f(x_2)$, the value of the function on the interval $[x_1, x_2]$ is always below the line.

Assuming f to be twice differentiable, then f is convex if and only if D_f, the domain of f, is a convex set and its Hessian is positive-definite for all $x \in D_f$. For a single-variable function, this reduces to $f''(x) > 0$.

Optimizing a neural network is generally not a convex problem, introducing local and global minima. Since gradient descent is deterministic, it often gets stuck in a local minimum for non-optimal initial values.

Stochastic Gradient Descent (SGD)

Moving on, let's delve into stochastic gradient descent, a method designed to address the challenges of optimizing non-convex neural network problems.

$$C(\beta) = \sum_{i=1}^{n} c_i(x_i, \beta) \tag{2.47}$$

This implies that the gradient can be computed as the sum of gradients:

$$\nabla_\beta C(\beta) = \sum_{i}^{n} \nabla_\beta c_i(x_i, \beta) \tag{2.48}$$

Stochasticity is introduced by computing the gradient on a subset of the data, known as minibatches. If there are n data points and the size of each minibatch is M, there will be $\frac{n}{M}$ minibatches, denoted as B_k, where we iterate over k up to M.

For each gradient step, we approximate the gradient with:

$$\sum_{i \in B_k}^{n} \nabla_\beta c_i(x_i, \beta) \tag{2.49}$$

In other words, we update the parameter values in the next step with:

$$\beta_{j+1} = \beta_j - \gamma_j \sum_{i}^{n} \nabla_\beta c_i(x_i, \beta) \tag{2.50}$$

The batches are shuffled randomly for each epoch, where one epoch means iterating over the whole dataset.

This approach has two key benefits. First, it introduces randomness, decreasing the chance of getting stuck in a local minimum. Second, if the size of the minibatches is small relative to the number of data points ($M < n$), the computation of the gradient is much cheaper since we sum over the k-th data points in the minibatch and not all n data points for each iteration.

An additional improvement is introducing momentum to the gradient process, creating a relationship between the $t+1$ value of the parameter being estimated and the $t-1$ value of the momentum vector.

$$v_t = \gamma v_{t-1} + \eta_t \nabla_\beta E(\theta_t) \tag{2.51}$$
$$\theta_{t+1} = \theta_t - v_t \tag{2.52}$$

In both stochastic gradient descent, with and without momentum, a schedule for tuning the learning rates must be specified. However, this presents challenges, as the learning rate is limited by the steepest direction, which can change depending on the current position in the landscape. To circumvent this problem, the ideal algorithm would keep track of curvature and take large steps in shallow, flat directions and small steps in steep, narrow directions. Second-order methods accomplish this by calculating or approximating the Hessian and normalizing the learning rate by the curvature. However, this is computationally expensive for extremely large models.

Adagrad

Adagrad, short for Adaptive Gradient Algorithm, is another gradient algorithm. Mathematically, Adagrad adapts the learning rate for each parameter (θ) based on the historical behavior of gradients, calculating individualized step size for optimal convergence. The algorithm's update scheme is defined by the following equations:

$$G_{t+1} = G_t + (\nabla J(\theta_t))^2 \tag{2.53}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_{t+1} + \epsilon}} \cdot \nabla J(\theta_t) \tag{2.54}$$

where $\nabla J(\theta_t)$ represents the gradient of the objective function at iteration t, η is the learning rate, ϵ is a small constant to avoid division by zero, G_t maintains the sum of squared gradients up to iteration t, and θ_t is the parameter vector.

Despite this adaptability, Adagrad is not without its challenges. The accumulation of squared gradients over time can lead to diminishing learning rates for parameters with infrequent updates, potentially resulting in slow convergence. Additionally, the algorithm's memory requirements for storing historical squared gradients might pose challenges in resource-constrained environments, especially when dealing with high-dimensional datasets. Understanding Adagrad involves appreciating its mathematical formulation and balancing its advantages in adaptability with the potential drawbacks related to accumulated information and memory usage.

RMSprop

RMSprop not only keeps track of the first moment of the gradient but also the second.

$$g_t = \nabla_\beta E(\beta) \tag{2.55}$$
$$s_t = \beta s_{t-1} + (1-\beta) g_t^2 \tag{2.56}$$
$$\beta_{t+1} = \beta_t - \eta_t \frac{g_t}{\sqrt{s_t + \epsilon}} \tag{2.57}$$

s_t tracks the second moment, and β represents how much the weighted average should focus on the previous iteration of s compared to the current second moment. The parameter is updated based on the ratio between the first moment and the square root of the second moment, reducing in directions where the norm of the gradient is consistently large. This significantly speeds up convergence by allowing us to use a larger learning rate for flat directions.

ADAM

$$g_t = \nabla_\beta E(\beta) \tag{2.58}$$
$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \tag{2.59}$$
$$s_t = \beta_2 s_{t-1} + (1-\beta_2) g_t^2 \tag{2.60}$$
$$m_t = \frac{m_t}{1-\beta_1^t} \tag{2.61}$$
$$s_t = \frac{s_t}{1-\beta_2^t} \tag{2.62}$$
$$\beta_{t+1} = \beta_t - \eta_t \frac{m_t}{\sqrt{s_t} + \epsilon} \tag{2.63}$$

ADAM is another algorithm that keeps track of the first and second moments of the gradient, using this information to adaptively change the learning rate for different parameters. The method is efficient when working with large problems involving lots of data and/or parameters.The tuning parameters β follow the same logic as RMSprop.

The choice of algorithm often depends on the specific characteristics of the problem at hand, and experimentation is key to finding the most suitable approach.

2.6 Algorithm

psudocode FFN

Algorithm 1 shows a pseudo-code of Feed Forward Neural network.

psudocode logistic regression

The same setup as Algorithm 2 can be used for multinominal logistic regression replacing the sigmoid with the softmax and choosing a pivot class.

Algorithm 1 Feedforward Neural Network

```
function FeedforwardNeuralNetwork(InputFeatures)
    # Initialize network architecture
    InitializeWeightsAndBiasesForAllLayers

    # Forward pass
    InputLayerOutput ← InputFeatures
    for each hidden layer do
        WeightedSum ← DotProduct(InputLayerOutput, Weights) + Biases
        Activation ← ApplyActivationFunction(WeightedSum)
        InputLayerOutput ← Activation
    end for

    # Output layer
    OutputLayer ← DotProduct(InputLayerOutput, OutputWeights) + OutputBiases
    PredictedOutput ← ApplyOutputActivationFunction(OutputLayer)

    return PredictedOutput
end function
function TrainFeedforwardNeuralNetwork(TrainingData, Labels, LearningRate, Epochs)
    for each epoch in range(Epochs) do
        for each data point, label in zip(TrainingData, Labels) do
            # Forward pass
            PredictedOutput ← FeedforwardNeuralNetwork(data_point)

            # Compute loss (e.g., mean squared error)
            Loss ← ComputeLoss(PredictedOutput, label)

            # Backward pass (Backpropagation)
            ComputeGradientsOfLoss()

            # Update weights and biases using an optimization algorithm
            UpdateParametersWithGradientsAndLearningRate()
        end for
    end for

    return TrainedFeedforwardNeuralNetwork
end function
```

Algorithm 2 Logistic Regression with Gradient Descent

```
function LogisticRegression(Features, Labels, LearningRate, Epochs)
    # Initialize weights and biases
    Weights ← InitializeWeights()

    for each epoch in range(Epochs) do
        Predictions ← Sigmoid(DotProduct(Features, Weights) )
        Loss ← ComputeBinaryCrossEntropy(Predictions, Labels)
        Gradients ← ComputeGradients(Features, Labels, Predictions)

        # Update weights and bias using gradient descent
        Weights ← UpdateWeights(Weights, Gradients, LearningRate)

    end for

    return Weights
end function
function Predict(NewData, Weights, )
    Predictions ← Sigmoid(DotProduct(NewData, Weights) )
    return Predictions
end function
```

Simulation study

In this simulation study, we explore various gradient methods and predictive models a classification scenario and a prediction scenario. Our task will have four distinct segments, each shedding light on the performance and efficacy of different algorithms in specific contexts.

The initial phase centers around a comparison of various gradient methods theoretically defined above. These methods will be put to the test on a straightforward first-order response $4+3*x+np.random.randn(n,1)$, enabling us to discern their relative strengths and weaknesses in handling a fundamental regression scenario.

Moving forward, our focus shifts to predictive modeling with a particular emphasis on feedforward neural networks (FFN). We will go into comparing the prediction mean squared error (MSE) obtained from FFN, a Keras implementation of FFN, and the outcomes derived from ordinary linear regression applied to the $SRTM\_data\_Norway\_1.tif$ dataset,which is an image from a particular location in Norway. This comparative analysis aims to elucidate the nuances in predictive accuracy and computational efficiency among these different modeling approaches.

Transitioning into the third facet of our study, we pivot towards classification tasks. Leveraging the lecture notes' FFN and its Keras counterpart, we assess their performance on a dataset sourced from scikit-learn. Specifically, we use the breast cancer dataset, measuring the accuracy of our models in classifying instances. This exploration into classification offers insights into the adaptability and robustness of the FFN from [Hjo23] compared to our Keras model.

Lastly, our study extends to a comparison between FFNN (Feedforward Neural Network) from the lecture notes and logistic regression in the context of a classification task on the cancer dataset. This final leg of our exploration aims to unravel the nuanced distinctions between neural network approaches and classical statistical models in tackling real-world datasets.

Gradient method result example using analytical gradient vs autograd

	Method	t_1	t_0	Batch Size	Epochs	Beta
0	SGD_linear	50	5	5	60	[[3.9633033118263667], [2.967630028118333]]
2	SGD_linear	50	5	5	200	[[4.017710805421551], [2.992404373035771]]
4	SGD_linear	50	5	10	60	[[3.979301179372519], [3.0202331054570206]]
6	SGD_linear	50	5	10	200	[[3.9857920037135344], [3.023026354309524]]
8	SGD_linear	50	10	5	60	[[4.013787691472411], [3.029523042386755]]
10	SGD_linear	50	10	5	200	[[3.9908947797101911], [3.0399703250408483]]
12	SGD_linear	50	10	10	60	[[3.906175487531253], [2.9481374032515046]]
14	SGD_linear	50	10	10	200	[[3.9482095808607434], [2.9978913569813206]]
16	SGD_linear	70	5	5	60	[[3.92255811939144961, [3.0125171774953925]]
18	SGD_linear	70	5	5	200	[[3.9304844168643647], [3.0415795887905]]
20	SGD_linear	70	5	10	60	[[3.9242237552088057], [2.898282357531859]]
22	SGD_linear	70	5	10	200	[[3.947695668550655], [2.9706518253395844]]
24	SGD_linear	70	10	5	60	[[3.945536474577046], [3.047732909847983]]
26	SGD_linear	70	10	5	200	[[4.081992153046897], [3.013530375722556]]
28	SGD_linear	70	10	10	60	[[3.970290802930665], [3.018665593675248]]
30	SGD_linear	70	10	10	200	[[3.952240972415152], [2.988099315373878]]

Figure 3.1: Sgd linear adagrad for regularization t1=50,70 and t0=5,10

	Method	t_1	t_0	Batch Size	Epochs	Beta
0	SGD_linear	50	5	5	60	[[3.9633033118263667], [2.967630028118333]]
2	SGD_linear	50	5	5	200	[[4.017710805421551], [2.992404373035771]]
4	SGD_linear	50	5	10	60	[[3.979301179372519], [3.0202331054570206]]
6	SGD_linear	50	5	10	200	[[3.9857920037135344], [3.023026354309524]]
8	SGD_linear	50	10	5	60	[[4.013787691472411], [3.029523042386755]]
10	SGD_linear	50	10	5	200	[[3.9908947797101911], [3.0399703250408483]]
12	SGD_linear	50	10	10	60	[[3.906175487531253], [2.9481374032515046]]
14	SGD_linear	50	10	10	200	[[3.9482095808607434], [2.9978913569813206]]
16	SGD_linear	70	5	5	60	[[3.92255811939144961, [3.0125171774953925]]
18	SGD_linear	70	5	5	200	[[3.9304844168643647], [3.0415795887905]]
20	SGD_linear	70	5	10	60	[[3.9242237552088057], [2.898282357531859]]
22	SGD_linear	70	5	10	200	[[3.947695668550655], [2.9706518253395844]]
24	SGD_linear	70	10	5	60	[[3.945536474577046], [3.047732909847983]]
26	SGD_linear	70	10	5	200	[[4.081992153046897], [3.013530375722556]]
28	SGD_linear	70	10	10	60	[[3.970290802930665], [3.018665593675248]]
30	SGD_linear	70	10	10	200	[[3.952240972415152], [2.988099315373878]]

Figure 3.2: Sgd linear analytical for regularization t1=50,70 and t0=5,10

Rest of the results could be found on the following link: https://github.com/gituser1234566/some_coding/blob/main/project2fys-stk4155/code/gradient_method_results_ridge_linear.ipynb

FFNN vs Keras FNN vs OLS

	alpha	degree	mse_train	mse_test	5cv_train_mse	5cv_test_mse	10cv_train_mse	10cv_test_mse
0	0.00001	1	0.055752	0.055752	0.063413	0.063413	0.057972	0.057972
1	0.00001	2	0.107061	0.107061	0.070858	0.070858	0.071358	0.071358
2	0.00001	3	0.120939	0.120939	0.057358	0.057358	0.057747	0.057747
3	0.00001	4	0.126853	0.126853	0.084650	0.084650	0.084616	0.084616
4	0.00001	5	0.155479	0.155479	0.064370	0.064370	0.063279	0.063279
5	0.00002	1	0.055752	0.055752	0.063413	0.063413	0.057972	0.057972
6	0.00002	2	0.107061	0.107061	0.070858	0.070858	0.071358	0.071358
7	0.00002	3	0.120939	0.120939	0.057358	0.057358	0.057747	0.057747
8	0.00002	4	0.126853	0.126853	0.084650	0.084650	0.084616	0.084616
9	0.00002	5	0.155479	0.155479	0.064370	0.064370	0.063279	0.063279
10	0.00004	1	0.055752	0.055752	0.063413	0.063413	0.057972	0.057972
11	0.00004	2	0.107061	0.107061	0.070858	0.070858	0.071358	0.071358
12	0.00004	3	0.120939	0.120939	0.057358	0.057358	0.057747	0.057747
13	0.00004	4	0.126853	0.126853	0.084650	0.084650	0.084616	0.084616
14	0.00004	5	0.155479	0.155479	0.064370	0.064370	0.063279	0.063279
15	0.00050	1	0.055752	0.055752	0.063413	0.063413	0.057972	0.057972
16	0.00050	2	0.107061	0.107061	0.070858	0.070858	0.071358	0.071358
17	0.00050	3	0.120939	0.120939	0.057358	0.057358	0.057747	0.057747
18	0.00050	4	0.126853	0.126853	0.084650	0.084650	0.084616	0.084616
19	0.00050	5	0.155479	0.155479	0.064370	0.064370	0.063279	0.063279

Figure 3.3: Prediction ffn for regularization 0.001,0.005,0.01,0.05,0.1

	alpha	degree	mse_train	mse_test	5cv_train_mse	5cv_test_mse	10cv_train_mse	10cv_test_mse
0	0.00001	1	0.045072	0.080021	0.055305	0.055137	0.054382	0.055750
1	0.00001	2	0.037948	0.074132	0.051650	0.059629	0.047613	0.053943
2	0.00001	3	0.029908	0.082917	0.047916	0.051606	0.047991	0.055472
3	0.00001	4	0.031027	0.086768	0.046201	0.051190	0.049384	0.055792
4	0.00001	5	0.026189	0.101850	0.047523	0.055198	0.046419	0.052910
5	0.00002	1	0.040707	0.080918	0.053272	0.057679	0.052056	0.056665
6	0.00002	2	0.032397	0.087437	0.048543	0.049785	0.048972	0.056176
7	0.00002	3	0.035326	0.083067	0.047838	0.052717	0.047513	0.054694
8	0.00002	4	0.026130	0.109350	0.046758	0.052868	0.047632	0.051881
9	0.00002	5	0.028162	0.100511	0.046614	0.049350	0.049765	0.057456
10	0.00004	1	0.041158	0.075914	0.052154	0.055914	0.055169	0.057774
11	0.00004	2	0.033059	0.084251	0.048381	0.055180	0.049560	0.052794
12	0.00004	3	0.031432	0.083422	0.048869	0.055832	0.046898	0.054865
13	0.00004	4	0.028921	0.097853	0.047599	0.051966	0.050163	0.056248
14	0.00004	5	0.028966	0.104548	0.047868	0.056046	0.047679	0.054189
15	0.00050	1	0.038473	0.074085	0.054845	0.060899	0.053467	0.059821
16	0.00050	2	0.035732	0.074231	0.048538	0.053679	0.047393	0.051977
17	0.00050	3	0.029705	0.083565	0.047824	0.054047	0.048419	0.052624
18	0.00050	4	0.029048	0.087273	0.048674	0.056311	0.048254	0.055087
19	0.00050	5	0.027319	0.104056	0.045577	0.054193	0.047173	0.054001

Figure 3.4: Keras prediction result for regularization 0.0001,0.0002,0.0004,0.0005

	alpha	degree	mse_train	mse_test	bootstrap_train_mse	bootstrap_test_mse	5cv_train_mse	5cv_test_mse	10cv_train_mse	10cv_test_mse
0	0	1	0.050754	0.096376	0.050079	0.098209	0.059221	0.061775	0.059306	0.062387
1	0	2	0.040425	0.076516	0.038880	0.080098	0.045827	0.051558	0.046096	0.050906
2	0	3	0.035026	0.082119	0.032583	0.092208	0.040887	0.050714	0.041289	0.051752
3	0	4	0.024931	0.064892	0.022952	0.076739	0.030146	0.047456	0.030619	0.049840
4	0	5	0.022438	0.104061	0.019383	0.187752	0.026528	0.073444	0.027416	0.079353

Figure 3.5: Ols result prediction for regularization 0.0001,0.0002,0.0004,0.0005

Comparison between Sigmoid,Relu,LeakyRelu

	alpha	degree	mse_train	mse_test
0	0.00001	1	0.034506	0.034506
1	0.00001	2	0.031164	0.031164
2	0.00001	3	0.090950	0.090950
3	0.00001	4	0.063844	0.063844
4	0.00001	5	0.151807	0.151807
5	0.00002	1	0.034506	0.034506
6	0.00002	2	0.031164	0.031164
7	0.00002	3	0.090950	0.090950
8	0.00002	4	0.063844	0.063844
9	0.00002	5	0.151807	0.151807
10	0.00004	1	0.034506	0.034506
11	0.00004	2	0.031164	0.031164
12	0.00004	3	0.090950	0.090950
13	0.00004	4	0.063844	0.063844
14	0.00004	5	0.151807	0.151807
15	0.00050	1	0.034506	0.034506
16	0.00050	2	0.031164	0.031164
17	0.00050	3	0.090950	0.090950
18	0.00050	4	0.063844	0.063844
19	0.00050	5	0.151807	0.151807

Figure 3.6: Sigmoid activation for regularization 0.0001,0.0002,0.0004,0.0005

	alpha	degree	mse_train	mse_test
0	0	1	1.490930e+07	1.490930e+07
1	0	2	2.452181e+07	2.452181e+07
2	0	3	1.411978e+07	1.411978e+07
3	0	4	4.600550e+08	4.600550e+08
4	0	5	1.824034e+09	1.824034e+09

Figure 3.7: Relu activation Figure showing for regularization 0.0001,0.0002,0.0004,0.0005

	alpha	degree	mse_train	mse_test
0	0.00001	1	1.483754e+07	1.483754e+07
1	0.00001	2	2.459653e+07	2.459653e+07
2	0.00001	3	1.422732e+07	1.422732e+07
3	0.00001	4	4.584313e+08	4.584313e+08
4	0.00001	5	1.822756e+09	1.822756e+09
5	0.00002	1	1.483754e+07	1.483754e+07
6	0.00002	2	2.459653e+07	2.459653e+07
7	0.00002	3	1.422732e+07	1.422732e+07
8	0.00002	4	4.584313e+08	4.584313e+08
9	0.00002	5	1.822756e+09	1.822756e+09
10	0.00004	1	1.483754e+07	1.483754e+07
11	0.00004	2	2.459653e+07	2.459653e+07
12	0.00004	3	1.422732e+07	1.422732e+07
13	0.00004	4	4.584313e+08	4.584313e+08
14	0.00004	5	1.822756e+09	1.822756e+09
15	0.00050	1	1.483754e+07	1.483754e+07
16	0.00050	2	2.459653e+07	2.459653e+07
17	0.00050	3	1.422732e+07	1.422732e+07
18	0.00050	4	4.584313e+08	4.584313e+08
19	0.00050	5	1.822756e+09	1.822756e+09

Figure 3.8: Lrelu activation Figure for regularization 0.0001,0.0002,0.0004,0.0005

Effect of using L2 norm on costfunction for classification

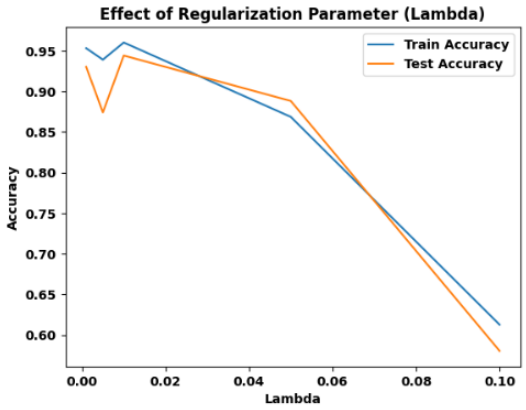

Figure 3.9: FFNN result classification for regularization 0.001,0.005,0.01,0.05,0.1

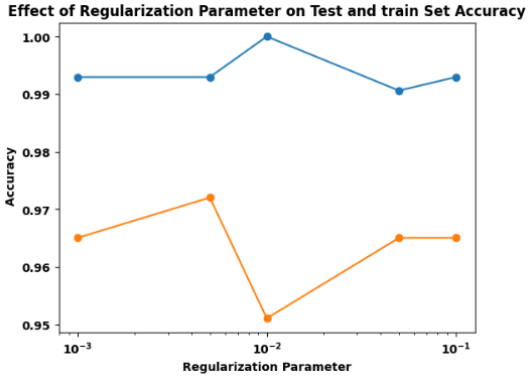

Figure 3.10: Keras regularization result classification for regularization 0.001,0.005,0.01,0.05,0.1

Accuracy comparison between FFNN ,implemented Logistic regression classification and skitlearn Logistic regression

	Lambda	Iteration	Train Accuracy	Test Accuracy
0	0.1	5	0.974178	0.937063
1	0.1	5	0.969484	0.930070
2	0.1	5	0.953052	0.895105
3	0.1	5	0.887324	0.818182
4	0.1	5	0.819249	0.818182

Figure 3.11: accuracy FFNN classification for regularization 0.001,0.005,0.01,0.05,0.1

	Lambda	Iteration	Train Accuracy	Test Accuracy
0	0.001	1	0.981221	0.958042
1	0.005	2	0.981221	0.958042
2	0.010	3	0.981221	0.958042
3	0.050	4	0.981221	0.958042
4	0.100	5	0.981221	0.958042

Figure 3.12: accuracy Logsitic classification for regularization 0.001,0.005,0.01,0.05,0.1

	Lambda	Iteration	Train Accuracy	Test Accuracy
0	0.001	1	0.957973	0.944056
1	0.005	2	0.960354	0.937063
2	0.010	3	0.962680	0.937063
3	0.050	4	0.967386	0.944056
4	0.100	5	0.969712	0.944056

Figure 3.13: Skitlearnaccuracy Figure for regularization 0.001,0.005,0.01,0.05,0.1

Confusion matrix comparison between FFNN ,implemented Logistic regression classification and skitlearn Logistic regression

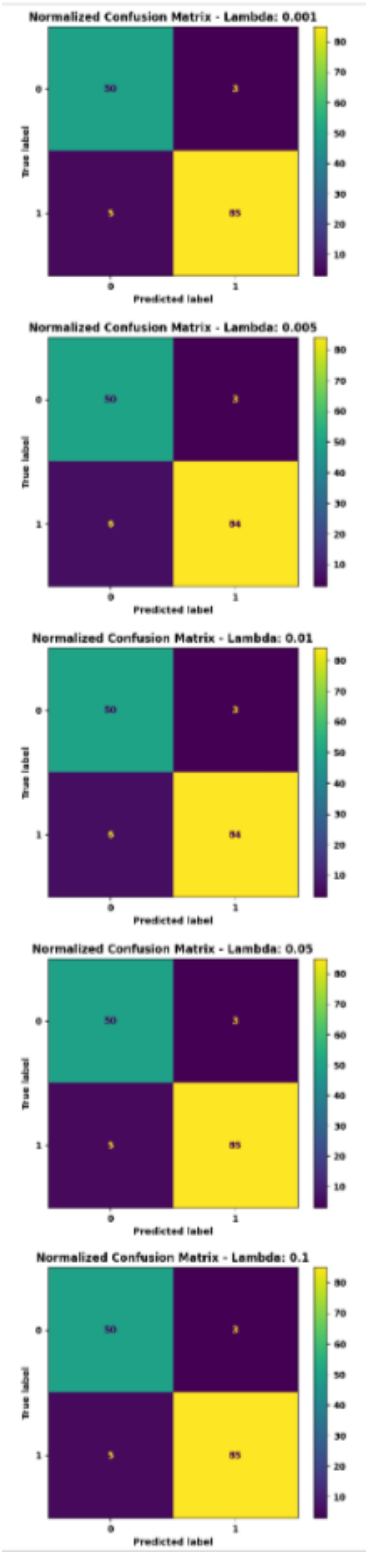

Figure 3.14: Skitlearn confusion matrix result Figure for regularization 0.001,0.005,0.01,0.05,0.1

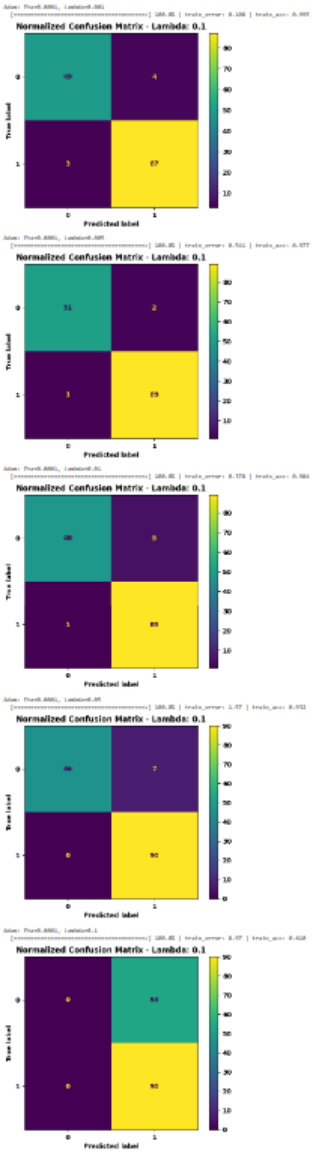

Figure 3.15: Figure showing Confusion matrix ffnn for regularization 0.001,0.005,0.01,0.05,0.1

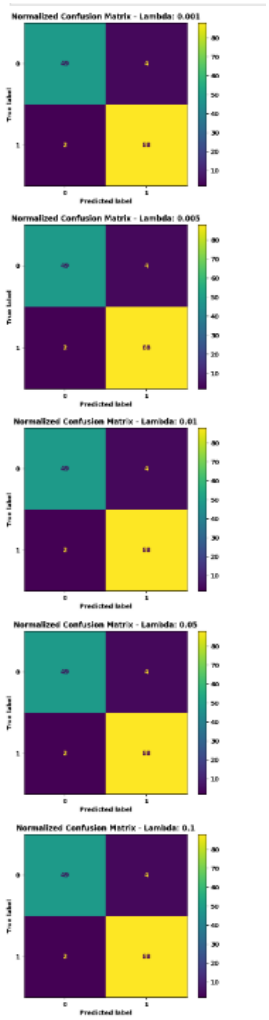

Figure 3.16: Figure showing Confusion matrix logistic for regularization 0.001,0.005,0.01,0.05,0.1

Discussion

All methods based on SGD(Stochastic gradient decent) with and without momentum and regularization perform somewhat equally well.Figure 3.1 and Figure 3.2 shows a comparison between using analytical gradient and Autograd,and the result match closely.For ordinary SGD we apply time decay rate since SGD those not promise convergence.Adagrad method and the linear method based on plain gradient decent(GD) show lack of convergence signalling that more iteration should be considered for convergence to be reached. Interesting thing to note that ridge based on GD converges as well,indicating that introducing regularization makes it easier for the method to converge.This makes sense for the reasons why regularization is introduced,one of the main being flattening out the regions where the method can get stucked (local minimums).Adagrad seem to be really slow which can be expected since its construction makes it conservative whenever the gradient is steep. To see the advantages of Adam an RMS a more complex polynomial should have been considered which are based on having a memory of previous gradient outputs by running a running average and addition incorporating more information about the gradient to determine the learning rate through keeping track of the second moment of the gradient as well.Also to see the advantage of momentum a more complex polynomial could have been used to see: "SGD momentum helps the gradient descent algorithm gain speed in directions with persistent but small gradients even in the presence of stochasticity, while suppressing oscillations in high-curvature directions". [HTF01]

Going further, the FFNN for prediction task did not improve its MSE which can be seen in Figure 3.3, 3.4 and 3.5. Figure 3.3 is for the custom FFNN from the lecture notes [Hjo23], Figure 3.4 is the result of keras model and 3.5 is the result from linear regression(LR).What can be seen however is the custom FFNN do worse than bought the keras model and LR,while the keras model do almost equally well,indicating that more epochs might have been needed for custom FFNN.This was however computer intensive, 1000 epochs where used.The activation function of the FFNN was the sigmoid and was initialized with *np.random.rand*.The bias should be initialized to be none zero but not large,so that the contribution of each hidden node is included and even if the gradient update yields an update that makes the weight close to zero so that the node stay alive.Since this is a prediction task we would use the identity function for the final layer. Issues that Glorot and Bengio mentioned in section

2.1 where not observed for the datasets used.The regularization had a positive effect for the keras model and the ols.

Next we compared the FFNN with activation function sigmoid RELU and LRELU.The results for RELU and LRELU (Figure 3.6,3.7,3.8) indicated exploding gradient.To fix this problem one might have tried to batch normalize to mitigate this problem, where we normalize the input of each activation function of each layer before the activation function is applied, or tried different initial values for the weights.

Moving on to the classification task we saw that logisitc regression did well compared to FFNN and the keras model (Figure 3.10,3.9).A comparison between FFNN and the keras model.The regularization parameter did not have a big effect for the accuracy(Figure 3.11,3.12,3.13). and the confusion matrices(Figure 3.14,3.15,3.16),except for the FFNN where regularization made the result worse.

For the datasets used the simpler models,linear regression and logistic regression performed equally well considering if optimal tuning was done on custom FFNN,indicating that neural networks should be used in scenarios where other methods are to simple to explain the patterns in the dataset.

Conclusion

In this study, we explored the application of Feedforward Neural Networks (FFNN) in classification and prediction tasks. We commenced with a theoretical overview of their setup, encompassing various components, before delving into the data fitting process through both the forward pass and the crucial backward pass. Our attention then turned to popular activation functions. Additionally, we investigated different gradient descent methods, pivotal for the success of the backward pass. Subsequently, we presented simulation results and discussed their implications.

This study serves as a procedural high-level overview. Future investigations should encompass diverse datasets, with a particular emphasis on parameter tuning for a more detailed and in-depth analysis. For example, Figures 3.7 and 3.8 results might benefit from tuning when the ReLU and LeakyReLU functions are unsatisfactory, highlighting the intricacies of fitting a Neural Network to data. While certain activation functions may work seamlessly across various datasets, the study emphasizes the critical role of weight initialization, stability, and the ability of different gradient methods to converge to an optimal solution, all while considering their tuning parameters.As demonstrated, even for lower degree polynomials, some cases yielded suboptimal results, indicating the challenges present. Although one might argue that given enough time, the methods could eventually converge, it underscores the intricacies involved. The study also reveals that for tasks with relatively smaller datasets, traditional methods such as linear regression and logistic regression perform equally well.

Interestingly, the Keras model employed for classification and prediction did not exhibit these behaviors, suggesting that further research is needed to unravel the complexities, aiming for a more nuanced understanding. In general, the neural network architecture sourced from [Hjo23] should be subjected to more epochs in the classification task. Future work could prioritize addressing these challenges.

Bibliography

[GBC16] Goodfellow, I., Bengio, Y. and Courville, A. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[Hjo23] Hjorth-Jensen, M. *Lecture Notes Fys-stk 4155*. https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html.. 2023.

[HTF01] Hastie, T., Tibshirani, R. and Friedman, J. *The Elements of Statistical Learning*. Springer Series in Statistics. New York, NY, USA: Springer New York Inc., 2001.