University of Oslo.

Project 2.

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

Ingar Andre Benonisen, Martin Hansen Bolle

FYS-STK4155/Oblig2/Department of Informatics

04.11.2024

Abstract.

This project explores both classification and regression problems through the development of a custom feed-forward neural network (FFNN) and the implementation of logistic regression. Building upon foundational algorithms studies in previous work, we have analyzed and compared the performance of our FFNN with traditional regression techniques, including linear regression and stochastic gradient decent (SGD), and found that classical linear regression techniques provided a better MSE than the FFNN. For the regression analysis, we utilize datasets such as the Franke Function and simpler one-dimensional functions, allowing us to evaluate the model's capability in fitting continuous functions. On the classification segment we use the Wisconsin Breast Cancer Dataset provided by sklearn as our primary case study. This comprehensive approach not only facilitates an understanding of the strengths and limitations of neural networks in comparison to classical methods but also enhances our ability to apply robust evaluation techniques such as accuracy. We aim to provide and gain valuable insight into the application of machine learning techniques, thereby fostering a deeper understanding of their practical implications in classification and regression tasks.

Content

Introduction

In recent years, neural networks have revolutionized the field of machine learning and artificial intelligence, providing powerful tools for tackling complex problems in areas such as image recognition, natural language processing, and predictive modeling. Among the diverse types of neural networks, the feed-forward neural network (FFNN) is one of the simplest, most widely used architectures. Despite its simplicity, the FFNN can effectively address both regression and classification tasks, making it an essential model for gaining insight into deeper, more advanced architecture.

Our primary goal behind this project is to deepen our understanding of the practical applications of FFNNs by developing a comprehensive code framework that manages both classification and regression problems. By writing our own FFNN code from scratch, we aim to not only reinforce theoretical concepts, such as backpropagation and gradient decent, but also gain firsthand experience in implementing machine learning algorithms.

With project one recent in mind, we also have the opportunity to compare network-based approaches with traditional machine learning algorithms, like linear regression and logistic regression. We can assess the performance and applicability of neural networks in different contexts. For instance, we will conduct a regression analysis on the Franke Function, the well-known synthetic function used for testing and benchmarking regression models. We aim to assess how well our FFNN captures its patterns and predict target values.

In addition, we explore our FFNN's performance on a classification task, using the Wisconsin Breast cancer dataset. Here we compare the neural network's classification accuracy and generalization capabilities to a traditional logistic regression model.

The structure of this report is as follows. In the Method chapter we will provide background theory behind the algorithms used, alongside the technical details necessary for their implementation. The following chapter will embrace our code structure and implementation, where we will explain the design of our code and the specific choices made during development. In the result chapter, we will present our main findings, including key insight gained from the experiments. Then, under discussion, we will critically assess the results, analyze potential limitations, and offer reflections on the strengths and weaknesses of our approach. Finally, we will summarize the key findings and provide perspectives and future work and improvements.

Method

Gradient decent

Most deep learning algorithms rely on optimization to enhance model performance. This refers to the task of either maximizing or minimizing a function $f(x)$ by altering x. We want to assess the fit of our model by finding the values of x that minimizes the cost function, which quantifies the difference between the predicted and actual outcomes. Ideally, we want to be able to solve for x analytically, but this is not possible in general. Therefore, we must use numerical methods to compute the minimum.

Suppose we have a function $f(x) = y$ where both x and y are real numbers. The derivative of this function, $f'(x)\ or\ \frac{dy}{dx}$, gives us the slope of $f(x)$ at point x. The derivative is therefore useful for minimizing a function because it tells us how to change our x to make small improvements in y. By moving x in small steps with the opposite sign of the derivative we can decrease $f(x)$. This is what we call gradient decent (Goodfellow et al., 2016).

The idea of gradient decent is that a function F(x) decreases fastest if one goes from x in the direction of the negative gradient $-\nabla F(x)$. It can be shown that if,

$$x_{k+1} = x_k - \gamma_k \nabla F(x_k),\ \gamma_{k>0}$$

for small γ_k, then $F(x_{k+1}) \leq F(x_k)$. This means that for a sufficient small γ_k we are always moving towards smaller function values, i.e., a minimum.

This observation is the basis of the method steepest decent, also referred to as the gradient descent (GD). One starts with an initial guess of x_0 for a minimum of F, and compute new approximations according to

$$x_{k+1} = x_k - \gamma_k \nabla F(x_k),\ \gamma_{k>0}$$

Where γ_k refer to as the learning rate.

Gradient decent with momentum

An improvement to the well-known gradient decent is the addition of momentum. Momentum handles some of the limitations of plain gradient decent, like initial conditions or convergence to local minima. The algorithm works like GD, but we introduce a momentum parameter γ where,

$$0 < \gamma < 1$$

This leads to

$$\Delta\theta_{t+1} = \gamma\Delta\theta_t - \eta_T \nabla_\theta E(\theta_t)$$
$$\Delta\theta_t = \theta_t - \theta_{t-1}$$

Stochastic gradient decent

Stochastic gradient decent also address some of the shortcomings of the plain gradient decent method. The underlying idea comes from the observation that the cost function can almost always be written as the sum over n data points $\{x\}_{i=1}^{n}$

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

This in turn means that the gradient can be computed as a sum over i-gradients.

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

Stochasticity is introduced by only taking the gradients of a random selection data points. We can also do this over a subset called minibatches. If we have **n** data points and the size of each minibatch is **M**, there will be n/M minibatches, denoted as B_k

Now we can approximate the gradient by replacing the sum over all datapoints with the sum over the points in the minibatches picked at random in each gradient decent step.

Adagrad

It is often the case in sparse data, that the infrequent features are highly informative and discriminative. This motivated the development of the adaptive gradient method Adagrad (https://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf). Essentially, Adagrad adapts the learning rate to each parameter based on pasts gradients of that parameter.

The parameter for a given time step is given by the update rule

$$x_{t+1} = x_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \cdot\, g_t$$

Where G_t is defined as

$$G_t = \sum_{i=0}^{t} \nabla \mathrm{f}_t(x)\; \nabla f_t(x)^T$$

Adagrad's weakness lies in the squared gradients in the denominator, which keeps growing and subsequently decreases the learning rate (https://arxiv.org/pdf/1609.04747).

RMSProp

Root mean square propagation (RMSProp) tackles the decreasing learning rates by dividing the learning rate with an exponentially decreasing average of squared gradients. It's update rule can be expressed as

$$x_{t+1} = x_t - \frac{\eta}{\sqrt{\mathrm{E}[g^2]_t + \epsilon}} g_t$$

Where $\mathrm{E}[g^2]_t$ is defined as

$$\mathrm{E}[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

(https://arxiv.org/pdf/1609.04747)

Adam

The last method adaptive learning rate this report looks at is Adaptive Moment Estimation (Adam), which leverages the mean (first moment) and the variance (second moment) of the gradients to provide a stable and efficient convergence.

The parameter for a given time step is given by the following update rule

$$\theta_{t+1} = \theta_t \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$$

Where the first and second moment m_t and v_t are defined as

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

As m_t and v_t are initialized as zero, they are biased towards zero. The is especially the case if β_1 and $\beta_2$are close to one. Subsequently, bias-corrected first and second moments $\hat{m}_t$ and $\hat{v}_t$ must be estimated accordingly

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Logistic regression.

In statistics, logistic regression is used to model the probability of a binary outcome. This could be true or false, yes or no etc. To achieve this logistic regression uses a function called logistic or sigmoid function. This transform a linear combination of input features to a probability between 0 and 1.

Like linear regression, which you can ready more about in appendix a, logistic regression starts computing a linear combination of the input features

$$z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n$$

To ensure our output is a probability we apply the logistic function to the linear model.

$$p = \frac{1}{1 + e^{-z}}$$

Where **z** is the linear combination. The function maps the input from the range $(-\infty, \infty)$ to (0,1).

Logistic regression is based on the concept of odds, which are used to express the likelihood of one outcome compared to another. If **p** is the probability of success, the odds of success are

$$odds = \frac{p}{1-p}$$

The log-odds, also called the logit, is then given by

$$\log\left(\frac{p}{1-p}\right)$$

Amazingly, log-odds is a linear function of the inputs

$$\log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n$$

To estimate our parameters $\beta_0, \beta_1, \ldots, \beta_n$ we use maximum likelihood estimation (MLE). The likelihood function for logistic regression is the probability of observing the actual labels in the dataset given the models parameters.

For a dataset with m samples, where each sample i has labels y_i (0 or 1), the likelihood **L** is:

$$L(\beta) = \sum_{i=1}^{m} p(y_i | x_i; \beta)$$

Since $p(y_i = 1 \,|x_i \beta) = p$ and $p(y_i = 0 | x_i; \beta) = 1 - p$, we can expand this to:

$$L(\beta) = \sum_{i=1}^{m} p_i^{y_i} (1 - p_i)^{1-y_i}$$

For computational efficiency, it is easier to work with the log of the likelihood function (log-likelihood)

$$l(\beta) = \log\, L(\beta) = \sum_{i=1}^{m} [y_i \log(p_i) + (1 - y_i) \log\,(1 - p_i)]$$

By maximizing this log-likelihood function with respect to the parameters β gives us the best estimates for the parameters of the model.

Neural networks

Artificial neural networks are computational framework capable of learning to perform task by analysing examples, generally without being programmed with any task-specific rules. These networks are designed to emulate biological systems, wherein neurons interact by transmitting signals in the form of mathematical function across layers. Each layer can contain an arbitrary number of neurons, and each connection is represented by a weight variable. There are several different types of NN, and in this project we use a feed forward neural network, where the signal will only go in one direction, and every node in one layer is connected to all nodes in the next. During this section we will review the essential parts of how a neural network functions.

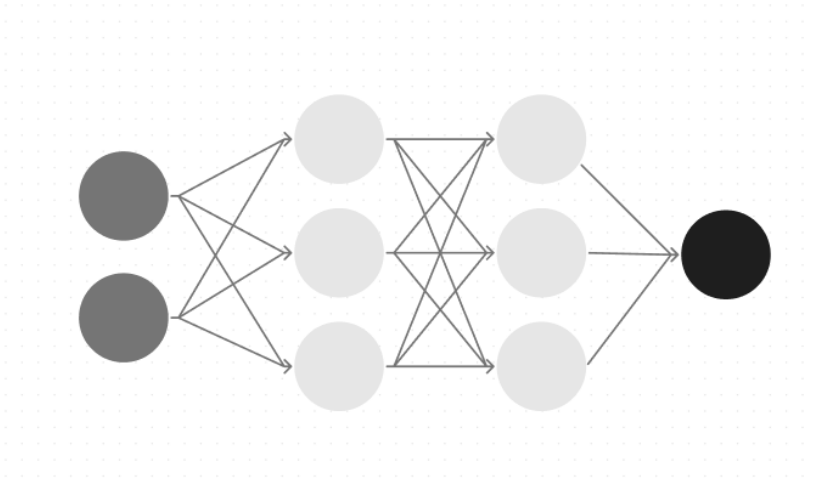

Figure 1 Schematic neural network. Circles represent nodes and arrows represent their connection. Dark grey represent input layer, light grep represent hidden layers and black represent output layer.

Feed-forward pass

This is the first of two basic steps. It takes a given input to produce an output, which is then compared to the target vales using a cost function. Figure 1 show an example of a small network with two input nodes, two hidden layers with three nodes each, and a single output layer. Each input is multiplied by a weight w_i and added to a bias term. The result, often referred to as the signal z_i, is passed through an activation function producing the activation $a(z_i) = a_i$, which becomes the input for the next layer. Since we are dealing with multiple observations, we assemble weights into a matrix W. The output a_l of layer **l** then becomes

$$a_l = a(a_{l-1}W_l + b_l)$$

Activation functions and weight initialization.

The choice of activation functions may have a huge effect on the performance of the model and there are several options that may work depending on what data we have and how it is represented. As stated in the project description we have taken advantage of several activation functions to analyse what works best for the selected task. This can be selected when running the program. Our only constraint is that a regression task require a linear activation function for the final layer.

Common choices for activation functions are the sigmoid function, the Rectified Linear Unit function (ReLU) and the leaky ReLU. For our regression method we have provided the possibility to choose between these three, and for our classification method, the sigmoid function is used.

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

$$ReLU(x) = f(x) = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases}$$

$$f(x) = \begin{cases} x, & x > 0 \\ \alpha x, & x \leq 0 \end{cases}$$

It is also worth noting that choice of weight and bias initialization is essential for performance. In this project we have used the Xavier initialization of our weights, simply because we experienced the occurrence of exploding gradient when training our network with different activation functions, learning rates and lambda values.

Xavier helped us mitigate these issues by ensuring the scale of the weights is appropriate for the architecture. The goal is to ensure that the variance from layer **l** matches the variance of the inputs to layer **l+1**. This can be expressed as

$$Var(z) = Var\left(\sum_{i=1}^{n} w_i a_i\right) = \sum_{i=1}^{n} Var(w_i) Var(a_i)$$

If we want the variance of the outputs to stay constant, the weights should be initialized with a variance that depends on the number of input neurons **n** and output neurons **m.** The weights are then sampled from a normal distribution with mean 0 and variance given by

$$W \sim N\left(0, \frac{2}{n+m}\right)$$

Output layer and backpropagation.

After completing the feed-forward pass, the network produces an output that reflects its performance on the give task. This output is compared with actual values, resulting in a score that quantifies the error or discrepancy between the predicted and true values.

To improve this performance, we need to adjust the weights and biases of the network. This adjustment process begins with the calculation of the gradient of the cost function, which measures how far the network's predictions are from actual values. The gradient provided information about both direction and magnitude of change needed to reduce the error.

The gradients are propagated backwards through the network, starting from the output layer, moving towards the input layer. At each layer the weights and biases are updated based on the calculated gradient, using gradient descent methods. This iterative process of calculating gradients and updating weights continues until the network converges to a set of parameters that minimized the cost function and increase the accuracy of its predictions.

Based on the derivative of the chosen cost function, and by applying the chain rule, one can show that in general terms the expression for the output error δ_L becomes

$$\delta_L = \frac{\partial C}{\partial a_L} \frac{\partial a_L}{\partial z_L}$$

We then get the back propagate error for the other layers $l = L-1, L-2, \ldots, 2$ as

$$\delta_l = \delta_{l+1} W_{l+1}^T a'(z_l)$$

Where a' is the derivative of the activation function. The update of the weights and biases in a general layer **l** is calculated by

$$W_l \leftarrow= W_l - \eta a_{l-1}^T \delta_l,$$
$$b_l \leftarrow= b_l - \eta \delta_l,$$

Where η is the learning rate. This specifies how much the weights and biases should be adjusted for each back propagation.

Cross entropy loss

In binary classification with two classes (0,1) we define the sigmoid function as the probability that a particular input is in class 0 or 1, just as discussed under logistic regression.

We assume now that we have two classes, y_i, zero or one. Also assume we only have two parameters beta in our fitting of the sigmoid function, that is we define probabilities

$$p(y_i = 1|x_i, \beta) = \frac{exp(\beta_0 + \beta_1 x_i)}{1 + exp\ (\beta_0 \beta_1 x_i},$$
$$p(y_i = 0|x_i, \beta) = 1 - p(y_i|x_i, \beta),$$

Where β are the weights, we wish to extract from data. To define the total likelihood for all possible outcomes from a dataset $D = \{(y_i, x_i)\}$, with the binary labels $y_i \in \{0,1\}$ and where the data points are drawn independently, we use the maximum likelihood estimator principle (MLE). We approximate the likelihood in terms of the product of individual probabilities of a specific outcome y_i that is

$$P(D \mid \beta) = \prod_{i=1}^{n} [p(y_i = 1 \mid x_i, \beta)]^{y_i} [1 - p(1 - p(y_i = 1|x_i, \beta)]^{1-y_i}$$

From which we obtain the log-likelihood and our new cost function after some reordering.

$$l(B) = \sum_{i=1}^{n} (y_i(\beta_0 + \beta_1 x_i) - log\ (1 + exp(\beta_0 + \beta_1 x_i)))$$

The maximum likelihood estimator is defined as the set of parameters that maximize the log-likelihood where we maximize with respect to β. Since the cost function is just the negative log-likelihood, we have

$$C(B) = -\sum_{i=1}^{n} (y_i(\beta_0 + \beta_1 x_i) - log\ (1 + exp(\beta_0 + \beta_1 x_i)))$$

In statistics, this is what we call the cross entropy.

Performance metrics

The performance metrics used throughout this project are mainly mean squared error (MSE) and accuracy. In addition, we looked at the R2 score only when comparing with results from the linear regression.

Mean squared error is defined as

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (y_i - \tilde{y}_i)^2$$

And R2 as

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

Accuracy is formulated as

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

Where $1(y_i = \hat{y}_i)$ is an indicator function that returns 1 if the prediction is correct and 0 otherwise.

Results

Gradient descent

In this chapter, we evaluate the performance of the various gradient descent methods with different values of lambda, learning rate, momentum, mini-batch size and epoch on OLS and ridge regression. The gradient descent methods in question are plain gradient descent (hereby just referred to as gradient descent) and stochastic gradient descent. These methods are also implemented with Adagrad, RMSProp and Adam. In the chapter we will be looking further into the results from the optimal hyper-parameters.

Upon studying the results, it's clear that gradient descent's MSE is much worse than stochastic gradient descent's, in all variations (with momentum, adagrad, etc.). However, gradient descent converges quicker than stochastic gradient descent in most variations as well. This indicates that the gradient descent variations hit a saddle point or local minima early on in the gradient descent. The exception is when the gradient descent methods are used in combination with Adam, where they converge much slower. Nonetheless, the MSE is still the same.

As discussed in the Methods, Adagrad's main weakness is the decaying learing rate. Comparing gradient descent methods with and without Adagrad does not give any insights into this, as the results are already inflicted by early convergence in a local minima or saddle point. However, looking at an average convergence rate (mini-bach size * epochs before convergence + iteration at convergence) of stochastic gradient descent methods, shows that the improved RMSProp convergences faster than Adagrad. Whilst the Adam method converged slower than Adagrad. Note that this was also the exception for the gradient descent methods.

Neural network.

In this chapter, we evaluate the performance of our neural network against the regression methods employed in project one, specifically focusing on Ordinary Least Squares (OLS) and Ridge regression. To facilitate a transparent comparison, we utilize the Franke function as the basis for fitting all models, ensuring that the data generating and scaling process were consistent across all approaches. This methodical approach allows for a clear assessment of how our neural network performs relative to traditional regression techniques. In addition, we examine the impact of different activation functions on the performance of our feed forward

neural network. Lastly, we will use our neural network for a classification task on a dataset provided by sklearn.

Regression.

We will look at the comparison of regression, using the classic linear methods like OLS and Ridge and compare them with our own neural network. In this paper we will explain the linear methods briefly. A more in-depth analysis can be found in the Appendix (project one).

In figures 2 and 3 we can see that the results for both OLS and Ridge provide a decent result. For both methods, as the complexity increases, our MSE for both test and training decreases. Similarly, the R^2 score for both datasets increase with model complexity. When the model complexity reaches a certain point (degree three in our case), we see the opposite trend for the test dataset. This observation implies that our model is overfitting, meaning that the model has learned to fit the training data very closely, but does not generalize to new, unseen data.

In comparison, figure 4 reveals a significant difference in the MSE-scores when using the neural network. The MSE-scores range between 0.6 to 1.2 on the test set depending on the activation function used. Notably, employing the sigmoid activation function with learning rate 10^{-2} and a lambda value of zero gave us the best result. Additionally, we observe that, in general, a higher learning rate results in lower MSE-score regardless of the activation function.

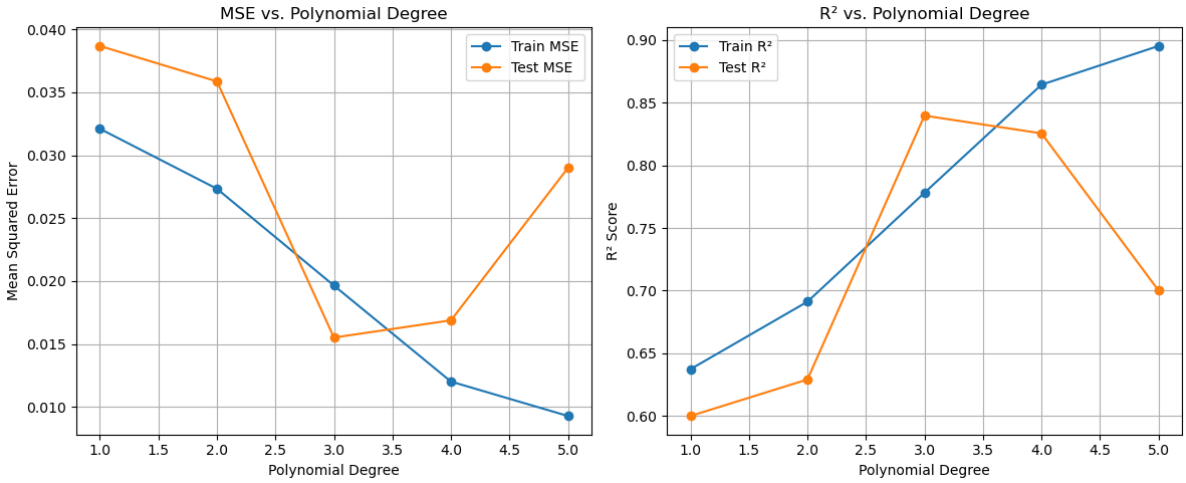

Figure 2 Accuracy metrics of Ordinary Least Square regression for different polynomial degrees. The left-hand side display the MSE for both train and test. Right-hand side displays R2 score for the same method.

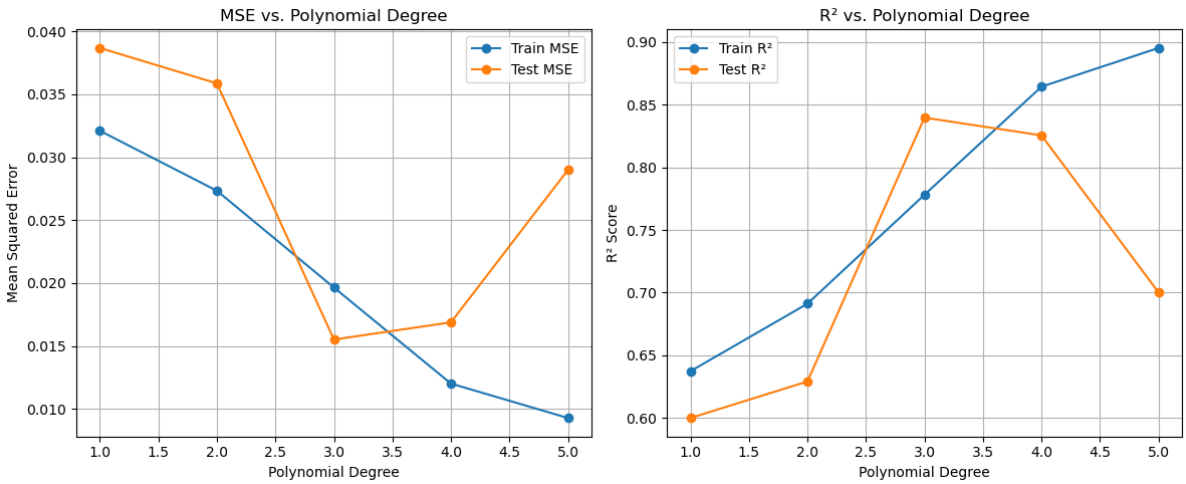

Figure 3 Accuracy metrics of Ridge regression for different polynomial degrees. The left-hand side display the MSE for both train and test. Right-hand side displays R2 score for the same method.

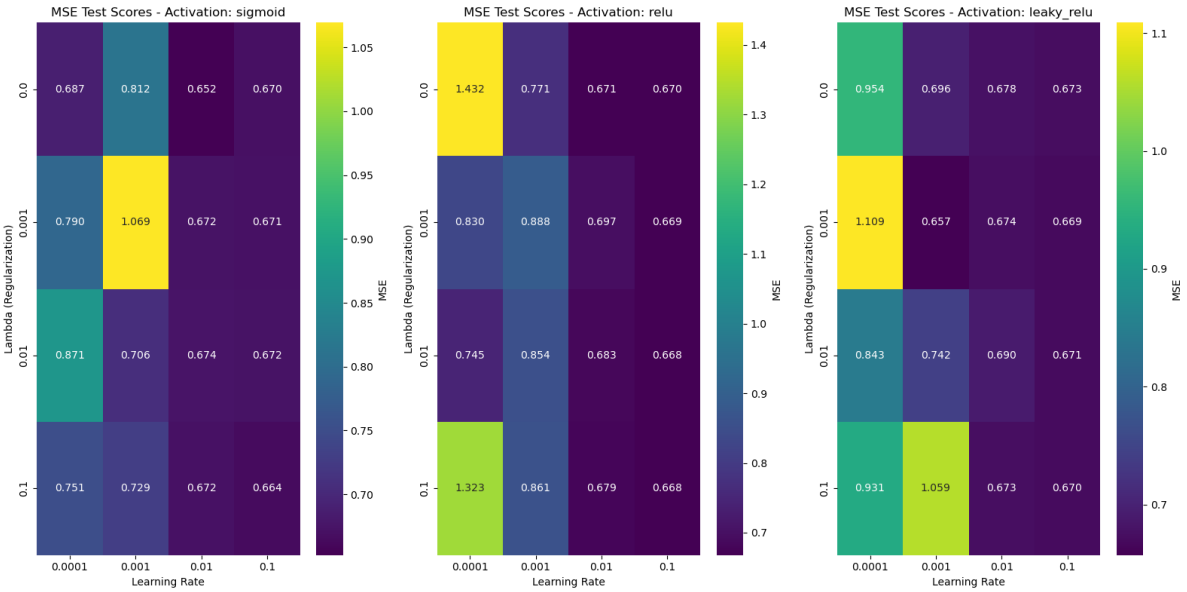

Figure 4 Heatmap of the MSE score on the test set for three different activation functions, Sigmoid, ReLU, and leaky ReLU. The different activation functions is combined with different learning rates and lambda values. A darker colour implies a lower score, while a brighter colour implies higher scores. Here we have used a hidden layer with 50 nodes and an epoch size of 100

We also notice that increasing our neural network in terms of number of layers, up to a certain point, tends to improve our best performance. This also happens when we increase the number of epochs. Looking at the size of our mini batches we see an improvement in speed for each calculation, but not a huge difference in performance metrics.

Classification.

For our classification analysis we have used the breast cancer dataset from sklearn. This dataset returns a dataset containing two classes of tumors, one benign and one malignant, with different statistics.

Taking a closer look at figure 5 we observe the accuracy of our binary classification with different parameters. In this case we have used a lambda value of 0.1 and mini-batch size of 10. A learning rate of 10^-1 with 200 epochs resulted in the best result in our case, and generally a higher learning rate has resulted in better accuracies. This might be because the number of iterations is not big enough to reach a smaller value with smaller learning rates. It also allows for faster and potentially more effective convergence. Here we also observe that a higher number of epochs yields a better accuracy for all learning rates.

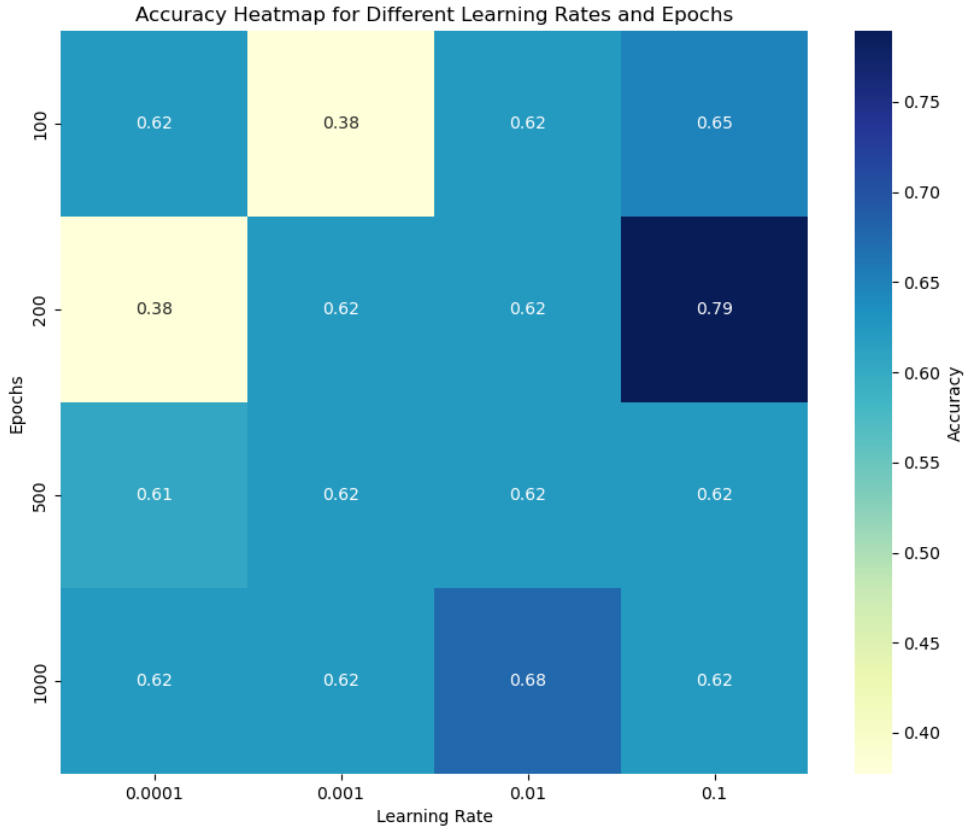

Figure 5 Heatmap illustrating the accuracy of a Feed-forward Neural network across various learning rates and epochs. Each cell in the heatmap represents the accuracy achieved for a specific combination of Learning rate and number of epochs.

Table one above also shows the confusion matrix for out best-case accuracy. With this we notice an imbalance in our dataset, where we have 71 positive cases against 43 negative ones. This imbalance can lead to biased predictions, as the model may prioritize the more frequent positive class to achieve the higher overall accuracy.

Table 1 Confusion matrix for the best-case accuracy on the classification of breast cancer dataset from sklearn.

	Predicted Negative.	Predicted Positive.
Actual negative	19	24
Actual positive	0	71

FFNN vs. Logistic Regression

Lastly, we will analyze the results from the FFNN classification with different values for the hyper-parameters and compare these with logistic regression.

Upon studying the figures genereted when evaluating the FFNN classification (which can be found under the folder partE), we can see that there is a clear trend upon all variations of epoch and mini-batch size, where the score increases when lambda is higher. Although the results are better with higher lambda values, the FFNN is not able to produce a result.

Upon studying the figures generated when evaluating logistic regression (which can be found under the same folder), we do not see the same trend. In this case, all variations of different hyper-parameters yield in more or less a high score.

When testing the methods with the optimal hyper-parameters we found when evaluating the methods, we found that the FFNN classification got a test score of 0.62. On the other hand, the logistic regression model resulted in a test score of 0.96. Scikit's Logistic Regression model was also tested and got a test score of 0.97.

Discussion

Gradient descent

The high MSE and fast convergence of the gradient descent methods indicates that they converged at a local minima, saddle point or could not "fit" into the global minima, perhaps due to too large learning rate for example.

As the Methods and Results chapters pointed out, Adagrad weakness in decaying learning rate should result in slower convergence than RMSProp and Adam. This was the case for RMSProp, but not for Adam. The reasoning behind Adam's slow convergence could be the slow initial convergence. On the other hand, Adagrad converges faster in the beginning.

Regression.

As our results demonstrated the classical linear regression techniques, Ordinary Least Square and Ridge regression, both outperformed our feed-forward neural network (FFNN) on the regression task on fitting the Franke function. Several reasons might be the cause of this, but reasons like model complexity, efficiency, training stability, and bias-variance trade-off might be reasonable answers. Linear regression models are simpler with fewer parameters. When the underlying function is relatively low-dimensional and lacks complex patterns, simpler models can often approximate quite well. In addition, FFNN needs large datasets to

learn effectively and avoid overfitting. In our comparison we have drawn a few data samples, which lead to our FFNN struggle to generalize due to its high capacity and flexibility.

Linear regression models also have close-form solutions (OLS) or use straightforward optimization method (Ridge), making them highly stable and efficient. They are also convex, meaning they will not reach local minima's like FFNN.

Classification

The results of how learning rate affected our results align with a common trade-off in the field of machine learning. Higher learning rates enable faster convergence but can risk overshooting optimal solutions if it is too high. On the other hand, lower learning rates did not perform as well as higher. This is because its needs in general more time or iterations to reach the optimal solution, as the step size is too small. The trend that more epochs yield higher accuracy suggests that the model benefited from additional iterations to refine its weights.

Even though we tuned our parameters we did not achieve an accuracy score higher than 0.79, which is not particularly high. Whether this score can be considered good depends on the context. In the field of medicine, it would be considered insufficient, and as the stakes are too high. Some datasets have overlapping feature distribution, making it harder for models to achieve high accuracy. In these cases, scores close to 0.8 might indicate a decent performance. It might be smart to consider multiple metrics to provide a balanced view.

FFNN vs. Logistic regression

The fact that the FFNN classification model was outperformed by a simple logistic regression model can be explained by various reasons. First, the poor performance of the FFNN may be a consequence of overfitting. The neural network has far more parameters than the logistic regression model, and in combination with the small dataset, the model may have learned the training data's patterns rather than generalizing well. In addition, the higher number of parameters in the neural network makes it harder to optimize than the logistic regression model.

Second, as briefly mentioned above, the dataset is small. Logistic regression models tend to excel on smaller datasets, whilst neural networks excel on larger datasets with complex relationships between the variables.

Imbalance.

We would like to emphasize that accuracy might not be an appropriate metric for this method. Typically, medial datasets like the one we have used, class imbalance is a prevalent issue, often arising when the occurrence of a particular condition or disease is significantly lower than the absence of it. This imbalance can lead to misunderstanding evaluations of a

model performance when using accuracy as performance metric. Given a scenario where 99 percent of the samples are negative and only one percent are positive, a model that predicts all cases as negative would achieve an accuracy of 99%. It is important to consider alternative metrics or look at accuracy in combination with others.

Conclusion and Perspectives

In this study, we explored both regression and classification tasks using linear regression techniques (OLS and Ridge), logistic regression and a custom feed-forward neural network. For our regression task, Ordinary least Square and Ridge regression achieved a lower MSE score on the test data compared to the FFNN, highlighting the efficiency of linear models in capturing underlying structure of simple, low-dimensional functions. The FFNN's on the regression tasks demonstrated its flexibility, through it was less effective without substantial hyperparameter tuning, emphasizing the importance of careful model configuration when managing simpler functions. We found that our neural network performs best with the sigmoid function and a learning rate $\eta = 10^{-2}$.

For classification, we applied both logistic regression and the FFNN to the Wisconsin Breast Cancer dataset. It showed that the simplistic logist regression model outperformed the neural network, indicating that for smaller and less complex datasets, a simpler model performs better.

Several gradient descent methods have also been evaluated. In this analysis, we found that the gradient descent methods performed worse than stochastic gradient descent methods. Furthermore, we found that Adam converged slower than Adagrad, as expected on a smaller dataset. Lastly, RMSProp converged the fastest of the adaptive learning rate methods.

Further work.

Our analysis opens avenues for future research. One potential direction is to explore experimenting with more complex neural network architectures, such as convolutional neural networks or recurrent neural networks, which may offer improved performance. Additionally, implementing ensemble methods like random forest or gradient boosting could provide a robust comparison against our current logistic regression and neural network models.

Another aspect that would have been sensible to look at is the implementation of the possibility of multiclass classification. As the project did not ask for this, we did not prioritize implementing this. A comparison of multiple performance metrics would also be potential future research.

References

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. The MIT press.

Hastie, T., Tibshirani, R., & Friedman, J. H. (2017). *The elements of statistical learning: Data mining, inference, and prediction* (Second edition, corrected at 12th printing 2017). Springer.

Morten Hjorth-Jensen. (2024). *CompPhysics/MachineLearning* [Computer software]. CompPhysics. https://github.com/CompPhysics/MachineLearning (Original work published 2017)

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, É. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, *12*(85), 2825–2830.

Raschka, S., Liu, Y., Mirjalili, V., & Dzhulgakov, D. (2022). *Machine learning with PyTorch and Scikit-Learn: Develop machine learning and deep learning models with Python*. Packt.

Appendix

GitHub repository:

https://github.com/martin-og-ingar/FYS-STK4155-Project2

Project one.

https://github.com/martin-og-ingar/FYS-STK4155-Project1