Classification and Regression:
From Linear and Logistic Regression to Neural Networks

Alessio Canclini, Sophus Gullbekk, Jakob Wiig Ryther

November 5, 2024

Abstract

This study explores machine learning techniques for regression and classification, focusing on a comparative performance analysis between traditional methods — Ordinary Least Squares (OLS), Ridge Regression, and Logistic Regression — and a more complex Feed Forward Neural Network (FFNN). We investigate various optimization algorithms, such as Gradient Descent (GD), Stochastic Gradient Descent (SGD), and adaptive schedulers like Adam, to enhance model training. For regression tasks, we approximate the Franke function, achieving a minimum test Mean Squared Error (MSE) of 5.9×10^{-4} using a FFNN with four layers and 64 nodes, optimized with the Adam scheduler and a learning rate of 0.001. In comparison, OLS and Ridge regression with analytical solutions produced an MSE of 3.7×10^{-4}. Classification performance on the Wisconsin Breast Cancer dataset was evaluated using accuracy metrics. The best FFNN model, with two hidden layers and eight nodes per layer, achieved an accuracy of 0.99, seemingly outperforming our Logistic Regression implementation whit an accuracy of 0.98. The false negative rate of Logistic regression was 0.02 compared to 0 for the FFNN. Running both models 300 times for different random seeds shows that Logistic regression on average achieves an accuracy close to 1, whilst the FFNN accuracy distribution is centered around 0.98.

Contents

1 Introduction

The field of Machine Learning (ML) has grown into a rapidly evolving and vast landscape of methods. Although seemingly new, the field has a long history. The term *machine learning* was coined as early as 1959[1], though its roots arguably reach back decades earlier, when we (humans) began to study cognition. Thus, modern ML builds on foundational works in neuroscience such as early mathematical models of neural networks by McCulloch and Pitts [2]. This year the Nobel Prize in physics was awarded to John J. Hopfield and Geoffrey E. Hinton for their pioneering work in the field of ML [3] which signifies its impact on both research and industries.

Two problems commonly addressed with ML are regression and classification. In this study we implement a Feed Forward Neural Network (FFNN) capable of solving both problems and present a comparative analysis of its performance compared to traditional methods. The traditional methods implemented are Ordinary Least Squares (OLS) and Ridge regression for the regression problem and Logistic regression for classification. To compare the different regression methods we approximate the Franke function as done in Gullbekk, Canclini, and Ryther [4]. The classification accuracy is assessed on the Wisconsin Breast Cancer dataset [5]. The latter is a data set of images representing various features of tumors with the goal being a cancerous or benign classification of each tumor.

The overarching question is whether traditional methods will be outperformed by the more complex FFNN. As already alluded to, the FFNN seeks to mimic the architecture of neurons in the brain. This is achieved by developing an interconnected network of nodes which each have adjustable weights. In fact, a universal approximation theorem [6] tells us that a FFNN with non-linear activations can approximate any continuous function, albeit at the cost of needing a large set of trainable parameters. The optimization of these parameters is a major bottle neck for Neural Network (NN) training — motivating the development of a variety of optimization methods.

Optimization methdods, along with the theory of a FFNN and a variety of schedulers and activation functions are presented in Section 2. This is followed by an overview of the traditional regression and classification methods as well as general implementation of the FFNN in Section 3. Results of the various optimization methods as well as the classification and regression approaches are presented and discussed in Section 4. Finally, Section 5 offers some concluding remarks and suggestions for future improvements. All code for the different methods, along with notebooks to reproduce all results can be found in the GitHub repository[1].

2 Theory

2.1 Optimization

In the majority of machine learning problems, we start with a dataset $\mathbf{X}$ and a model $g(\boldsymbol{\theta})$ that uses the parameters $\boldsymbol{\theta}$ to explain the values in $\mathbf{X}$. To measure how well the model performs we use a cost function $C(\mathbf{X}, \boldsymbol{\theta})$. The goal of optimization is to find the parameter values $\boldsymbol{\theta}$ that minimize the cost function and thus results in the best possible fit of our data. For simpler models like OLS and Ridge regression we can find the optimal values for $\boldsymbol{\theta}$ analytically, albeit at a potentially great computational cost. For more complex models, it might not be possible to calculate the analytical solution. Instead, we have to rely on iterative optimization methods such as Gradient Descent (GD) and Stochastic Gradient Descent (SGD) that use the gradient of the cost function $\nabla C(\mathbf{X}, \boldsymbol{\theta})$ to move gradually towards a better model. It can be shown that if we choose

[1] `github.uio.no/sophusbg/FYS-STK4155/tree/main/project2`

a cost function that is convex, any local minimum we find will also be a global minimum. In most deep learning applications, it is not feasible to have convex cost functions and a central problem for our optimization methods becomes how to avoid getting stuck in bad local minima. [7].

2.1.1 Gradient Descent

Standard GD is the most basic implementation which all subsequent gradient-based methods will build on. The main idea revolves around using the gradient of the cost function to find its minimum, i.e. where the gradient is equal to zero. This is done by guessing an initial $\boldsymbol{\theta}_0$ value and then moving a small step in the opposite direction of the gradient as shown in Figure 1. Using the Newton-Raphson method [8], this is computed iteratively as,

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}) \text{ for } t = 0, 1, ... \tag{1}$$

Where, η is the learning rate, a positive parameter which determines the size of each step, and $\nabla_{\boldsymbol{\theta}}$ refers to the gradient with respect to $\boldsymbol{\theta}$. This process is repeated for a set number of iterations or until a certain minimum threshold value of $||\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta})||$ is reached [7]. Even if one could compute an analytical solution, a large motivation for GD is it's simplicity and reduced computational cost compared to matrix inversions.

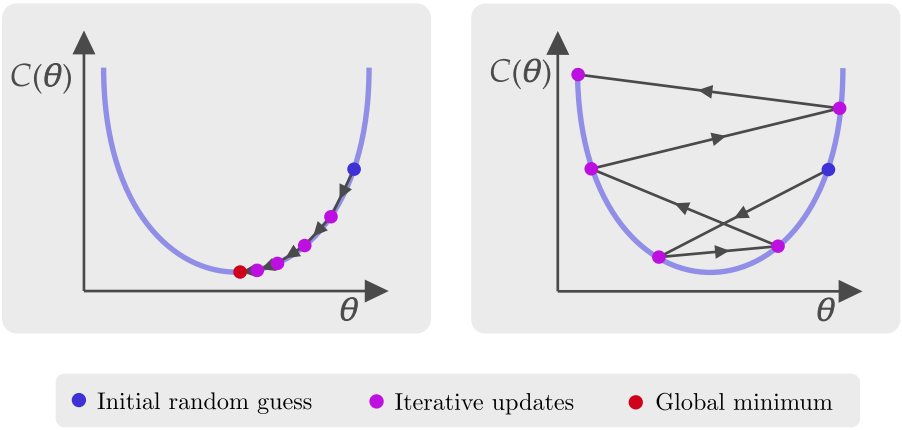


Figure 1: *A depiction of how the iterative process used in GD. The left figure shows a typical path towards a global minimum, while the right figure highlights what happens when the step size is too large.*

2.1.2 Stochastic Gradient Descent

Standard GD suffers from a few important pitfalls. Notably, it is not able to easily escape bad local minima and it is very computationally expensive to sum over all the data points each time we calculate the gradient. The SGD algorithm tries to remedy this by only calculating the gradient with respect to a randomly selected subset of the data points called a mini batch each iteration. This is not only more computationally efficient, but has the added benefit of adding randomness in the sense that local minima might be different between different batches. Thus SGD often becomes more likely to escape local minima.

Lets say we divide the data $\mathbf{X}$ in k mini batches $\mathbf{B}_1, \mathbf{B}_2, \ldots, \mathbf{B}_k \subseteq \mathbf{X}$. Then for each iteration, we would select a given mini batch $\mathbf{B}_i$ and calculate the gradient using its samples. The computation of $\boldsymbol{\theta}$ then becomes

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \sum_{\mathbf{x} \in B_i} \nabla_{\boldsymbol{\theta}} C_i(\mathbf{x}, \boldsymbol{\theta}). \tag{2}$$

After we iterate over all minibatches, completing one pass through the entire dataset, we call this an epoch. It is important to consider the trade off between different mini batch sizes. By using smaller batches, SGD reduces the computational load and, due to its stochastic nature, is less likely to get stuck in local minima. However, a larger mini batch size would make the updates more stable [9].

2.2 Schedulers

Although the algorithm for SGD is an improvement over GD it still suffers from many of the same problems. Two of the big problems is the sensitivity to the chosen learning rate η and that it treats all directions in the parameter space uniformly [9]. The schedulers try to address these problems by changing the learning rate dynamically during training. We will investigate two categories of schedulers: Momentum-based methods and adaptive methods. Momentum-based methods incorporate velocity into the update equations, while adaptive methods leverage the magnitude of gradients to adjust the learning rate during training. Adaptive methods can treat different directions of the parameter space differently by tracking the second moment of the gradient. In this section, we will examine one momentum-based method and three adaptive methods: AdaGrad, RMSprop, and Adam.

2.2.1 Momentum

Both GD and SGD can be accelerated by the method of momentum [10]. This adds an intermediate step to the computation of updating $\boldsymbol{\theta}$ that incorporates the velocity from previous updates

$$\boldsymbol{v}_t = \gamma \boldsymbol{v}_{t-1} + \eta \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}_t) \quad \longleftarrow \text{calculate the velocity,} \tag{3}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \boldsymbol{v}_t. \tag{4}$$

Here the hyperparameter $\gamma \in [0, 1)$ dictates how much the previous gradients i.e. momentum (or memory) affect the current direction. The momentum helps escaping regions in the cost landscape with low gradients while at the same time reducing oscillations where the gradients are high.

2.2.2 AdaGrad

AdaGrad [11] stands for Adaptive Gradient and it will adapt the learning rate for all the different model parameters individually. The idea behind the algorithm is to have greater progress in more gently sloped regions of the parameter space [12, chapter 8]. It achieves this because the parameters with the largest gradients get a more decreased learning rate than other parameters with smaller gradients. The steps for updating $\boldsymbol{\theta}$ using AdaGrad for a given mini-batch B_k are given by [12]:

$$\begin{aligned}
\mathbf{g} &= \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_{i \in B_k} C(\mathbf{x}_i, \theta) && \longleftarrow \text{calculate the gradient} \\
\mathbf{r} &= \mathbf{r} + \mathbf{g} \odot \mathbf{g} && \longleftarrow \text{accumulate the square gradient} \\
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \frac{\eta}{\varepsilon + \sqrt{\mathbf{r}}} \odot \mathbf{g}
\end{aligned}$$

Here, ε is a small constant typically set to 10^{-6}. Full pseudocode for the algorithm is given in Appendix A.2, Algorithm 1.

2.2.3 RMSprop

RMSprop was introduced in a lecture by the Nobel laureate Geoffrey Hinton [13]. It changes the gradient accumulation of AdaGrad into an exponential decay, and thus aims to keep the learning rate adaptable over time. The decay is decided by the new decay parameter ρ. The steps are similar to AdaGrad and is given by [12]:

$$\begin{aligned}
\mathbf{g} &= \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_{i \in B_k} C(\mathbf{x}_i, \theta) && \longleftarrow \text{calculate the gradient} \\
\mathbf{r} &= \mathbf{r} + \rho\mathbf{r} + (1-\rho)\mathbf{g} \odot \mathbf{g} && \longleftarrow \text{accumulate the square gradient} \\
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \frac{\eta}{\sqrt{\varepsilon + \mathbf{r}}} \odot \mathbf{g}
\end{aligned}$$

A more complete pseudocode can be found in Appendix A.2, Algorithm 2.

2.2.4 Adam

Adaptive Moment Estimation (Adam) was first introduced by Kingma and Ba in 2014 [14], designed to combine the advantages of the previous two algorithms (Sections 2.2.2 and 2.2.3). It adapts the learning rates from estimates of first and second moments of the gradients. Adam incorporates momentum directly as an estimate of the first order moment (the mean) of the gradients, This is achieved through exponential moving averages of the gradients, which helps in smoothing and accelerating the optimization process. In addition to the first moment, Adam maintains an exponentially decaying average of past squared gradients, similar to RMSprop, to estimate the second moment. However, a key distinction is that Adam includes bias correction terms for both moment estimates. Since these moment estimates are initialized at zero, they are biased towards zero, especially in the first couple of time steps. The bias correction compensates for this, providing more accurate estimates. The algorithm for Adam is given by [12]:

$$\begin{aligned}
\mathbf{g} &= \frac{1}{m}\nabla_{\theta} \sum_{B_k} C(\mathbf{x}_i, \boldsymbol{\theta}) && \longleftarrow \text{calculate the gradient} \\
\mathbf{s} &= \rho_1\mathbf{s} + (1-\rho_1)\mathbf{g} && \longleftarrow \text{update first order moment estimate} \\
\mathbf{r} &= \rho_2\mathbf{r} + (1-\rho_2)\mathbf{g} \odot \mathbf{g} && \longleftarrow \text{update second order moment estimate} \\
\hat{\mathbf{s}} &= \frac{\mathbf{s}}{1-\rho_1^t} && \longleftarrow \text{correct bias in first moment} \\
\hat{\mathbf{r}} &= \frac{\mathbf{r}}{1-\rho_2^t} && \longleftarrow \text{correct bias in second moment} \\
\theta_{t+1} &= \boldsymbol{\theta}_t - \varepsilon\frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}} + \delta}
\end{aligned}$$

A more complete pseudocode with explanaitions of the different parameters is included in Appendix A.2, Algorithm 3.

2.3 Feed Forward Neural Network

A FFNN is a NN where none of the connections between different nodes form a cycle, meaning that data is propagated or fed forward through the network. A FFNN consists of three types of layers: the input layer, the hidden layers, and an output layer. In addition, we will assume that our FFNN is fully connected, meaning that each neuron receives a

weighted sum from all the neurons in the previous layer. It has been shown that a FFNN with just a single hidden layer with a sigmoid activation function can approximate any continuous function [6]. This is called the universal approximation theorem, and is a testament to how useful FFNN can be to solve a large variety of different problems. A diagram of the architecture of a FFNN is shown in Figure 2.

The input layer consists of the input vector $\mathbf{x} \in \mathbb{R}^{N_0}$, with a corresponding bias vector $\mathbf{b} \in \mathbb{R}^{N_1}$. Here N_ℓ is the number of nodes in layer ℓ of our FFNN. As we feed the input into the first hidden layer, each neuron in the hidden layer calculates a weighted sum of the inputs. For the first hidden layer, the value $\mathbf{z}^{(1)} \in \mathbb{R}^{N_1}$ for the intermediates can be written as

$$\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)},$$

where $\mathbf{W}^{(1)} \in \mathbb{R}^{N_0 \times N_1}$ is the weight matrix. The output of the first layer is usually passed through an activation function $f^{(1)}$ such that the output $\mathbf{y}^{(1)} \in \mathbf{R}^{N_1}$ fed into the second layer becomes

$$\mathbf{y}^{(1)} = f^{(1)}(\mathbf{z}^{(1)}).$$

In general, for each layer ℓ, this process can be written as

$$\mathbf{z}^{(\ell)} = \mathbf{W}^{(\ell)}\mathbf{y}^{(\ell-1)} + \mathbf{b}^{(\ell)},$$
$$\mathbf{y}^{(\ell)} = f^{(\ell)}(\mathbf{z}^{(\ell)}).$$

After the input has been fed through all the hidden layers, we use the output layer to compute the predictions of the FFNN. This is done by putting the activations from the last hidden layer through a final activation function. This activation function will differ based on what task we want the FFNN to perform [15].

Notice the convenience of writing all the operations in vector notation. The whole propagation through the FFNN can be explained as simple operations between vectors and matrices. In general for a given layer ℓ, we have that $\mathbf{W}^{(\ell)} \in \mathbb{R}^{N_{\ell-1} \times N_\ell}$, while $\mathbf{y}^{(\ell)}$, $\mathbf{b}^{(\ell)} \in \mathbb{R}^{\mathbb{N}_\ell}$. When intitializing the FFNN all weights are set randomly to a value close to zero drawn from a normal distribution and biases initialized as 0.1. If these were simply initialized with zeros, all neurons would have the same output and render the FFNN unable to learn effectively.

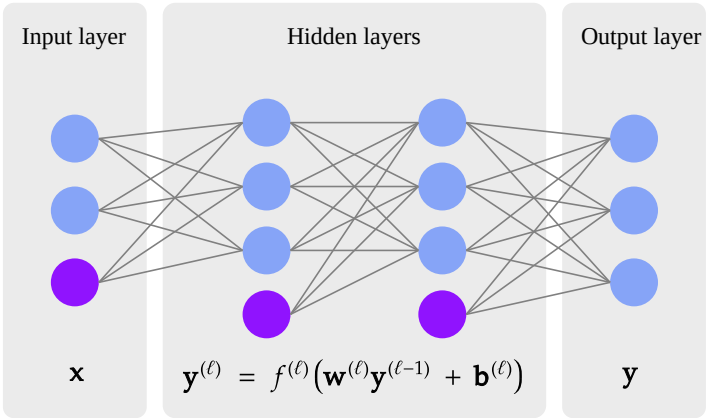


Figure 2: *A diagram of the architecture of a feed forward neural network. The blue nodes represent the activations, while the purle nodes represent the bias.*

2.4 Back Propagation

The back propagation algorithm [16] is the backbone of the FFNN. It allows us to calculate the gradients of all the parameters of the network such that they can be updated

with respect to the cost function. In other words, it allows the network to learn and improve. The parameters that can be changed in the FFNN are the weights $\mathbf{W}^{(\ell)}$ and biases $\mathbf{b}^{(\ell)}$. As before, we want to minimize the cost function $C(\mathbf{W}^{(\ell)}, \mathbf{b}^{(\ell)})$ with respect to the parameters. Since the FFNN has several hidden layers with possibly non-linear activations, calculating the gradient of the cost function with respect to the parameters deep within the network is not trivial. This complexity is why we need backpropagation.

The backpropagation algorithm is based on an iterative application of the chain rule to calculate the gradient of the cost function C with respect to all the parameters in the network. There are three equations needed for the backpropagation algorithm:

$$\boldsymbol{\delta}^{(\ell)} = f'(\mathbf{z}^{(\ell)}) \odot (\mathbf{W}^{(\ell+1)})^T \cdot \boldsymbol{\delta}^{(\ell+1)}, \tag{5}$$

$$\frac{\partial C}{\partial \mathbf{W}^{(\ell)}} = \boldsymbol{\delta}^{(\ell)} \cdot (\mathbf{y}^{(\ell-1)})^T, \tag{6}$$

$$\frac{\partial C}{\partial \mathbf{b}^{(\ell)}} = \boldsymbol{\delta}^{(\ell)}, \tag{7}$$

where $\odot$ refers to the Hadamard product. Equation 5 defines the value of $\boldsymbol{\delta}^{(\ell)}$ for a given layer ℓ which is the gradient of the cost with respect to the intermediate $z^{(\ell)}$. Notice that both the gradients for the weights and the biases only depend on $\boldsymbol{\delta}^{(\ell)}$ and the activation from the previous layer. Therefore, if we store their values as we propagate the data through the network in what we call a forwards pass, we have all the information we need to calculate the gradients. When we calculate the gradients using the difference equations described above, we refer to this process as a backwards pass, which is the origin of the term backpropagation[15].

To find the gradients using backpropagation we use a set of methods called Automatic Differentiation (AD). An AD method numerically evaluates a partial derivative of a function specified by a computer program [9]. It exploits the fact that all operations calculated on a computer are broken down into a finite set of elementary arithmetic operations and elementary functions. Therefore, by applying the chain rule iteratively on these operations, it is possible to automatically compute partial derivatives of arbitrary orders to numerical precision. If we use the equation for the activation in layer ℓ of our neural network we could decompose it as follows into components given by $\boldsymbol{\omega}_i$

$$\mathbf{y}^{(\ell)} = f^{(\ell)}(\mathbf{z}^{(\ell)}) = f^{(\ell)}(\boldsymbol{\omega}_0) = \omega_1$$

The derivatives with respect to the components $\boldsymbol{\omega}_i$ can then be solved by applying the chain rule. In fact, it is not difficult to show that using AD in the reverse mode is equivalent to the backpropagation algorithm [17].

2.5 Activation Functions

As previously mentioned, the intermediates of a layer in the NN are passed through an activation function $f^{(\ell)}(\mathbf{z}^{(\ell)})$. This idea has it's roots in cognitive research, where the activation function replicates the 'activation' of biological neurons as a response to stimuli. It is most common to specify the same activation function for all hidden layers in the FFNN and change it for the final output layer depending on the nature of the task. We want our NN to be able to model more than just linear relations, this is where the activation functions come into play, as they introduce non-linearity and thus allow the NN to learn complex patterns. We will introduce several activation functions seen in Figure 3, each having different properties which dictate how gradients will flow through the NN during backpropagation.

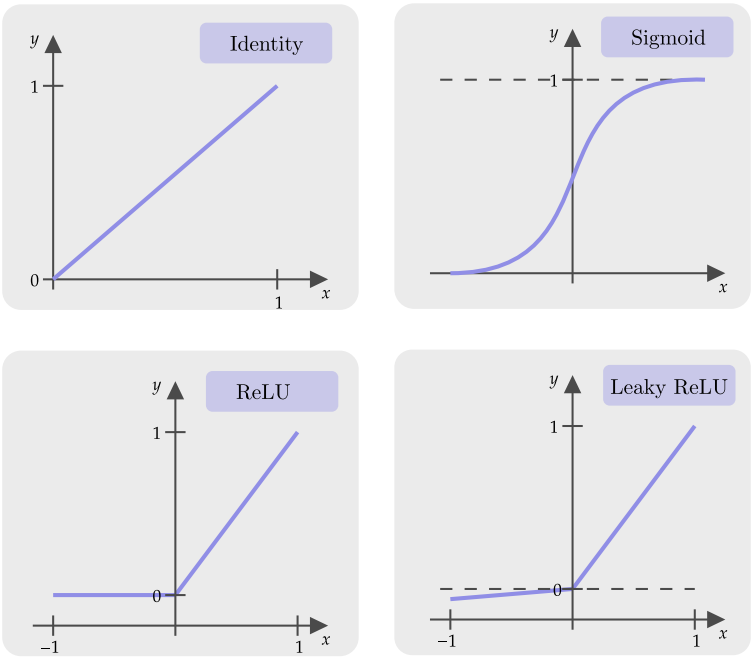


Figure 3: *Four activation functions which can be used in a NN architecture: Identity (top left), Sigmoid (top right), ReLU (bottom left), and Leaky ReLU (bottom right).*

2.5.1 Sigmoid

The Sigmoid function, named after its characteristic S-shape, maps any real number to the domain (0,1) with an intercept at $\sigma(0) = 0.5$. The function is defined as,

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

It is a commonly used activation function in logistic regression[18]. In binary classification problems, the output of the sigmoid is interpreted as the probability of the input belonging to class 1 (the positive class).

2.5.2 ReLU

Another frequently used activation function is the rectified linear unit (ReLU),

$$\mathrm{ReLU}(x) = \max(0, x) = \frac{x + |x|}{2} = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{if } x \leqslant 0. \end{cases}$$

This function addresses the typical *vanishing gradient* problem encountered in the sigmoid function (amog others). For large input values, the sigmoid gradient becomes close to zero, making the learning process extremely slow or even stop completely. ReLU addresses this by having a gradient of 1 for all positive inputs.

2.5.3 Leaky ReLU

Notably, the ReLU activation function mentioned above still has a vanishing gradient problem for negative input values. This is not always an issue and may even aid models with classification[18], but in many applications gradients for the full range of inputs are

beneficial. This can be achieved by making a small modification to the ReLU. The Leaky ReLU is given as,

$$\text{LeakyReLU}(x) = \begin{cases} x & \text{if } x > 0, \\ \delta x & \text{if } x \leqslant 0, \end{cases}$$

where $\delta \in \mathbb{R}$ is some small positive value to achieve a slight slope for negative x values. In our case we have used $\delta = 10^{-4}$. This results in non-zero gradients for negative inputs and a more expressive NN.

3 Methods

We perform both regression and classification using our FFNN. The following will take you through how the FFNN is tailored to the specific type of problem, such as the choice of cost function. Additionally, for regression, we implement the algorithms OLS and Ridge used in Gullbekk, Canclini, and Ryther [4]. These are chosen as a benchmark to assess the FFNN performance on fitting the Franke function. OLS and Ridge have been chosen because they are easy to interpret, have analytical solutions and performed well on the Franke function in previous studies [4]. Classification is performed on the Wisconsin Breast Cancer dataset[5]. Here, the FFNN is compared to an implementation of Logistic regression and both are assessed against scikit's `LogisticRegression` during development.

3.1 Regression

3.1.1 Linear Regression

Linear Regression aims to approximate the target variable using a linear combination of predictors. We will implement the linear regression methods OLS and Ridge using a design matrix $\mathbf{X}$ as the input to minimize the cost function as defined in [4]:

$$C_{\text{OLS}}(\boldsymbol{\theta}) = \frac{1}{n}\|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|_2^2,$$
$$C_{\text{Ridge}}(\boldsymbol{\theta}) = \frac{1}{n}\|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|_2^2 + \lambda\|\boldsymbol{\theta}\|_2^2.$$

Where $\boldsymbol{\theta}$ represents the weights and biases as usual and λ is the regularization parameter. To solve the minimization problems stated, we will use different gradient descent methods in addition to calculating the analytical solution. A key difference between these linear regression methods is the use of the design matrix. This means that the polynomial order of the method is fixed and a suitable order has to be chosen beforehand.

3.1.2 Regression with a Feed Forward Neural Network

To be able to use a FFNN for regression, we need to make sure that the cost function and activation functions are suitable. For a fair comparison, we will use the cost function used in Ridge regression because it is equivalent to OLS when the regularization parameter λ is set to zero. Further, to be able to leverage the universal approximation theorem we need to have non-linear activation functions in the hidden layers (or else the FFNN will be equivalent to a linear regressor). Lastly, the final activation function should be the identity function. This is because we do not want to introduce any bias for how the target variables are distributed. The key advantage of the FFNN is that it takes the original data points as input and outputs the targets directly. This means that we do not have to construct a design matrix and the network itself will find a suitable order for the function we are fitting.

3.2 Classification Methods

3.2.1 Logistic Regression

While OLS and Ridge regression are effective methods for predicting continuous outcomes, they are unsuited for classification problems. Classification tasks require methods such as Logistic regression, where the output is interpreted as a probability between 0 and 1. Logistic regression is one of the most widely used binary classification methods in industry due to its simplicity and strong performance on linearly separable classes [18]. In Logistic regression, each data point $\mathbf{x}_i$ is assigned a category $y_i \in \{0, 1\}$. The conditional probability is modeled using the sigmoid function $p(y_i = 1 \mid \mathbf{x}_i, \boldsymbol{\theta}) = \sigma(\mathbf{x}_i)$ (see section 2.5.1), which maps any real-valued number to the interval $(0, 1)$. The predicted class $\hat{y}$ is then typically determined by applying a threshold function

$$\hat{y} = \begin{cases} 1 & \text{if } \sigma(x) \geqslant 0.5, \\ 0 & \text{otherwise.} \end{cases} \tag{8}$$

where $\sigma(x)$ is the sigmoid function and x is the output of a linear regression function. In this case, we assign a probability of 0.5 to category 1.

Unlike the function approximation cases above, which minimize the mean squared error, Logistic regression minimizes the negative log-likelihood, also called Binary Cross-Entropy (BCE) in the binary case. This is motivated by the fact that the Mean Squared Error (MSE) is not convex when using the sigmoid function, whilst the BCE is, and thus easing the convergence to a global minimum during gradient descent. The BCE is given by [19]

$$\mathcal{C}(\boldsymbol{\theta}) = -\sum_{i=1}^{n} \left(y_i \log \sigma(z_i) + (1 - y_i) \log \left[1 - \sigma(z_i) \right] \right). \tag{9}$$

Due to the non-linearity of the sigmoid function, Logistic regression has no closed form solution [19]. Therefore we use iterative optimization algorithms such as gradient descent to find the parameters that minimize the cost function.

3.2.2 Classification with a Feed Forward Neural Network

To use the FFNN for classification, we use BCE (see Equation 9) as the cost function instead of the MSE. Furthermore, as for Logistic regression, the output layer uses the sigmoid activation function followed by a threshold function such that final outputs are 0 or 1. The input values do not differ from logistic regression, where we simply use the data points with all their features without using a design matrix as we do with linear regression.

3.3 Error Analysis

When comparing our different regression setups we use MSE to quantify the quality of the model. MSE is given as

$$MSE = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2, \tag{10}$$

where N is the number of samples, y_i is the actual sample value, and $\hat{y}_i$ is the predicted sample value.

While this works well for regression tasks it is not suited for evaluating the quality of a model applied to classification. To compare different classifiers, we use the *accuracy*

metric, which is the proportion of labels predicted correctly out of all predicted labels.

$$\text{Accuracy} = \frac{T_P + T_N}{T_P + F_P + T_N + F_N}, \tag{11}$$

where T_P and T_N are correctly predicted labels, and F_P and F_N are wrongly predicted labels, with subscripts P and N denoting positives and negatives respectively. We use this metric because its is a good metric to asses the overall quality of the model. In addition to using accuracy, we can construct a confusion matrix that gives additional information such as precision and recall.

3.4 Data Sets

For regression, we will test the methods on the Franke function as done for OLS and Ridge regression in previous work [4]. The Franke function is given by:

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

Notice that this function is a weighted sum of exponentials that make up two Gaussian peaks and can therefore not be represented by a finite degree polynomial. In addition to enabling a comparative analysis with previous work, applying the FFNN on the Franke function gives us the opportunity to see if the non-linearity of the network proves advantageous. For the purpose of comparison, we will use the same data points as in the previous work.

For classification, we will test the different methods on the Wisconsin Breast Cancer data set. This data set contains 569 data points with 30 features each along with the target diagnosis for each data point. The target diagnosis are binary values indicating the absence (0) or precense (1) of cancer in the patients. The features are derived from digitized images of fine needle aspirates of a breast mass, an example is shown in Figure 4.

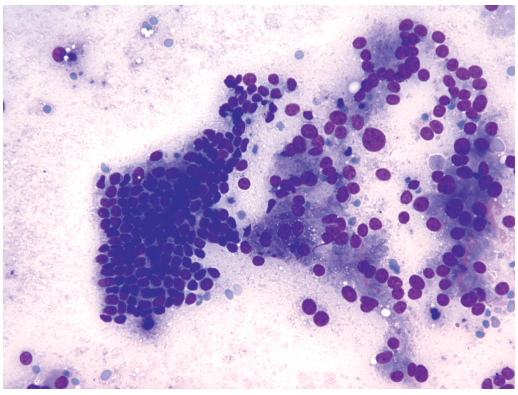


Figure 4: *An example of a fine needle aspirate similar to those used to generate the Wisconsin breast cancer data set [20].*

3.5 Scaling and Splitting of the Data Sets

It is always good practice to scale our data before each algorithm is run. For regression, this might seem unnecessary at first, since all the features have the same dimension. However, since we have defined a cost function based on Ridge regression, the regularization terms might be affected by how the data is scaled. This is especially important if the intercept is high. In addition scaling makes it more intuitive to compare results across different datasets and between different methods. For classification, scaling is often essential. This is because the features can have many different dimensions. For instance, for the Wisconsin Breast Cancer data set we have thirty different features. If one of the features has very large values it might dominate in the sense that the cost will be disproportionally affected by that feature. Thus, scaling with normalization is chosen such that the different features have values within similar ranges which leads to faster convergence and more accurate predictions. We will use the scaler `StandardScaler` from the python library `scikit-learn`.

Data splitting is done to ensure that we do not overfit to the training data. For better comparison with the regression analysis performed in [4] we will use a separate training and test set with an 80% and 20% split. For the classification task there are only 569 data points in the Wisconsin Breast Cancer data set. Although it would be beneficial to introduce a separate validation data set in addition to the train and test data sets, we argue that this would lead to an insufficient amount of data in each data set. Therefore we use the same data splitting for classification as we do with regression. To see a more representable score, closer what we could have achieved with more data, we will also run the classification models for 300 different random seeds. This essentially just changes how the dataset is split into test and train sets, thus adding an additional element of stochasticity. To perform the data splitting we use the method `train_test_split` from the Python library `scikit-learn`.

3.6 Implementation

The code is written in Python using standard packages in addition to functions from scikit-learn [21]. All plots have been made using the matplotlib [22] package in python. We have only tuned the different schedulers by varying the learning rate, all other parameters are given the values as shown in table 1. The implementation of the FFNN is based upon the implementation from [23]. All code used to generate the results can be found in the GitHub repository [2].

Table 1: *Default hyperparameter values for different optimization algorithms. N/A indicates that the parameter is not applicable for that particular scheduler.*

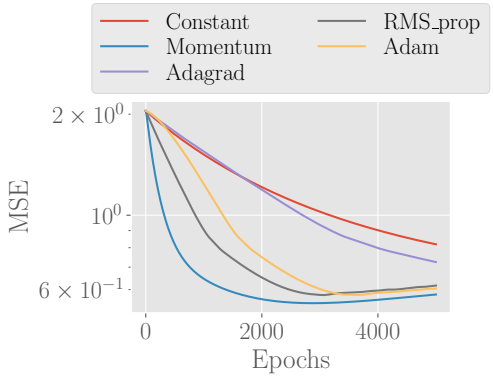
Optimization Algorithm	Hyperparameters		
	Momentum (β)	First Moment (ρ)	Second Moment (ρ_2)
Momentum	0.9	N/A	N/A
AdaGrad	0.9	N/A	N/A
RMSProp	N/A	0.9	N/A
Adam	N/A	0.9	0.999

[2] GitHub repository: `github.uio.no/sophusbg/FYS-STK4155/tree/main/project2`

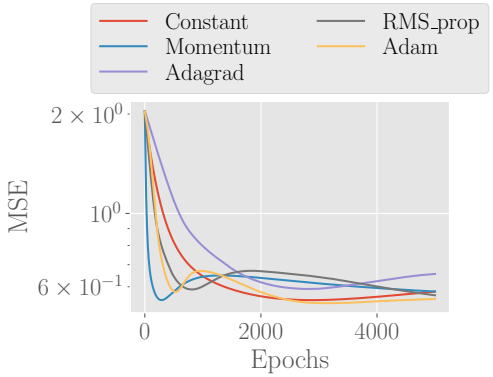
4 Results and Discussion

4.1 Regression

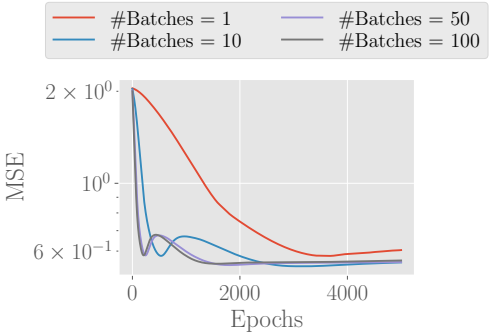
Before constructing the FFNN, we perform an initial comparison of GD and SGD using OLS. The goal of the analysis is to highlight their differences as well as decide which one should be used for the FFNN. In Figure 5a and Figure 5b we see the resulting MSE test scores for OLS using both gradient descent methods for five different schedulers. SGD is run with ten mini batches. It is clear from the figures that SGD has a much faster convergence rate than standard GD. In fact, for standard GD, neither the constant scheduler nor Adagrad manages to converge before 5000 epochs. Despite this, both methods achieve similar MSE scores. Additionally, SGD normally has a lower computational cost. Thus, it is not unreasonable to use SGD when we design the FFNN. Interestingly, we see that the test MSE starts to increase at some point in both plots. This might indicate a point where the model starts to overfit the data and suggests that we could benefit from terminating the algorithm sooner than 5000 epochs, therefore we will run subsequent models for 1000 epochs. This is both to prevent overfitting and to save time as later parameter tuning will demand a lot of computational power.



(a) OLS SGD



(b) OLS SGD



(c) OLS SGD with Adam

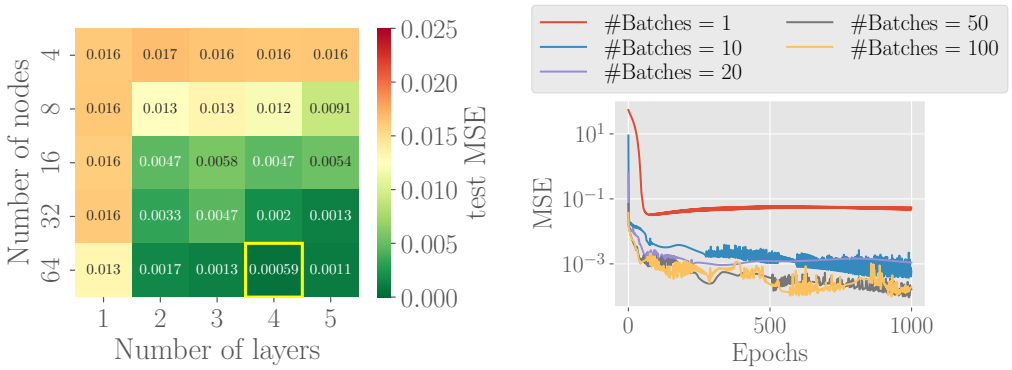
Figure 5: *Figure (a) and (b) gives a comparison of GD and SGD with 10 batches done on OLS with different schedulers. Figure (c) compares how the number of batches affects the performance of the regressor. All schedulers use the same learning rate of* 10^{-4}*, and other standard parameters defined in table 1 They are run with order 8 polynomials.*

We also show how the Adam scheduler performs with different batch sizes on OLS in Figure 5c. It is clear that the algorithm benefits from a larger batch size, but that there are diminishing returns. We therefore conclude that ten mini batches, as run with the previous two figures, is suitable for this problem. Later, we will also look at the impact of batch size for the more complex FFNN. Note that although OLS is a convex

optimization problem, none of the methods manage to come close to the MSE for the analytical solution of 3.7×10^{-4} found in Gullbekk, Canclini, and Ryther [4] using the same order polynomial and data points.

Having decided to use SGD over GD and a batch size of 10 for 1000 epochs, we now decide the architecture of the FFNN. Figure 6a shows a grid search over number of nodes and layers for the network using the sigmoid activation function for all hidden layers. We choose to start with the sigmoid function as it should perform well in fairly shallow networks. Similarly, we begin our tuning using the Adam scheduler as it has become an industry standard and is expected to perform well. Here, the lowest test MSE of 5.9×10^{-4} is achieved for an architecture of 4 layers with 64 nodes each. It should be noted that this is only the best combination found in the sampled parameter domain, and even better architectures may exist for a higher number of layers or nodes. These combinations have not been explored in this study as the computational cost was deemed too large in relation to the possible marginal gains in precision.

The suitability of the chosen batch size and number of epochs is verified for the 4 layers and 64 node architecture in Figure 6b. We conclude that 10 batches are still a suitable compromise between computational efficiency and precision, especially considering the MSE becomes relatively stable within 1000 epochs.



(a) *Grid search over the number of layers and hidden nodes in the FFNN. Here Adam was used as the scheduler with default parameters (Table 1), and learning rate* 10^{-3} *and with the sigmoid activation function. The yellow square indicates the chosen architecture.*

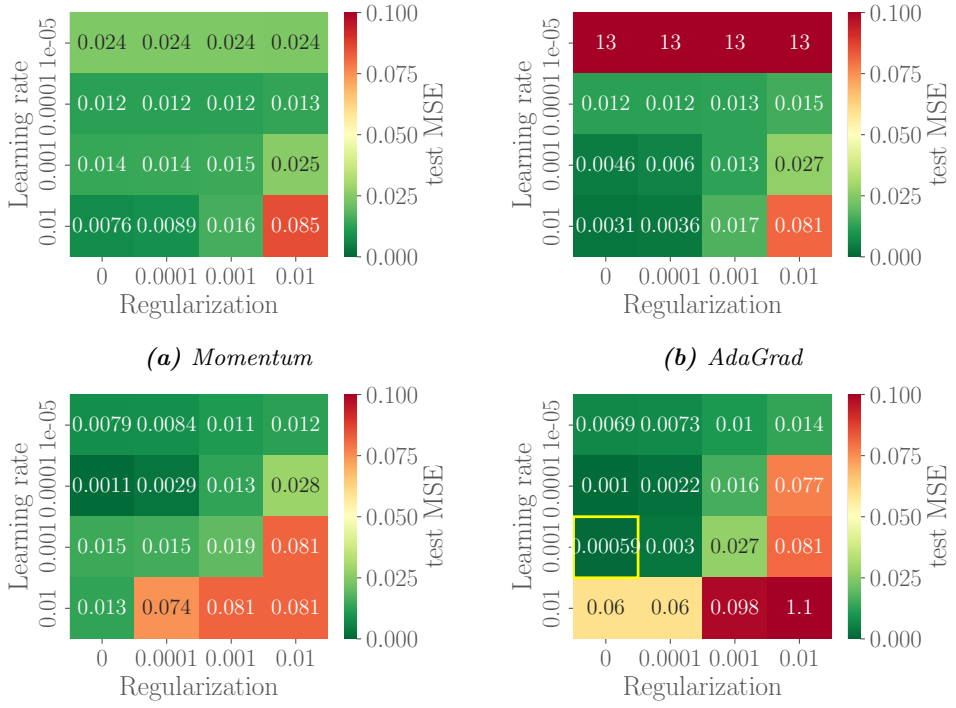
(b) *MSE scores for theFFNNfor different numbers of batches. Here the network structure is 64 nodes and four layers with Adam as the scheduler with default values (Table 1) and learning rate of* 10^{-2}.

Figure 6: *Finding a suitable network architecture (a) for the FFNN, as well as comparing different batch sizes over epochs (b).*

So far, we have only used the Adam scheduler in the FFNN as it empirically has been shown to work well on a wide spectrum of different problems. However, we also assess how the performance varies when we use different schedulers. We have defined four different scheduler based on SGD: Momentum, Adagrad, RMSprop and Adam. To test the performance of the different schedulers we use aFFNNwith 4 layers with 64 nodes each and perform a grid search over different learning rates η and regularization parameters λ. The results are shown in Figure 7. The best MSE score is still 5.9×10^{-4} for $\lambda = 0$ and $\eta = 0.001$ using the Adam scheduler, which is considerably better than the best scores for the other schedulers: 10^{-3} for RMSprop, 7.6×10^{-3} for Momentum and 3.1×10^{-3} for AdaGrad.

There is an interesting trend for all schedulers towards both a smaller regularization parameter and learning rate. The smaller regularization parameter is in accordance to the findings in [4], where they concluded that OLS performed better than Ridge regression — even with a regularization parameter of order 10^{-6}. Thus it is not surprising that the

best MSE achieved is with a regularization parameter of zero which is equivalent to OLS. The trend towards smaller learning rates might indicate that we could have run the grid search for even smaller learning rates. However, the best result was accomplished with a learning rate of 0.001.

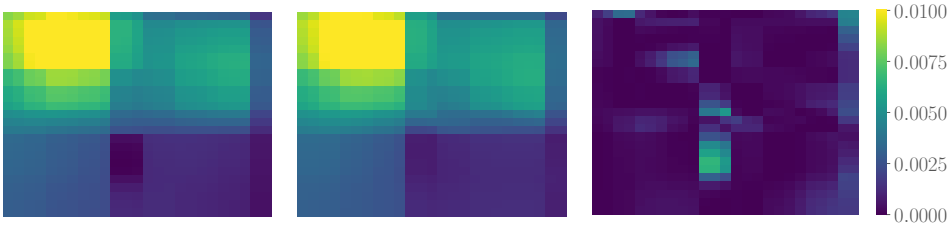


(a) Momentum *(b) AdaGrad*

(c) RMSprop *(d) Adam*

Figure 7: *A comparison of the different schedulers on the so far best network configuration of 64 nodes and four layers. All other values have been set to default according to Table 1.*

After tuning the architecture, batch size, scheduler, learning rate and regularisation parameter for the FFNN, we are ready to generate a final prediction. The data we are fitting is shown in Figure 8a, the final prediction is shown in Figure 8b and the squared error for each pixel is shown in Figure 8c. We have used a FFNN with four hidden layers with 64 nodes in each hidden layer and the Adam scheduler with a learning rate of 0.001 based on the cost function from OLS. We see in the final prediction that the surface is predicted fairly well and that it is difficult to distinguish the two at a glance. However, when looking at the squared error for each pixel it is clear that the FFNN struggles to accurately predict the global minimum close to the center of the surface.

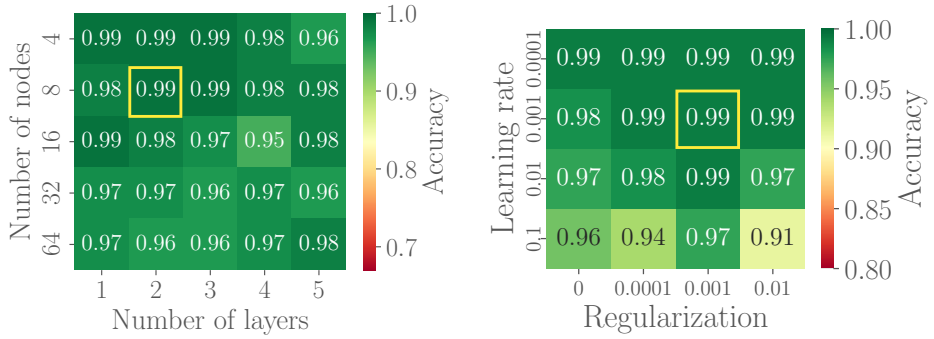


(a) The Franke function *(b) FFNN prediction* *(c) The squared error*

Figure 8: *The performance of our bestFFNNmodel, with 64 nodes, four layers, learning rate of 10^{-3}, and λ set to 0, on the franke function. Figure 8c show the squared error between the two for each data point on the surface.*

4.2 Classification

Having used the FFNN for regression, we now perform a similar analysis using it for classification of the popular Wisconsin Breast Cancer dataset [5]. We use 10 batches and 1000 epochs as for the regression above. This gives good results but has not been tuned specifically to the dataset, something which could be done in future studies. To find a suitable network architecture we start by using the Adam scheduler and asses the test accuracy of the model for different numbers of nodes and layers as presented in Figure 9a. Here we use the sigmoid activation function for all hidden layers and the output layer. The same grid search for network architecture has been performed using ReLU and LeakyReLU as activation functions for hidden layers, but gave inferior and unstable results (see Appendix A.3, Figure 13). Figure 9a already shows high accuracy for all shown architectures. Here we choose to continue using a network with 2 hidden layers and 8 nodes each, as this seems to be the best region in the parameter space. This also strikes a nice balance between computational cost and network expressivity.



(a) *Combinations of layers and nodes for cancer data classification. Each combination of nodes and layers is run using the Adam scheduler with an initial learning rate of* 10^{-4}.

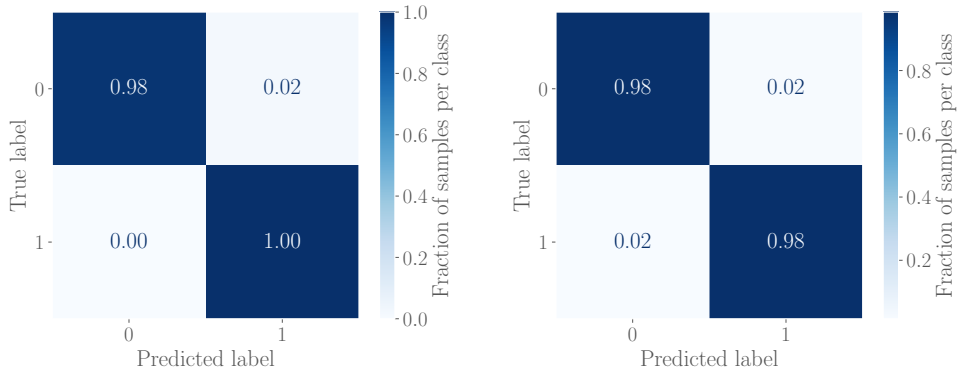
(b) *Grid search over learning rate and regularization parameter for a network of two layers with eight nodes each, using the Adam scheduler.*

Figure 9: *Tuning of theFFNNfor cancer data classification. All hidden layers have the same number of nodes and use the sigmoid activation function. Performance is shown as classification accuracy and each combination is run using 10 batches and 1000 epochs using the Adam scheduler with default values from Table 1. The yellow squares indicate chosen combinations with an accuracy of 0.99.*

This architecture is then assessed further using four different schedulers. Performing a grid search over learning rate and regularization parameter for Momentum, Adagrad, RMSprop and Adam shows that all of these options could result in an accurate model (see Appendix A.3, Figure 14). Adam with a leaning rate and regularization term of 0.001 gives an accuracy of 0.99 and is chosen as our best model as seen in Figure 9b. This is again based on comparably high accuracy in the surrounding parameter space which is marginally better when compared to other schedulers. It should be noted that all schedulers achieve accuracies of at least 0.98 in the sampled parameter space, there is thus no strong motivation to use Adam over other methods.

Figure 10 shows confusion matrices for cancer classification using the FFNN as well as our own Logistic regression. A parameter search reveals that the Logistic regression model also performs well for both a learning rate and regularization parameter of 0.001, achieving an accuracy of 0.98 (see Appendix A.3, Figure 15). This allows for a relatively fair comparison of the two models. These are both run using the Adam scheduler and outperform an implementation using Scikit-Learn's Logistic regression method which does not allow for the use of Adam (not shown). We observe marginal differences, with false positive rate of 0.02 for both the FFNN and Logistic regression. In the cancer case,

our most important score concerns the false negative which could result in untreated cancerous tumors. The FFNN has a false negative rate of 0 which means that all cancerous tumors were identified, while Logistic regression has a false negative rate of 0.02. Based on this, we can consider the FFNN a bit more appropriate for the task of cancer classification, although marginally so. The FFNN should be capable of modeling more complex relationships, even with this simple two layer structure. Nonetheless, results show that Logistic regression captures the relations in this dataset very well, this may change for larger more complex datasets where a deeper network could be beneficial.



(a) *FFNN with two layers and eight nodes in each.*

(b) *Own Logistic regression*

Figure 10: *Confusion matrices for classification of the Wisconsin Breast Cancer data using aFFNNand Logistic regression. Both theFFNNand Logistic regression are run using Adam with an initial learning rate of* 10^{-3}*, default values from Table 1, a regularization of* 10^{-3} *and 10 batches over 1000 epochs.*

It should be noted that more exhaustive parameter tuning for both models may lead to even better accuracies. At a glance, the Adam scheduler performed best in our study, but additional tuning using other schedulers could possibly lead to an even better model. We leave this deeper exploration to future work since any marginal improvements of accuracy so close to 1 would require an disproportionate investment of CPU cycles. In addition it is clear that without a separate validation data set, further improvement gains could simply be a consequence of added bias to the models.

Model tuning has been performed using a set of train and test data as we deem the dataset too small for an additional validation set. This unfortunately biases the final accuracy results since the same data has been used for parameter tuning. Additionally, a set random seed has been used for test and train data splitting. This is great in terms for reproducibility, but also an opportunity to introduce another level for stochasticity.

Figure 11 shows the accuracy score distribution of both final models for 300 different random seeds. Notably, Logistic regression outperforms the FFNN. Logistic regression has a distribution centered close to 1, whereas the FFNN is below 0.98. This result agrees well with baseline model performance presented by William et al. [5], where Logistic regression outperformed NN classification.

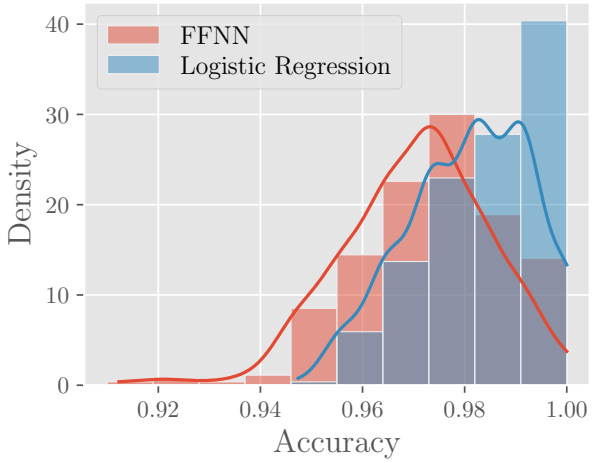


Figure 11: *Bar plot showing the distribution of accuracy scores using theFFNN(red) compared to Logistic regression (blue) for 300 different random seeds. Both theFFNNand Logistic regression are run using Adam with a n initial learning rate of* 10^{-3}*, default values from Table 1 and a regularization of* 10^{-3}*.*

These results also shows that the FFNN tuning was performed on a random case in the right tail rather than the center of the accuracy distribution, meaning the tuning showed the model to be better than it actually is. Similarly, tuning of the Logistic regression was performed left of the center of the observed distribution indicating that the confusion matrix in Figure 10b reflects a somewhat conservative estimate of the models accuracy. We believed SGD would introduce enough randomness to achieve a representative accuracy score, but clearly the small size of the dataset makes it very sensitive to the initial train-test splitting. This could be addressed by implementing bootstrapping or k-fold cross-validation in future studies of this data. Lastly, one should be aware that although results in Figure 11 give an indication of the models overall accuracy, we do not know if miss-classification is dominated by false positives or false negatives. Such distinctions are incredibly important when considering the classification of cancerous tumors as a false negative could lead to devastating outcomes.

5 Summary and Conclusions

This study demonstrates that FFNNs, when optimized appropriately, can closely match traditional regression and classification models on the chosen datasets. Despite a minimum MSE of 5.9×10^{-4} using a well-tuned FFNN to fit the Franke function, the analytical OLS solution remains superior with an MSE of 3.7×10^{-4}. For classification, the FFNN achieves a perfect false negative rate of 0 and an overall accuracy of 0.99, although 300 randomized runs show that the true accuracy is closer to 0.98. This highlights the importance of extensive and well split data for training and testing to avoid any bias introduced during model tuning. Logistic Regression still yields competitive results with a lower computational cost, proving highly effective with an accuracy distribution centered close to 1 for 300 random runs. The effectiveness of optimization methods, particularly Adam , is underscored by their significant improvement in convergence speed and model accuracy. Future research should focus on exploring other optimization strategies, more advanced network architectures, and robust cross-validation techniques to minimize bias introduced by limited data sizes. On the presented data there does not seem to be an obvious need for more advanced methods such as FFNNs, but we expect traditional methods to be outperformed on larger and more complex datasets where the high non-linearity of a deeper NN should perform well.

Acronyms

AD Automatic Differentiation. 8

Adam Adaptive Moment Estimation. 6

BCE Binary Cross-Entropy. 11

FFNN Feed Forward Neural Network. 3, 6–8, 10–19, 21

GD Gradient Descent. 3–5, 14, 15

ML Machine Learning. 3

MSE Mean Squared Error. 11, 14–16

NN Neural Network. 3, 6, 8–10, 18, 19

OLS Ordinary Least Squares. 3, 10–12, 14–16

ReLU rectified linear unit. 9, 10

SGD Stochastic Gradient Descent. 3–5, 14, 15, 19

References

[1] Arthur L Samuel. "Some studies in machine learning using the game of checkers". In: *IBM Journal of research and development* 3.3 (1959), pp. 210–229.

[2] Warren S McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5 (1943), pp. 115–133.

[3] Nobel Prize Outreach AB. *The Nobel Prize in Physics 2024*. `https://www.nobelprize.org/prizes/physics/2024/summary/`. Accessed: 2024-11-04. 2024.

[4] Sophus B. Gullbekk, Alessio Canclini, and Jakob W. Ryther. *Regression Analysis and Resampling Methods*. 2024. URL: `https://github.uio.no/sophusbg/FYS-STK4155/tree/main/project1`.

[5] Wolberg William et al. *Breast Cancer Wisconsin (Diagnostic)*. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5DW2B. 1993.

[6] G. Cybenko. "Approximation by superpositions of a sigmoidal function". en. In: *Mathematics of Control, Signals and Systems* 2.4 (Dec. 1989), pp. 303–314. ISSN: 1435-568X. DOI: `10.1007/BF02551274`. URL: `https://doi.org/10.1007/BF02551274` (visited on 11/01/2024).

[7] Morten Hjort-Jensen. *Applied Data Analysis and Machine Learning*. UiO, 2021. Chap. Optimization, the central part of any Machine Learning algortithm.

[8] Joseph Raphson. *Analysis Aequationum Universalis*. Originally published in Latin; English translation available as "Universal Solution of Equations". London, 1690.

[9] Morten Hjort-Jensen. *Applied Data Analysis and Machine Learning*. UiO, 2021. Chap. Optimization and Gradient Methods.

[10] Boris T Polyak. "Some methods of speeding up the convergence of iteration methods". In: *USSR Computational Mathematics and Mathematical Physics* (1964).

[11] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *Journal of Machine Learning Research* 12.61 (2011), pp. 2121–2159. URL: `http://jmlr.org/papers/v12/duchi11a.html`.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[13] Geoffrey Hinton. *Lecture 6e rmsprop: Divide the gradient by a running average of its recent magnitude*. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf. Coursera Lecture: Neural Networks for Machine Learning. 2012.

[14] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG]. URL: https://arxiv.org/abs/1412.6980.

[15] Morten Hjort-Jensen. *Applied Data Analysis and Machine Learning*. UiO, 2021. Chap. Week 41 Neural networks and constructing a neural network code.

[16] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". en. In: *Nature* 323.6088 (Oct. 1986). Publisher: Nature Publishing Group, pp. 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0. URL: https://www.nature.com/articles/323533a0 (visited on 10/22/2024).

[17] Atilim Gunes Baydin et al. *Automatic differentiation in machine learning: a survey*. arXiv:1502.05767. Feb. 2018. URL: http://arxiv.org/abs/1502.05767 (visited on 11/02/2024).

[18] S. Raschka et al. *Machine Learning with PyTorch and Scikit-Learn: Develop Machine Learning and Deep Learning Models with Python*. Expert insight. Packt Publishing, 2022. ISBN: 9781801819312. URL: https://books.google.no/books?id=UHbNzgEACAAJ.

[19] Morten Hjort-Jensen. *Applied Data Analysis and Machine Learning*. UiO, 2021. Chap. Logistic Regression.

[20] Shahla Masood and Marilin Rosa. "Cytopathology of the Breast". In: *Differential Diagnosis in Cytopathology*. Ed. by Ji-Weon Park et al. Cambridge University Press, 2021, pp. 340–388.

[21] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[22] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[23] Morten Hjorth-Jensen and contributors. *Machine Learning: Neural Networks for Physics*. https://github.com/CompPhysics/MachineLearning/blob/master/doc/pub/NeuralNet/ipynb/NeuralNet.ipynb. Accessed: 2024-11-04. 2023.

A Appendix

A.1 Deriving the Equations for Backpropagation

The derivation of the backpropagation equations introduced in section 2.4 builds upon an iterative application of the chain rule. Using the chain rule we can find the gradient of the cost C with respect to all the different parameters in the FFNN. Lets define a general FFNN with L layers, weights $\mathbf{W}^{(\ell)}$ and biases $\mathbf{b}^{(\ell)}$ for a given layer ℓ. Using the chain rule, we get the following gradients with respect to a weight $w_{i,j}^{(\ell)}$ and bias $b_i^{(\ell)}$ for a layer ℓ:

$$\frac{\partial C}{\partial w_{i,j}^{(\ell)}} = \sum_{k=1}^{N^{(\ell)}} \frac{\partial C}{\partial z_k^{(\ell)}} \frac{\partial z_k^{(\ell)}}{\partial w_{i,j}^{(\ell)}} = \frac{\partial C}{\partial z_i^{(\ell)}} \frac{\partial z_i^{(\ell)}}{\partial w_{i,j}^{(\ell)}}$$

$$\frac{\partial C}{\partial b_i^{(\ell)}} = \sum_{k=1}^{N^{(\ell)}} \frac{\partial C}{\partial z_k^{(\ell)}} \frac{\partial z_k^{(\ell)}}{\partial b_i^{(\ell)}} = \frac{\partial C}{\partial z_i^{(\ell)}} \frac{\partial z_i^{(\ell)}}{\partial b_i}^{(\ell)}$$

Since $\mathbf{z}^{(\ell)} = \mathbf{W}^{(\ell)}\mathbf{y}^{(\ell-1)} + \mathbf{b}^{(\ell)}$, we can generalize the above equations to vector form as follows:

$$\frac{\partial C}{\partial \mathbf{W}^{(\ell)}} = \frac{\partial C}{\partial \mathbf{z}^{(\ell)}} \cdot (\mathbf{y}^{(\ell-1)})^T.$$
$$\frac{\partial C}{\partial \mathbf{b}^{(\ell)}} = \frac{\partial C}{\partial \mathbf{z}^{(\ell)}}$$

Notice that for an arbitrary layer ℓ, the expressions only depend on the gradient of the cost with respect to the intermediates $\mathbf{z}^{(\ell)}$ and the activations from the previous layer $\mathbf{y}^{(\ell-1)}$. Since the activations are known, we just need to figure out an expression for the intermediates. Again, starting with the expression for a single intermediate $z_i^{(\ell)}$ in layer ℓ we get

$$\frac{\partial C}{\partial z_i^{(\ell)}} = \sum_k \frac{\partial C}{\partial z_k^{(\ell+1)}} \frac{\partial z_k^{(\ell+1)}}{\partial y_k^{(\ell)}} \frac{\partial y_k^{(\ell)}}{\partial z_i^{(\ell)}} = \sum_k \frac{\partial C}{\partial z_k^{(\ell+1)}} w_{k,i}^{(\ell+1)} f'(z_i^{(\ell)})$$

Which in vector notation can be written as:

$$\boldsymbol{\delta}^{(\ell)} = \frac{\partial C}{\partial \mathbf{z}^{(\ell)}} = f'(\mathbf{z}^{(\ell)}) \odot (\mathbf{W}^{(\ell+1)})^T \cdot \boldsymbol{\delta}^{(\ell+1)},$$

where $\odot$ is the Hadamard product. Thus, we can summarize the backpropagation algorithm into the following three equations:

$$\boldsymbol{\delta}^{(\ell)} = f'(\mathbf{z}^{(\ell)}) \odot (\mathbf{W}^{(\ell+1)})^T \cdot \boldsymbol{\delta}^{(\ell+1)},$$
$$\frac{\partial C}{\partial \mathbf{W}^{(\ell)}} = \boldsymbol{\delta}^{(\ell)} \cdot (\mathbf{y}^{(\ell-1)})^T,$$
$$\frac{\partial C}{\partial \mathbf{b}^{(\ell)}} = \boldsymbol{\delta}^{(\ell)}.$$

A.2 Schedulers

The algorithms behind the different schedulers are taken from [12].

Algorithm 1 AdaGrad

Require: Global learning rate ε
Require: Initial parameter $\boldsymbol{\theta}$
Require: δ small value for numerical stability
  Initialize $\mathbf{r} = 0$
  while stop criteria not met **do**
    Sample minibatch of m examples from the train-set $\{\mathbf{x}_1, ..., \mathbf{x}_m\}$ with corresponding targets $\mathbf{y}_i$.
    Compute gradient: $\mathbf{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i \mathcal{C}(\mathbf{x}_i, \boldsymbol{\theta})$
    Accumulate squared gradient: $\mathbf{r} \leftarrow \mathbf{r} + \mathbf{g} \odot \mathbf{g}$
    Compute and apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \frac{\varepsilon}{\delta + \sqrt{\mathbf{r}}} \odot \mathbf{g}$
  end while

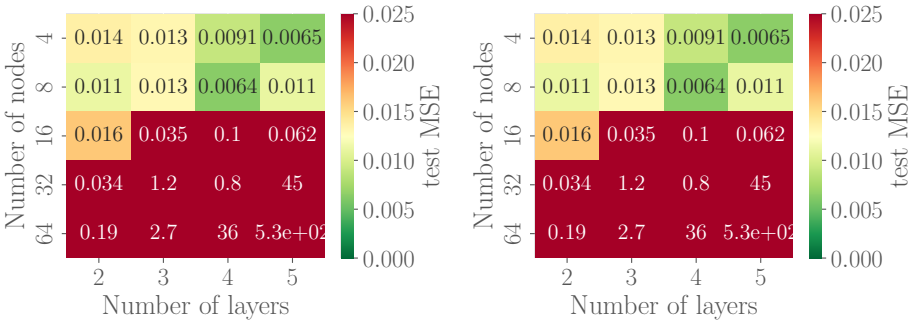
Algorithm 2 RMSprop

Require: Global learning rate ε, decay rate ρ
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ for numerical stability
Initialize $\mathbf{r} = 0$
while stop criteria not met **do**
Sample a minibatch of m examples from training set $\{\mathbf{x}_1, ..., \mathbf{x}_m\}$ with corresponding targets $\mathbf{y}_i$.
Compute gradient: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i \mathcal{C}(\mathbf{x}_i, \boldsymbol{\theta})$
Accumulate squared gradient: $\mathbf{r} \leftarrow \rho \mathbf{r} + (1 - \rho) \mathbf{g} \odot \mathbf{g}$
Compute and apply parameter update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\varepsilon}{\sqrt{\delta + \mathbf{r}}} \odot \mathbf{g}$
end while

Algorithm 3 Adam

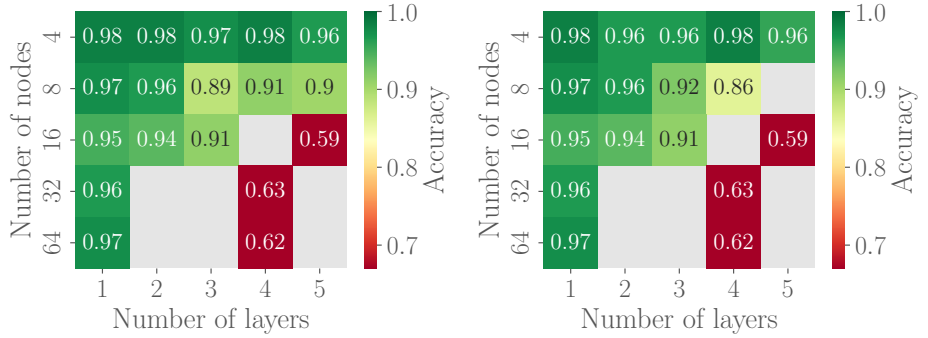
Require: Stepsize ε,
Require: Exponential decay rate for moment estimates ρ_1 and ρ_2.
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ for numerical stability
Initialize $\mathbf{s} = 0$ and $\mathbf{r} = 0$
Initialize $t = 0$
while stop criteria not met **do**
Sample a minibatch of m examples from training set $\{\mathbf{x}_1, ..., \mathbf{x}_m\}$ with corresponding targets $\mathbf{y}_i$.
Compute gradient: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i \mathcal{C}(\mathbf{x}_i, \boldsymbol{\theta})$
$t \leftarrow t + 1$
Update first moment estimate: $\mathbf{s} \leftarrow \rho_1 \mathbf{s} + (1 - \rho_1) \mathbf{g}$
Update second moment estimate: $\mathbf{r} \leftarrow \rho_2 \mathbf{r} + (1 - \rho_2) \mathbf{g} \odot \mathbf{g}$
Correct bias in first and second moment: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \rho_1^t}$, $\hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1 - \rho_2^t}$
Compute and apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \varepsilon \frac{\hat{\mathbf{s}}}{\sqrt{\mathbf{r}} + \delta}$
end while

A.3 Additional Figures



(a) ReLU *(b) Leaky ReLU*

Figure 12: *Grid searches over network structure for Regression. Each combination of nodes and layers is run for 1000 epochs and 10 batches using the Adam scheduler with an initial learning rate of* 10^{-3}*, and default values from Table 1*



(a) *ReLU* ***(b)*** *Leaky ReLU*

Figure 13: *Grid searches over network architectures for classification. Each combination of nodes and layers is run for 1000 epochs and 10 batches using the Adam scheduler with an initial learning rate of* 10^{-4}*, default values from Table 1 and no regularization.*

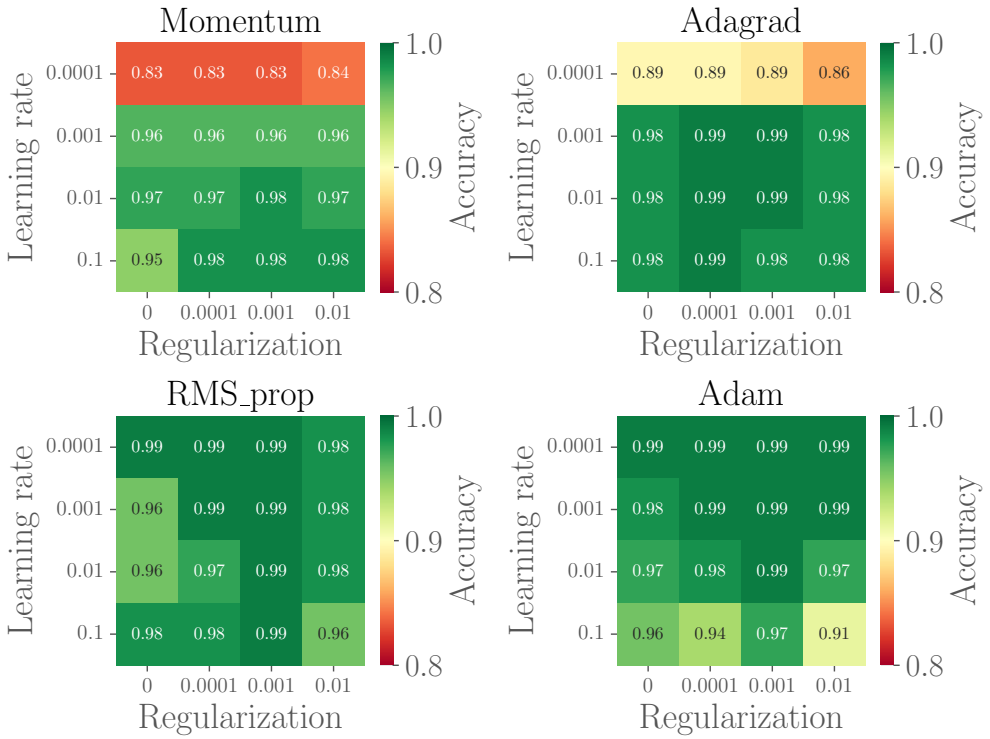


Figure 14: *Comparison of different schedulers for FFNN classification. Grid search over learning rate and regularization parameter using a network of 2 layers with 8 nodes each and 10 batches for 1000 epochs.*

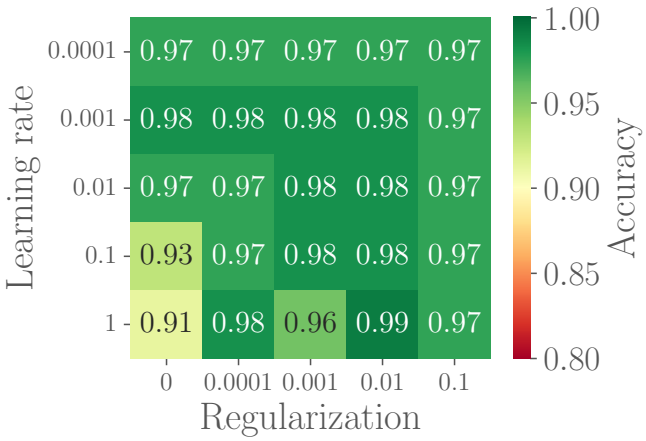


Figure 15: *Grid search over learning rate and regularization parameter for our own Logistic regression using a network structure of 8 nodes, 2 layers and the Adam scheduler with default values in Table 1.*