Project 2 FYS-STK4155

Trond Skaret Johansen, Eirik Salberg Pedersen,
Jonatan Winsvold

November 19, 2023

Abstract

Two problems of great importance in the field of machine learning are regression and classification. The goal of this project has been to implement and study methods for gradient descent and then applying them on these important problems. We use OLS and neural networks for regression and neural networks and logistic regression for classification. We find that with an equal amount of data, a neural network outperforms OLS in the regression case. For classification, we find logistic regression to be best.

1 Introduction

At the heart of all optimization are the various methods for computing gradients and performing gradient descent in order to minimise a cost function. We therefore begin by implementing methods for various descent methods. The methods include simple gradient descent, stochastic gradient descent, RMS propagation, AdaGrad and Adam. All methods are implemented with the option of including a momentum parameter γ to speed up training. The descent methods will be used for minimising multiple cost functions, and we explore the difference between gradient minimization through analytical matrix inversion and automated differentiation using JAX.

When experimenting with the methods above, our goal is an analysis of the results for OLS and Ridge regression as function of the chosen learning rates, the number of mini-batches and epochs as well as algorithm for scaling the learning rate. For Ridge regression we also study results as functions of the hyper-parameter λ.

As computers grow faster, neural networks are becoming increasingly viable as large models for prediction, classification and many more applications. We take steps into the vast space of neural network applications, by implementing functionality for a flexible neural network. The network is then trained by minimizing the MSE and Ridge cost functions on the Franke function as in Project 1 [2], before delving into a classification problem using the Wisconsin Breast Cancer Dataset [6]. This dataset not only tests the accuracy of our methods in a binary classification problem, but also provides an assessment of their applicability to real-world, high-stakes data.

2 Methods

2.1 Data handling

To evaluate the effectiveness of the newly implemented methods, we apply them to two distinct problems. The first problem is to interpolate the Franke function, which we previously analyzed using OLS, Ridge, and Lasso regression in Project 1 [2]. Since already know how these previous methods perform, we have a notion of what to expect as predictions, making it an ideal test case for assessing the capabilities of our methods. In this project, we minimise the gradient using descent methods rather than the analytical matrix inversion as used in Project 1 [2]. A comparison with the previous results will be given.

In addition to the Franke function, our methods are further tested against the Wisconsin Breast Cancer Dataset. This dataset, a benchmark in medical data analysis, includes features derived from digitized images. Each instance is classified as benign or malignant, providing a clear binary classification problem. The complexity of this dataset lies in its high dimensionality and the critical need for accurate classification in a medical context. Our previous exploration of this dataset has not been as extensive as with the Franke function, offering us a fresh perspective and a challenging opportunity to validate the effectiveness of our newly implemented methods. The ability to accurately classify instances in this dataset will demonstrate the robustness and adaptability of our approaches to real-world, high-stakes data.

With OLS and Ridge we are minimising the cost functions described in Project 1 [2]. For OLS, we minimize the average square error, while for Ridge there is an additional regularisation term $||\boldsymbol{\beta}||_2^2$ penalising large parameters. Other cost functions appear when optimising the neural network and performing logistic regression.

We scale the feature matrix by subtracting the mean and dividing by the standard deviation. The reasons for this are explained in detail in Project 1 [2]. To make sure there is no sorting of the breast cancer data, we do an initial shuffling when loading it.

2.2 Descent methods

Before implementing methods for neural networks and logistic regression, we build a solid library of descent methods, experimenting with various ways of tuning the learning rate. We have implemented five popular methods and we will compare their performance. The methods are plain gradient descent, stochastic gradient descent, RMSprop, AdaGrad and Adam. All of them share a similar structure as plain gradient descent, but with small augmentations which give a significant difference in performance. For each method there is also an optional momentum parameter to learn faster. Algorithm 1 shows a general description of stochastic gradient descent. In the following we explain how the various methods differ in each step.

Algorithm 1: Gradient Descent

```
Input  : Initial parameters $\beta$, method for computing gradient of loss
         function, $\nabla C(\beta)$ and various hyperparameters.
Output: New parameters $\beta$.
1  for $i = 1, ..., n$ epochs do
2      Initialise accumulation variables
3      for $i = 1, ..., m$ minibatches do
4          Draw minibatch $D \subset X$
5          Compute gradient of cost using minibatch: $\nabla_\beta C(\beta, D)$
6          Accumulate parametres
7          Compute step
8      end
9  end
10 return New parametres $\beta$
```

2.2.1 Plain gradient descent

We employed a plain gradient descent (GD) approach as a baseline for comparison. This method iteratively adjusts the model parameters to minimize the loss function, which quantifies the discrepancy between the predicted and actual values. In each iteration, the model parameters, denoted by β, are updated in the opposite direction of the gradient of the loss function with respect to β. The update shown in Algorithm 1 can also be mathematically represented as:

$$\beta_{\text{new}} = \beta_{\text{old}} - \eta \nabla C(\beta_{\text{old}}), \tag{2.1}$$

where η is the learning rate, and $\nabla C(\beta_{old})$ is the gradient of the cost function computed using the old values. In this plain version we use all data in each step. We shall see that this corresponds to stochastic gradient descent with a single batch, i.e. $m = 1$ in Algorithm 1. There are also no accumulation variables to update and accumulate before making the step.

2.2.2 Stochastic gradient descent

As our next model we implemented the variation, *stochastic gradient descent* (SGD). The difference from plain GD is that we update the gradient from a smaller random subset of the dataset. This is often more computationally efficient, and the hope is that we get a better outcome after each epoch. Our implementation executes this update rule in mini-batches. This approach helps reduce the computational burden and can lead to faster convergence. Specifically, our training dataset is divided into batches, and in each epoch, the gradient is computed for a randomly selected batch. The parameter update rule is then applied using this batch gradient, enabling the model to iteratively learn from different segments of the data across epochs.

2.2.3 Root Mean Square Propagation

The *Root Mean Square Propagation* (RMSProp) optimization algorithm operates by adapting the learning rates for each parameter through the division by the root mean square (RMS) of recent gradients. Specifically, for each parameter, it maintains a running average of the squared gradients, q, which is updated as

$$q = \rho q_{old} + (1 - \rho) \nabla C(\beta_{\text{old}})^2, \tag{2.2}$$

where ρ is the decay rate. The parameter updates are then computed as

$$\beta_{\text{new}} = \beta_{\text{old}} - \frac{\eta}{\sqrt{\delta + q}} \nabla C(\beta_{\text{old}}), \tag{2.3}$$

where η is the learning rate and δ is a small constant (a safe division parameter) to avoid numerical errors.

This scaling by $\sqrt{\delta + q}$ normalizes the gradient step, taking larger steps for parameters with small gradients and smaller steps for parameters with large gradients.

2.2.4 AdaGrad

In our approach, we utilized AdaGrad (Adaptive Gradient Algorithm) to dynamically adjust the learning rate for each parameter during training. AdaGrad is designed to assign different learning rates to different parameters, particularly useful in scenarios with sparse data. For each parameter, it maintains a running total of the squared gradients, which is used to adjust the learning rate inversely proportional to the square root of this accumulated sum. The update rule for each parameter is given similarly as for RMSprop by:

$$\beta_{\text{new}} = \beta_{\text{old}} - \frac{\eta}{\sqrt{\delta + g}} \nabla C(\beta_{\text{old}}), \tag{2.4}$$

The difference here however is g which represents the accumulated squared gradients (as opposed to the running average maintained in RMSprop). This adaptive learning rate mechanism allows parameters with frequent updates to experience a reduction in learning rate, contributing to more stable and robust convergence, particularly beneficial in handling noisy gradient information.

2.2.5 Adam

Additionally, we extended the SGD with the Adam optimizer, which combines the benefits of both AdaGrad and RMSprop, further improving the stability of the convergence. Adam adapts the learning rate for each parameter based on first and second moments of the gradients:

$$\beta_{\text{new}} = \beta_{\text{old}} - \frac{\eta}{\sqrt{\hat{r}} + \delta} \hat{s}, \tag{2.5}$$

where $\hat{s}$ and $\hat{r}$ are the bias-corrected first and second moment estimates, respectively. By bias-corrected we mean we divide by 1 minus the exponential decay rate for each time step; $\hat{s}_t = \frac{s_t}{1-\beta_1^t}$, $\hat{r}_t = \frac{r_t}{1-\beta_2^t}$. Where we use the recommended values $\beta_1 = 0.9, \beta_2 = 0.99$ [3].

2.2.6 Momentum

For all the methods we have also incorporated the option to add momentum with a coefficient γ, to accelerate convergence by combining the gradient direction from the previous step, capturing the momentum of the parameters.

The momentum mechanism works by maintaining a velocity vector (v), which is updated at each iteration and then used to update the parameters (β). The update rule for the velocity and the parameters is as follows:

$$v_{\text{new}} = \gamma v_{\text{old}} + \eta \nabla C(\beta_{\text{old}}), \tag{2.6}$$

$$\beta_{\text{new}} = \beta_{\text{old}} - v_{\text{new}}, \tag{2.7}$$

where v_{old} is the velocity from the previous iteration, γ is the momentum coefficient, η is the learning rate, and $\nabla C(\beta_{\text{old}})$ is the gradient of the cost function with respect to the parameters. The momentum coefficient γ typically takes a value between 0 (no momentum) and 1 (very high momentum). When γ is set to 0, the method reduces to standard gradient descent without momentum. We shall see $\gamma = 1$ can cause very poor results. We will compare convergence with different γ using plain gradient descent.

2.2.7 Methods for computing the gradient

For the OLS, Ridge and Lasso optimisation problems one can derive closed form expressions for the gradient. When turning to neural networks we will need methods for automatically calculating the gradient. We therefore import tools from the library JAX. We also make big improvements in runtime by using the function jit provided by JAX [1]. We see in figure 21 of the appendix that there is no difference between the gradient derived by hand and the one calculated using JAX, verifying this implementation.

2.3 Implementing the Neural Network

Our implementation of neural networks will rely on gradient descent methods implemented in the previous section. We now look at some definitions and theoretical background relating to neural networks.

2.3.1 Neural Network - theoretical background

We begin by recalling the structure of a neural network. A visualisation is given in Figure 1. The network consists of *nodes*. The leftmost are known as *input nodes*. Each column is referred to as a *layer*. In the case of the Franke function, we have 2 input nodes, one for each coordinate of the grid.

The final layer is known as the *output layer*. For the Franke function the height z is the only node in the output. Between these layers are the *hidden layers*. We denote vertex i of layer l as h_i^l. The input layer will be considered the 0th layer, i.e. $h_0^0 = x, h_0^1 = y$. Between the layers are *edges* with given *weights*. We denote the weight from node i in layer $l-1$ into node j in layer l by w_{ij}^l. Each node is also assigned a bias b_i^l.

A final important component is the *activation function* $f : \mathbb{R} \to \mathbb{R}$. In principle this can be different for each layer or even for each node. We will use one activation function for the hidden layers, and also have the option of using a different one for the output. Some examples of activation functions are given in Figure 2.

A prediction is made through *forward propagation*. For the description of forward propagation, assume that we have computed $h_1^{l-1}, \dots, h_n^{l-1}$ for the n nodes in layer $l-1$. The value of node j in the next layer is then computed as:

$$h_j^l = f\left(\sum_{i=1}^{n} w_{i,j}^l h_i^{l-1} + b_i^l\right). \tag{2.8}$$

This process of signals "propagating" through the network motivates the name.

Given a cost funtion, we can optimise the weights and biases of the neural network using the backpropagation algorithm. In our implementation, this is taken care of by the automatic differentiation provided by JAX as described in section 2.2.7.

2.3.2 Activation functions

We will run experiments using different activation functions. The ones we test are sigmoid, ReLu, leaky ReLu and tanh, all shown in Figure 2.

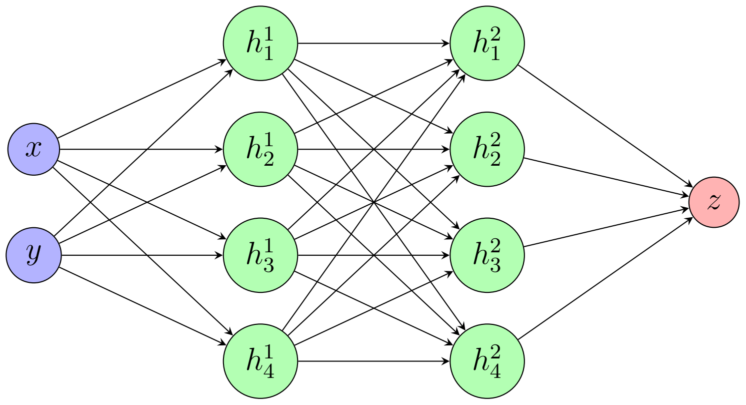


Figure 1: One choice of Neural Network architecture for the Franke Function.

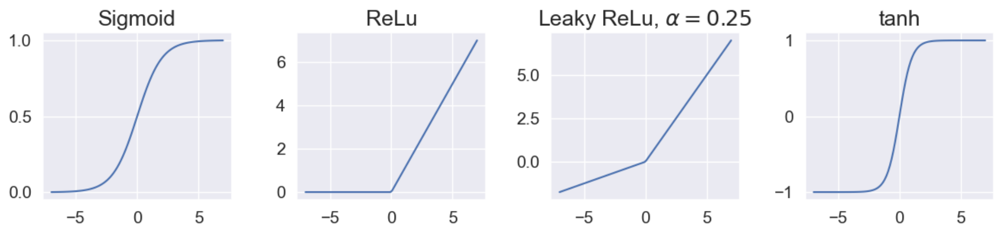


Figure 2: The activation functions used in this project.

We will see that the choice of activation function is of great importance. For the Franke function we choose the identity function for the final layer, for reasons explained later.

2.3.3 Initialisation

Through our experimentation, we found the following initialisation of weights and biases to perform well. We initialize the weights using a normal distribution. Some adjustment are made in the form of Xavier initialisation [5].

The biases are initialised to 0, as to not cause large errors in the beginning due to an unlucky bias selection in the final layer. For the Franke Function for instance, we might have started out with too high of a surface.

To provide an example of initialisation gone wrong, we mention that when performing classification we first initialised all weights in the range $[0, 1]$. This often led to a vanishing gradient as the positive numbers accumulated and when passed through the sigmoid gave a very low gradient.

2.3.4 Initial regression experiments

We focus first on the regression problem given by the Franke Function from project 1 using the MSE with and without ridge regularisation to be able to compare with our previous regression results.

We will experiment with different numbers of hidden layers and nodes, as well as different activation functions, learning rates, epoch numbers and regularisation. Some of the results will be useful in the remainder of the project.

2.3.5 Classification analysis

Moving on from the Franke function, our next goal is to perform classification analysis. In particular, we study the Wisconsin Breast Cancer data set [6]. Here the inputs are not x and y coordinates, but rather numbers relating to the attributes of the tumour. Thus, the input layer shown in figure 1 changes.

The most important metric for studying the performance of our classification models is the *accuracy score*; that is the proportion of correctly guessed targets:

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} I(t_i, y_i)}{n}, \tag{2.9}$$

where I is the *indicator function* equaling one whenever $t_i = y_i$ and 0 otherwise. Here t_i represents the target and y_i the outputs of our classifier. n is the number of targets t_i evaluating the accuracy on.

We will perform analysis of how the model depends on the various hyperparameters like learning rate η, regularisation parameter λ, activation functions $f : \mathbb{R} \to \mathbb{R}$, number of hidden layers and number of nodes per layer. Results on optimal number of minibatches are assumed equal to previous results. We conclude the experiments with a surface plot of the model prediction.

2.4 Logistic regression

We will also compare the results from the neural network (high complexity model) with the results obtained from Logistic regression. In this model, the prediction $\hat{y}^{(i)}$ corresponding to datapoint $\boldsymbol{x}^{(i)}$ is given as:

$$\hat{y}^{(i)}(\boldsymbol{\beta}, \boldsymbol{x}^{(i)}) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1^{(i)} + \cdots + \beta_n x_1^{(d)})}}. \tag{2.10}$$

Here d is the dimensionality; the number of features. In our case features are properties of the possibly malignant tumor in the Wisconsin breast cancer dataset. We notice that this predictor gives a special case of a neural network with no hidden layers. We recognise the expression $\beta_0 + \beta_1 x_1^{(i)} + \cdots + \beta_n x_1^{(d)}$ as bias in the output plus the contributions from the input layer. This sum is passed through the sigmoid function to produce the output in the final layer.

The Logistic regression is performed by minimising the so called *cross entropy* cost function given by:

$$C(\boldsymbol{\beta}) = \frac{1}{n} \sum_{i=1}^{n} -y^{(i)} \log(\hat{y}^{(i)}) - (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}), \tag{2.11}$$

where n is the number of data points. We minimise this cost function using the Adam algorithm developed earlier, as we find this to be the best algorithm. Having found good values for batch size, we only use learning rate as hyper parameter and study the dependence of this. We then add an l_2 regularization parameter λ, obtaining the modified cost function:

$$C_\lambda(\boldsymbol{\beta}) = C(\boldsymbol{\beta}) + \lambda \|\boldsymbol{\beta}\|_2^2. \tag{2.12}$$

In Section 5.5 of the appendix, we compare our result with results obtained using logistic regression functionality provided by Scikit-Learn[4] to verify that our implementation performs as expected.

3 Results and analysis

3.1 Descent methods

We begin by looking at some experiments performed by minimising the OLS and Ridge cost functions from project one. In all of the below, we have used a dataset with 100 points from the Franke function. We have scaled the points by subtracting the mean and dividing by the standard deviation. The dataset is split 80-20 for training and test. We fit the points to a polynomial with 10 features, chosen as in the first project. This corresponds to a complete degree 3 polynomial [2].

$$p(x, y) = \beta_{0,0} + \beta_{1,0}x + \beta_{0,1}y + \cdots + \beta_{0,3}y^3. \tag{3.1}$$

Except for the final Ridge experiment, all plots optimise the MSE error, i.e. we perform OLS regression.

3.1.1 Epoch experiment

Our first experiment is shown in Figure 3. It is simply to plot the test error during training for each method. One thing we immediately notice is that the test error is below train for plain GD. This is a result of initialisation and datasplit; in this run the test error starts out smaller, and as the model learns the surface it increases along with the train error until train catches up at 1000 epochs.

The plot shows that plain gradient descent is significantly outperformed by all other methods. AdaGrad shows slight improvement in performance over stochastic gradient descent. The clear winners are RMSprop and Adam. Some parameter tuning might change the results, so we continue our experiments.

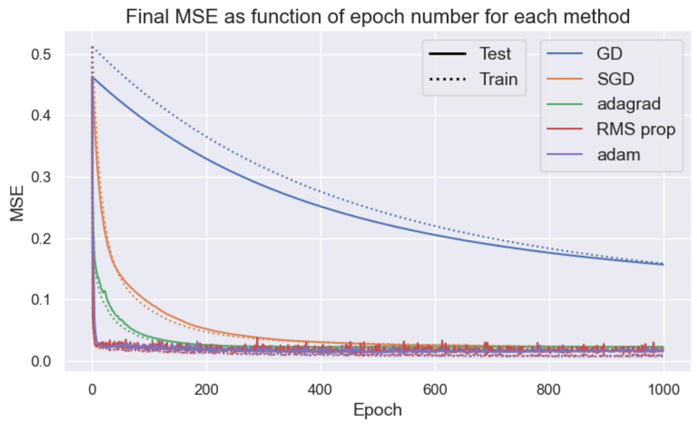


Figure 3: Final MSE for each method using up to 1000 epochs. Train is shown as dotted lines and test as whole. For the SGD based methods, number of batches is 5.

3.1.2 Batch size

For the methods based on stochastic gradient descent, an important parameter is the batch size. The final test error results for various batch sizes are shown in Figure 4, where we first remark that with 1 batch, GD and SGD coincide as expected.

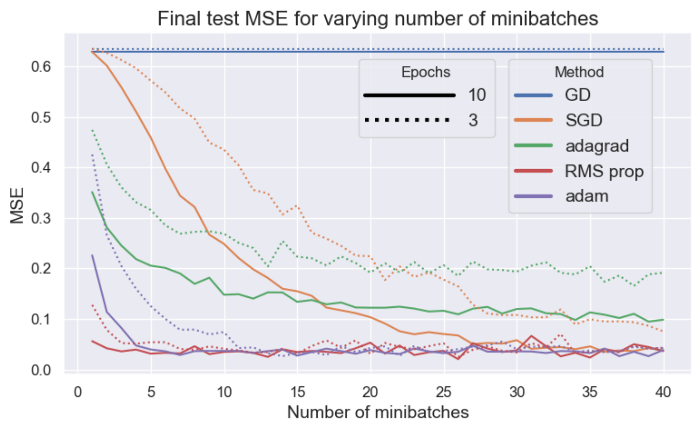


Figure 4: The final training error for each method for different number of mini-batches. The dotted lines show the effect of training for fewer epochs. A learning rate of 0.01 was used.

We see that the loss is a decreasing function in number of batches. The strategy of doing one small step for each batch means that we get new information for the later batches, so the steps are more precise.

This does not however mean that we want to choose as many batches as possible. The computational cost increases with the batch number since we need to do the forward and backward propagation more times per epoch. In conclusion, we must strike a balance between gain in convergence and cost in computation. We find that 5 mini-batches give good results.

3.1.3 Learning rate

Tuning the learning rate is clearly of great importance: if it is too big we might jump across all minima, if it is to small we might not get there in time. Figure 5 shows the results from an experiment with varying learning rate.

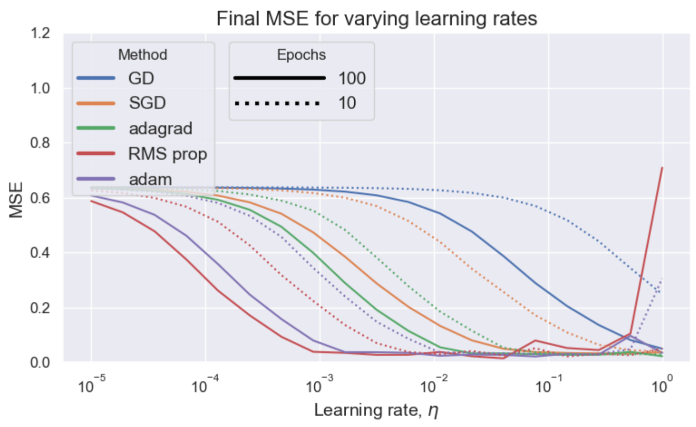


Figure 5: The final training error for each method for different learning rates. The dotted lines show the effect of training for fewer epochs. For the SGD based methods, 5 batches are used

This plot shows that, while RMSprop is slightly better for small learning rates, Adam seems overall more stable. For instance we see that RMSprop performs poorly if the learning rate gets a little too big. During experimentation, we also plotted errors across the epochs of training, and we observed that RMSprop jumps between low and high error at high learning rates. AdaGrad is here comparable to SGD when looking at final error for 100 epochs. The learning rate 0.01 seems to be optimal for Adam and RMSprop.

3.1.4 Regularisation

Having found Adam with 100 epochs to be a good model, we use this in the following experiment.

The goal of Ridge regression is to penalise increase in the parameters to get a more stable result. Detailed discussion on the ups and downs of regularisation can be found in Project 1 [2]. Figure 6 shows how regularisation above $\lambda = 100$ gives about the same results; the parameters are set to 0 to minimise penalisation and the squared error of the points is incomparable to the error introduced by changing the parameters. For low learning rates we do not converge to a good solution.

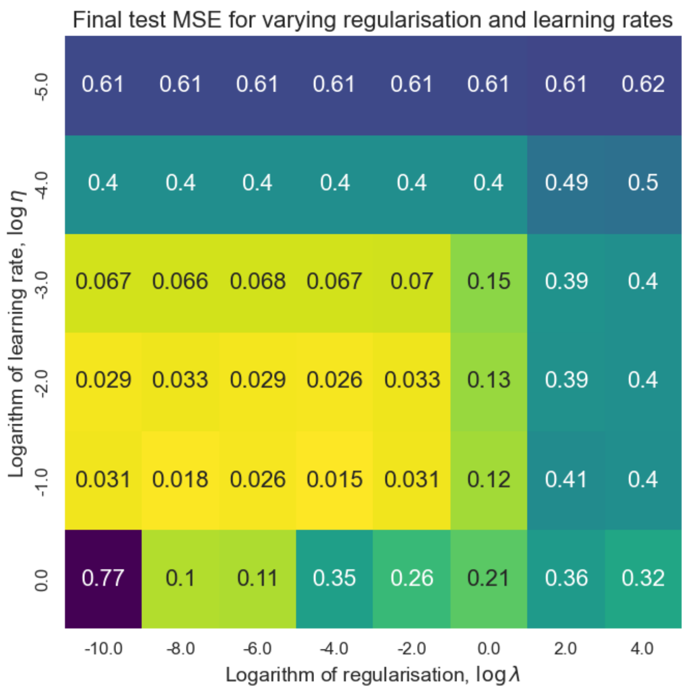


Figure 6: Final test MSE for different values of regularisation λ and learning rates η using 100 epochs. Experiment done with the Adam optimizer.

3.1.5 Momentum

Momentum was added to each method as a way of speeding up the learning process. Since plain gradient descent had worst convergence rate, we show the results for this using momentum in Figure 7, to see if we can improve. We include one plot using 1000 epochs and one with 10000 epochs to emphasize the following point; momentum only speeds up the training. We see in the leftmost plot that for 1000 epochs, a momentum parameter of 1 gives the best convergence for all learning rates. In the rightmost plot for the learning rate $\eta = 10$, we have reach about the same point independent of γ. There is one notable exception, namely $\gamma = 1$. The issue here is that there is no dampening of the first steps, meaning that these are carried through the whole training, giving significantly worse results.

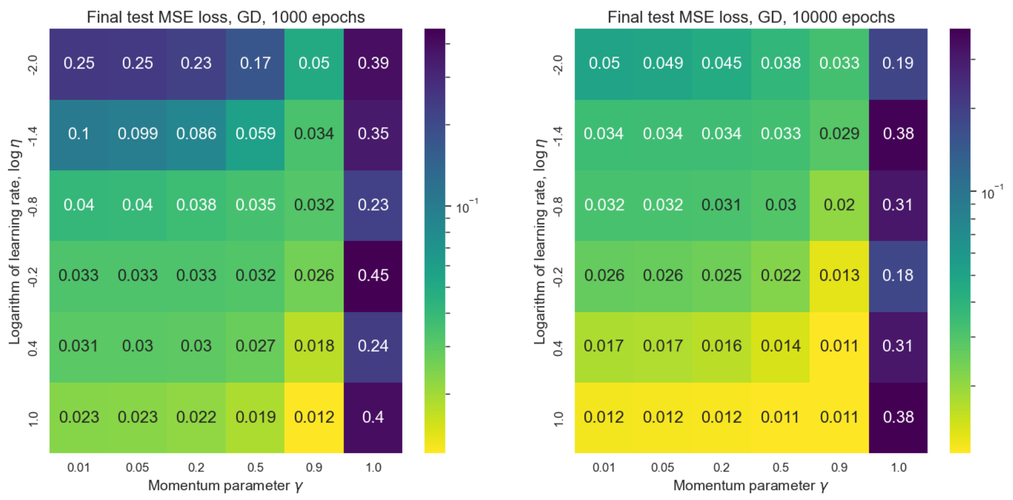


Figure 7: Final test MSE for different momentum γ and learning rates η for 1 000 and 10 00 epochs using plain gradient descent.

3.1.6 A closer look at the result

While plots of error are nice, it can give new insights to plot the our predicted surface. This will also be important when evaluating how OLS compares to a neural network model in the next section. Figure 8 shows this result. This is not a very good fit, but it does roughly the highs and lows of the surface. This motivates our attempts with fitting a neural network to the data.

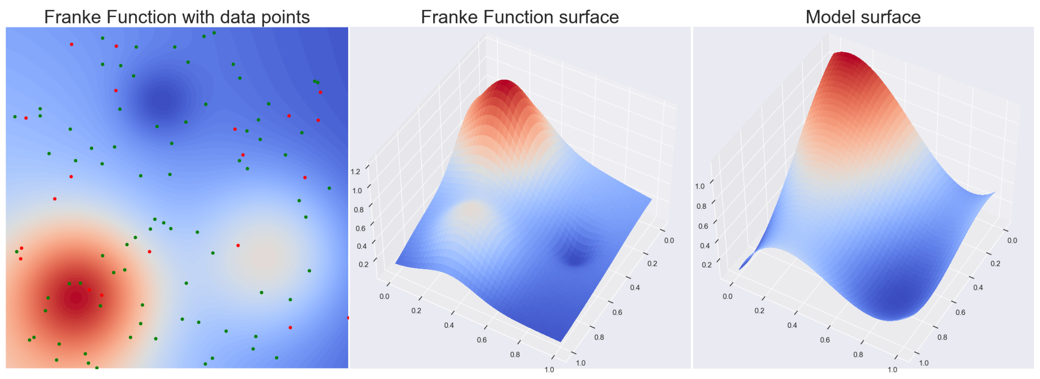


Figure 8: The Franke Function and the resulting surface for training with Adam using 1000 epochs and learning rate 0.01. The leftmost plot shows the sampled data points; training in green and test in red.

3.2 Neural networks for regression

With a solid toolbox of descent methods we are ready to investigate neural networks. Before making a choice of what is the best model, we investigated how the various parameters affect training, namely learning rate, regularisation and choice of activation function.

3.2.1 Activation function for final layer

When making the choice of activation function, we initially made the error of assuming that the Franke Function is non-negative. With this in mind, we chose ReLu for the final layer. This led to issues with the gradient sometimes being initialised to zero. This happens when enough weights in the final layer are initialized with negative values putting the output in the zero-gradient region of the ReLu function.

A potential solution could involve initializing all weights as positive. However, we anticipate that this approach might lead to the issue of gradient explosion. Therefore, a more effective strategy would be to modify the final activation function. Additionally, upon further examination, we realized that the Franke Function occasionally dips below zero. Given this observation, opting for the identity function as the final layer's activation proved to be the best solution.

This experience gave the following insight; one must be careful with using ReLu in the final layer.

3.2.2 Learning rate

We start by looking into the effect of the learning rate on how the model learns and how it affects the stability of the training process. We will use a regularization parameter of 10^{-5} and a model architecture consisting of 3 hidden layers and 5 nodes in each hidden layer. These parameters will be looked into more deeply in later sections, but it should be noted that the choice of these parameters might affect the relationship between the learning rate and model performance.

We also used a constant number of epochs in all the training runs which may impact which of the learning rates are seen as viable as some of them might have converged to a better model if given more training time.

Due to limited computational resources available during our experiments, we had to impose certain constraints. To investigate the impact of activation functions on the learning process, we conducted experiments using four different activation functions. This approach allowed us to assess whether these functions influence the relationship between the learning rate and the model's learning efficacy.

To mitigate the influence of outliers stemming from the random initialization of parameters, we averaged the results of five distinct training runs for each learning rate value. This approach enabled us to gain a more accurate understanding of the performance across different segments of the learning rate spectrum, identifying which ranges consistently yielded better outcomes.

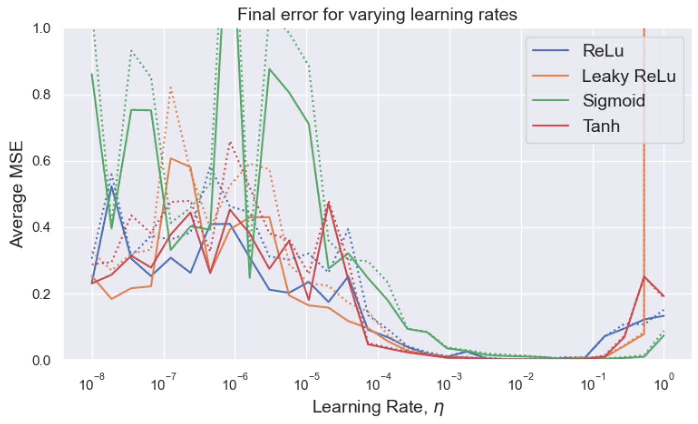


Figure 9: The final test error for each activation function for different learning rates. The dotted lines show the respective final training error. Training done with three hidden layers, five nodes per hidden layer and regularization $\lambda = 10^{-5}$. Adam optimization scheme was used with 300 epochs. The training was repeated 5 times and the values were averaged to get the final values in the plot.

Figure 9 reveals that there exists an optimal 'Goldilocks zone' for the learning rate, specifically within the range of 10^{-4} to 10^{-1}. In this zone, training consistently yields high-performing models, irrespective of the activation function used in the hidden layers. Additionally, we observe a notable decline in model performance when the learning rate falls below 10^{-4}. This decline becomes particularly pronounced and unstable below 10^{-5}. This pattern suggests that, with the given number of epochs, the model struggles to converge to an effective solution and remains relatively close to the random initialization.

The observed phenomena might also be attributed to the vanishing gradient problem, where gradients approach zero due to the recurrent application of low-gradient activation functions. This is particularly evident in the plot of Figure 9, where models using the sigmoid function display the most unstable performance, with average MSE values exceeding one in some instances. The inherent nature of the sigmoid function, having gradients less than 0.25 across its domain, makes this issue even worse. In our model with three hidden layers employing sigmoid activations, the initial layers' gradients are likely quite small. This, compounded with a low learning rate, results in minuscule updates to the model's weights, rendering the learning process ineffective.

When the learning rate exceeds 0.1 we also see that the training error starts increasing. This is most likely due to the model struggling to stabilize at a minimum due to each update of the weights being too large and the model 'jumps' over the minimum causing the gradient of the model to not decrease and getting stuck jumping back and forth across the minimum. For the leaky ReLu activation function we also see tendencies of the learning rate being large enough to cause the gradient to start diverging making the loss of the function explode.

Since 10^{-2} is roughly in the middle of the region where the loss is low and stable for the model we will use this as the learning rate going forward when experimenting with the other hyperparameters of the model. Although this will not necessarily give us the absolute best combination of hyperparameter it will still be a good baseline which we can use to investigate how the model improves or worsens as we vary the other hyperparameters.

3.2.3 Regularization

A second important parameter to tune is the regularisation parameter λ. This parameter ensures that the model parameters do not explode by penalising large parameters. Figure 10 shows how the final test error depends on the regularisation parameter.

We observe that large regularisation makes the model unable to converge to something usable. With ReLu we see some instability for low regularisation. The dotted lines represent the training error, so it seems surprising that this is greater than the test error. The explanation is surprisingly simple; for large enough regularisation it is most beneficial to put all parameters to zero. Our estimate of the Franke function then becomes the 0-surface, and the error in this plot is the average distance from this. In other words it only shows that the test set on average is nearer 0 for this specific sampling.

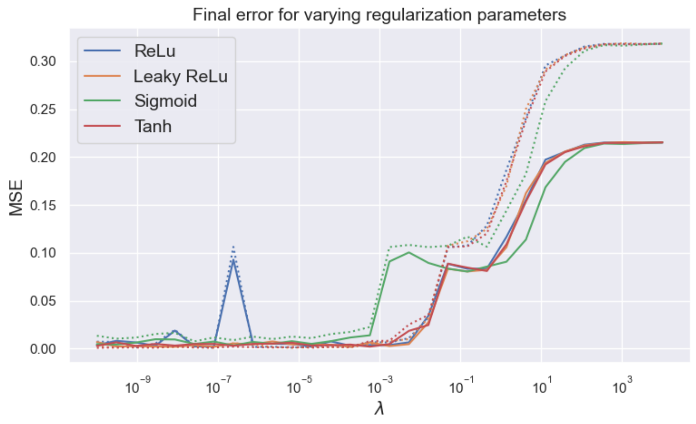
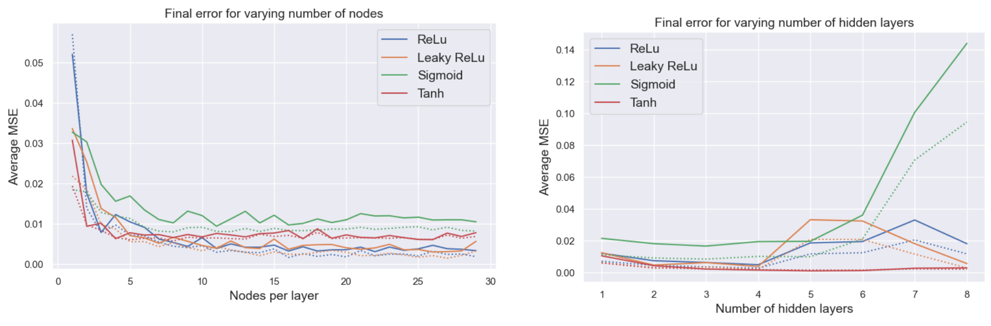


Figure 10: The final test error for each activation function for different regularization parameters, λ. Training done with three hidden layers, five nodes per hidden layer and a learning rate of 0.01. The Adam optimization scheme was used with 300 epochs.

We repeat that the use of regularisation means that we solve a slightly different problem. What we measure here is the error we want to minimise, and we see that for large regularisation we deviate too much from our original problem. In other words there is a trade-off between the stability and robustness obtained from regularisation and the deviation from the original optimisation objective.

3.2.4 Model Architecture

We do two initial experiments with varying number of nodes and hidden layers before making final decisions on activation functions. We first ran an experiment where we used one hidden layer and varied the number of nodes in the hidden layer. We ran the model with the learning rate and regularization parameter that we found in the previous sections. After finding a good value for the number of nodes per hidden layer we ran an experiment varying the number of hidden layers. We ran these experiments for all four hidden layer activation functions to get a good view on how they affect the scaling of the model architecture.



(a) Node experiment

(b) Layer experiment

Figure 11: The final test error for each activation function for different number of nodes and hidden layers. For a) we use 1 hidden layer and for b) 5 nodes per hidden layer. A learning rate of 0.01 and a regularization parameter of 10^{-5} is used in both, with the Adam optimization scheme with 300 epochs. The training was repeated 10 times and the values were averaged to get the final values in the plot.

In Figure 11a we see a clear trend that when using a low number of nodes in the hidden layer the model has an MSE value closer to what we got using OLS. When the number of nodes in the hidden layer increases the loss decreases until it plateaus between 0.005 and 0.01. We also see that there are some differences in where the different activation functions converge. The runs using the sigmoid function stand out as the one which converges to a higher loss value than the other three, while ReLu and leaky ReLu are both slightly lower than the tanh function. This might be because the runs are done with a constant number of epochs and the sigmoid and tanh functions may learn slower due to having the effect of contracting the gradient, while the ReLu functions has no effect on it as long as the value of the nodes are above 0.

For Figure 11b, where the number of hidden layers are varied, we see a slightly different pattern where upon increasing the number of hidden layers from one to four we almost don't see any change in the loss of the model. However when the model has more than four hidden layers we see that for some of the activation functions the loss starts increasing. Especially we see that the sigmoid function struggles when the number of layers increase. This might be a symptom of the vanishing gradient problem. Since the gradient of the sigmoid function reaches its maximum at 0.25, each time the activation function is used the gradient will be contracted by a factor of 0.25 or less. This will cause the earlier layer in the models to have a very small gradient and might prevent them from being able to learn causing worse performance. We might expect to see a similar effect for tanh since its derivative also is lower than 1 at most points, however since the derivative reaches 1 when the value is close to zero it will keep the gradient at a reasonable level as long as the magnitude of the nodes in the network do not get too large. This is likely the case in our experiments since we have both scaled the input data and initialized the weights in a way such that the hidden nodes will not get a value close to zero. We also notice that the ReLu and leaky ReLu functions have an increase in loss when the model has more than four layers and although this is not as much of an increase as we see with the sigmoid function, it is still a significant increase when compared to the tanh function which stays at its low level of loss for all of the tested number of layers.

3.2.5 Hidden layer activation

Throughout the experiments in this section we have seen that the different activation functions have had different effects on the performance of the model. We will now make an analysis of which of the activation functions provide the best and most reliable performance and make a choice as to which activation function we will use for our final model.

From our experiment on learning rates we saw that the sigmoid function struggles relatively more than the other functions when trained with a low learning rate, possibly due to the parameter updates getting too low due to the already contracted gradients of the sigmoid function. A similar effect of the sigmoid function is likely seen when increasing the number of layers as the gradients contracts more for each layer added, causing the loss to drastically increase for larger models. This, combined with the models using the sigmoid function consistently performing worse across most layer sizes in Figure 11a lets us conclude that the sigmoid function is not the best choice for an activation function of the hidden layers, both due to worse performance and worse stability in terms of the specific hyper-parameters chosen.

For the ReLu and the leaky ReLu functions we see good performance on the regularization experiment as well as in the experiment with varying layer sizes. In the experiment with varying learning rates however we do see signs of the leaky ReLu function potentially experiencing the exploding gradient problem as the learning rate gets close to 1. Also when increasing the number of layers we see a trend of both functions performing almost one order of magnitude worse than the tanh function for most of the models with more than 4 layers. Since we may be limited to using a lower number of layers and a lower learning rate to achieve optimal performance with these functions we conclude that they will not be the best choice for this solving the task of predicting the Franke function. It should however be noted that since these experiments were ran with the same number of epochs for all methods it could be that the models using ReLu or leaky ReLu would have converged to as good or better solutions if they were given a longer training time. These scaling properties of the model is something we have not been able to experiment with extensively due to a limited compute during the experimentation process.

Lastly we have the tanh function which we see perform well throughout all of the experiments. It has similar or better performance than the other functions throughout all of the experiments we have performed. Since it performs the best across the largest range of hyper-parameters it gives us more freedom to explore the space of hyper-parameters without running into issues like the vanishing or exploding gradient problem which will make it easier to find a better performing model. We will now use this function for the final section on neural networks in regression.

3.2.6 Choosing the final architecture

Figure 22 of the appendix shows that we can keep using a learning rate of 0.01 even as we increase the number of layers. With this, we perform our final experiment whose results are shown in Figure 12. The figure indicates that the best architecture is 3 layers with 16 nodes. However, we argue that the better choice is a model around 4 layers and 7 nodes. This is both a simpler model and the neighbouring models are better, so it should be more stable to choice of data. We note that the parameter tuning process we are going through has the drawback that we are in a sense training on the test set, and the phenomenon of a good model lying in the midst of bad ones might be a consequence. Another benefit of choosing the simpler model is the reduction in computational cost when using it.

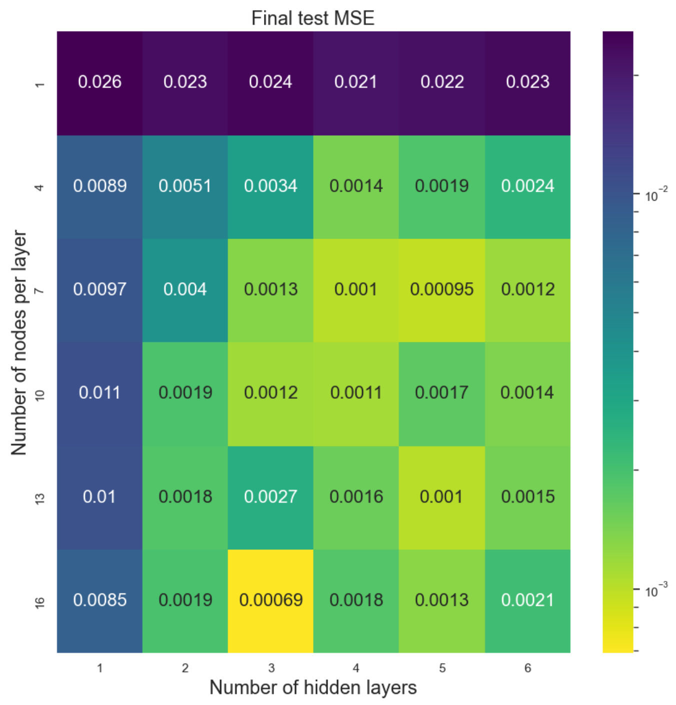


Figure 12: The final test error with varying architecture. Training done with $\lambda = 10^{-5}$. The Adam optimization scheme was used with 300 epochs and the activation function for the hidden layers was the tanh function. The training was repeated 5 times and the values were averaged to get the final values in the grid.

An important part of the exploration was to plot the surface obtained from the neural network. This makes it easier to make a judgement of whether the model is good. The result of our final model choice is shown in 13.

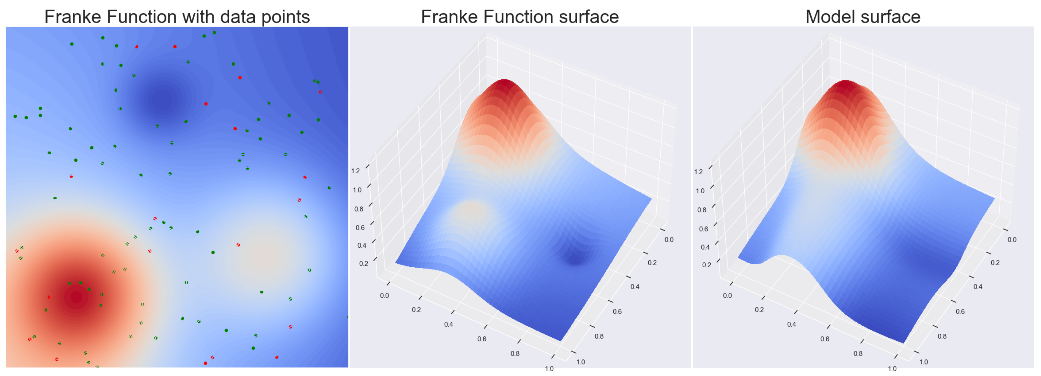


Figure 13: The result using 4 layers and 7 nodes. The leftmost plot shows the sampled data points; training in green and test in red.

The first thing we note is that some features of the surface are lost. One example is that the valley loses some depth. When looking at the data points, this is not that surprising as there are few points to capture this feature, and none lie in the deepest areas. The model complexity might not be great enough even if more data was included, but it still underscores the importance of having enough data so that the model is able to capture all edge cases. A similar argument can be made about points along the edges, where we see that there are some discrepancies.

Referring back to Figure 8, we believe this the neural network model to be much better. It seems to clearly capture the Franke function despite having no more data than was used for OLS. The drawback of OLS is that we have to make assumptions about the model; in our case we assume that it can be approximated well by a degree 3 polynomial. The neural network however does not have these assumptions forced into the model. We believe this freedom to be among the reasons it gives a better model.

3.3 Classification task

We will now turn to a different class of problems from the one studied in the previous section, namely binary classification problems, where we will try to make a model dividing a set of points into one of two classes. The dataset we will use is the Wisconsin breast cancer dataset where we have data on a set of tumours and whether they have been classified as malignant or benign. We will still apply similar techniques to the ones in the previous section, such as using a deep neural network to predict the class of the tumours, but with an important difference in the cost function used to optimize the model as well as the activation function used in the output layer. Since we want our output to be a value between zero and one, with one indicating a malignant tumour and zero indicating a benign one, we use the sigmoid function as the output activation function. We also use the binary cross-entropy loss function as it fits better for optimizing outputs binary class outputs. To assess the quality of our models we will use the accuracy measure given as Equation 2.9, since we primarily care about how many of the data points are correctly predicted and not necessearily how close the outputs are to either one or zero.

3.3.1 Learning rate

We start by investigating how the learning rate affects the accuracy of the model for the different hidden layer activation functions. To get more reliable results we have here also averaged the results over five training runs with the same parameters. This will help us look beyond the random changes in performance as a result of the random initialization of the weights. It should also be noted that in contrast with the previous section we have only ran the models with 100 epochs per training run instead of the 300 we used for the regression problem.

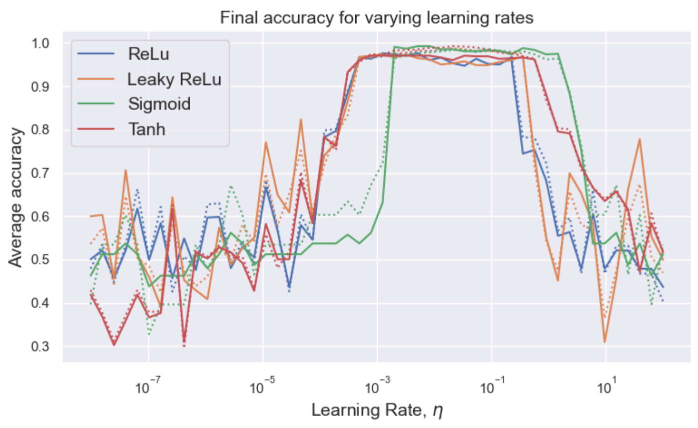


Figure 14: Final test MSE for each activation function for different learning rates. Training done with three hidden layers, five nodes per hidden layer and a regularization parameter of 10^{-5}. The Adam optimization scheme was used with 100 epochs. The training was repeated 5 times and the values were averaged to get the final values in the plot.

We see in Figure 14 a similar pattern here as we did for the regression problem in the previous section where for too low learning rates, the model performance is unstable, likely due to learning to slowly to reach a minimum in the time given to train. This is supported by the clustering of the models around an accuracy of 0.5, a fair coin-flip, indicating that the model has not actually learned anything about the data and is only making a random guess based on the randomly initialized weights. We also see that the model struggles when the learning rates above 0.05. This probably gives a worse performing model for the same reason as in the regression case, where the model struggles to converge due to too large jumps in each training step.

We also see some differences between the activation functions in how they respond to the varying learning rate. Most noticeably we see that the sigmoid function reaches a high accuracy only for a much higher learning rate than the others. This is likely due to the contracting gradient effect of the sigmoid function being worsened by a low learning rate and so it needs a higher learning rate to compensate. We also see that it falls in performance later than most of the other models. This is again likely due to the contracting gradient and the higher learning rate compensating for each other allowing it to be stable even at a significantly higher learning rate. Another difference is that the ReLu and leaky ReLu functions have a much steeper drop off in performance as the learning rate increases than the tanh function. This may be because they are more prone to hit issues with an exploding gradient as the learning rate passes some threshold which the tanh function does not experience due to its gradient's slightly contractive properties.

3.3.2 Regularization

We will now test how the regularization parameter affects the performance of our neural network model. Here we will also test all the four activation functions to assess whether they have any major differences.

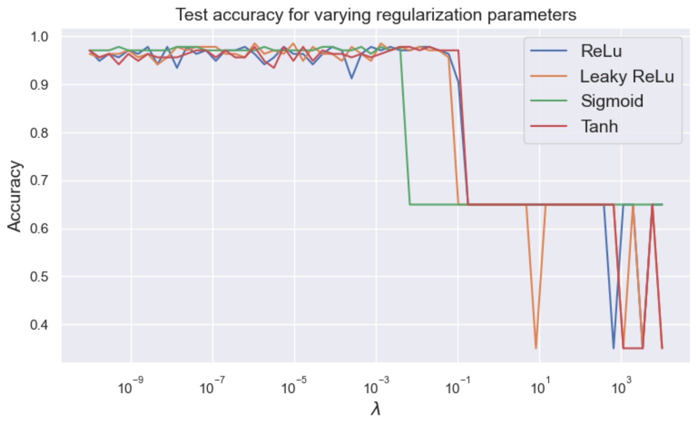


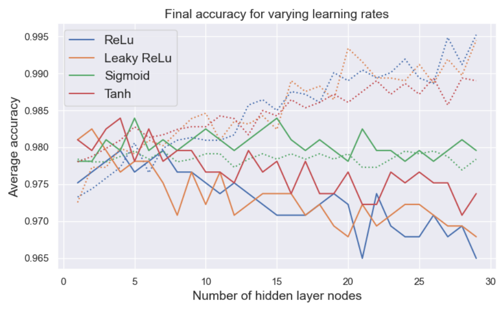
Figure 15: The final test accuracy for each activation function for different regularization parameters. Training done with three hidden layers, five nodes per hidden layer and a learning rate of 0.01. The Adam optimization scheme was used with 100 epochs.

We see in 15 that the model behaves in a similar way to the neural network model for the Franke function. It performs well with a low value for the regularization parameter and then at around 0.1 the accuracy starts degrading significantly. We also see here that the sigmoid function performs worse than the other models as it worsens already when the regularization parameter nears 0.01.

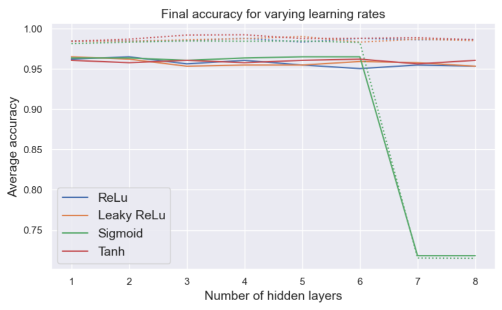
We also see that when the model reaches its low performing range of the regularization parameter it tends to jump between two values. The rationale for why this is happening is that the model has made most of the weights negligible and so the output before the final sigmoid activation of the output is very close to zero, meaning that the final output is right at the boundary of 0.5. This likely causes the small variation in the final layer bias to decide whether the model outputs a value above or below 0.5 for all the training examples, which would cause it to either predict all the examples as true or all the examples as false. The difference between the two accuracies are probably because the test set does not have a 50-50 distribution of malignant and benign examples.

3.3.3 Model Architecture

We now want to study how the model size affects the performance of the model. Similarly to the regression case we will first look at a varying number of nodes with a single hidden layer to find a good number of nodes per hidden layer and then vary the number of layers to see if it has any impact on the accuracy of the model.



(a) Node experiment



(b) Layer experiment

Figure 16: The final test accuracy for each activation function for different number of nodes per hidden layer and different number of layers. In a) one hidden layer is used and in b) 5 nodes per layer. Training done with a learning rate of 0.01 and a regularization parameter of 10^{-5}. The Adam optimization scheme was used with 100 epochs. The training was repeated 5 times and the values were averaged to get the final values in the plot.

In Figure 16a we see an interesting pattern emerge as we increase the number of nodes in the hidden layer. For all of the activation functions except the sigmoid function we see an increase in training accuracy as the model gets bigger and at the same time a decrease in the test accuracy. This strongly indicates that the model overfits to the training data more and more as the model gets more nodes per hidden layer. This might be because at only a single hidden node the model already performs very well with an accuracy of around 0.98. Since it is difficult to perfectly predict the pattern in the dataset due to the inherent noise it has as a real world dataset, the extra nodes are likely only used to predict the noise of the training data instead of the underlying pattern in the data. As the model tries to predict more of the noise in the training set it also loses accuracy on the test data which has noise unrelated to that of the training set. Interestingly we see that with the sigmoid activation function the model does not have the same overfitting pattern and it keeps its performance at the same level as the number of nodes increase. This might be because the sigmoid has a slight slowing effect on the learning making it more difficult for the model to learn the more intricate noise in the training set.

As the model performs best at a low number of hidden layer nodes we chose five nodes as the number of nodes per hidden layer as it still gives us enough complexity to potentially improve the model prediction, while also not experiencing the overfitting like the larger models did. When training the model with a varying number of hidden layers, and with each layer having five nodes, we don't see any clear pattern in most of the results. Both the training and test accuracy is kept relatively constant with the training accuracy being slightly higher. We do however see that when we hit seven and eight layers for our model the sigmoid function collapses to a far worse performance than the other models. This is most likely a similar phenomena as the one seen with the regression case where the number of layers gets so large that the gradient is contracted repeatedly by the sigmoid function causing it to vanish in many of the early layers. This would have prevented the model from efficiently learning the pattern in the training data and could be the cause of the worsened performance.

3.3.4 Hidden layer activation function

Now that we have seen how the different activation functions perform across all the different hyper-parameters investigated we are ready to make an analysis of which function is best suited to solve the classification task at hand.

After this we will use that function when we try to find the optimal model for solving the problem.

We start by looking at how the sigmoid function performed throughout our tests. While it reaches the same kind of performance as the others at its peak, we also have instances where its performance drops off when the other methods remain at their peaks. Most noticeably we see that in the experiment varying the number of hidden layers the sigmoid function is the only one where the performance degrades when trained with a high number of layers. In addition to this we also see that for the regularization experiment we would have to use a much lower regularization parameter in order to avoid the performance drop off that comes with too much regularization. These two factors mean that if we were to use the sigmoid function we would be more limited in what hyper-parameters we can choose and still get a good result. It should however be noted that although we will not use the sigmoid function it still had the advantage in the experiment looking at the hidden layer sizes, where it was the only function which managed to avoid overfitting when increasing the model size. This could be an important factor, but as the overfitting of the other functions were not too drastic we still conclude that they would be the better choice to reach optimal performance.

When it comes to deciding between the three activation functions ReLu, leaky ReLu and the tanh function we see throughout the experiments that there is not a significant difference in their performance in any of the four experiments. The only experiment where we see some sign of better performance is in the learning rate experiment where the tanh function performs better at higher learning rates than the two other activation functions. We will therefore choose the tanh function as our activation function going forward, while we note that we could probably have chosen any of the three remaining activation functions and still likely gotten a model performing at a similar level.

3.3.5 Final Architecture exploration

With the preceeding results in mind we will run one final test assessing how the model behaves as we vary both the learning rate and the number of hidden layers. The reason for doing this experiment is the learning rate has different paradigms where its performance is drastically different and since adding more hidden layers create gradients that go through more activation functions it might have an effect on the stability of the model at different learning rates. We therefore sample a grid of different learning rates and number of hidden layers to see if they have some effect on each others relationship to the model accuracy.

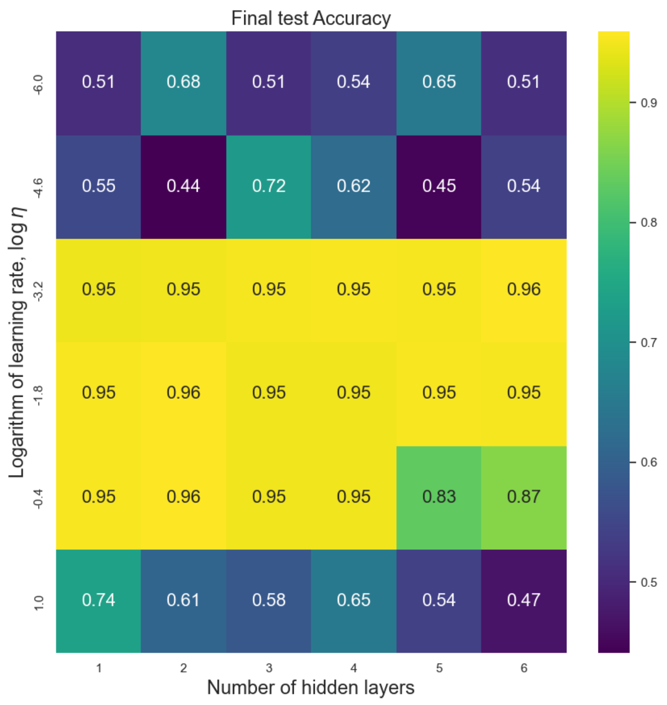


Figure 17: Grid with the final test accuracy with varying learning rates and number of hidden layers. Training done with five nodes per hidden layer and a regularization parameter of 10^{-5}. The Adam optimization scheme was used with 100 epochs and the activation function in the hidden layers was the tanh function. The training was repeated 5 times and the values were averaged to get the final values in the plot.

We see in Figure 17 that there is still a range of learning rate values where the model performs well fairly consistently. The only exception to this range is when the number of layers exceed 5 and the learning rate is in the higher part of the range, where the accuracy drops a significant amount, although not collapsing completely like some of the other values. There seems to be a pattern that for higher learning rates smaller models perform better, as we see that for a learning rate of 1 the model performs the best at a single hidden layer and worsens as more layers are added. For the lower learning rates it seems like the model fails to converge no matter how many layers is used as all of the accuracy's are fairly close to 0.5. A possible explanation for why the high learning rate models perform better with fewer layers is that the high learning rate causes the model to be unstable due to large model updates each step and that when the number of layers is too high there are too many parameters that have to stabilize at the same time for the model to improve. However when the number of layers is low the probability of getting an update where most of the parameters are updated in such a way that the model starts improving is likely higher due to fewer parameters being present.

Now that we have seen that a high number of layers can cause the training process to become unstable we can combine this with the other observations from our experiments to conclude on a model that we think will solve the classification problem the best. As we saw in Figure 16a the model starts overfitting when the number of nodes per layer grows, while the performance does not improve. Because of this we would choose as few nodes as possible as a simpler model is both computationally less expensive and is less likely to overfit the training data. As for the number of hidden layers we have seen that when the layer count grows to around five or higher the model starts exhibiting some unstable characteristics. We also saw that the performance did not improve much as the number of layers increased. For these reasons we will choose the simplest model having only one hidden layer.

3.3.6 Final Model Evaluation

To evaluate out final model, we will create an ROC curve where we plot the true positive rate against the false positive rate as we vary the boundary at which we consider the output to predict a positive outcome or a negative outcome. This is a post model processing step where if the model outputs a number above some value s we round up to one indicating a malignant case and if it is below s it will be rounded down to zero indicating a benign case.

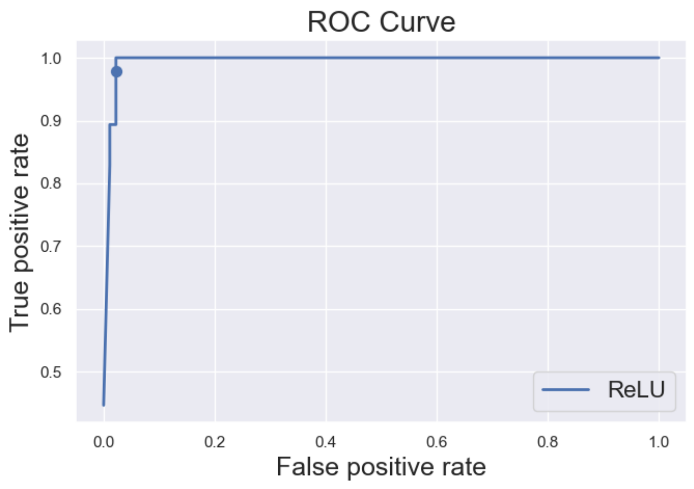


Figure 18: ROC curve for the final model using 3 layers with 5 nodes. 100 epochs and $\lambda = 0.01$.

We see in the Figure that the model has a very steep curve as we increase the boundary meaning that the model is able to recognize most of the cases without accidentally assigning a malignant prediction to a case that is actually benign. This signifies that the model performs very well on most of the dataset. If one were to use the model in the real world one should also do a cost analysis of how costly or dangerous having the model falsely predict a malignant case versus falsely predicting a benign one.

3.3.7 Logistic regression

After seeing in the previous section that the models with only a few hidden layers and only a few nodes per hidden layer perform just as well as the larger models we might suspect that a more simple kind of model might do the job of predicting our dataset just as well as a neural network. We therefore will attempt to train logistic regression model to solve the problem and compare the results with the ones in the previous section. The logistic regression model is equivalent to the neural network model we trained earlier, but with zero hidden layers.

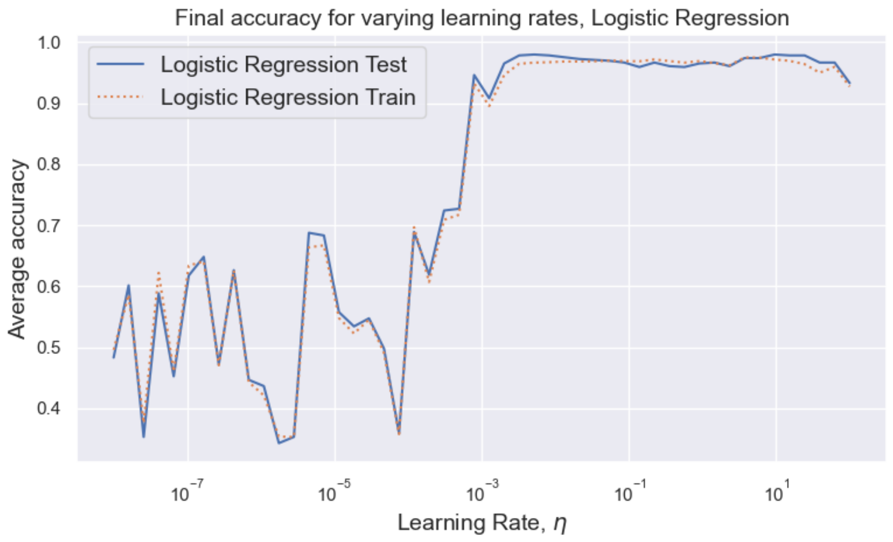


Figure 19: Accuracy as function of learning rate when training with 100 epochs. Results are average over 5 runs.

When varying the learning we see in Figure 19 that the logistic regression model exhibits a similar pattern to the neural network model as it is unstable at too low learning rates and reach a peak performance of roughly the same accuracy. One difference we see is that the logistic regression model is more stable at higher learning rates and even manages to converge to a good solution when the learning rate exceeds ten. For the neural network model we saw that the model started failing when the learning rate was higher than one. These findings indicate that the logistic regression model might be more stable than the neural network models. An explanation for this might be that the gradient of the neural network model has a tendency to diverge as it is passed through more layers, while the simpler logistic regression architecture has a simpler and more stable gradient.

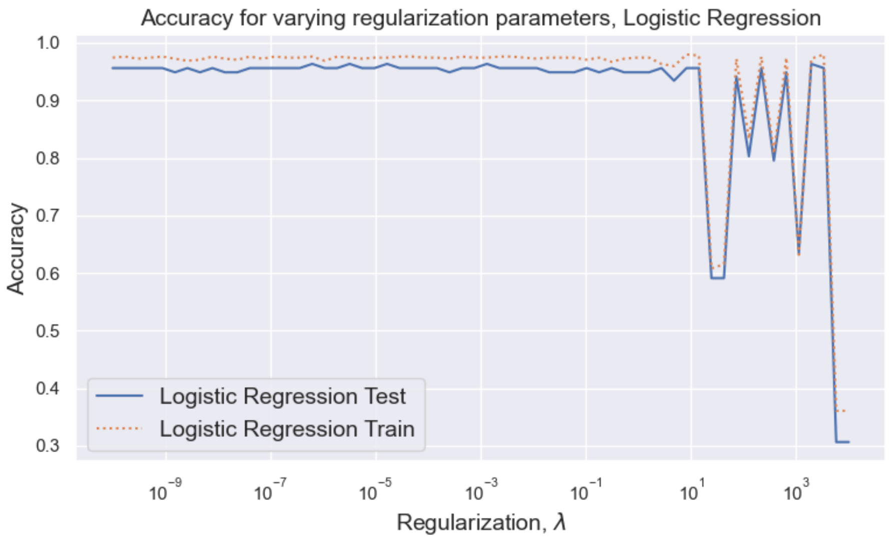


Figure 20: Accuracy as function of learning rate when training with 100 epochs.

We see that for the regularization experiment in Figure 20 the logistic regression model performs similarly well at most of the low levels of the regularization parameters. Here we also see that the model outperforms the neural network model in terms of stability at higher regularization parameter values. Here the model only starts exhibiting unstable behaviour when the regularization parameter exceeds ten while the respective value for the neural network model was 0.1.

We see that basic logistic regression is a strong competitor to our more complex neural network and in many ways is better in terms of training stability with different sets of hyper-parameters. For these reasons in addition to the model being smaller and thus computationally cheaper we would recommend using the logistic regression model for solving this classification task. It might be that since the problem only correlates a few independent variables with the probability of the tumour being malignant a neural network model is too complex for what is needed to solve the problem. For larger and more complicated relationships however the neural network model might gain an edge over the simpler logistic regression model.

4 Conclusion

In this paper we have explored methods for descent and some applications of these methods. We have found the adam descent method to give best results. A learning rate of $\lambda = 0.01$ gives good balance between computation time and closeness to the minimum.

We have seen that neural networks give better fits to the Franke function than the polynomial fits done both in this project and in Project 1 [2]. This model looks good when comparing the surface with that of the Franke function.

For the classification problem, we find the neural network models to be overly complex. It seems the logistic model is able to capture the complexity of the data, and to save computational power, we recommend this model.

Accuracy of above 95% shows that these are all good models.

4.1 Future direction

For future exploration of methods in this project, it would be interesting to do more exploration of the data looking into concepts like dimensionality reduction. To better compare parameters, it might be a good idea to implement early stopping once the gradients are sufficiently small.

Another potential project of interest would be exploration of a more complex dataset where we need the complexity of a neural network to find a good fit.

During our experimentation we also noticed a few flaws in our methodology that might have affected the results. One of these is that we ran the experiments with the same number of epochs for all the models. This might make some models which scale better with more epochs perform worse than they would if they were trained until they converged. Additionally we noticed that the exact train/test split used strongly impacted how well the models performance, probably due to some data being easier to predict in the test set than others. A further study could also try to take these into account or measure their effect on the results.

5 Appendix

5.1 Github

Our code as well as jupyter noteboooks used to generate the figures in this paper can be found at:

`https://github.com/Trond01/Project2_FYS-STK4155`

The neural network codes and descent methods have been written to be flexible enough to be used in future projects as well as serving their purpose for this project.

5.2 Descent methods for OLS and Ridge

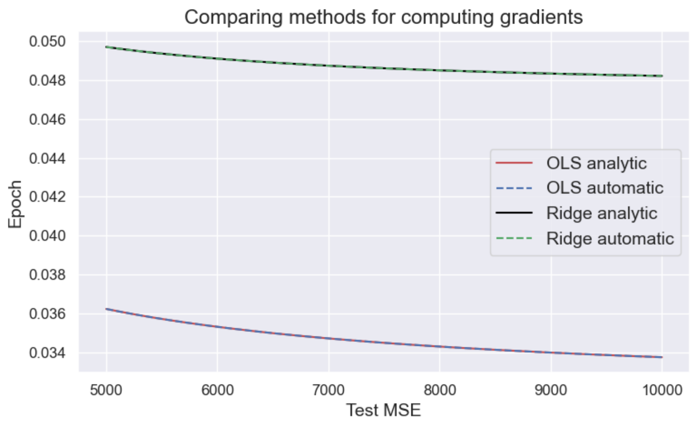


Figure 21: This plot shows the performance of analytic and automatic gradients for performing plain gradient descent. This plot serves as a verification of our implementation of automatic gradients using JAX.

5.3 Learning rate with tanh for increasing layers

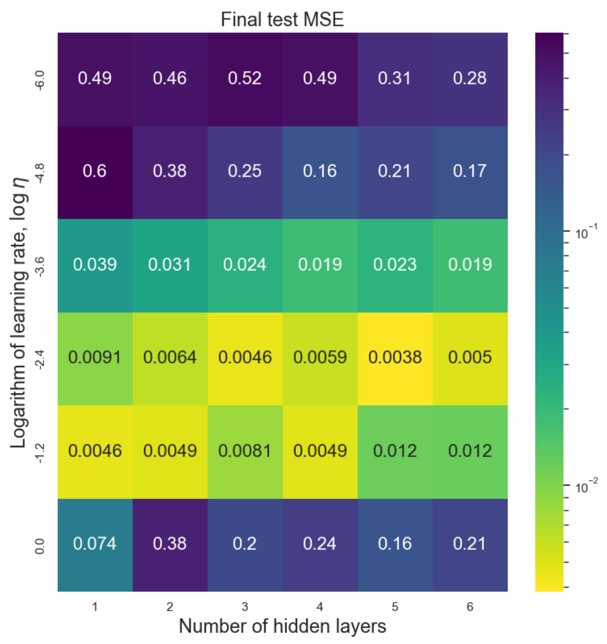


Figure 22: The final test MSE with varying learning rates and number of hidden layers. Training done with 5 nodes per hidden layer and a regularization parameter of 10^{-5}. The Adam optimization scheme was used with 300 epochs and the activation function for the hidden layers was the tanh function. The training was repeated 5 times and the values were averaged to get the final values in the grid.

5.4 Neural net confusion matrices

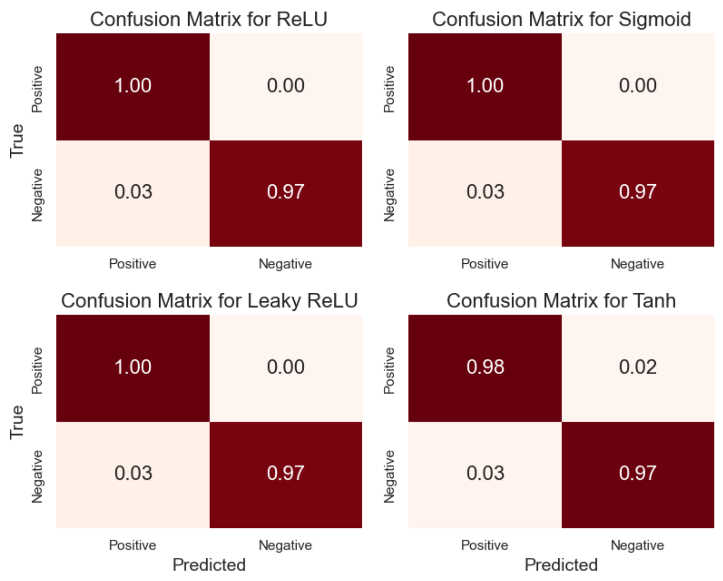


Figure 23: Confusion matrices using different activation functions in the hidden layers and an architecture of 3 hidden layers with 5 nodes.

5.5 Scikit-Learn

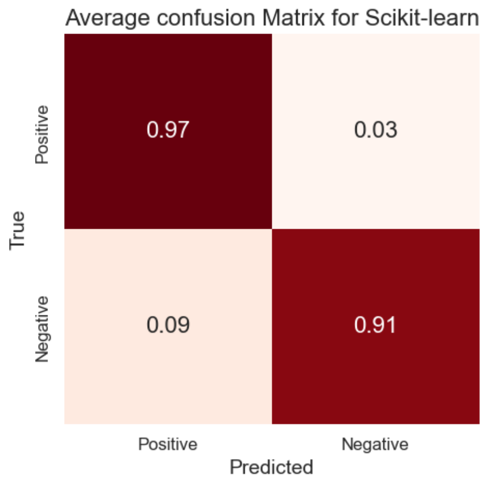


Figure 24: Average confusion matrix values for 10 runs of the logistic regression tools provided by Scikit-Learn. These values are consistent with our results, and serves as verification of our implementation. The accuracy for this model typically lies in the range 0.92-0.98.

References

[1] Bradbury, J. et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.3.13. 2018. URL: `http://github.com/google/jax`.

[2] Johansen, T. S., Pedersen, E. S., and Winsvold, J. "Project 1". Project report for FYS-STK4155. Oct. 2023. URL: `https://github.com/Trond01/Project1_FYS-STK4155`.

[3] Kingma, D. P. and Ba, J. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: `1412.6980 [cs.LG]`.

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

[5] Stanford University. *Section 4 (Week 4): Xavier Initialization and Regularization*. `https://cs230.stanford.edu/section/4/`. CS230 Deep Learning. 2022.

[6] Wolberg, W. *Breast Cancer Wisconsin (Original)*. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5HP4Z. 1992.