Breast cancer detection using a self-developed Feed-Forward Neural Network code

Frieda Wik

October 2023

1 Abstract

Breast cancer is the most common cancer worldwide, affecting millions of women every year and ranking as a leading cause of mortality [4]. Following the progress in medical technology and the availability of extensive medical data, machine learning techniques have emerged as promising tools in breast cancer diagnosis. This study focuses on developing a Feed-Forward Neural Network (FFNN) code to predict tumor types from histopathological data, utilizing the well-established Wisconsin Breast Cancer dataset. This dataset includes measurements from breast mass biopsies and is often used as a benchmark in machine learning and cancer research for distinguishing between benign and malignant tumors.

Our best FFNN model outperformed the logisistic regression model in terms of accuracy and misclassification rates. The accuracy score reached 99.3 % for the FFNN model, compared to 97.8 % for logistic regression. The FFNN classified all malign tumors correctly, while the logistic regression resulted in one out of 54 cancerous tumors being misclassified as benign. Although this is not a large number of errors, the consequences could be fatal for the patient concerned if it leads to delayed treatment. It is therefore worthwhile striving for a non-erroneous classification tool even though the complexity of the model increases. For the false positive rates, the FFNN only resulted in one out of 89 benign tumors being classified as malign, while the logistic regression resulted in three benign tumours being misclassified.

2 Introduction

With the advancement of medical technology and the access of abundant medical data, machine learning techniques have emerged as a valuable tool in breast cancer diagnosis. Breast cancer tumors are typically diagnosed either through genomic analysis or histopathological image analysis [7]. The aim of this project was to contribute to the development of the latter method by building a robust classification tool to predict the tumor type from the histopathological data available in the Wisconsin Breast Cancer dataset. The Wisconsin dataset is a well-known benchmark dataset in the field of machine learning and cancer research [5]. It comprises various features extracted from fine-needle aspirate biopsies of breast mass, which can be used to distinguish between benign and malignant tumors. To detect cancerous tumors among the patient data, a fully connected FFNN with flexible number of hidden layers and nodes was developed, including several options for activation functions and learning schedules. An extensive grid search was performed to find the optimal hyper parameters, where all combinations of activation functions and learning schedules were tested over a range of learning rates and regularisation parameters for three different network architectures. Finally, the classification results were compared to those obtained by using logistic regression model.

The neural network model was also tested on a one-dimensional polynomial and compared to Ordinary Least Squares (OLS) regression and Ridge regression, along with a FFNN implemented using the PyTorch library. The same polynomial was used to analyse the effect of different optimisation parameters such as the number of epochs and batch size.

The rest of this report is structured as follows: Section 3 describes the methods used in this project, with an emphasis on explaining the structure of an FFNN. Section 4 describes the results of the breast cancer classification study, along with some hand-picked results showing the performance of the FFNN used for regression and some results showing the behaviour of the optimisation parameters. Section 5 and 6 discusses and concludes the project, and suggests directions for future studies. All in all, this report summarises the knowledge that I have gained through assembling my first FFNN and performing parameter tuning to find the best hyperparameters. Some runs were not presented in the text but can be found in the additional material at the GitHub repository https://github.com/friedawik/FYS-STK4155-.

3 Methods

In this project, a Feed-Forward Neural Network code was developed with interchangeable cost functions to study both regression and classification problems. Several parameters were evaluated and adjusted to increase performance, including learning rate, activation function, hidden function, network architecture and gradient descent method. The best performing model was found by performing a grid search over the relevant hyperparameters.

Figure 1 gives a schematic drawing of a feed-forward network with one hidden layer. The notation introduced below will be used throughout this report. The input data $x = x_1, x_2, ..., x_p$ where p is the number of features is indicated with blue circles. Each feature is multiplied with a weight ${w_{ij}}^l$ indicated by black lines where i and j are the indices of the two nodes that are connected in layer l and $l-1$, respectively. At each layer l, a bias of b_i^l is added to the data, such that every node (indicated as red circlees) receives an input

$$z_i^l = \sum_j {w_{ij}}^l a_j^{l-1} + b_i^l$$

before it is fed into the activation function $a_i^l(z_i^l)$. Later in the report, the matrix notation will be used, where all datapoints are stacked together in one matrix and the N_{l-1} x N_l matrix $\hat{W}$ contains all the weights connecting two adjacent layers. A final transformation is performed at the output layer (yellow circles) to produce the predicted values of the model, here indicated by $y = y_1, y_2, ..., y_k$. These values are then compared to the target values through a cost function and their errors are propagated backwards to update the weight and biases, as described in further in section 3.5. The algorithm for updating the weights and biases as a function of the cost function is called backpropagation.

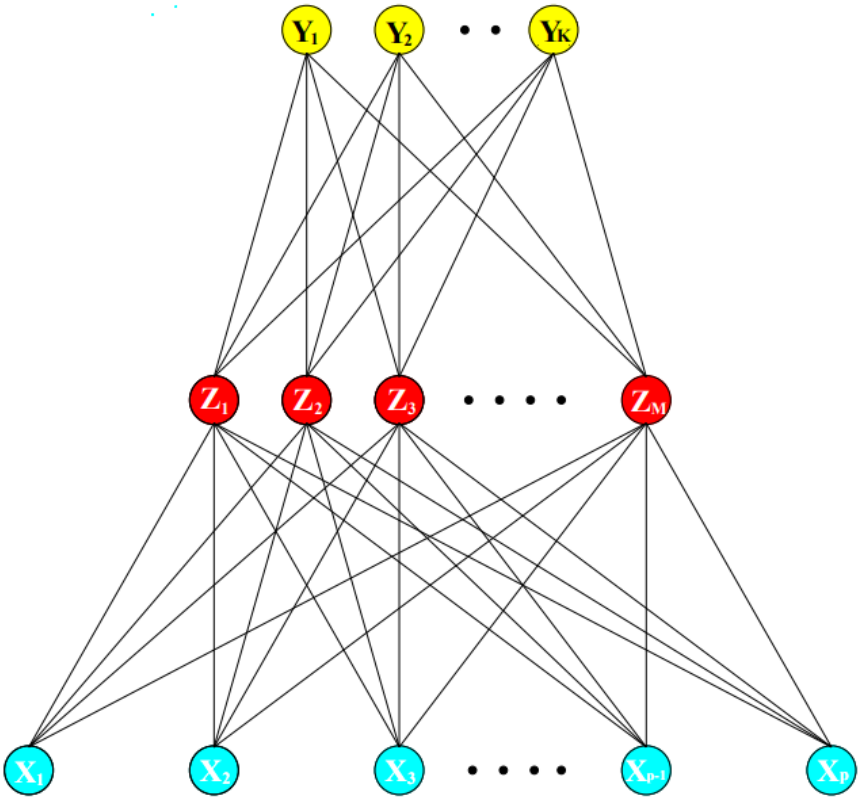


Figure 1: Schematic of a feed-forward neural network with a single hidden layer. Hastie et al., 2009.

Most of the results in this project were obtained by using the FFNN code written by Gregor Kajda and Eric Reber made available through the course lecture notes. This code will be referred to as FFNN K&R throughout this report. This code provided a ready-to-be-run code with all the functionality already implemented. However, I also developed a FFNN code which, although less elegant, could perform most of the same tasks. Since the code by Kajda and Reber was published mid-ways through the project time, the self-written code was not developed further, and therefore lacks certain functionalities, such as the ability of calling functions when initialising the class. Therefore, only a few benchmark runs were prepared were the activation function, output functions and loss functions were defined manually. ChatGPT was used throughout this project and a conversation example can be found following this link: https://chat.openai.com/share/780c8542-0a70-4132-b8e0-dae366028481.

3.1 Collection and pre-processing of data

The Wisconsin Breast Cancer data was imported directly from the Scikit-learn library and scaled with the function *standard_scaler*() prior to performing the analysis. Since the weights of a FFNN are directly affected by the scaling of the inputs, a standardisation is often recommended before analysis such that all inputs are treated equally [2].

The regression analysis was performed on the second-order polynomial given in equation 1. The data was produced with stochastic noise. Since this polynomial is already zero-centered and symmetric around the y-axis, the Ridge regression shrinkage method could be performed without penalising the intercept even if the data was not scaled [2].

$$f(x) = a_0 + a_1 x + a_2 x^2 \tag{1}$$

The datasets were divided into train and test datasets using the Scikit-Learn function *train_test_split*().

3.2 Model architecture

The FFNN code was developed as a fully connected network architecture with a flexible number of nodes and hidden layers. In this type of NN, the information moves only in the forward direction. For a binary classification problem a network with only one hidden layer can often be sufficient while more complex problems may need a deeper network [3]. In this project three different architectures were tested:

1. FFNN with 1 hidden layer with 100 nodes
2. FFNN with 3 hidden layers with 50 nodes each
3. FFNN 3 hidden layers with 101 nodes each

The architectures 1-3 that were studied were chosen such that the complexity ranges from low to high, without running into issues of the models exploding. During the development stage several deeper networks were tested, but they were unstable probably as a result of overfitting of the data. For instance, the architecture 3 was first tested with 100 and 99 nodes, which resulted in most of the runs exploding. However, with 101 nodes the results were overall quite good. The reason for this was not found, but it was a curious phenomena and the architecture was included in the analysis. According to Hastie et al. [2], overfitting due to an excessive number of nodes can usualy be balanced by an increase in the regularisation parameter λ. Therefore, they advice to better use many nodes than too few.

The weight and biases in the FFNN were initialised as random values near zero. At values near zero, the sigmoid function is roughly linear. Through learning, the weights can increase and the function output becomes more and more nonlinear [2].

3.3 Activation functions

After the linear transformation by the weights and biases, the activation functions at each hidden layer introduce non-linearity to the network. Since the backpropagation algorithm includes the derivative of activation function, the choice of activation function will govern the computational and training properties of the nodes [6]. In this project three different activation functions were tested: The sigmoid, ReLU and Leaky ReLU functions.

The sigmoid function has a smooth gradient that allows for efficient backpropagation during training. It has a S-shaped curve that outputs values between 0 and 1 (equation 2). Its derivative (equation 3) will be needed when looking at how the cost function is used in the backpropagation of the error as descibed in section 3.5.

$$a_i = f({z^l}_i) = \frac{1}{1 + e^{-{z^l}_i}} \tag{2}$$

$$\frac{\partial {a^l}_i}{\partial {z^l}_i} = {a^l}_i(1 - {a^l}_i) = f({z^l}_i)(1 - {z^l}_i) \tag{3}$$

One disadvantage of the sigmoid function is that its gradient can become very small when the absolute value of z is large, meaning that it saturates. This can lead to the "vanishing gradient" problem, where the gradients become too small to effectively update the weights during training. Additionally, the sigmoid function is not zero-centered, which can make training slower and less stable. Goodfellow et al. [1] therefore discourage the use of sigmoid altogether, recommending to use hyperbolic tangent activation function if a S-shaped curve is desired.

To mitigate the "vanishing gradient" problem, the ReLU activation function can be used (equation 4). As can be seen by the gradient given in equation 5, the ReLU function is not differentiable at $z = 0$ but gradient descent still performs well enough since the network seldom reaches the local minima of the cost function [1].

$$a_i = f({z_i}^l) = max(0, z_i^l) \tag{4}$$

$$\frac{\partial {a^l}_i}{\partial {z^l}_i} = \begin{cases} 1 & \text{if } {z^l}_i > 0 \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

One potential drawback of ReLU is that it can "die" or become inactive for inputs that result in a negative output. In other words, the slope of the function becomes zero, and the neuron stops learning. To address this issue, variants of ReLU have been proposed, such as Leaky ReLU. With the LReLU, instead of setting negative inputs to zero, the function allows a small, non-zero output for negative inputs [1]. The LReLU with its gradient is given in equation 6 and 7, respectively.

$$a_i = f(z_i^l) = max(cz_i^l, z_i^l) \tag{6}$$

$$\frac{\partial a_i^l}{\partial z_i^l} = \begin{cases} 1 & \text{if } z_i^l > 0 \\ c & \text{otherwise} \end{cases} \tag{7}$$

where c is a small constant that controls the amount of "leakage". When c is set to a very small value, the LReLU behaves similarly to the standard ReLU function for positive inputs, but allows a small, non-zero output for negative inputs. In this project, the constant was set to $c = 10^{-4}$. By allowing non-zero gradients for negative inputs, the LReLU helps to prevent neurons from dying and encourages the model to learn more diverse representations of the data.

3.4 Output functions

Once the feed-forward pass has been completed through all hidden layers, the last activation function, or output function, is reached. Typically, the sigmoid or the Softmax function is used for classification, and the identity function is used for regression [6]. Since the breast cancer problem is a binary problem, the sigmoid function was used as output function for this problem. For the regression analysis the identity function was used.

3.5 Cost functions

Once the feed-forward pass has been completed and the model has produced an output y, the predicted values are compared to the target values t using a cost function. For the breast cancer problem, the cross entropy loss function was used. This is the standard choice for binary classification problems [8]. The cross entropy loss function is given in equation 8.

$$C(\hat{W}^L) = -\sum_{i=1}^{n} t_i \log {a_i}^L + (1 - t_i) \log(1 - a_i)^L \tag{8}$$

Where the $\hat{W}$ matrix is a N_{l-1} x N_l matrix containing all the weights connecting two adjacent layers (here the output layer with the targets), n is the number of nodes in layer l (here equal to the number of targets) and t is the target vector.

Taking the derivative with respect to the output function gives

$$\frac{\partial C(\hat{W}^L)}{\partial {a_i}^L} = \frac{{a_i}^L - t_i}{{a_i}^L(1 - {a_i}^L)} \tag{9}$$

Which when using the sigmoid function as activation function as described in section 3.3 results in the error ${\delta_j}^L$ of the output layer given by equation 10:

$$\delta_j{}^L = f'(z_j{}^L)\frac{\partial C}{\partial a_i{}^L} = a_j{}^L(1-a_j{}^L)\frac{(a_j{}^L - t_j)}{a_i{}^L(1-a_i{}^L)} = a_j{}^L - t_j \tag{10}$$

This error can be propagated back through the hidden layers by the following expression:

$$\delta_j{}^l = \sum_k \delta_j{}^{l+1} w_{kj}^{l+1} f'(z_j{}^l) \tag{11}$$

For the regression problem, the Residual Sum of Squares (RSS) was chosen as cost function. For simplicity, the expression was multiplied with the constant 1/2. For the output layer, the cost function reads as seen in equation 12.

$$C(\hat{W}^L) = \frac{1}{2}\sum_{i=1}^{n}(y_i - t_i)^2 = \frac{1}{2}\sum_{i=1}^{n}(a_i{}^L - t_i)^2 \tag{12}$$

The linear output function, or identity function, was used as output function in the output layer, $a_i{}^L$. Taking the derivative with respect to the output function gives

$$\frac{\partial C(\hat{W}^L)}{\partial a_i{}^L} = a_i{}^L - t_i \tag{13}$$

Which results in the error of the output layer:

$$\delta_j{}^L = f'(z_j{}^L)\frac{\partial C}{\partial a_i{}^L} = a_j{}^L(1-a_j{}^L)(a_j{}^L - t_j) \tag{14}$$

Which again can be propagated back to the hidden layer by the expression given in equation 11.

3.6 Optimization

The backpropagation algorithm involves calculating the gradient of the loss with respect to the weights and biases in the network using the gradient descent method. In this project, both standard gradient descent and stochastic gradient descent were developed and tested on the same simple one-dimensional polynomial that was defined in equation 1. The regression was performed using the FFNN without hidden layer and with the identity function as output function. When using the mean squared error as cost function, this corresponds to OLS regression. In addition, several different types of learning schedules were tested, along with regularisation of different magnitude. The gradient descent method was implemented both using the analytical expression for the gradients and by using the JAX library.

The analytical gradients are typically more accurate and computationally efficient than automatic differentiation, but can be complex to find for some functions. The following section describes how the analytical expression of the gradients were found for the one-dimensional polynomial following the lecture notes of the course [3]:

Given the simple function

$$f(x_i) = y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2$$

We can define the design matrix as:

$$\mathbf{X} = \begin{bmatrix} 1 & x & x^2 \\ . & . & . \\ . & . & . \\ . & . & . \\ 1 & x & x^2 \end{bmatrix}$$

and the OLS cost function as:

$$C_{OLS}(\beta) = \frac{1}{n}\|\mathbf{X}\beta - \mathbf{y}\|_2^2 = \frac{1}{n}\sum_{i=1}^{n}[(\beta_0 + \beta_1 x_i \beta_2 x_i^2)^2 - 2y_i(\beta_0 + \beta_1 x_i \beta_2 x_i^2) - y_i^2]$$

Now we can find the β that minimizes $C(\beta)$ by taking the derivative of the cost function with respect to β

$$\nabla_\beta C(\beta) = \frac{2}{n}\begin{bmatrix} \sum_{i=1}^{n}(\beta_0 + \beta_1 x_i \beta_2 x_i^2) - y_i \\ \sum_{i=1}^{n}(\beta_0 + \beta_1 x_i \beta_2 x_i^2)x_i - y_i x_i \\ \sum_{i=1}^{n}(\beta_0 + \beta_1 x_i \beta_2 x_i^2)x_i^2 - y_i x_i^2 \end{bmatrix} = \frac{2}{n}\mathbf{X}^T(\mathbf{X}\beta - \mathbf{y})$$

Similarly, for Ridge regression,

$$C_{Ridge}(\beta) = \frac{1}{n}\|\mathbf{X}\beta - \mathbf{y}\|_2^2 + \|_\lambda\|_2^2, \quad \lambda \geq 0$$

gives the gradient

$$\nabla_\beta C_{Ridge}(\beta) = \frac{2}{n}(\mathbf{X}^T(\mathbf{X}\beta - \mathbf{y}) + \lambda\beta)$$

These gradients, along with the JAX computed gradients, were applied in the SGD and GD methods as described in algorithms 1 and 2.

The choice of learning rate, defined through a learning schedule, strongly affects the model's performance [1]. In the simplest case, the learning schedule is just a constant. It usually necessary to perform a search for the optimal learning rate for each learning problem [3]. Since the constant learning rate can result in slow learning, the momentum method is sometimes used to speed up the convergence. This method accumulates the gradients, such that the step size depends on previous gradients and their alignment. If many gradients point the same direction, the learning speeds up in this direction[1]. The momentum method was implemented as described in 3.

Algorithm 1 Gradient descent to find optimal β

Define learning schedule
Initialise β
while stop criteria not met **do**
    Compute gradient of cost function $C(\beta)$
    Compute change to β using a learning schedule
    Update β
return Optimal β

Algorithm 2 Stochastic gradient descent to find optimal β

Define learning schedule
Initialise β
while stop criteria not met **do**
    Sample a mini-batch from the full data set
    Compute gradient of cost function $C(\beta)$ on the mini-batch
    Compute change to β using a learning schedule
    Update β
return Optimal β

Algorithm 3 SGD with momentum

Define momentum parameter α
Define learning rate η
Initialise β
Initialise velocity v
while stop criteria not met **do**
    Sample a mini-batch from the full data set
    Compute gradient g of cost function $C(\beta)$ on the mini-batch
    Compute velocity update: $v \leftarrow \alpha v - \eta g$
    Update $\beta \leftarrow \beta + v$
return Optimal β

However, the momentum method has only limited ability to accommodate the fact that some directions in the parameter space are more important for the learning. There exists several methods to adapt the learning rate to the model parameters. The following adaptive learning methods were implemented as described by Goodfellow et al [1].

The Adaptive gradient method (AdaGrad) uses a separate learning rate for each parameter, which is updated based on the sum of the squares of the gradients for that parameter. This means that the learning rate decreases over time for parameters that have large gradients, which can help prevent the algorithm from overshooting the minimum. The AdaGrad method was implemented with and without momentum.

The Root Mean Squared Propagation (RMSprop) uses an exponentially weighted moving average, which means that it doesn't keep the entire history of squared gradients. This can be useful for problems that are non-convex [1].

Finally, the Adam algorithm was used. Goodfellow et al. [1] describe this method as a mix between RMSprop and momentum, where the momentum is added as an estimate of the running average of the gradient. It also includes a bias corrections of the first and second-order moments.

Algorithm 4 The AdaGrad algorithm for adaptive learning rate

Define step size ϵ
Define initial parameter β
Define small number δ for numerical stabilization
Initialize gradient accumulation variable
while stop criteria not met **do**
    Sample a mini-batch from the full data set
    Compute gradient of cost function $C(\beta)$ on the mini-batch
    Accumulate gradients: $\mathbf{r} \leftarrow \mathbf{rg} \odot \mathbf{g}$
    Compute update:$\Delta\beta = -\epsilon/(\delta + \sqrt{\mathbf{r}}) \odot g$
    Update $\beta \leftarrow \beta + \Delta\beta$
return Optimal β

Algorithm 5 The RMSprop algorithm for adaptive learning rate

Define step size ϵ and decay rate ρ
Define initial parameter β
Define small number δ for numerical stabilization
Initialize gradient accumulation variable
while stop criteria not met **do**
    Sample a mini-batch from the full data set
    Compute gradient of cost function $C(\beta)$ on the mini-batch
    Accumulate gradients: $\mathbf{r} \leftarrow \mathbf{r}\rho + (1-\rho)\mathbf{g} \odot \mathbf{g}$
    Compute update: $\Delta\beta = -\epsilon/(\delta + \sqrt{\mathbf{r}}) \odot g$
    Update $\beta \leftarrow \beta + \Delta\beta$
return Optimal β

Algorithm 6 The Adam algorithm for adaptive learning rate

Define step size ϵ
Define exponential decay rates ρ_1 and ρ_2
Define small number δ for numerical stabilization
Initialize 1st and 2nd moment variables
while stop criteria not met **do**
    Sample a mini-batch from the full data set
    Compute gradient of cost function $C(\beta)$ on the mini-batch
    Update t $\leftarrow t + 1$
    Update estimate of first moment $\mathbf{s} \leftarrow \rho_1 \mathbf{s} + (1 - \rho_1)\mathbf{g}$
    Update estimate of second moment $\mathbf{r} \leftarrow \rho_2 \mathbf{r} + (1 - \rho_2)\mathbf{g} \odot \mathbf{g}$
    Correct bias in first moment: $\mathbf{s} \leftarrow \mathbf{s}/(1 - \rho_1^t)$
    Correct bias in second moment: $\mathbf{r} \leftarrow \mathbf{r}/(1 - \rho_2^t)$
    Compute change to $\beta : \Delta\beta = -\epsilon \mathbf{s}/\sqrt{\mathbf{r}} + \delta$
    Update $\beta \leftarrow \beta + \Delta\beta$
return Optimal β

The number of epochs is an important hyperparameter that determines the number of times the training algorithm will iterate over the entire training dataset. In this project, the number of epochs was set to a fixed value to ensure that the training process was reproducible along across the different FFNN set-ups. The two main problems with this approach is that overfitting can be a problem for the cases were convergence occurs early and the model starts to memorize the training data instead, and that the computational cost becomes unnecessarily high in some cases. The number of epochs was initially set to 100 to prevent underfitting, but was indeed lowered to 50 and then further to 10 as it became clear that convergence happened within the first epochs. The accuracy score was plotted against epochs for each run so that it would be possible to inspect the convergence rate. After each loop over learning rate η and regularisation parameter λ, a heatmap was plotted to visualise the best performing combinations.

The number of batches was set to 5, corresponding to a batch size of 85. Larger batches provide a more accurate estimate of the gradient, but with less than linear returns [1]. Another important hyper parameter that impacts the optimisation, is the regularisation term λ. The regularisation is employed to avoid reaching the global minimaliser of the cost function, as this often would imply overfitting the data [2].

3.7 Classification analysis using Neural Network

The classification analysis was performed on the Wisconsin Breast Cancer data available as a built-in Scikit-learn dataset. This data set includes 30 measurements computed from images collected from fine needle aspirates of breast masses that describe the cell nuclei. The binary target corresponds to malignant or benign cancer. The summary statistics of the data set can be found in the jupyter notebook plots_regression.ipynb in the Github repository of the project.

The Pearson correlation matrix is given in figure 2. This matrix displays the pairwise relationships between variables. Positive correlations (values closer to 1) indicate a positive linear relationship, while negative correlations (values closer to -1) indicate a negative linear relationship. Several variables have a strong positive correlation, such as "worst perimeter" and "worst area". This is as expected, since both are measurements of size. The distribution of all variables are shown as histograms in figure 3. Each histogram shows the number of observation that fall within a discrete interval. Some variables, such as "worst concave points" have values that separate well between the target values, and will most probably be important for the classification. Others, such as "smoothness error" have overlapping distribution, and will most probably be less important for the classification analysis.

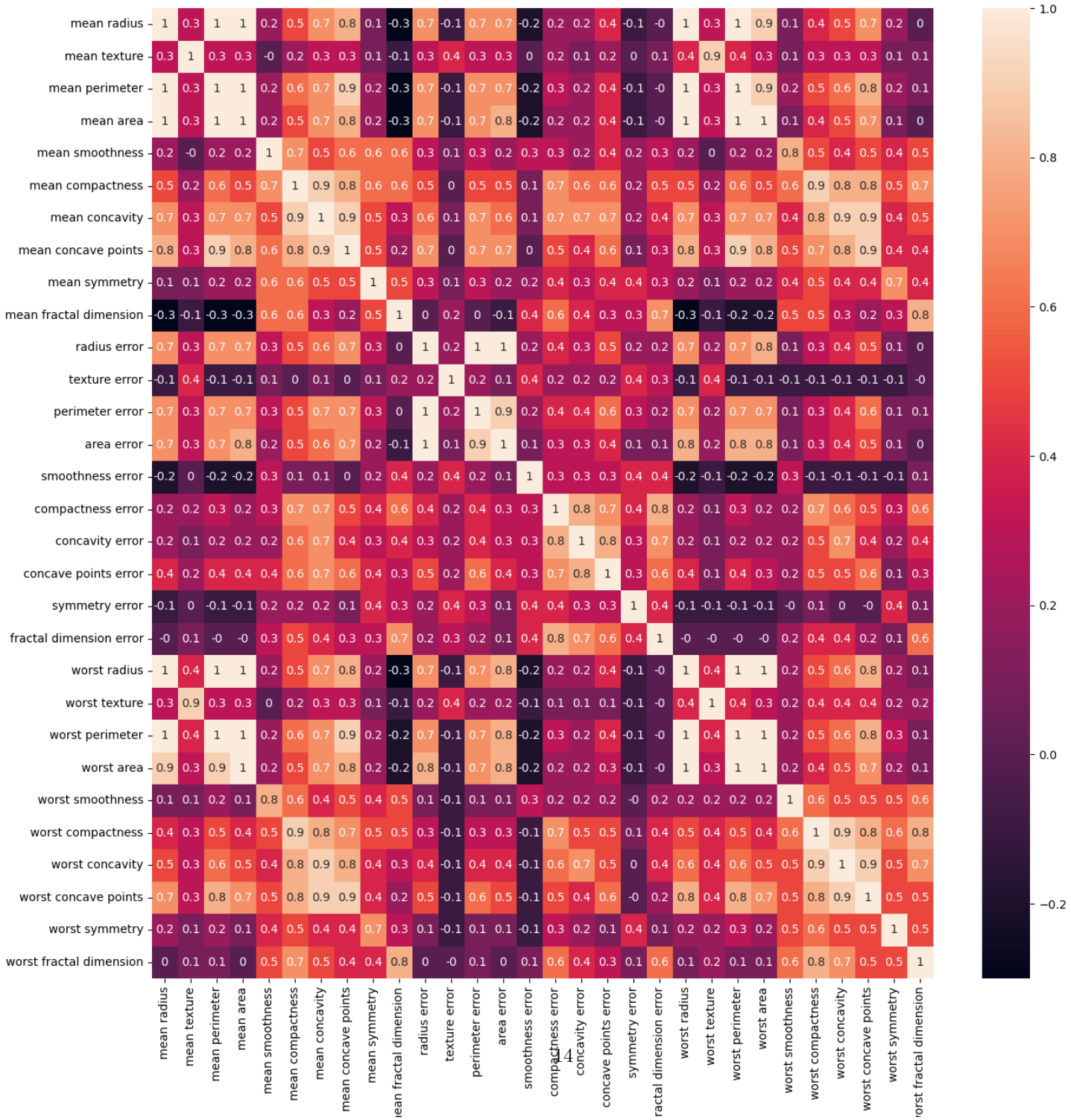
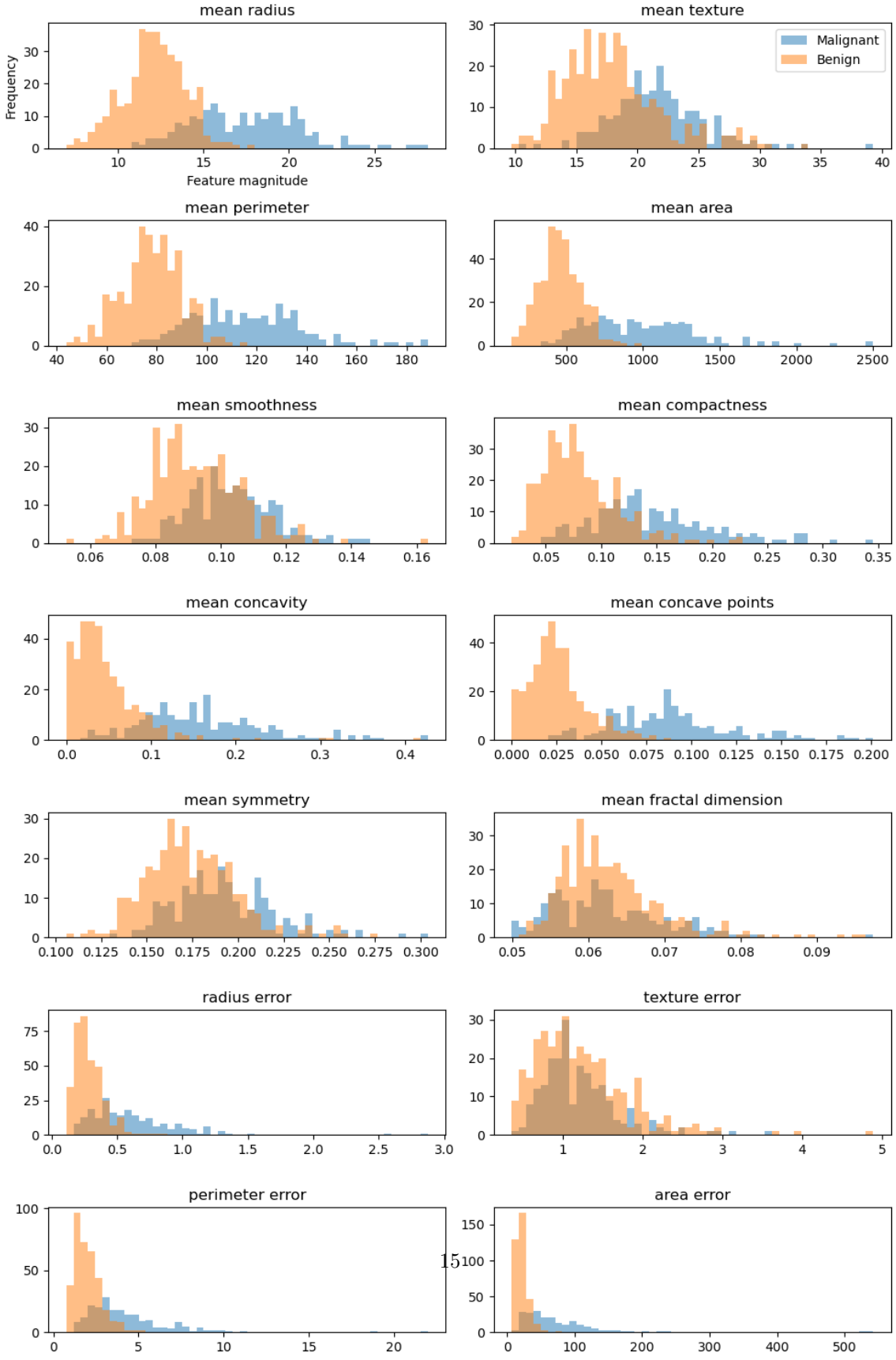


Figure 2: Correlation matrix for cancer data.


mean radius
mean texture
mean perimeter
mean area
mean smoothness
mean compactness
mean concavity
mean concave points
mean symmetry
mean fractal dimension
radius error
texture error
perimeter error
area error
Malignant
Benign
Frequency
Feature magnitude

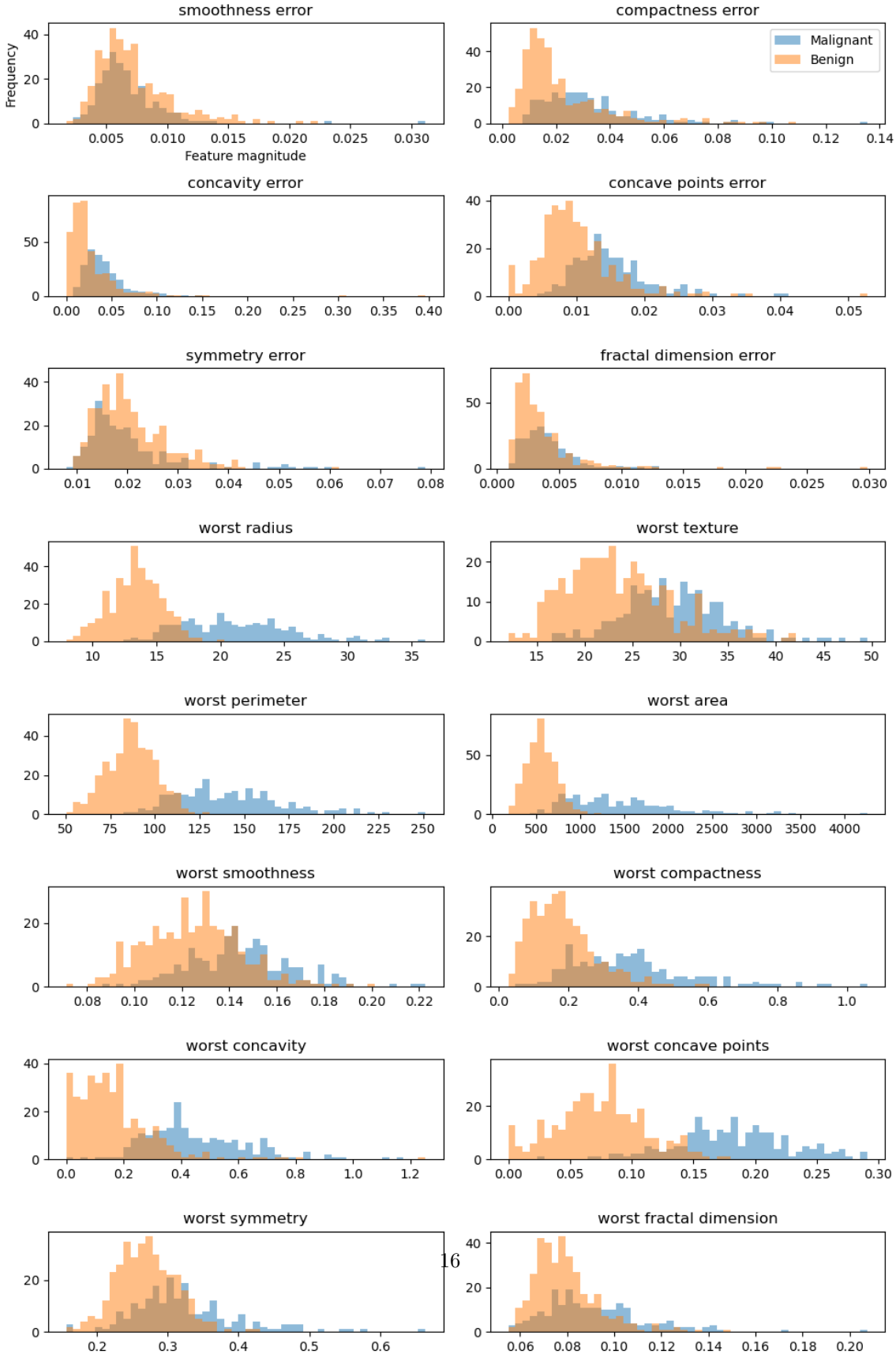


Figure 3: Histograms for the Wisconsin breast cancer data.

The best models for the binary classification of the Wisconsin Breast Cancer Data were found by looping over all three architectures, hidden functions and optimisation methods (see table 3.7) while performing a grid search over different learning rates η and regularisation λ. While doing so, the momentum parameter of the Momentum and AdaGrad w/momentum learning schedules were set to $momentum = 0.3$. After having found the optimal η and λ parameters, a search for the best momentum was performed on the best performing Momentum and AdaGrad w/momentum methods. The results were then updated with the new, best-performing momentum parameter. The sigmoid function was used as output function and the binary cross-entropy loss function was used as cost function for all runs. The results were compared to the results when using logistic regression. The logistic regression model was build by using the FFNN without hidden layers, since this is equivalent to logistic regression [2].

Architecture	Hidden function	Scheduler
1: 1 hidden layer, 100 nodes	Sigmoid	Constant
2: 3 hidden layers, 50 nodes	ReLU	Momentum
3: 3 hidden layers, 101 nodes	LReLU	AdaGrad
		Adagrad w/momentum
		Adam
		RMS-prop

Table 3.7: The model combiantions used for the classification of tumors.

4 Results

4.1 Classification of breast cancer tumors

The best models for the binary classification of the Wisconsin breast cancer data were found by looping over the architectures, activation functions and optimisation methods while performing a grid search over different learning rates η and regularisation λ.

The accuracy scores of the best models of each combination are plotted in figure 4 and the top five best models are described in table 3.7. As can be seen from these presentations, the simplest architecture with only one hidden layer containing 100 nodes reached the highest accuracy score of 99.3 % when using ReLU or LReLU in combination with the momentum learning schedule.

Otherwise, using the sigmoid function as hidden function generally resulted in higher accuracy scores than ReLU and LReLU, where it was most obvious for architecture 3. This

indicates that the sigmoid function was not suffering from the problem of "vanishing gradients" described in section 3.3. The ReLU and Leaky ReLU resulted in identical accuracy scores for architecture 1 and similar scores for architecture 2. For architecture 3 however, the ReLU outperformed the Leaky ReLU for especially the RMSprop method.

The parameter space of the optimal learning rates η and regularisation λ were plotted for each architecture in figure 5. For all models, the optimal learning rate was ranging from $10^{-3} < \eta < 1$ and the optimal regularisation was ranging from $10^{-10} < \lambda < 100$. The values that are touching the bottom of the plot do not need any regularisation since λ is essentially zero at this level. However, for architecture 2 and architecture 3 more regularisation results in higher results, indicating that overfitting otherwise could become a problem with increasing complexity. The models did not display any clear clustering based on learning schedule, which was contrasting to the regression case, where the constant and momentum learning schedule required smaller learning rates than the adaptive learning schedules.

As can be seen from table 6, the two best performing models only differed in type of activation function, meaning that the ReLU and the Leaky ReLU performed equally good. This was confirmed by plotting the convergence of the five models in the table (see figure 7), where the two methods converge in an identical manner. Since the ReLU and LReLU only differ if the input is negative, it is not surprising that they behaved the same way, indicating that all inputs must have been positive.

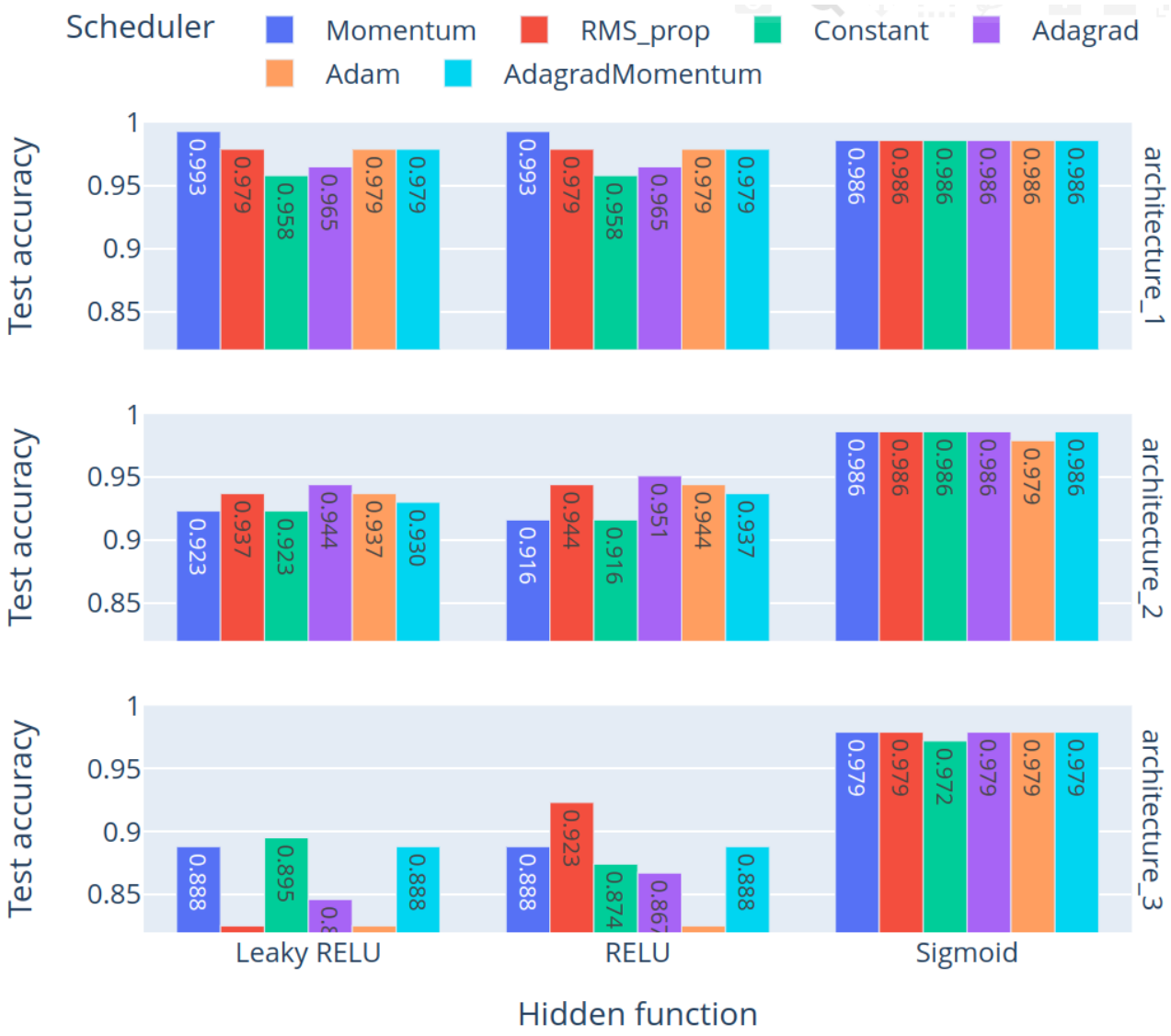


Figure 4: Accuracy score for the best model of each combination of architecture, hidden function and learning schedule.

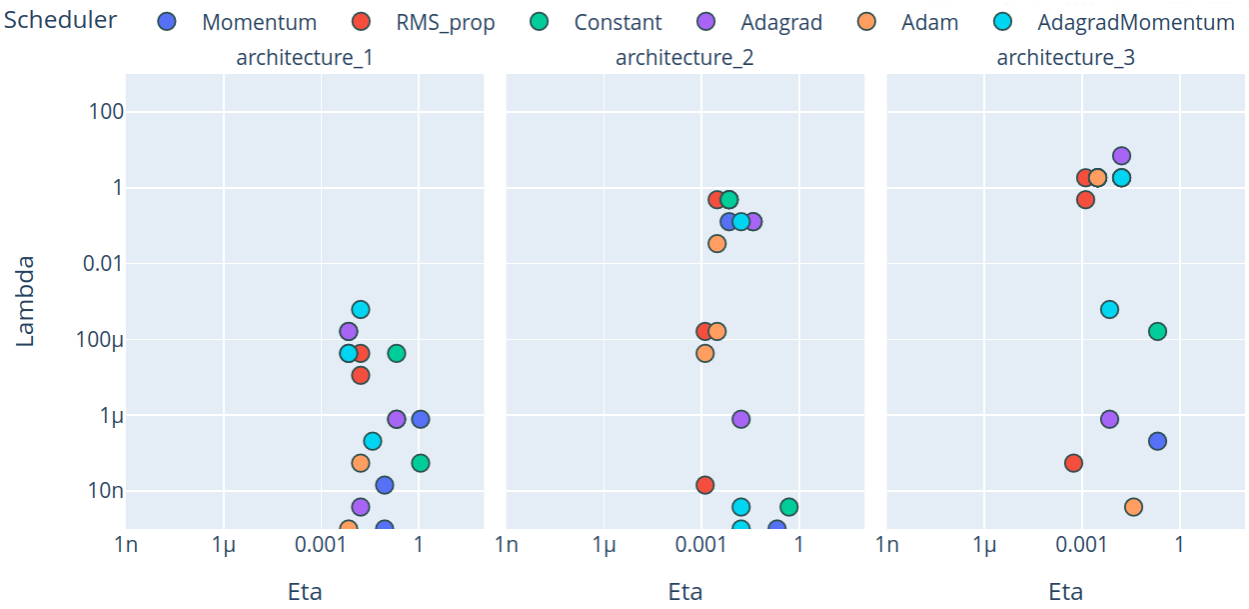


Figure 5: The parameter space for the best learning rates η and regularisation parameters λ.

Architecture	Hidden function	Scheduler	Eta	Lambda	Train accuracy	Test accuracy	Epochs	Batches	Momentum
architecture_1	RELU	Momentum	8.9e-02	1.0e-09	0.962	0.993	50	5	0.70
architecture_1	Leaky RELU	Momentum	8.9e-02	1.4e-08	0.962	0.993	50	5	0.70
architecture_1	Sigmoid	Adagrad	1.6e-02	1.6e-04	0.955	0.986	50	5	NaN
architecture_2	Sigmoid	RMS_prop	1.3e-03	4.3e-05	0.965	0.986	50	5	NaN
architecture_1	Sigmoid	RMS_prop	1.6e-02	4.3e-05	0.953	0.986	50	5	NaN

Figure 6: Accuracy scores for the top five best combinations sorted by highest value. The full table can be found in the jupyter notebook plots_classification.ipynb at the GitHub repository of the project.

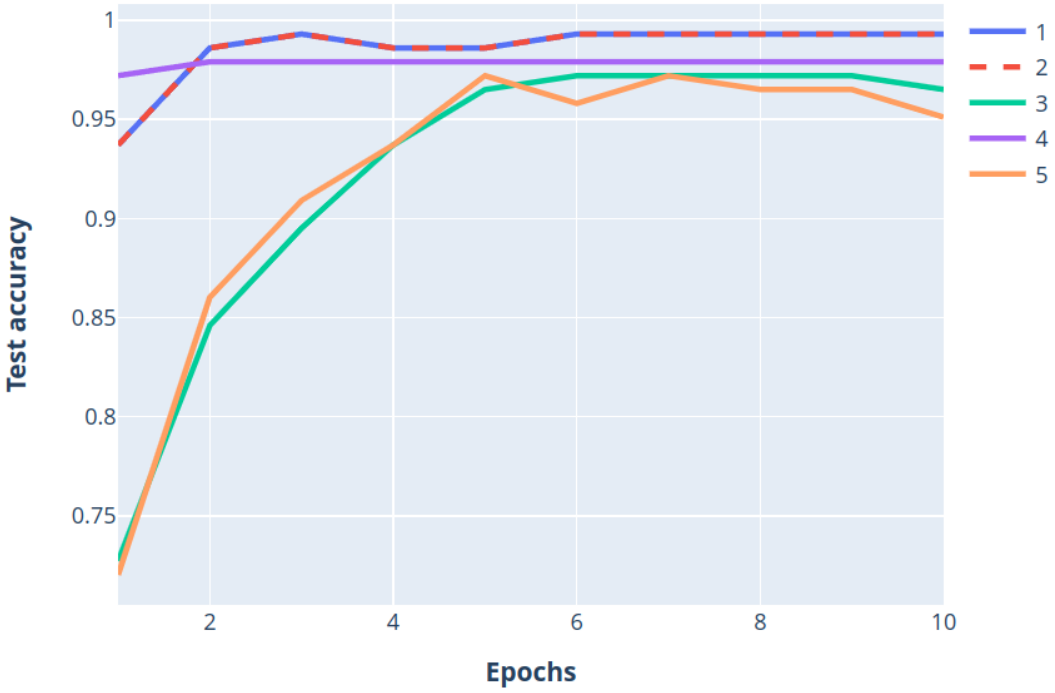


Figure 7: The convergence for the five best models as the number of epochs is increased. The numbering of the lines correspond to the appearance of the models in table 6, with 1 being the top row and 5 the bottom row.

The model with the ReLU activation function was chosen for further analysis and the results were compared to logistic regression. Figure 8 shows the accuracy of each combination of learning rate η and λ plotted in a heatmap. The close-to yellow area indicates the area of combinations that result in high accuracy, while the grey areas correspond to the combinations that lead to the model not converging. Since the plot was automatically generated during a grid search that was design to cover all reasonable combinations of η and λ for all model set-ups, it was expected that a lot of the combinations would fall outside the converging range of each model.

Figure 9 shows the Receiver Operating Charecteristic (ROC) curves of the best FFNN model and the logistic regression model. This plot shows the trade-off between true positive rate (also known as recall) and false positive rate across a range of threshold values. The FFNN model reaches a 100 % true positive rate without resulting in high false positive rate. The logistic regression reaches almost as high true positive rate, but on the cost of having some more falsely reported positive cases.

The confusion matrices of the logistic regression and the FFNN results are plotted in figure 10. The plots display the number of true positive (upper left), false negative (upper right), false positive (lower left) and true negative. In this context the value zero indicates a malign tumor or a positive result, and the value 1 indicates a benign tumor or a negative result. The logistic regression model misclassified one malignant tumors to be non-cancerous, and three benign tumors to be cancerous. For the FFNN model, the corresponding numbers were zero false negative and one false positive.

The accuracy, precision, recall and F1 score of the two models are given in table 11. The precision measures the proportion of correctly predicted positive instances among all instances predicted as positive. This metric depends on the rate of positive to negative cases, which for the Wisconsin breast cancer data corresponds to 357 benign to 212 malign cases. The F1 score is the harmonic mean of the precision and recall and can be used to give equal weight to both metrics. The FFNN scored higher on all the test metrics.

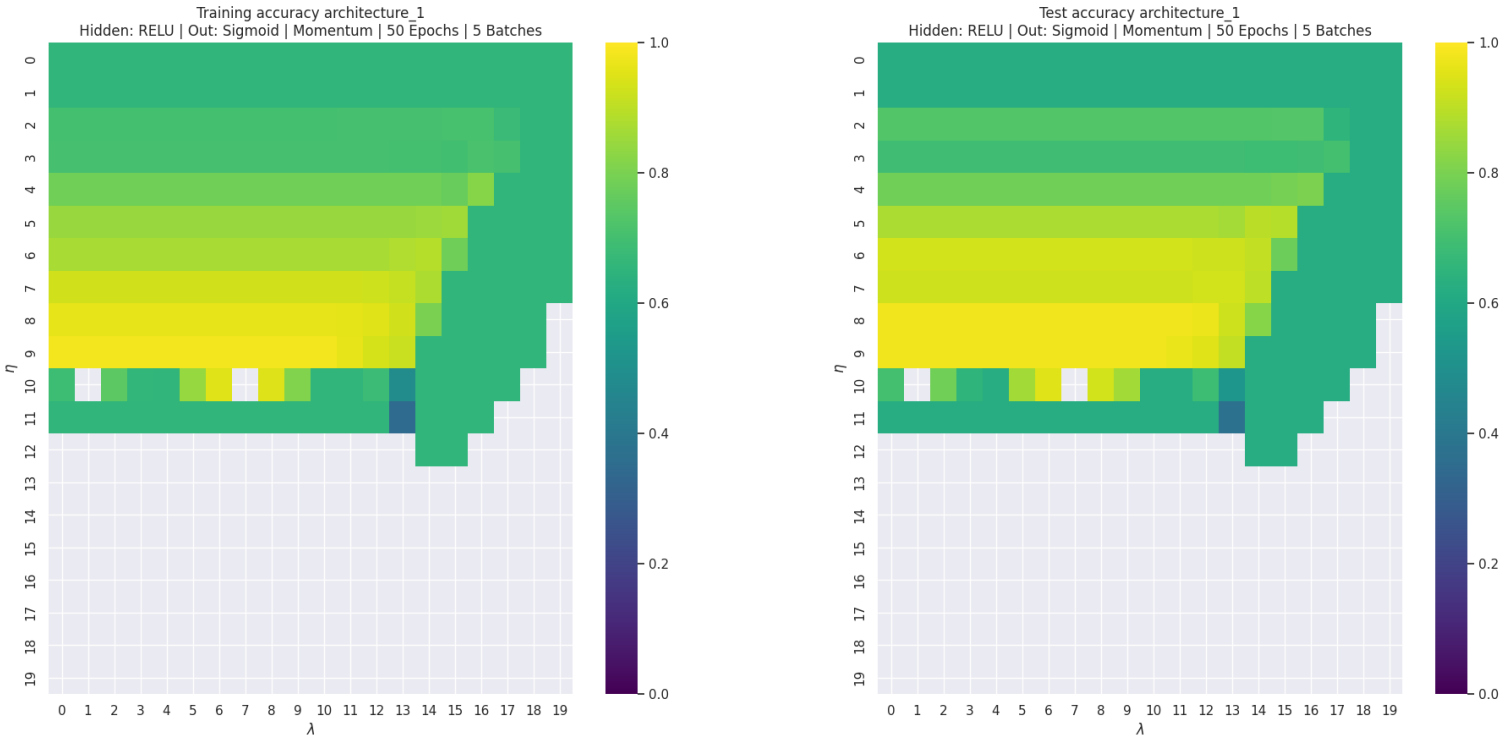


Figure 8: Heatmaps showing the accuracy of each combination of η and λ for the training data (left) and test data (right). A separate heatmap was generated for each combination of architecture, activation function and learning schedule.

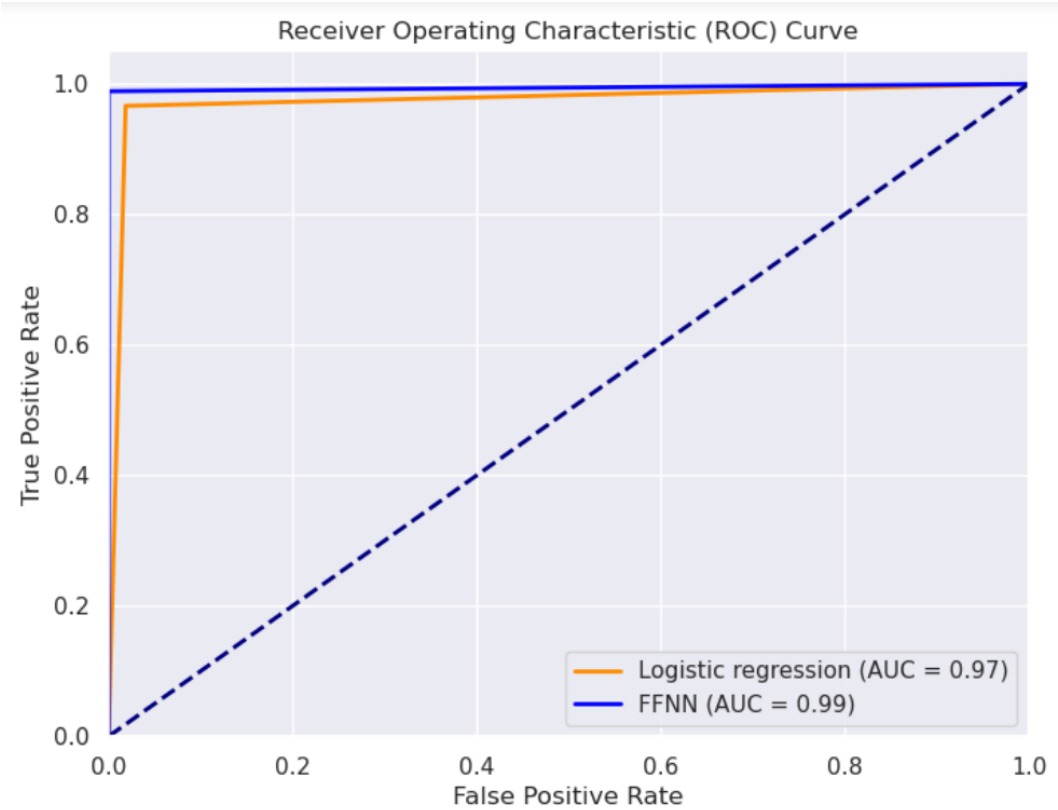


Figure 9: The ROC plot shows the trade-off between true positive rate and false positive rate across a range of threshold values.

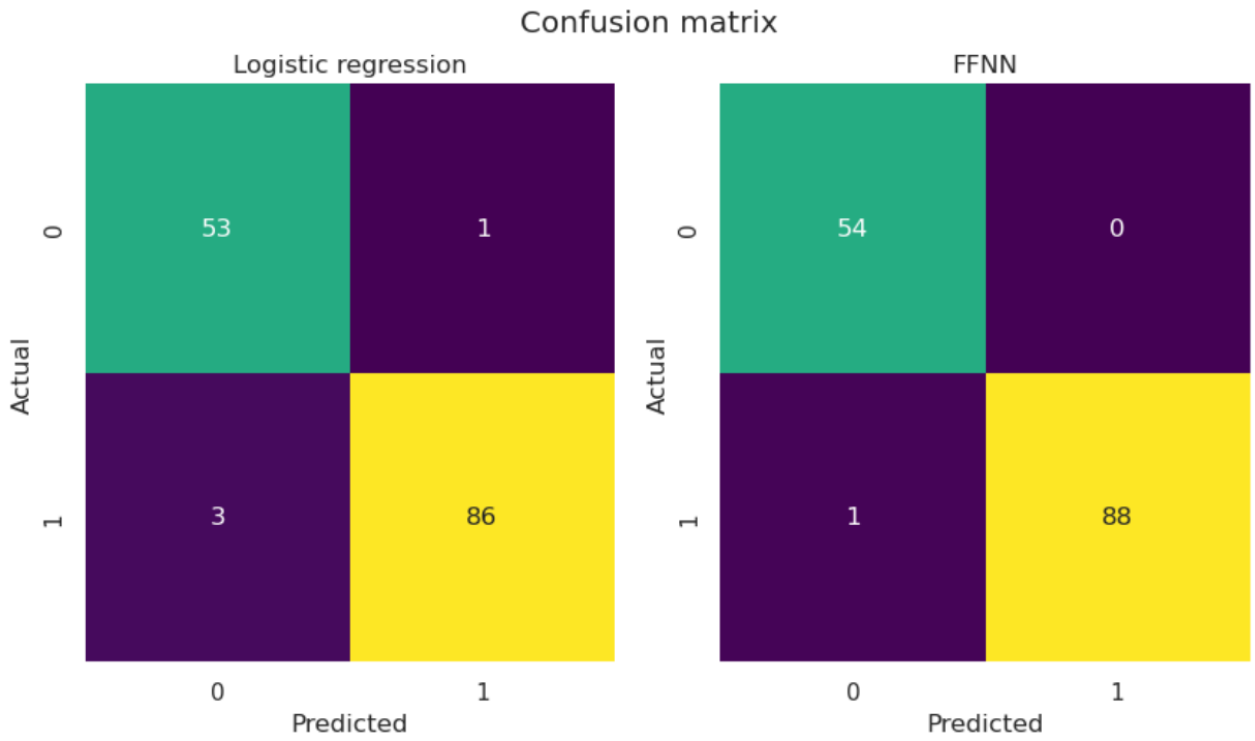


Figure 10: The confusion matrices of the results obtained with the logistic regression (left) and the FFNN model (right). The zeros correspond to malignant tumors and ones correspond to benign tumors.

Dataset	# Samples	Model	Accuracy	Precision	Recall	F1 score
Train	426	Logistic regression	0.967	0.981	0.978	0.979
		FFNN	0.981	0.981	0.985	0.983
Test	143	Logistic regression	0.972	0.989	0.966	0.977
		FFNN	0.993	1.000	0.989	0.994

Figure 11: Relevant train and test scores for the logistic regression and the best FFNN model.

4.2 Regression on simple polynomial

The regression analysis was performed using the self-developed FFNN with the sigmoid activation function and the mean squared error as cost function. Since the function was rather simple (see section 3.1), only one hidden layer with 50 nodes was used. The results were compared with OLS and Ridge regression, as well as the results of the K&R FFNN code and PyTorch. Since the function had its intercept at zero and was symmetric around the y-axis, the intercept of the Ridge regression was not penalised even without scaling.

The MSE of all models are given in table 12. The Ridge and OLS regression resulted in lowest MSE, with the FFNN models just behind. All FFNN models introduce randomness during the stochastic descent method, which could explain the differences between their final errors. Also, the K&R FFNN code and the PyTorch code used automatic differentiation during optimisation, while the self-written FFNN code used the analytical expression of the gradient.

Dataset	# samples	OLS	Ridge	Own FFNN	K&R FFNN	PyTorch
Train	160	3.6	3.7	3.5	4.0	3.9
Test	40	4.3	4.4	4.6	4.6	4.7

Figure 12: The mean squared error for all models.

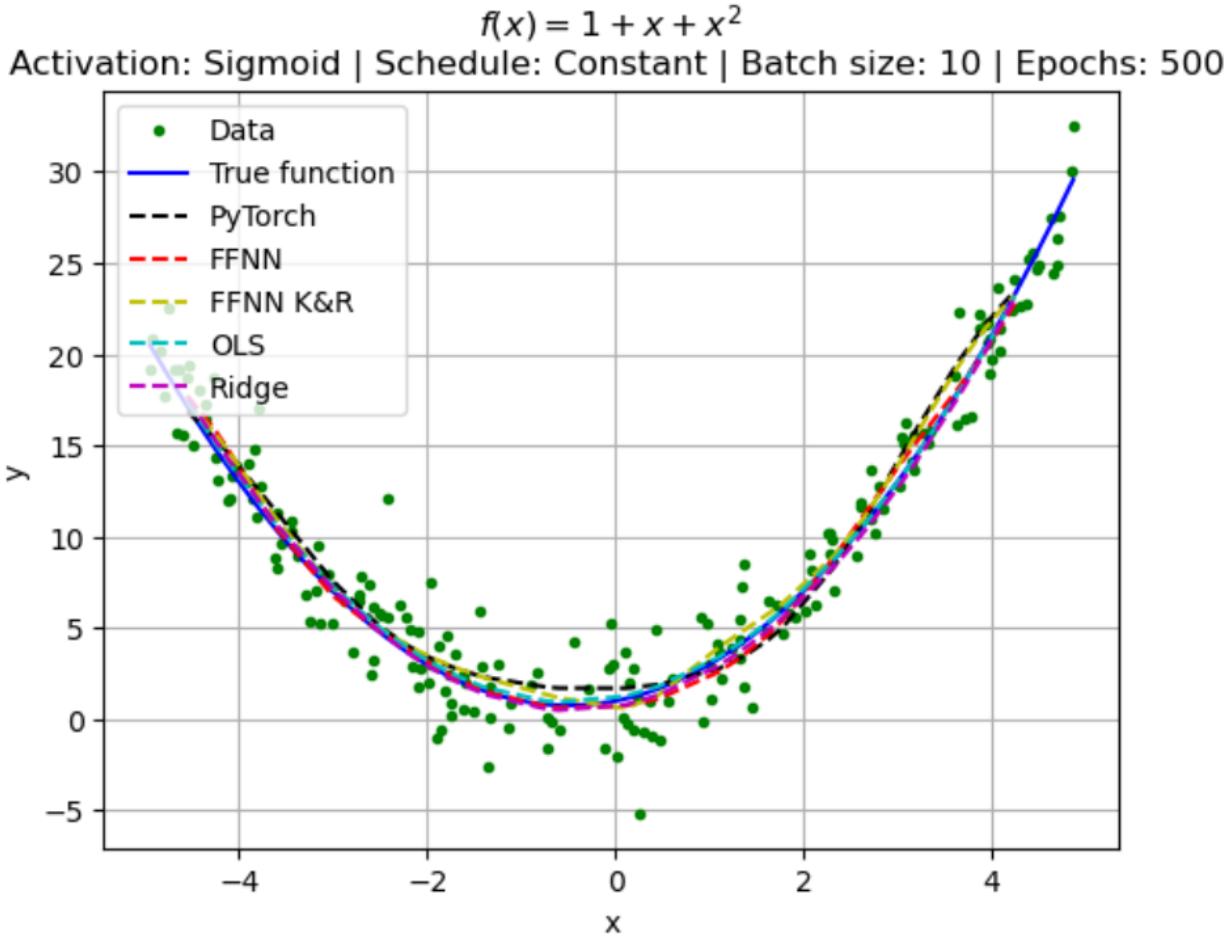


Figure 13: The predicted values of all models.

4.3 Optimization

The gradient descent methods applied in the FFNN code were first developed and tested on the same polynomial function as the regression analysis. A plain gradient descent with and without momentum was developed, along with the adaptive AdaGrad, RMSprop and Adam learning schedules. The code was extended to a stochastic gradient descent code containing tunable epochs and mini-batching.

The analytical expressions were compared with automatic differentiation from the Python JAX library. In figure 14 the true function was modelled with OLS using both gradient descent (GD) and stochastic gradient descent (SGD) and the JAX automatic differentiation functionality.

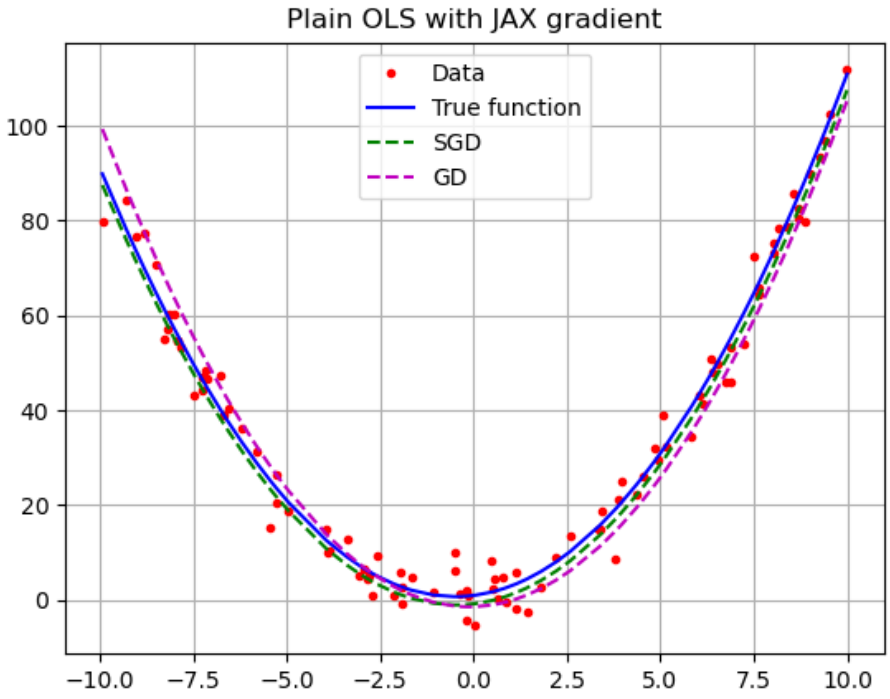
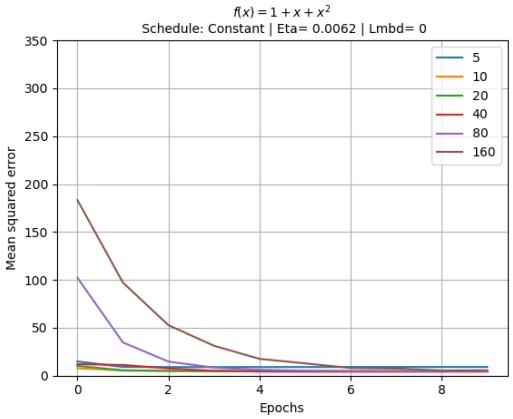
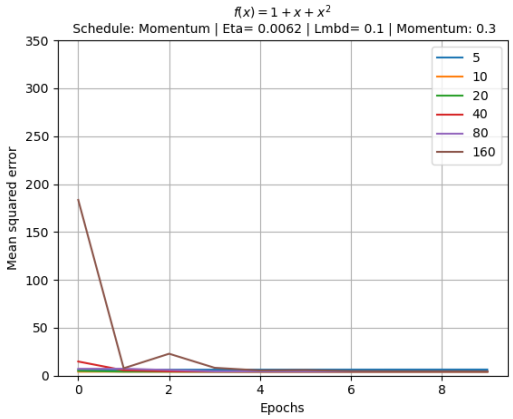


Figure 14: The OLS solution when using GD and SGD with JAX gradient.

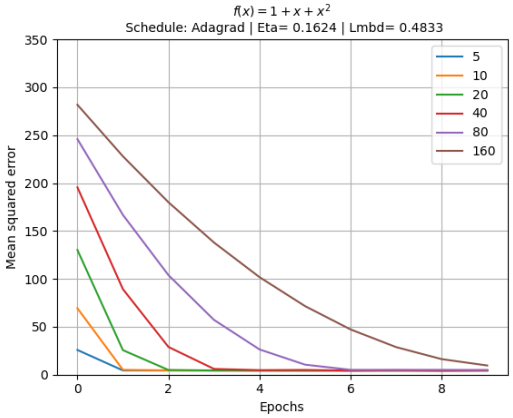
The number of epochs and batches was varied to investigate their effect on convergence. For this part of the analysis, the FFNN K&R code was used without hidden layer and with the mean squared error as cost function. When feeding a design matrix with the polynomial structure into such a network, the model approximates classical linear regression. This was done so that the FFNN K&R code could be used while keeping to the same set-up as project 1, as was asked in the project description. In general, smaller batch size increased convergence rate, since more iterations were performed per epoch. The smaller batch sizes also seemed to allow the model to generalize better to the test data, since they systematically flattened out at a lower error than larger batch sizes. All convergence rates are plotted in figure 15. No sign of overfitting was present, since the test error reached a plateau. Some of the methods did not yet converge when using batch size of 80 or 160. The convergence rate of each method depends on the chosen hyperparameters. Finding the optimal parameters was done by a grid search over different learning rates and regularisation parameters. For this specific problem the momentum method converged the fastest.



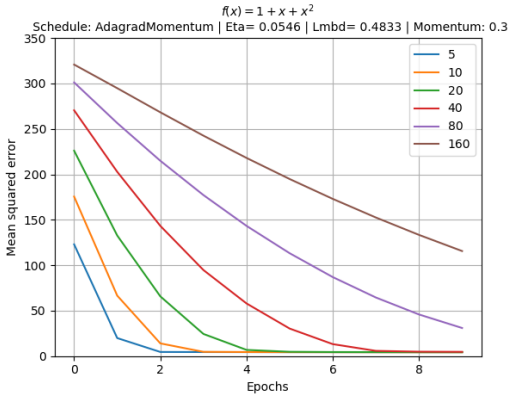
(a) Constant learning rate



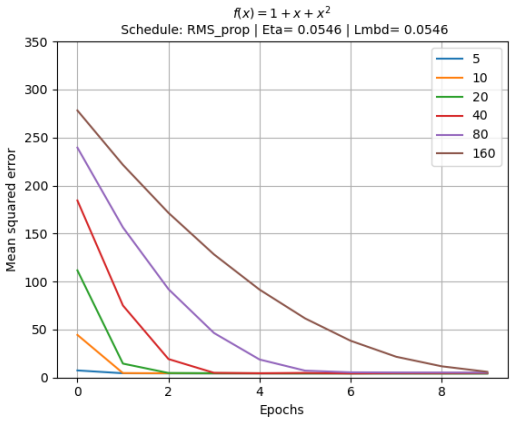
(b) Constant learning rate w/momentum



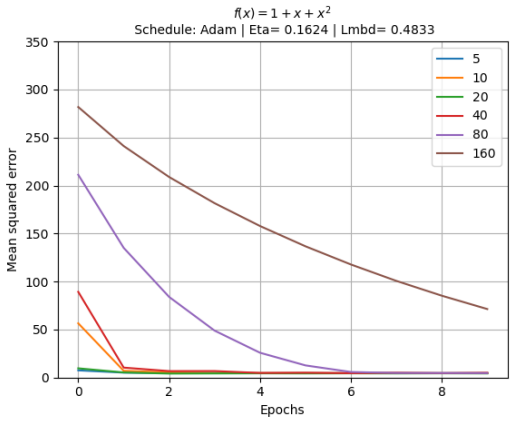
(c) AdaGrad



(d) AdaGrad w/momentum



(e) RMSprop



(f) Adam

Figure 15: Convergence rate at different batch size for all learning schedules.

5 Discussion

The grid search for optimal hyperparameters was performed with a fixed number of epochs. This was done to facilitate the comparison of different models, but resulted in unnecessarily high computational cost. Such a strategy could also result in issues of under- or overfitting since the model could stop either before or after the minimum test accuracy was used. Since all train and test accuracy curves were plotted, such a failure to converge could easily be discovered by inspecting the plots from each run. To avoid the above-mentioned problems, an early stopping criteria could have been implemented instead.

Although the FFNN code outperformed the logistic regression, the logistic regression also classified most tumor correctly. This indicates that the Wisconsin Breast Cancer Data does not need a deep network to perform well, which is also confirmed by the fact that the simplest of the three architectures performed the best. This architecture only included one hidden layer with 100 nodes.

When running the models with different randomness the results differed, both in terms of which model performed the best and what accuracy they achieved. The seed() function in Python was used to draw the same random number every run to facilitate the analysis of the results but it should be kept in mind that the accuracy scores obtained may not be reproducible if another randomness was to be used.

6 Conclusion and directions for future work

The early detection of malignant tumors is crucial for the outcome of breast cancer patients. Machine learning techniques have proven to be promising tools for distinguishing between malignant and benign tumors based on histopathological measurement [7]. In this project a Feed-Forward Neural Network model was applied to classify the tumors in the Wisconsing Breast Cancer Dat and the results were compared to those of logistic regression.

The best FFNN model outperformed the logsistic regression model in terms of accuracy and misclassification rates. The accuracy scores was 99.3 % for the FFNN model and only 97.8 % for logistic regression. The false positive rates and false negative rates are important metrics in cancer diagnostics, since the first can lead to unnecessary testing and treatment and the latter can lead to a fatal outcome for the patient if treatment is delayed. While the logistic regression misclassified one out of 54 cancerous tumors as benign, the FFNN correctly diagnosed all cancerous tumors as malign. For the false positive results, the logistic regression resulted in three out of 89 benign tumours being misclassified as malign and the FFNN only resulted in one tumor being incorrectly classified.

Breast cancer classification is an active area of research, and there are several directions for future work within machine learning:

Image analysis using convolutional networks: Performing classification on the tumor images directly may give access to information that is filtered away when only specific metrics are reported.

Understanding histopathological data using explainable learning: Using explainable learning on tumor images may reveal new features of importance.

References

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

[2] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference and prediction*. 2nd ed. Springer, 2009. URL: http://www-stat.stanford.edu/~tibs/ElemStatLearn/.

[3] Morten Hjorth-Jensen. *Lecture notes in Applied Data Analysis and Machine Learning*. Oct. 2023.

[4] Ruoxi Hong and Binghe Xu. "Breast cancer: an up-to-date review and future perspectives". In: *Cancer Communications* 42.10 (2022), pp. 913–936. DOI: https://doi.org/10.1002/cac2.12358. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cac2.12358. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cac2.12358.

[5] Sara Ibrahim, Saima Nazir, and Sergio A Velastin. "Feature selection using correlation analysis and principal component analysis for accurate breast cancer diagnosis". en. In: *J. Imaging* 7.11 (Oct. 2021), p. 225.

[6] Pankaj Mehta et al. "A high-bias, low-variance introduction to Machine Learning for physicists". In: *Physics Reports* 810 (May 2019), pp. 1–124. DOI: 10.1016/j.physrep.2019.03.001. URL: https://doi.org/10.1016%2Fj.physrep.2019.03.001.

[7] Ali Bou Nassif et al. "Breast cancer detection using artificial intelligence techniques: A systematic literature review". In: *Artificial Intelligence in Medicine* 127.102276 (2022), pp. 913–936. DOI: https://doi.org/10.1016/j.artmed.2022.102276. URL: https://www.sciencedirect.com/science/article/pii/S0933365722000410.

[8] Michael A. Nielsen. *Neural Networks and Deep Learning*. Oct. 2023.