Using Feed-Forward Neural Networks for linear and logistic regression Project 2 – FYS-STK4155

Anders Thorstad Bø

Department of Mathematics (Fluid Mechanics),
University of Oslo, Oslo,
Norway[*]

In this project, I am implementing three different algorithms with the aim of studying their performance against two different types of datasets, namely a continuous 2D-dataset for regression, and a binary classification dataset. The algorithms are one general purpose Feed-Forward Neural Network (FFNN), two Stochastic Gradient Descent (SGD)-algorithms, one for linear regression, and one for logistic regression. The FFNN is the main focus of the study, and I am using the two others, in addition similar Scikit-learn methods and standard linear regression, to validate the performance of the FFNN-algorithm. My study uses five different SGD-methods, and five activation functions to investigate how this choice impacts the behavior and outcomes for the network. The analysis will also address the effect from different parameters such as learning rate, regularization and mini-batches. For testing I'm using datasets generated from the 1st Franke function for the regression analysis, and the Wisconsin Breast Cancer Data for the classification analysis. I found the best performance for the regression case with the ReLU-activation function in the hidden layers, but the network struggled with this dataset. For the classification case, the network performed well, producing predictions that scored similarly to other model.

I. INTRODUCTION

When dealing with real world data, there is a lot of inherent uncertainty in the datasets. Developing general purpose methods and algorithms for dealing with this uncertainty for predictions and approximations based on the data is a major driver behind the development of the class of methods named neural networks. Having general-purpose templates to produce specialized models have introduced a flexible approach to solving these real-world challenges. (Bishop, 2006)

The aim of this project is to show how to use a general **Feed-Forward Neural Network** (FFNN) to make predictions on two types of data. These types are a 2D-dataset generated for the first Franke function, and a classification dataset generated for the Wisconsin Breast Cancer Data (Wolberg et al., 1993).

I will be presenting how the network behavior changes with different choices of parameters and method. Examples of these parameters are learning rate, regularization, and mini-batch sizes. The methods are different **Stochastic Gradient Descent** (SGD)-methods, such as momentum gradient descent and ADAM (Kingma and Ba, 2017). I will also see how the choices of different so-called activation functions for the layers in the FFNN impact the performance of the network.

As part of the project, I also implemented two SGD-regression algorithms, one for linear regression, and one for logistic regression, in part for comparison purposes against the FFNN-implementation, and also to investigate the application to these kinds of datasets. These two methods also use the same SGD-methods for their gradient calculations, and will be using *Mean Squared Error* and *negative log-likelihood* as cost functions, respectively.

For the purposes of verification and validation, I also use **scikit-learn**'s **MLPRegressor** and **MLPClassifier**-classes, which I treat as well-tested implementations with trustworthy outputs (Pedregosa et al., 2011). In addition, I am using the my own results from a linear regression study looking at the Franke-function for validation of the linear regression case[1].

The report will, in Sec. II, through the theoretical concepts relevant for the FFNN-algorithm, explaining the general building blocks of the network, the gradient descent methods, a concept named automatic differentiation, different activation functions for the network layers. This part will also address model selection, specifically with regards to parameter selection, and model assessment with different error metrics. Examples of these are mean squared error, R^2-score, prediction accuracy, and the confusion matrix.

[*] andetb@uio.no; https://github.com/andersthorstadboe/project-2-fys-stk4155-ffneural-network

[1] Report can be found here: `https://github.com/andersthorstadboe/project-1-fys-stk4155-lin-regression/tree/main/02-report`

The Sec. III gives an overview of how algorithms for the FFNN, and the two SGD-methods, are implemented. It also presents the general structure of the programs and a short introduction on how to access and use the different programs used in this analysis.

The final parts, Secs. IV,V of the report addresses a selection of results from the analysis of the two datasets, and presents some of main findings. Some notable results are that the network struggles with the regression dataset, but performs a lot better with the classification case. The best performance for the two datasets are MSE ≈ 0.007, $R^2 \approx 0.92$ for the regression analysis, and an accuracy of ≈ 0.96 for the classification dataset.

The report also has an appendix that presents some of the theoretical concepts in more detail, and also some additional results that may be of the interest to the reader.

II. THEORY AND METHODS

This section outlines the theoretical concepts and methods used in this project, first giving an overview of the stochastic gradient descent method, then the general structure of a FFNN, as well as central concepts such as backpropagation, automatic differentiation and activation functions. Lastly, the section comments on the concepts necessary for model selection and assessment, such as different error measures for scoring the output, the confusion matrix, cumulative gains and the ROC-curve.

A. Stochastic Gradient Descent

Central to the concept of approximation is to maximize the likelihood of the resulting model from some input replicates the target model or function. The opposite is also true, that the difference between the model and the target is as small as possible. Both these can be seen as the need to optimize some model to the best possible fit to some data. The so-called **gradient descent** (GD)-algorithm seeks to do this by using the gradient of some function to step towards the function minimum. (Deisenroth et al., 2020).

The gradient of a real-valued function, $f(\boldsymbol{x})$, for $\boldsymbol{x}$ as a vector, points in the direction of the steepest ascent of the function at a given $\boldsymbol{x}_i$. The algorithm utilizes the fact that, for some starting $\boldsymbol{x}_0$, $f(\boldsymbol{x}_0)$ will decrease, or descend, fastest in the direction of the negative of $\nabla f(\boldsymbol{x}_0)$. Calculating

$$\boldsymbol{x}_1 = \boldsymbol{x}_0 - \eta_0 \nabla f(\boldsymbol{x}_0) \tag{A.1}$$

for some small $\eta_0 \geqslant 0$, usually known as the **learning rate** of the algorithm. That is, this ensures that $\nabla f(\boldsymbol{x}_1) \leqslant \nabla f(\boldsymbol{x}_0)$ for an appropriate choice of η_0. The simplest form of a GD-algorithm then seeks to calculate

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i - \eta_i \nabla f(\boldsymbol{x}_i) \tag{A.2}$$

in order to ensure that the sequence $\{f(\boldsymbol{x}_i)\}_{i=0}^{N}$, for large N, is decreasing, and converges to a minimum $f(\boldsymbol{x}_*)$. Here, η_i may be different from η_{i-1} and η_{i+1}, but this is not a requirement (Deisenroth et al., 2020).

In general, it is not possible to classify the minimum $f(\boldsymbol{x}_*)$ as a global minimum. Finding the global minimum is the aim for some prediction model, as this gives the lowest possible error. To ensure that the minimum is indeed a global one, the function needs to be convex, as all local minima are also global ones for that class of functions. A first condition put on the function is that, for $f \in C^1$ differentiable, then f is convex if and only if

$$f(\boldsymbol{y}) \geqslant f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})(\boldsymbol{y} - \boldsymbol{x}), \quad \forall\ \boldsymbol{x},\ \boldsymbol{y}\ \in D \tag{A.3}$$

for some convex set D. If $f \in C^2$, then the convexity of f is ensured if the Hessian

$$\boldsymbol{H} = \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x} \partial \boldsymbol{x}^T} = \nabla_x^2 f(\boldsymbol{x}) \tag{A.4}$$

is positive, semi-definite $\forall\ \boldsymbol{x} \in D$ (Deisenroth et al., 2020).

The previous descriptions assumes that the gradients are calculated from the entire dataset for each iteration. In many applications, these datasets can become very large, leading to a huge computational cost. The **Stochastic Gradient Descent**-approach seeks to lighten the load here by reducing the size of the set the gradients is calculated over.

The SGD must be seen as a approximation to a full GD-method, as reducing the size of the set will introduce stochastic noise into the calculation. In the most extreme case the size of this set is a single data point, which traditionally is what is known as the SGD-approach. This report treats all methods dividing the full dataset into smaller subsets as a SGD-method (Raschka et al., 2022).

In the context of this project, the implementation will seek to optimize a set of parameters based on the input dataset. Lets denote the vector of model parameters as $\boldsymbol{\theta}$. The task is to find a $\boldsymbol{\theta}_*$ that minimizes some cost function, C, generally defined as

$$C(\boldsymbol{\theta}) = \sum_{n=0}^{N-1} C_n(\boldsymbol{\theta}) \tag{A.5}$$

where $n = 0, \ldots, N-1$ is the data points, and C_n the cost of data point n. Its gradient is

$$\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}) = \sum_{n=0}^{N-1} \nabla_{\boldsymbol{\theta}} C_n(\boldsymbol{\theta}) \tag{A.6}$$

and the parameters update is

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \eta_i \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}_i) \tag{A.7}$$

To introduce randomness into the model, lets divide the full set of C_n into smaller subsets, usually called *mini-batches* of size M. Denote these as B_k, for $k = 1, \cdots, N/M$. The gradient calculation then becomes

$$\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}) = \sum_{n=0}^{N-1} \nabla_{\boldsymbol{\theta}} C_n(\boldsymbol{\theta}) \rightarrow \sum_{n \in B_k}^{N-1} \nabla_{\boldsymbol{\theta}} C_n(\boldsymbol{\theta}) \tag{A.8}$$

To keep this random, k must be picked by random from [1,N/M]. The algorithm then has to iterate over the number of the B_k's. One such iteration is usually referred to as an *epoch* (Hjorth-Jensen, 2023). As a final remark, an advantage of introducing mini-batches is that it may prevent the algorithm to get stuck in local minima, due to the introduced randomness. One can think of it in this way. Due to the noise, the parameter update may be "lifted up and out" of the local minimum, giving the next iteration the chance to continue the search with a new, larger gradient (Deisenroth et al., 2020).

1. Different SGD-method

The approach in the previous section is a general, plain SGD-approach with three hyperparameters η, the learning rate, M, the mini-batch size, and E, the number of epochs. Adjusting these will in turn impact the performance of the method. Several different additions and modifications to the plain SGD-method have been proposed to improve it. One modification is known as **momentum gradient descent**, which takes inspiration from the concept of momentum from physical systems and their models. Including momentum introduces inertia, or memory, to the system. This affects the parameter update by increasing the "speed" for small, but steady gradients, allowing for faster convergence, as well as dampening oscillations. The addition of momentum in the parameter update looks like this

$$\boldsymbol{v}_i = \gamma \boldsymbol{v}_{i-1} + \eta_i \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}_i) \boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \boldsymbol{v}_i \tag{A.9}$$

where $\gamma \in [0, 1]$ is the momentum.

Another class of modified methods are the ones making use of the second moment of the gradient to give a so-called adaptive learning rate, η. Relevant for this study are the **Adagrad, RMSprop** and **ADAM** algorithms. These methods adapts the learning rate based on information of the system behavior during the iterations in an effort to improve the convergence rate.

Common to all three is that they scale the global η by using an accumulated parameter calculated based on the current gradient and its *Hadamard* or *element-wise* product, i.e. a version of the squared gradient. The three methods are based on each other, where ADAM is a recent, well-received and robust method with broad applications (Goodfellow et al., 2016). Details on these method in the appendices, in App.C.

B. Application to regression analysis

The SGD-algorithms above can be applied directly to regression analysis, for a different approach than standard linear regression analysis. For this project, two types of regression analysis is of relevance, namely **linear regression**, which deals with continuous function approximation, and **logistic regression**, applied to so-called classification problems. The latter seeks to predict the state of a system from a collection of features. The predicted and target state can be as simple as a binary (0 or 1)-outcome, or more outcomes, known as classes (Hjorth-Jensen, 2023). This project only deal with binary classification, and the following will assume this binary outcome.

The main distinction when analyzing these different cases lies in what cost function to use to calculate how well the prediction is compared to some target dataset. For the linear regression case, a least squares error is usually the metric, and the cost function is the *mean-squared error* (MSE), looking like

$$C(\tilde{\boldsymbol{y}}, \boldsymbol{y}) = \mathrm{MSE}(\tilde{\boldsymbol{y}}, \boldsymbol{y}) = \frac{1}{n}(\tilde{\boldsymbol{y}} - \boldsymbol{y})^T(\tilde{\boldsymbol{y}} - \boldsymbol{y}) \tag{B.1}$$

where $\tilde{\boldsymbol{y}} = \tilde{\boldsymbol{y}}(\boldsymbol{x}, \boldsymbol{\theta})$ is the predicted function, and $\boldsymbol{y}(\boldsymbol{x})$ is the target function. Here, $\tilde{\boldsymbol{y}} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}$, where $\boldsymbol{X}$ is the design matrix, with features $\boldsymbol{\beta}$. More details on the MSE cost function can be found in App.A.

The gradient of Eq.(B.1) wrt. to the parameters $\boldsymbol{\theta} = \boldsymbol{\beta}$ is then

$$\nabla_{\boldsymbol{\beta}} C(\tilde{\boldsymbol{y}}, \boldsymbol{y}) = \frac{2}{n}\boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{\beta} - \boldsymbol{y}) = \frac{2}{n}\boldsymbol{X}^T(\tilde{\boldsymbol{y}} - \boldsymbol{y}) \tag{B.2}$$

Using an SGD-implementation, the parameters to update are $\boldsymbol{\beta}$, and the gradient is $\nabla_{\boldsymbol{\beta}} C(\boldsymbol{X}\boldsymbol{\beta}, \boldsymbol{y})$. Optimizing wrt. $\boldsymbol{\beta}$ then generates a model that approximates the target $\boldsymbol{y}$.

The logistic regression case uses a slightly different cost function due to the nature of the task. As it seeks to classify a outcome into a certain class, it is suitable to deal with the probability that the outcome belongs to a class. For a system with two classes, the probability that a data point, x_n belongs to one of the classes, $y_n = 0, 1$ can be expressed using the so-called *logistic*, or *Sigmoid*, function

$$p(y_n = 1|\boldsymbol{x}, \boldsymbol{\beta}) = \frac{1}{(1 + \exp(\sum_{j=0}^{p-1} \beta_j x_j))} \tag{B.3}$$

where $\boldsymbol{x} = [1, x_1, \ldots, x_{p-1}]$, $\boldsymbol{\beta} = [\beta_0, \beta_1, \ldots, \beta_{p-1}]$, for a model with p features.

To end at a cost function for such a setup, lets use the notion of the *likelihood* of observing a given outcome (Hjorth-Jensen, 2023). Using the **Maximum Likelihood Estimation** (MLE)-principle, which assumes that the best possible choice of the model parameters is where the probability of a observation is at its largest (Hastie et al., 2009). With this it is possible to express the approximate likelihood of a model predicting outcomes correctly as the product

$$P(y_n|\boldsymbol{x}, \boldsymbol{\beta}) = \prod_{n=1}^{N} [p(y_n = 1)]^{y_n} [1 - p(y_n = 1)]^{(1-y_n)} \tag{B.4}$$

where $p(y_n = 1)$ is the short form of the expressions in Eq.(B.3), and $p(y_n = 0) = 1 - p(y_n = 1)$. Taking the negative natural logarithm of $P(y_n|\boldsymbol{x}, \boldsymbol{\beta})$ gives the negative log-likelihood function as

$$\begin{aligned} C(\boldsymbol{\beta}) = -\sum_{n=1}^{N} (y_n \log[p(y_n = 1)] + \cdots \\ \cdots + (1 - y_n) \log[1 - p(y_n = 1)]) \end{aligned} \tag{B.5}$$

or more compactly as

$$C(\boldsymbol{\beta}) = -(\boldsymbol{y}^T \log[\boldsymbol{p}] + (1 - \boldsymbol{y})^T \log[1 - \boldsymbol{p}]) \tag{B.6}$$

This function will take the role the cost function to be minimized for the classification cases. Its gradient can be calculated as

$$\nabla_{\boldsymbol{\beta}} C(\boldsymbol{\beta}) = -\boldsymbol{X}^T (\boldsymbol{y} - \boldsymbol{p}) \tag{B.7}$$

where $\boldsymbol{X} \in \mathbb{R}^{n \times p}$ is the feature matrix containing the x_n's, $\boldsymbol{y}$ is the target vector of N elements, and $\boldsymbol{p}$ the vector of probabilities $p(y_n|x_n, \boldsymbol{\beta})$ (Hjorth-Jensen, 2023). More details on the differentiation can be found in App.A.

As with the optimizing the linear regression model, an SGD-methods now use this gradient to update the model features $\boldsymbol{\beta}$ to find the optimum values to minimize Eq.(B.5).

C. Feed-Forward Neural Network (FFNN)

A **Feed-Forward Neural Network** (FFNN) is one of the simplest types of the group of computational models called **Artificial Neural Networks** (ANNs). A general ANN builds on attempts of creating a mathematical model of how biological systems, such as a human brain, processes information (Raschka et al., 2022).

The network is a system of layers of artificial neurons that are connected to each other by mathematical functions between the layers. The resulting model is a nonlinear model , and the challenge is solving the resulting nonlinear optimization problem in an efficient way (Bishop, 2006).

1. General structure

The general structure of the FFNN is a collection of layers, l with associated weights, $\boldsymbol{W}$, and biases, $\boldsymbol{b}$ that are connected to the layers upstream and downstream by a mathematical function, σ, the activation function. Lets define the network input as $\boldsymbol{x}$, the network output as $\tilde{\boldsymbol{y}}$, and the target as $\boldsymbol{y}$. Then call the first layer, $l = 1$, the **input layer**, the last layer, $l = L$, the **output layer**, and the layers for $l = 2, 3, \ldots, L-1$, the **hidden layers**. The depth of the network is then L, and it has $L - 2$ hidden layers. A NN with more than one hidden layer is what is known as a **deep neural network** (Raschka et al., 2022).

To define the connections between the layers mathematically, lets define the layer output, $\boldsymbol{a}^{(l)}$, of layer, l as

$$\boldsymbol{a}^{(l)} = \sigma^{(l)} \left((\boldsymbol{W}^{(l)})^T \boldsymbol{a}^{(l-1)} + \boldsymbol{b} \right) \tag{C.1}$$

where the contents of the activation function is usually denoted

$$z^{(l)} = \boldsymbol{W}^{(l)})^T \boldsymbol{a}^{(l-1)} + \boldsymbol{b} \tag{C.2}$$

Here, $\boldsymbol{W}^{(l)}$, is a matrix, while the remaining symbols are either vectors or matrices, and $\boldsymbol{z}$ is the **net input?**. The dimensions of the different parts of (C.1) depends on the structure of the dimension of inputs, $\boldsymbol{x}$, and outputs, $\tilde{\boldsymbol{y}}$, to and from the FFNN. As an example, for a network with one hidden layer, with input $\boldsymbol{x} \in \mathbb{R}^{n \times 1}$, the target $\boldsymbol{y} \in \mathbb{R}^{m \times 1}$, then the weight matrix, $\boldsymbol{W}^{(1)}$, and the biases, $\boldsymbol{b}^{(1)}$ must have dimension $(n \times m)$, $(m \times 1)$, respectively (Hjorth-Jensen, 2023).

Fig.II.1 shows a conceptual representation of the information flow through a FFNN. This project deals with a so-called fully connected FFNN. This means that every unit in the hidden layers are connected to the layer inputs, and the layers output is fully connected to the previous layer (Raschka et al., 2022).

The feed-forward part of the network concept deals with the process of feeding the information from the input layer to the output layer, through the hidden layers. For a network with L layers, this involves transforming the layer input, $\boldsymbol{a}^{(l-1)}$ with the layer weights, $\boldsymbol{W}^{(l)}$, and adding the layer biases, $\boldsymbol{b}^{(l)}$. The resulting net input, $\boldsymbol{z}^{(l)}$, is then feed to the the next layer through the activation function, σ. Repeating the process for all hidden layers gives the network its structure, as well as the final network output as

$$\tilde{\boldsymbol{y}} = \boldsymbol{a}^{(L)} = \sigma^{(L)}\left(z^{(L)}\right) \tag{C.3}$$

It is now possible to compare the network output with the target. This comparison utilizes a cost function, $C = C(\tilde{\boldsymbol{y}}, \boldsymbol{y})$, giving a measure of the network performance. Examples of cost functions was introduced above, in Sec.II.B. Stopping here would most likely result in a poor performance, if the weights and biases in the network does not contain any special information of the problem. The strength of the FFNN concept lies in the network training through an algorithm called **backpropagation**, which updates the weights and biases in the network using information of the performance from the cost function by calculating the various gradients for the network.

2. Backpropagation

The backpropagation algorithms attempts to make use of the information gained of the network performance by stepping back through the network carrying this information in the gradients of the different weight matrices and biases vectors in the different layers. The following discussion and expressions are based on the derivation made in ch. 13 in (Hjorth-Jensen, 2023).

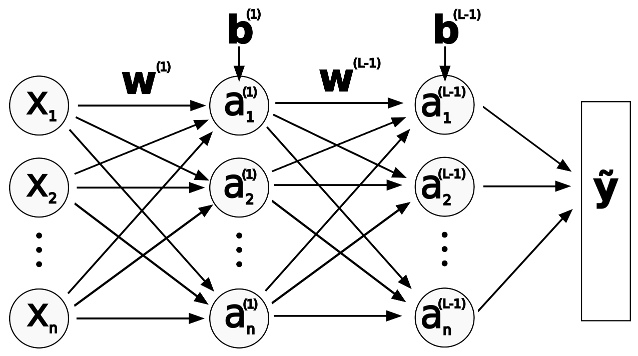

FIG. II.1: Conceptual figure of the feed-forward structure of a neural network. Based on figure 11.2 (p.339) in (Raschka et al., 2022)

By applying the chain rule in an appropriate way, it is possible to create expressions that will carry this information of the error of the current network prediction back to the first hidden layer. The point of interest is the derivatives of the cost function wrt. the weights and biases, in order to use these to update these components through a SGD-algorithm. For the last layer, L, they can be found, using the chain rule, as

$$\frac{\partial C}{\partial w_{ij}^L} = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L}\frac{\partial z_j^L}{\partial w_{ij}^L} \tag{C.4}$$

and

$$\frac{\partial C}{\partial b_j^L} = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L}\frac{\partial z_j^L}{\partial b_j^L} \tag{C.5}$$

Here it is convenient to define the output error, δ_j^L as

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} \tag{C.6}$$

to make the expression more compact.

For the general hidden layer, l, the derivatives take the form

$$\frac{\partial C}{\partial w_{ij}^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{ij}^l} \tag{C.7}$$

and

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial b_j^l} \tag{C.8}$$

For consistency, lets also to define the hidden layer error, δ_j^l, as

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \frac{\partial C}{\partial a_j^l}\frac{\partial a_j^l}{\partial z_j^l} \tag{C.9}$$

Now, there is a need to relate the hidden layer error to the previous layer output, such that it can be carried backwards through the network. That can be done by using the chain rule, and writing,

$$\delta_j^l = \sum_k^{M_{l+1}} \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l} \tag{C.10}$$

with M_{l+1} as the number of nodes in layer $(l+1)$. The last part of this can be written as

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = \frac{\partial z_k^{l+1}}{\partial a_j^l}\frac{\partial a_j^l}{\partial z_j^l} \tag{C.11}$$

With this, it is now possible to find δ_j^l as

$$\delta_j^l = \sum_k^{M_{l+1}} \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial z_j^l} \tag{C.12}$$

Breaking this down, using the relations set-up in the feed-forward part, expressions for each derivative can be found as

$$\frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} = \left(\frac{\partial}{\partial z_j^{(l)}} \, \sigma^{(l)}(z_j^{(l)}) \right) \tag{C.13}$$

$$\frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} = 1, \qquad \frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}} = a_i^{(l-1)} \tag{C.14}$$

and

$$\frac{\partial z_k^{l+1}}{\partial a_j^l} = w_{kj}^{l+1} \tag{C.15}$$

Substituting these into the expressions for the output and hidden layer errors gives

$$\delta_j^L = \left(\frac{\partial}{\partial z_j^{(l)}} \, \sigma^{(l)}(z_j^{(l)}) \right) \frac{\partial C}{\partial a_j^L} \tag{C.16}$$

$$\delta_j^l = \sum_k^{M_{l+1}} \delta_k^{l+1} w_{kj}^{l+1} \left(\frac{\partial}{\partial z_j^{(l)}} \, \sigma^{(l)}(z_j^{(l)}) \right) \tag{C.17}$$

Using these in Eq.(C.4) and (C.5), the gradients need for the backpropagation update become

$$\frac{\partial C}{\partial w_{ij}^l} = \delta_j^l a_i^{l-1} \, , \qquad \frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{C.18}$$

which holds for all layers $l = 1, \ldots, L$. Armed with these expressions, the gradients of the weights and biases can be computed for each layer, going from layer L to the first hidden layer. This also makes it possible to use a SGD-algorithm to update the weights and biases in each layer based on information from the network output. For layer l, the update becomes

$$w_{ij}^l = w_{ij}^l - \eta \frac{\partial C}{\partial w_{ij}^l} \; = \; w_{ij}^l - \eta \delta_j^l a_i^{l-1} \tag{C.19}$$

and

$$b_j^l = b_j^l - \eta \frac{\partial C}{\partial b_j^l} \; = \; b_j^l - \eta \delta_j^l \tag{C.20}$$

These updates can be implemented using any of the methods presented in Sec.II.A.1

3. Automatic Differentiation

The gradient expressions from Sec.II.C.2 needs to be implemented in a FFNN, and the derivatives of the layer activation function must, in general, be written out explicitly for implementation into the network. For deep networks with complicated activation function, this is a time-consuming task, prone to introduce errors and bugs in the expressions and codes. The method of **Automatic Differentiation** (AD) is a method that repeatedly applies the chain rule on a set of well-known elementary operations where the derivatives are known to compute derivatives to numerical precision (Baydin et al., 2018).

The backpropagation process above is a specialized case of the AD-process known as *reverse mode*, where derivatives at the end of the chain are evaluated first, the the process is propagated backwards. The opposite mode is the *forward mode*, where the first derivative in the chain is evaluated first, and then stepped forward through the chain. The reverse mode is computationally less costly, especially for large inputs (Baydin et al., 2018). It is possible to use this in an implementation of a FFNN in a way that allows for inputting the expression of a cost function defined using the feed-forward pass, and then automatically differentiate wrt. to the layer weights and biases to obtain the various gradients (Hjorth-Jensen, 2023).

4. Activation functions

Each of the layers in the FFNN are connected through activation functions. These are usually non-linear functions that scales the input to the function, giving a scalar as a output. In the presentation above, the activation functions, σ, take the net input, $\boldsymbol{z}^{(l)}$, as input, and gives the next layer input, $\boldsymbol{a}^{(l)}$ as output.

The main reason for using activation functions between the layers, is to introduce non-linearity into the network. Inside each of the layers, the net input is a linear function consisting of matrix-vector multiplication and addition. Feeding this forward directly, or using a linear function, will then lead to a network that is connected by one linear transformation after the other. This means that a equivalent, linear network with only one hidden layer, no matter how many layers the original network consists of (Bishop, 2006).

Two examples of popular activation functions are the **Sigmoid**-function, taking a similar form as in Eq.(B.3) and the **Rectified Linear Unit** (ReLU)-function, defined as

$$\mathrm{ReLU}(z) = \sigma_{\mathrm{R}}(z) = \begin{cases} z & \mathrm{if} z > 0 \\ 0 & \mathrm{if} z \leqslant 0 \end{cases} \tag{C.21}$$

and Fig.II.2 shows these two over a suitable domain to show their behavior (Raschka et al., 2022).

Their behavior makes them suitable for different purposes. The Sigmoid-function, as it effectively outputs values between $(0, 1)$, usually makes it a good choice when dealing with probabilities, as this keeps the layer inputs in the target range when the targets are probabilistic.

One drawback to this function is that it may result in vanishing gradients, as the exponential in the function introduces an asymptotic behavior for large positive and negative values. An an example, a $z = 15$ gives $\sigma_S(15) = 0.9999$, and $z = 24$ gives $\sigma_S(24) = 1.0$. Since these are used in the calculation of the gradients of the weights and biases in the backpropagation algorithm, the asymptotic behavior will give little to no change(Raschka et al., 2022).

Another issue with σ_S that may limit its usability is the narrow range give strictly positive values. An alternative to the sigmoid function is the hyperbolic tangent, $\tanh(z)$. This has a broader range, $(-1, 1)$, and it does not limit the output to only positive values. App.A.2 shows this function, along with others that has been used in the analysis.

The ReLU-function scales the output linearly for values larger than zero, and forces the other values to zeros. This has effect that the activations in the layers become non-negative, and since it's derivative is always 1 for positive values, it eliminates the issue with vanishing gradients, making it more suitable for deeper network, and where the activations tend to become small (Raschka et al., 2022).

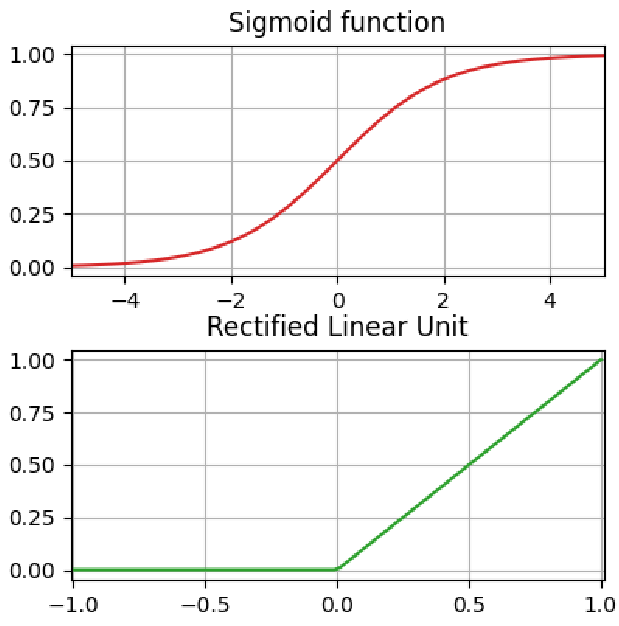

FIG. II.2: The Sigmoid and Rectified Linear Unit (ReLU)-functions in the ranges $(-5, 5)$, and $(-1, 1)$. The Sigmoid-function scales the variable to a value between $(0, 1)$, while the ReLU-function terminates all values equal to or below zero.

A last comment regarding the choice activation function for the output layer. The aim is to have an output that can be compared to the target. For a classification problem, the output should often be a probability. For the other type of problem in this study, a regression problem, forcing the output values into a certain region will defeat the purpose. The usual approach here is to simply to have no activation function for the output layer (Raschka et al., 2022).

D. Model selection and assessment

The goal of the methods described in the previous sections is to obtain the best possible prediction of a systems behavior based on input parameters. The algorithms this project uses to create these predictions contain a multitude of parameters with various impacts on the model performance.

The notion of **model-selection** refers to using different methods to figure out the best possible parameters for a given dataset. Examples of such methods are **k-fold cross-validation, bootstrapping** and **grid searches**.

The **model assessment** then refers to how well the resulting model performs against test datasets. This is a quantitative assessment, looking at different error metrics to check how much the final prediction deviates from the "real" test case (Hastie et al., 2009). Due to the nature of the two types of analysis this project presents, it is necessary to define separate measures for the two.

1. Grid search

The k-fold cross-validation and bootstrap method was extensively covered in my report on linear regression approximation[2]. Some of the key points from that report can be found in App.B. A general grid search method is a straightforward process that checks model performance for different combinations of the model hyperparameters. An an example, lets consider the SGD-algorithm with momentum. Choosing a cost function with regularization

[2]See `https://github.com/andersthorstadboe/project-1-fys-stk4155-lin-regression/tree/main/02-report` for more details

gives

$$\hat{C}(\boldsymbol{\theta}) = C(\boldsymbol{\theta}) + \lambda||\boldsymbol{\theta}||_2^2 \tag{D.1}$$

where $C(\boldsymbol{\theta})$ denotes the original cost function, and $\hat{C}(\boldsymbol{\theta})$ is the new cost function with added L^2-regularization for some scalar $\lambda \geqslant 0$. In the GD-algorithm, the learning rate, η, momentum, γ, mini-batch size, M, and number of epochs also shows up as parameter that will have an effect on model performance (Raschka et al., 2022).

Choosing lists of values for these parameters, the method then asks to run the algorithm with each possible combination of the items of the lists. By collecting measures of model performance of each of these runs, the user can choose optimal combinations of hyperparameters. In practice, the method compares two hyperparameters while other model parameters are kept fixed. (Raschka et al., 2022).

Choosing what combinations to pick for each run, can be done either by sequentially picking one values from the first parameter, and iterating over the entire list of the second hyperparameter. This is what is usually called a *grid search*, as it keeps picking parameters from a grid fixed before the search. The picking procedure can be randomized by choosing two and two hyperparameters from a distribution of values in the desired range. This method is usually named *randomized grid search* (Raschka et al., 2022).

The advantage of this is that it may increase the probability of picking a pair that "lands" at the optimal setting, rather than hoping that the chosen grid also falls within this setting. This may be a more efficient method, as it takes away the need for a refined fixed grid. Combining the grid search with a cross-validation method, such as the k-fold, allowing for better model comparison (Raschka et al., 2022).

2. Model assessment measures

The datasets the project uses are one sampled from a continuous function, and one attempting to classify binary outcomes, there is a need for different types of measures. Common measures for the continuous data is the *mean squared error* (MSE), presented in Eq.(B.1), which has the optimal values zero, and the R^2-score, which is defined as

$$\mathrm{R}^2(\boldsymbol{y}, \tilde{\boldsymbol{y}}) = 1 - \frac{\sum_{i=0}^{n-1}(y_i - \tilde{y})^2}{\sum_{i=0}^{n-1}(y_i - \bar{y})^2} \tag{D.2}$$

where $\bar{y}$ is the mean value of the dataset $\boldsymbol{y}$. The best possible score is $\mathrm{R}^2(\boldsymbol{y}, \tilde{\boldsymbol{y}}) = 1$.

For the classification data, which is binary, a couple of different measures are more informative. One of these is the so-called *accuracy-score*, which measures the fraction of correctly classified instances wrt. the target data. This simply compares the network output with the target and counts the number of correct, and returns the fraction of that number with the total number of outputs, making it a rational number between zero and one (Raschka et al., 2022).

An issue with this measure is that is lends no insight into the actual classification. For a binary problem, the distinctions between *true positive* (TP), *true negative* (TN), *false positive* (FP) and *false negative* (FN) labels are important, especially in real-world applications where the outcome may have a real impact. A TP/TN is here a correctly classified instance of 1/True or 0False. The FN's are when the model outputs a 0/False, but the target was 1/True, and vice versa for the FP (Raschka et al., 2022).

Dependent on the actual nature of the system the model attempts to classify, the dangers lies in the number of FP's or FN's. Just using the accuracy score, an accuracy of 0.96 of a test sample of 100 instances, means misclassifying four instances. Say all four instances are FN's. If the model attempts to classify whether or not somebody has a malignant tumor, and the class 1/True denotes a malignant one. Then four FN's means that the model, on average will misclassify 4% of the cases as benign, or not dangerous, while they are, in fact dangerous to the patient. This relationship is usually represented in a *confusion matrix*. Fig-II.3 shows an example of this (Raschka et al., 2022).

Continuing with the TP/FP-distinctions, it is possible to calculate the rate at which the model classifies positive outcomes, and plot this in a so-called *receiver operating characteristic* (ROC)-curve. This seeks to give a measure

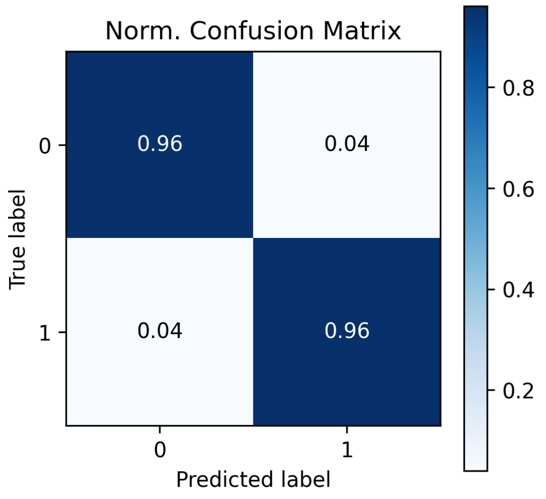

FIG. II.3: This shows an example of a confusion matrix for a binary classification dataset. Here, ...

of how the model is "guessing" the outcome by plotting the TP-rate against the FP-rate, calculated as

$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{FN} + \mathrm{TP}} \qquad \mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}} \tag{D.3}$$

It aims to visualize the rate at which the model correctly classifies a random sample as TP, and the closer this curve is to the vertical axis, the better the model generally performs. The area under the ROC-curve also serves as a performance measure between different kinds of models. Both the rates and area take values between $[0, 1]$ (Developers, 2023).

One last measure of classification models is the *cumulative gains*, which is best explained as a measure giving insight into how efficiently the model captures a certain %-age of the classes wrt. to the total %-age of the instances in the test data. This means that if the model is shown to capture 70% of TP's of just 20% of the total instances, the effectively targets a lot of the positive cases with a fairly limited number of samples for the total population(Sahakyan, 2019).

III. CODES AND CODE DESCRIPTION

This section presents the algorithms and methods implemented to perform the analysis of the two types of data, with the main focus on the feed-forward neural network implementation. It will also present the general program and class structures, how to access the programs and files, and give a short note on some of the software packages that the implementation uses to solve some tasks.

A. Program and class structure

This project organizes its programs in **Jupyter-notebooks** that runs the Python programs, and also allows for text, formulas and comments to be added to the code using the **Markdown**-markup language (Community, 2021).

The general notebook structure is:

1. Imports of supporting files and external packages
2. Setting notebook defaults like figure parameters and random seed default
3. Creation or loading of the dataset for the notebook. This also include splitting of datasets into training and test datasets and scaling
4. Initialization of the main class, like the neural network, with all necessary instance variables and functions
5. Setup of supporting classes and methods
6. Main calculations, parameter study, like a grid search, and figure plots.
7. Comparison with similar methods and studies

The notebook-approach was chosen to be able to annotate and comment the program, and make the re-usability of the program better.

One main neural network class was created as part of this project, which is suppose to be a general purpose FFNN. The same basic approach as described above generates predictions for both types of datasets without any modification other than changing out the activation functions, their derivatives and the choice of cost function.

The class structure follows the general order that was presented in Sec.II.C, with methods for creating layers, doing the feed-forward pass, backpropagation, and training, and calculating the different measures, as well as reset necessary instance variable for reusing the same class in loops.

In addition, two different classes was created that implements a general SGD-approach for linear approximation and classification. They also follow the same basic class structure with methods for scaling the datasets, fitting to the data, generating prediction, calculating measures and plotting results.

A number of supporting classes for computing gradients using the various SGD-algorithms with simple methods for computing the gradient change and resetting variables as needed. This also includes a class using the *Autograd*-package to compute the gradient using automatic differentiation(Maclaurin et al., 2015).

Lastly, a number of different methods are available that defines popular activation function, cost functions, and their derivatives, as well as several method for plotting relevant data for the analysis. All final versions of the program files containing these classes and methods, as well as a selection of notebooks, can be found in the public GitHub-repository created for this project, using this link: `https://github.com/andersthorstadboe/project-2-fys-stk4155-ffneural-network`

B. Analysis algorithms

The algorithms in the section tries to outline the general approach that was taken in the implementation. The classes the algorithm are written like **AClass**, and the class methods like *some_method*.

The following Alg.1 describes the general outline for all the neural network training performed in the project.

Algorithm 1 Algorithm for training the FFNN for a fixed set of hyperparameters, seen both from the program structure, and inside the class.

```
Initialization of dataset, and data split
Scale input (and output) data if necessary
Define network layer structure, activation functions and their derivatives
Init. **FFNNetwork**, call *create_layers*()
Setup of gradient descent parameters, init. **GDMethod**
Call *train_network*() using training dataset as input
**for** the number of epochs **do**
  **for** the number of mini-batches **do**
    Choose random index for mini-batch
    Call *back_propagation*() to start grad. computation
    Call *feed_forward*() to populate layers
    **for all** layers from L → 1 **do**
      Compute gradients of $(\boldsymbol{W}, \boldsymbol{b})$
    **end for**
    **for all** layers **do**
      Call **GDMethod**.*reset*() to reset SGD-variables
      Compute change based on gradient with **GDMethod**.*update_change*()
      Update $(\boldsymbol{W}, \boldsymbol{b})$
    **end for**
  **end for**
**end for**
Calculate final prediction model with *predict*() with test dataset as input
Model assessment using appropriate measures and plots
```

Algorithm 2 The general approach for performing a grid search of hyperparameters for the FFNN

```
Initialization same as in Alg.1
Create list of values to draw **param1**, **param2** from
Create storage array(s) for computed measure(s)
**for all** values in **param1**-list **do**
  **for all** values in **param2**-list **do**
    Call *create_layers*() and init. **GDMethod** with current param's
    Call *train_network*() using training dataset as input to train network
    Call *predict*() with test dataset as input
    Compute measure(s) and store in array(s)
    Call *reset*() to reset instance variables in network
  **end for**
**end for**
Create heatmap plot with the measure-scores for each combination of param's
```

Alg.2 presents the basic approach for performing a grid search using the **FFNNetwork**-class. This is easily modified to perform a randomized grid search by changing out the how the two hyperparameters are draw. As long as this is done in the beginning of the for-loop, the gradient descent class is still initialized with the current choice of hyperparameter pairs.

Lastly, Alg.3 shows the general procedure for running a general SGD-regressor for either linear or logistic regression. This can be combined with the grid search approach in Alg.2 to perform model selection with these implementations as well.

Algorithm 3 The general stochastic gradient descent regressor algorithm this study uses, both the the linear and logistic datasets

```
Initialize dataset and split data into training and test datasets, scale data if necessary
Init. **Regressor** with hyperparameters (and create design matrix if linear dataset)
Setup of GD-parameters, initialize **GDMethod** and **GMethod**
Call *fit*() to start SGD
Generate parameters to optimize from normal dist.
**for** the number of epochs **do**
  **for** the number of mini-batches **do**
    Call **GDMethod**.*reset*() to reset SGD-alg. instance variables
    Choose random index to pick data points for mini-batch
    Compute gradients by calling **GMethod**.*compute_gradient*()
    Update parameters by subtracting output from **GDMethod**.*update_change*()
  **end for**
**end for**
Call *predict*() to calculate measure and plot resulting model to visualize fit/accuracy
```

IV. RESULTS

The results section will present model predictions for both types of data, first the continuous dataset , and the classification dataset. Both parts will show relevant prediction measures, and be compared to models generated from other methods and similar implementations, such as the linear regression analysis from (Bø, 2024), the **MLPRegressor-** and **MLPClassifier-**classes from Scikit-learn, as well as the SGD-algorithms described in Sec.III.B.

The aim is to show how the parameter selection process, e.g. grid search, resampling and also activation function choice, and give a thorough model assessment based on the comparisons.

A. Regression results

The regression analysis uses a dataset generated from the *1st Franke function*, which is a much-used test function for assessing interpolation and regression methods, as its shape has some challenging features. The function domain is $(x, y) \in [0, 1] \times [0, 1]$ (Franke, 1979).

Tab.IV.1 shows the input variables and parameters to the programs running the **FFNNetwork-**, and **LinearRegressor-**codes. All results shown in this section uses the *sigmoid*-function as activation function for the hidden layers, and no activation function in the output layer.

In an attempt to improve the model performance, a grid search was performed, that resulted in the plots in Fig.IV.1. It indicates that the optimal range for the learning rate and regularization is around the middle of the grid. Similar studies was done for Tab.IV.2 show a selection of the different combinations of hyperparameters settings that produced good results.

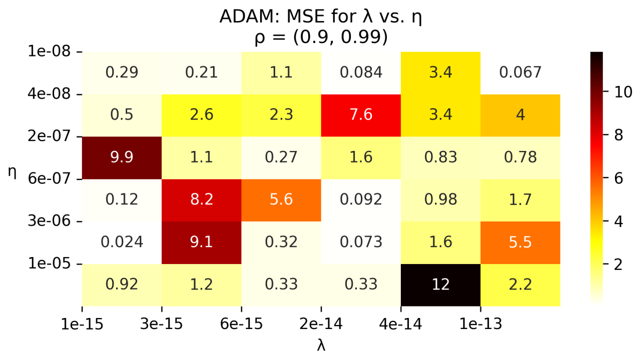

FIG. IV.1: Showing the MSE-results of a grid search for η and λ-values using ADAM as SGD-algorithm. Values indicate that the optimal range is around $\eta = 1 \cdot 10^{-5}$, $\lambda = 2 \cdot 10^{-14}$

TABLE IV.1: Input variables and parameters for the models trying to approximate the Franke function

Variable	**Value**	**Comment**
Domain (x, y)	$[0, 1] \times [0, 1]$	
Grid (Nx, Ny)	(100, 100)	
Test size	0.2	
Layers	[5, 5, 5, 1]	
Learn. rate	$\eta \in [10^{-10}, 10^{-6}]$	
Reg.parameter	$\lambda \in [10^{-16}, 10^{-13}]$	Regularization
Momentum	$\gamma \in [10^{-10}, 10^{-6}]$	
Decay rates	$\rho \in [0.01, 0.9]$	For *RMSprop*
# of batches	$m \in [16, 32, 64]$	
Epochs	$e \in [100, 2000]$	

TABLE IV.2: Three different combinations of hyperparameters that gave the best results after the model selection process. The table also show their corresponding MSE and R^2-score

Param	**Adagrad**	**RMSprop**	**ADAM**
η	$3 \cdot 10^{-14}$	$3 \cdot 10^{-8}$	$1 \cdot 10^{-5}$
λ	$3 \cdot 10^{-13}$	$5 \cdot 10^{-10}$	$5 \cdot 10^{-14}$
γ	$1 \cdot 10^{8}$	-	-
ρ	-	0.9	(0.9, 0.99)
m	32	128	64
e	1000	1000	1000
Measures			
MSE	0.0282	0.0276	0.0301
R^2	0.6789	0.6857	0.6569

The grid search was performed using the ADAM-algorithm for the gradient descent update of the weights and biases. The final model prediction is depicted in Fig.IV.2, and shows that the model is struggling to recreate the shape of the Franke function. The values for MSE and R^2 that the optimal hyperparameter produce also indicate that model is having a hard time reproducing the target-function. To investigate how different activation functions impact the model performance, the same training was repeated using a different set of functions for the hidden layers. Tab-IV.3 shows how this choice impacted the model performance measured using the MSE and R^2-score. This shows that ReLU activation function seems to perform better on this dataset, with a significantly better MSE-value and R^2-score. This can also be seen in Fig.IV.3,

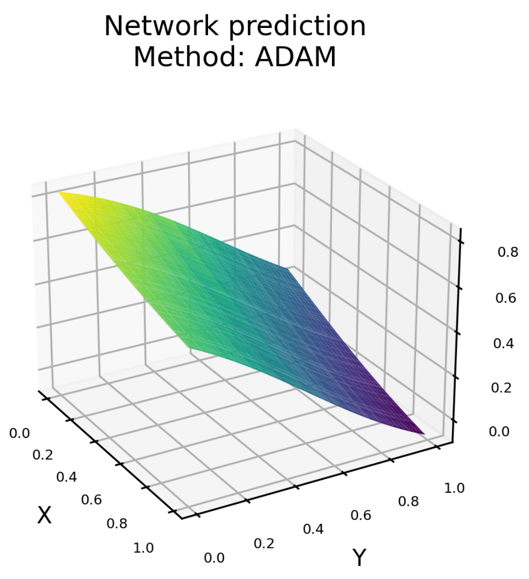

FIG. IV.2: Caption

1. Comparison with other methods

Comparing the performance of the FFNN-models with other well established methods and implementations, will give additional insight into how well the network are able to reproduce the target dataset. Tab.IV.4 shows the MSE and R^2-score from the network, scikit-learn's equivalent class, results from the project's implementation of a SGD-algorithm, and scikit-learn's equivalent class, and

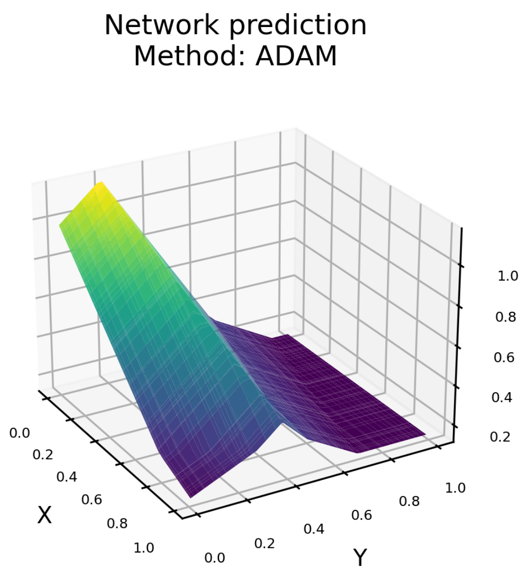

FIG. IV.3: Show a prediction using model trained with ReLU-activation functions in the hidden layers. The recreation of the shape looks better then when using the sigmoid function in the hidden layers.

TABLE IV.3: The MSE and R^2-score for different choices of activation functions in the hidden layers of the network, using ADAM as SGD-algorithm.

Activation func.	**MSE**	**R^2-score**
Sigmoid	0.0325	0.6292
tanh	20.370	-503.9
ReLU	0.0074	0.9160
LeakyReLU	0.0411	0.5307
ELU	24.885	-2032

results from a standard linear regression analysis of the same dataset using the *Ridge*-regression. As the table shows, it becomes clear that the network might be struggling a lot with the dataset, especially when using the sigmoid function.

B. Classification results

For the classification analysis uses the *Wisconsin Breast Cancer Data* as basis for the dataset. This is a dataset consisting of 569 samples with 30 different features used to classify whether or not a tumor with those specific features is malignant or benign, i.e. dangerous or not. Class 0 denotes a malignant tumor, Class 1 a benign. The target is a vector of length 569, with the values $\{0, 1\}$ corresponding to the input samples (Wolberg et al., 1993).

Tab.IV.5 shows the input variables and parameters for the programs running the **FFNNetwork-**, and **LogisticRegressor**-codes. For the classification task, the network runs with *sigmoid*-function as activation functions in the hidden layers, and the *sigmoid* as output layer activation function, as this is a binary classification case. An output scaled to fall between $[0, 1]$ is therefore desirable. App.D.2 shows a collection of confusion matrix, ROC- and cumulative gains curves for some combinations of hyperparameters.

TABLE IV.4: The MSE and R^2-score for the four different implementations creating model of the same dataset using the ADAM-algorithm for SGD. FFNNetwork, LinearRegressor and Ridge are my own implementations.

Method	**MSE**	**R^2-score**
FFNN, Sigmoid	0.0325	0.6292
FFNN, ReLU	0.0074	0.9160
LinearRegressor	0.0047	0.9471
MLPRegressor	0.0012	0.975
Ridge-regression	0.0030	0.9670

TABLE IV.5: Input variables and parameters for the models trying to classify the cases in the Wisconsin Breast Cancer Data

Variable	**Value**	**Comment**
Input shape	(569, 30)	
Target shape	(569, 1)	
Test size	0.1 - 0.2	
Layers	[10, 5, 1]	Last is output layer
Learn. rate	$\eta \in [10^{-4}, 10^{-2}]$	
Reg.parameter	$\lambda \in [10^{-9}, 10^{-4}]$	Regularization
Momentum	$\gamma \in [10^{-10}, 10^{-6}]$	
Decay rate	$\rho \in [0.01, 0.9]$	For *RMSprop*
# of batches	$m \in [16, 64, 128]$	
Epochs	$e \in [250, 1000]$	

The same process for model selection was repeated for the classification analysis. This produced Fig.IV.4, which shows a grid search for η and λ using the RMSprop-algorithm when updating the layers during backpropagation. This shows two areas of interest, namely the far left, and the far right of the grid, with η-values in the middle of the range.

Analysis of plot and similar searches conducted for the Adagrad and ADAM-method resulted in three combinations of hyperparameter that seems to produce good predictions. Tab.IV.6 shows these combinations, as well as different measures of model performance. Here, the metrics accuracy, the area under the ROC-curve and TP/TN-fraction are also listed. The final feed-forward pass with the ADAM produced the confusion matrix in Fig.IV.5, and ROC-curve in Fig.IV.6, and cumulative gains curve in Fig.IV.7. These shows that the model have a dominating TP-rate in the beginning, which is steadily increasing with with population %-age.

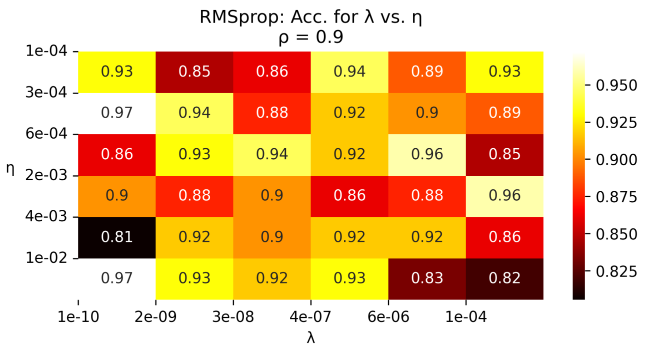

FIG. IV.4: Showing and example of one of the grid searches for η and λ using the SGD-algorithm RMSprop with $\rho = 0.9$. Scoring was done calculating the accuracy.

TABLE IV.6: Three different combinations of hyperparameters that gave the best results after the model selection process. The table also lists their corresponding accuracy, ROC-area-under curve (ROC-AUC) and TP/TN-values

Param	**Adagrad**	**RMSprop**	**ADAM**
η	$3 \cdot 10^{-4}$	$3 \cdot 10^{-4}$	$1 \cdot 10^{-2}$
λ	$3 \cdot 10^{-7}$	$3 \cdot 10^{-6}$	$1 \cdot 10^{-9}$
γ	$1 \cdot 10^{-8}$	-	-
ρ	-	0.9	(0.9,0.99)
m	64	64	16
e	1000	1000	1000
Measures			
Acc.	0.9583	0.9444	0.9583
ROC-AUC	0.9818	0.9575	0.9922
TP / TN	0.98 / 0.92	0.94 / 0.96	0.96 / 0.96

The cumulative gains curve in particular, shows that the model is captures "class 1/Benign"-cases at a higher rate than the "class 0/Malignant"-cases. This is also a case where there is a non-significant fraction of FP-classifications. With 20% of the sample as the test sample, this means that the model wrongly predicts $0.04 \cdot (0.2 \cdot 569) \approx 5$ dangerous tumors as benign, which is not great.

Finally, as with the other test case, a study of the impact from different activation functions in the hidden layer produced the results in Tab.IV.7. It therefore seems that it is the sigmoid activation function that produces the best performing model for this dataset. It should

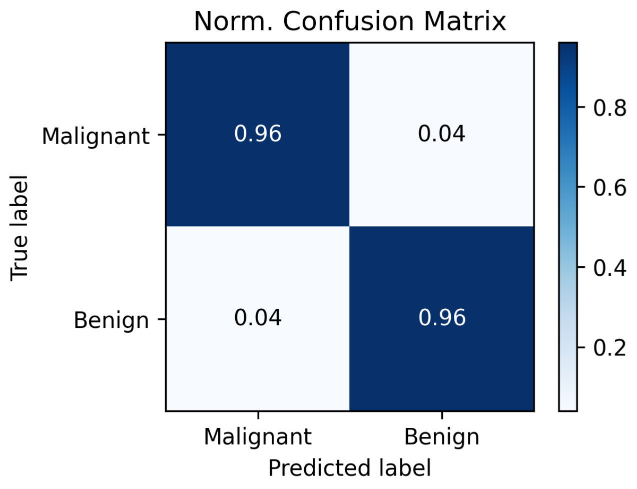

FIG. IV.5: This shows the confusion matrix for the final prediction using hyperparameter combinations from Tab.IV.6 for the ADAM-method

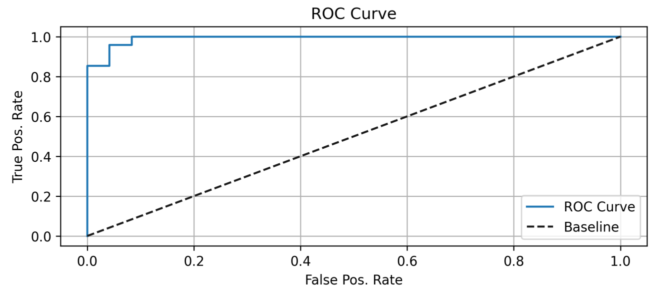

FIG. IV.6: The ROC-curve for the final prediction using ADAM with the optimal hyperparameter settings from Tab.IV.6.

TABLE IV.7: The accuracy and ROC-area-under-curve (ROC-AUC) for different choices of activation functions in the hidden layers of the network

Activation func.	**Acc.**	**ROC-AUC**
Sigmoid	0.9583	0.9835
tanh	0.9444	0.9887
ReLU	0.9444	0.9766
LeakyReLU	0.9028	0.9557

be noted that the other activation functions required a greater regularization to converge. They also performed better as the batch size decreased.

1. Comparison with equivalent implementations

For comparison purposes, the same approach was taken for both datasets. For the classification, the projects own implementations of the FFNN and SGD-algorithm is compared to scikit-learn's equivalent classes, as well as to the baseline accuracy reported for (Wolberg et al., 1993).

Tab.IV.8 shows this comparison for accuracy, ROC-AUC and TP/TN-values. The SGD-implementations are doing a better job than the neural network in this comparison, but only by a small amount. With respect to the baseline report by (Wolberg et al., 1993), all four perform comparably well.

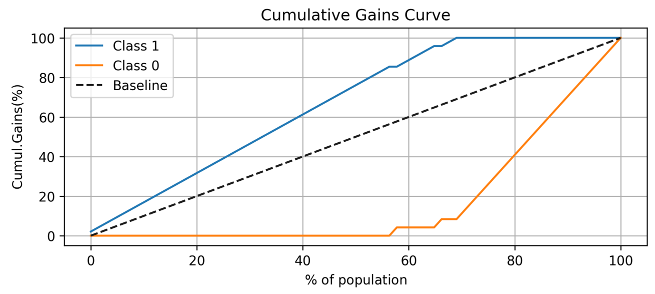

FIG. IV.7: The Cumulative gains curve for both classes for the final prediction using ADAM with hyperparameter as shown in Tab.IV.6.

TABLE IV.8: The accuracy, ROC-area-under curve (AUC) and confusion matrix values for the four different implementations creating model of the same dataset. FFNNetwork and LogisticRegressor are my own implementations.

Method	**Acc.**	**ROC-AUC**	**TP / TN**
FFNNetwork	0.9583	0.9922	0.96 / 0.96
LogisticRegressor	0.982	0.995	1.00 / 0.95
MLPClassifier	0.972	0.969	0.95 / 0.96
SGDClassifier	0.974	0.969	0.99 / 0.95
WBCD-baseline	0.92-0.97	-	-

V. DISCUSSION

This section will go through some of the major points from the results section. In short, it will present some notes on the overall model performance in the analysis, how model parameters impact this performance, such as regularization and learning rate, the role of the activation functions. Lastly, some comments to the suitability of the methods in analysis of these systems will be given.

A. Model performance

For the regression analysis, the overall model performance of the implemented feed-forward neural network is not great. The overall error and R^2-scores does not compare well to a standard linear regression analysis on the same dataset.

In the comparison, a straightforward SGD-analysis seems to do a much better job, with an error an order-of-magnitude lower than the network managed. That said, comparing this to the best performance achieved with Scikit-learn's MLPRegressor-class. It had the best overall performance in the study, so solving this task using a neural network can be done. The reasons for the poor performance of the network may be that the activation functions used are not a good match to the problem. The network was also very shallow, with few nodes in the layers.

For the classification analysis, the implemented network had a good overall performance. The accuracy and ROC-AUC measures is in a comparable range to the other methods it was compared against. It was the SGD-algorithm implemented in this project that performs best on the dataset. The network was still very sensitive to model parameters and the network structure. One reason for this is the amount of input data, which may make the variance of the input data large.

B. Model parameters

Both the network and the SGD-programs introduces a lot of hyperparameter to be studied and tuned. It is observed that small tweaks may have a large impact on model performance. This was especially noticeable for the classification dataset.

For example, choosing too few batches would result in a model that performed what amounts to pure guessing. The size of the dataset is a lot smaller than the regression dataset, which can be sampled as large as the user wants. With that, the network was less sensitive to the number of batches. This highlights the effect introducing stochastic effects can have on the training of the network.

Looking at the two figures showing grid searches in the results, it is clear to see that the model performance is very tightly linked to both learning rate and regularization. The two are also linked, in the way that choosing one while holding the other fixed can make or brake the model. The reason for this is most likely that they play a central role in the gradient descent update algorithm used to update the weights and biases in the layers. Even though methods like RMSprop and ADAM introduces an adaptive learning rate into the method, it was observed that the network still is sensitive to the choice of these two.

For both datasets, λ also had the desired effect of scaling down large values. This effect was very visible in the linear regression study in project 1 as well, which shows that having that as a tool in a machine learning toolbox is crucial.

Some attempts was done to try different layer structures than the ones presented in the input tables in the previous section. A too shallow network, e.g. one hidden layer, did make the model performance slightly worse, but it was making it deeper that had the most significant effect.

Going past 3 hidden layers made, for both datasets, the model unusable. The effect was that the model started to give escalating errors, giving predictions that grew. This might have something to do with the activation function that was used in the hidden layer. More on that in the next part.

Altering the size of the individual layers also made the predictions worse. The classification model was especially sensitive to this, and the configuration that Tab.IV.6 shows was one of the only ones that made usable predictions, but the same was also observed in the regression case. Since the classification dataset is fairly small, having many nodes in a layer might lead to overfitting, as the complexity of the layer increases with the number of nodes (Raschka et al., 2022).

C. Activation functions

The activation functions used in the analysis are all well-known and well-used. The sigmoid function in the hidden layers of both model makes a well-performing model, given a suitable setting of the hyperparameters. This was also the case for the ReLU-function.

For the regression case, it gave a model performing an order-of-magnitude better than the sigmoid with the same network configuration. An observation is that the ReLU allowed to modeled to be trained more extensively than when using the sigmoid. In theory, the sigmoid is suppose to be more sensitive to vanishing gradients, which results in network stagnation, as it forces the values in the range (0,1). This did happen in a lot of cases, with an error fluctuating slightly for each training iteration. For ReLU, the training usually converged to a tolerance. As can be seen in Fig.IV.3, this has the effect that the resulting fit resembles the target shape more, but it does produce flat and sharp shapes.

As mentioned above, deeper networks seemed to give escalating errors. A known effect is that many of the common activation functions tend to amplify large gradients in deeper networks. This may be the reason why a shallower network worked better.

D. Method suitability

From the discussion above, the FFNN implemented here is not well suited for a complex function like the Franke function. The SGD-method implemented seems to be better suited for this purpose, but it most be said that it is more specialized than the network. The program allows for setting up a design matrix, and the gradient is calculated with an analytical expression like with a standard linear regression method. This might be one of the reasons it makes better predictions. A prediction on a simpler 2D-exponential function made by the network is shown in App.D.1. Here the network performs a lot better.

The FFNN does seem more suited to a binary classification task. Based on the discussion above, and the results, one reason may be the activation functions. The classification network uses the sigmoid function on all layers, as this generated the best overall model. This scales all weights and biases in the network structures to the range where the output should be.

VI. CONCLUSIONS

This section will try to summarize the main parts of this report, and give some closing remarks with a critical view on the work.

One of the project's main goals was to implement a general-purpose feed-forward neural network (FFNN) that might be used on several different types of data. Here, the two types were data sampled from the continuous 1st Franke function, a regression analysis, and the Wisconsin Breast Cancer Data, which is a classification dataset. The performance of the model was to be assessed against methods and algorithms seeking to do the same general task.

The main trend observed for the regression case is that the FFNN implemented here struggles to create predictions that resemble the target function. Using the *sigmoid*-function in the hidden layers, the model is only able to reproduce the global gradient of the function, and manages to predict the maximum and minimum to a certain degree. The overall shape is better reproduced using the *ReLU*-function in the hidden layer, but it struggled to give the model smoothness. The resulting measures are MSE's of 0.0325 and 0.0074, and R^2-scores of 0.63 and 0.92 with optimal parameters, respectively. The latter performs comparably with similar methods.

The network did perform better with the classification case, being able to achieve an accuracy- and ROC-AUC-score of 0.9583, 0.9835 with *sigmoid*, and 0.9583, 0.9835 with *tanh* as activation functions.

An issue with the network model on this dataset is that it produced around five *false negatives*-predictions from the test data. For this type of data, that is equivalent to classifying a dangerous tumor as harmless. Compared to similar methods, the model performance is similar.

One other major task in this project was implementing two stochastic gradient descent (SGD)-methods, one for each of the datasets. The measures from their predictions is also shown in the results section, and they where able to produce models with a good performance compared to other implementations. For the regression data, the SGD-method did reproduce the shape of the Franke function to a similar degree as the linear regression methods for project 1.

A. Perspectives and Improvements

There are a number of extensions to the methods in the project that can make for better predictions. One issue during the simulations, was stagnation in the training of the network. Doing a more thorough study on other activation functions, or modifications to the ones already in use can be a remedy for the stagnation.

Another area that was not explored, was how the network's layers were initialized. The method now creates the weights and biases as random numbers from a normal distribution. Trying out different strategies here might be worth the effort, especially for making models for regression analysis.

Several SGD-methods, such as ADAM and RMSprop, was implemented as part of this project. One issue in the implementation might be the way the algorithm parameters are reset during the network training. Validating that the methods work as intended may be a worthwhile task.

Lastly, the hyperparameter tuning done for this project was limited to a standard grid search. The plan was to implement additional method for cross-validation, bootstrapping and a randomized grid search, but that was skipped due to the time constraint. Exploring more hyperparameter settings may improve the model performance.

REFERENCES

Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of machine learning research*, 18(153):1–43, 2018.

Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer, New York, 1st edition, 2006. ISBN 978-0-387-31073-2.

Anders Thorstad Bø. Approximation of topographical datasets using linear regression - project 1 - fys-stk4155. Written as part of course FYS-STK4155, delivered 07.10.2024, 2024. URL `https://github.com/andersthorstadboe/project-1-fys-stk4155-lin-regression`.

Executable Books Community. Jupyter book, feb 2021. URL `https://doi.org/10.5281/zenodo.4539666`.

Marc Peter Deisenroth, A. Aldo Faisal, and Cheng Soon Ong. *Mathematics for Machine Learning*. Cambridge University Press, 2020. URL `https://mml-book.com`.

Google Developers. Classification: Roc and auc. `https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc`, 2023. Accessed: 2024-10-28.

Richard Franke. A critical comparison of some methods for interpolation of scattered data. *Calhoun: The NPS Institutional Archive*, 1979.

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

T. Hastie, R. Tibshirani, and J.H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer series in statistics. Springer, 2009. ISBN 9780387848846. URL `https://books.google.no/books?id=eBSgoAEACAAJ`.

Morten Hjorth-Jensen. Introduction to machine learning. `https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html`, 2023. Accessed: 2024-09-27.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. URL `https://arxiv.org/abs/1412.6980`.

Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*, volume 238, page 5, 2015.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Sebastian Raschka, Yuxi (Hayden) Liu, and Vahid Mirjalili. *Machine Learning with PyTorch and Scikit-Learn*. Packt Publishing, 2022.

Raffi Sahakyan. Meaningful metrics: Cumulative gains and lift charts. `https://bit.ly/cumulativegains`, 2019. Accessed: 2024-10-28.

William Wolberg, Olvi Mangasarian, Nick Street, and W. Street. Breast Cancer Wisconsin (Diagnostic). UCI Machine Learning Repository, 1993. DOI: `https://doi.org/10.24432/C5DW2B`.

Appendix A: Details on cost functions and activation functions

1. Cost functions for regression and classification

For regression models, the most common cost function to measure the error of the model output against the target is the *mean squared error* presented in Sec.II.A.1. It looks like this

$$\begin{aligned}\mathrm{MSE}(\boldsymbol{y},\tilde{\boldsymbol{y}}) = C(\boldsymbol{y},\tilde{\boldsymbol{y}}) = \frac{1}{n}\left(\boldsymbol{y}-\tilde{\boldsymbol{y}}\right)^T\left(\boldsymbol{y}-\tilde{\boldsymbol{y}}\right) = \\ = \frac{1}{n}\sum_{i=0}^{n-1}(y_i-\tilde{y}_i)^2 = ||\boldsymbol{y}-\tilde{\boldsymbol{y}}||_2^2\end{aligned} \tag{1.1}$$

where $\boldsymbol{y},\tilde{\boldsymbol{y}}$ is the target and prediction, respectively, and $||\circ||_2^2$ is the L^2-norm. Adding regularization to $C(\boldsymbol{y},\tilde{\boldsymbol{y}})$, expressed with a general L^r-norms, takes the form

$$C(\boldsymbol{y},\tilde{\boldsymbol{y}}) = \frac{1}{n}||\boldsymbol{y}-\tilde{\boldsymbol{y}}||_2^2 + \lambda||\boldsymbol{\theta}||_r^r \tag{1.2}$$

assuming $\tilde{\boldsymbol{y}} = \tilde{\boldsymbol{y}}(\boldsymbol{\theta})$, with $\boldsymbol{\theta}$ as the model parameters. The most common regularization orders are $r\in\{1,2\}$. This has the derivative wrt. to the prediction, $\tilde{\boldsymbol{y}}$, $r=2$, as

$$\nabla_{\tilde{\boldsymbol{y}}}C = \frac{2}{n}(\boldsymbol{y}-\tilde{\boldsymbol{y}}) + 2\lambda\boldsymbol{\theta} \tag{1.3}$$

and $r=1$

$$\nabla_{\tilde{\boldsymbol{y}}}C = \frac{2}{n}(\boldsymbol{y}-\tilde{\boldsymbol{y}}) + \lambda\mathrm{sgn}(\boldsymbol{\theta}) \tag{1.4}$$

For the classification model, the project uses the negative log-likelihood function. This part gives details on the derivation of this function. This project has defined it as

$$\begin{aligned}C(\boldsymbol{y},\boldsymbol{p}) = -\sum_{n=1}^{N}(y_n\log[p(y_n=1)] + \cdots \\ \cdots + (1-y_n)\log[1-p(y_n=1)]) = \\ = -(\boldsymbol{y}^T\log[\boldsymbol{p}] + (1-\boldsymbol{y})^T\log[1-\boldsymbol{p}])\end{aligned} \tag{1.5}$$

where $\boldsymbol{p} = [p_1,\ldots,p_N] = p_n(y_n|x_n\boldsymbol{\theta})$, with $\boldsymbol{y} = [y_1,\ldots,y_N]$, $\boldsymbol{x} = [x_1,\ldots,x_N]^T$, for N as the number of sample. If p_n is given by the sigmoid function, its derivative is then

$$\frac{\partial C}{\partial p_n} = -\sum_{n=1}^{N}\left(\frac{y_n}{p_n}\frac{\partial p_n}{\partial\boldsymbol{\theta}} + \frac{(1-y_n)}{1-p_n}\frac{\partial p_n}{\partial\boldsymbol{\theta}}\right) \tag{1.6}$$

The derivative $\partial_{\boldsymbol{\theta}}p_n$ is

$$\frac{\partial p_n}{\partial\boldsymbol{\theta}} = \frac{x_n\exp(-\boldsymbol{\theta}^Tx_n)}{(1+\exp(-\boldsymbol{\theta}^Tx_n)^2} = p_n(1-p_n)x_n \tag{1.7}$$

with

$$\begin{aligned}(1-p_n) = 1 - \frac{1}{1+\exp(-\boldsymbol{\theta}^Tx_n)} = \\ = \frac{1+\exp(-\boldsymbol{\theta}^Tx_n)-1}{1+\exp(-\boldsymbol{\theta}^Tx_n)} = \frac{\exp(-\boldsymbol{\theta}^Tx_n}{1+\exp(-\boldsymbol{\theta}^Tx_n)}\end{aligned} \tag{1.8}$$

This gives the final expression for the derivative of $C(y_n,p_n)$ as

$$\begin{aligned}\frac{\partial C}{\partial p_n} = -\sum_{n=1}^{N}\left(p_n(1-p_n)x_n\left[\frac{y_n}{p_n}+\frac{(1-y_n)}{1-p_n}\right]\right) = \\ = -\sum_{n=1}^{N}(y_n(1-p_n)x_n + (1-y_n)p_nx_n = \\ = -\sum_{n=1}^{N}(y_n-p_n)x_n = -\boldsymbol{X}^T(\boldsymbol{y}-\boldsymbol{p}) = \nabla_{\boldsymbol{p}}C\end{aligned} \tag{1.9}$$

As with the MSE, it is also possible to add regularization to this cost function, to end up at the general expression for a regularized cost function as given in Eq.(D.1)

$$\hat{C}(\boldsymbol{\theta}) = C(\boldsymbol{\theta}) + \lambda||\boldsymbol{\theta}||_r^r \tag{1.10}$$

for $r \in 0,1,2$. Setting r $=$ 0 then simply gives the un-regularized cost function. These expressions are implemented in the program such that the FFNN and SGD-algorithms can use these with or without regularization.

2. Activation functions

Two activation function was presented in the main part of the report. In addition, other popular activation functions have been added and used to compare how the model performance changes with the choice of function. The full list of activation functions used, along with their derivatives, are:

The Sigmoid-function

$$\sigma_S = \frac{1}{1+\mathrm{e}^{-z}},\quad \partial_z\sigma_S = \frac{\mathrm{e}^{-z}}{(1+\mathrm{e}^{-z})^2} \tag{2.1}$$

The Hyperbolic tangent-function

$$\tanh(z) = \frac{\mathrm{e}^{2z}-1}{\mathrm{e}^{2z}+1},\quad \partial_z\tanh(z) = \mathrm{sech}^2(z) \tag{2.2}$$

The ReLU-function

$$\sigma_{\mathrm{R}}(z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{if } z \leqslant 0 \end{cases}, \; \partial_z \sigma_R = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leqslant 0 \end{cases} \quad (2.3)$$

The Leaky ReLU-function, $\alpha \geqslant 0$

$$\sigma_{\mathrm{LR}}(z) = \begin{cases} z & \text{if } z > 0 \\ \alpha z & \text{if } z \leqslant 0 \end{cases}, \; \partial_z \sigma_{LR} = \begin{cases} 1 & \text{if } z > 0 \\ \alpha & \text{if } z \leqslant 0 \end{cases} \quad (2.4)$$

The Exponential LU-function, $\alpha \geqslant 0$

$$\sigma_{\mathrm{E}}(z) = \begin{cases} z & \text{if } z > 0 \\ \alpha(\mathrm{e}^z - 1) & \text{if } z \leqslant 0 \end{cases}$$
$$\partial_z \sigma_E = \begin{cases} 1 & \text{if } z > 0 \\ \alpha \mathrm{e}^z & \text{if } z \leqslant 0 \end{cases} \quad (2.5)$$

The two variants of the ReLU-function seeks to not just disregard the values of the activation input, z_j^l, that are negative, but allow them to contribute. The default is to set α to a small value, so that the negative values contribution are kept small (Hjorth-Jensen, 2023).

Appendix B: Additional validation methods

For the purposes of model selection, this project also uses two other methods in addition to the grid search presented in Sec.II.D. These are the *bootstrap* method and *k-fold cross-validation*. Both these are so-called **resampling** methods. The following gives a quick rundown of the idea behind the methods, as well a short note on how they were implemented. The descriptions are taken from and adapted from my own report (Bø, 2024).

The bootstrap method leverages the *central limit theorem* by resampling a dataset a number of times and creating a model using a "new" dataset. The expectation is then that the variance of the predictions should tend towards the actual variance of the system, had the dataset been much larger. This may be an effective technique for small datasets, where the variance usually is too big to capture the actual trends in the system (Bishop, 2006).

By using the concept of a **Bias-Variance trade-off**, mean values of the model *bias*- and *variance*-functions for each of the iterations are then calculated. Repeating the bootstrap method with several different model parameters, it should be possible to identify a certain range of value of that parameter that should minimize the overall error of the predictions. The hope is that choosing parameters in that range will generalize the model performance for new datasets collected from the same system (Bishop, 2006).

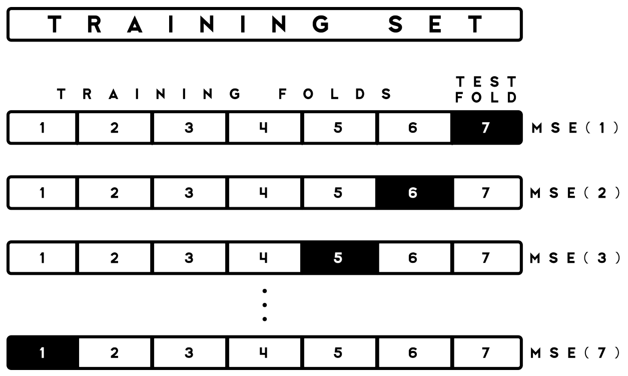

FIG. B.1: A conceptual illustration of how the k-fold algorithm works, and how the training data is divided into seven different folds (based on figure on page 117 from (Raschka et al., 2022)).

The k-fold cross-validation technique applies a different type of resampling method. It divides the training dataset into k so-called *folds*, with $k-1$ training folds, and one test fold. The training then happens on the training folds, and the evaluation of the prediction uses the test fold. The process repeats k-times, as shown in Fig.B.1, each iteration calculating an error for that fold. In the end, the individual error estimates is combine to give a average performance of the model. The splitting is done without replacement, ensure that the training and evaluation sets does not have overlapping values (Raschka et al., 2022).

As with the bootstrap method, the k-fold cross-validation is usually performed on a set of model parameters, such as the regularization hyperparameter, and individual *validation scores* are calculated for each iteration. Plotting the scores wrt. to the hyperparameter, then should give insight into what the optimal value-range for this parameter.

Appendix C: Adaptive learning rate methods

This section is based on the equations and descriptions from ch. 8 in (Goodfellow et al., 2016).

The Adagrad introduces a term named the *accumulated squared gradient*, $\boldsymbol{r}$, that is updated each iteration by adding the *Hadamard* or *element-wise* product of the current gradient. The actual gradient update is then done using a scaled η based on $\boldsymbol{r}$. Denote the gradient, $\boldsymbol{g}$, by

$$\boldsymbol{g}_i = \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}_i) \quad (0.1)$$

Then for each iteration, i, the $\boldsymbol{r}$ updates as

$$\boldsymbol{r}_i = \boldsymbol{r}_{i-1} + \boldsymbol{g}_i \odot \boldsymbol{g}_i \quad (0.2)$$

The then scales the η in the parameter update as

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \frac{\eta}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}_i \tag{0.3}$$

where the small constant δ ensures numerical stability by avoiding division by zero.

The RMSprop extends the approach of Adagrad by adding a *decay rate* parameter, ρ to the update of $\boldsymbol{r}$ to leverage the effect on an exponentially decaying average. This makes the contributions from gradients in the beginning of the iterations less and less relevant as i grows. This modifications is suppose to allow for better performance in non-convex settings. The modification to $\boldsymbol{r}$ is

$$\boldsymbol{r}_i = \rho \boldsymbol{r}_{i-1} + (1-\rho)\boldsymbol{g}_i \odot \boldsymbol{g}_i \tag{0.4}$$

while the parameter update remains the same.

Lastly, the ADAM-algorithm applies a more substantial update to the standard SGD-algorithm. It keeps $\boldsymbol{r}$, and adds a first moment $\boldsymbol{s}$ variable taken as

$$\boldsymbol{s}_i = \rho_1 \boldsymbol{s}_{i-1} + (1-\rho_2)\boldsymbol{g} \tag{0.5}$$

where ρ_1 is the first moment decay rate, usually with $\rho_1 \neq \rho$. In addition to this, a time step, t, which is incremented for each mini-batch. To scale η, the approach then calculates the biases, $\hat{\boldsymbol{s}}$ and $\hat{\boldsymbol{r}}$ as

$$\hat{\boldsymbol{s}} = \frac{\boldsymbol{s}}{1-\rho_1^t} \quad , \quad \hat{\boldsymbol{r}} = \frac{\boldsymbol{r}}{1-\rho^t} \tag{0.6}$$

that in turn is used in the parameter update approach as

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \eta \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta} \odot \boldsymbol{g}_i \tag{0.7}$$

These three methods are almost always used in the SGD-setting, i.e. the parameter update is based on the current mini-batch, applied to each mini-batch, and usually also over several epochs.

Appendix D: Additional results

1. Regression dataset

2. Breast cancer dataset