Project 2 - FYS-STK4155

Oda Langrekken

November 18, 2022

Abstract

We implement optimization methods, logistic regression and neural networks and use the methods for both regression and classification tasks. The performance of the neural network is investigated for different hyperparameters, and compared to the performance of the other models. We discover that for simple data sets the error using neural networks is of the same order as the error using simpler models such as linear regression. For classification, logistic regression outperforms the neural network, with an accuracy of 100% for logistic regression compared to 98% for neural networks. Based on this we conclude that neural networks are very powerful, but as they are hard to train we are better off using simpler models when our data is not that complicated.

Github repository: `https://github.com/OdaLangrekken/FYS-STK4155/tree/main/Project2`

1 Introduction

The development of Artificial Intelligence (AI) has intrigued people for centuries. With the recent evolution of deep learning, developing AI seems closer than ever before. Using deep learning scientists and engineers have developed algorithms for image recognition, speech to text translation and self-driving cars. Fundamental to deep learning are neural networks, which were initially developed to mimic how our brains process information. There is no doubt that neural networks are powerful, but do they always outperform simpler models?

In this project we develop a neural network from scratch. The neural network is used for both classification and regression tasks, and we investigate how the architecture and hyperparameters affect the performance of the network. Additionally we compare the performance of the neural network to more elementary methods such as logistic regression.

We start out by introducing the methods used, with a short recap of linear regression as a warm up. The next methods that are introduced are optimization methods, before we move on to logistic regression and neural networks. Then we briefly describe that data that is used for regression and classification. Next topic is results, where the results of optimization methods are first introduced and discussed. Afterwards we present and discuss the results for regression with neural networks, and then classification with neural networks and logistic regression. Finally some final thoughts around the project are offered in the conclusion.

2 Methods

2.1 A short recap of Linear Regression

In the first project [1] of this course we studied linear regression models . Given a dataset $\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_p)$, the goal of a regression model is to find a functional relationship between $\mathbf{X}$ and the output $\mathbf{y}$. In linear regression the function is assumed to be linear [2]

$$f(\mathbf{X}) = \beta_0 + \sum_{j=1}^{p} X_j \beta_j \tag{1}$$

where $\boldsymbol{\beta} = (\beta_0, \beta_1, \dots, \beta_p)$ are the coefficients of the model.

To evaluate our model we define a cost function that is a measure of how well our model explains the observed data $\mathbf{X}$. For linear regression a popular choice is the residual sum of squares:

$$RSS(\boldsymbol{\beta}) = \sum_{i=1}^{N} (y_1 - f(x_i))^2 = (\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T (\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) \tag{2}$$

Our model is fit to the data by choosing the coefficients $\boldsymbol{\beta}$ that minimize the residual sum of squares.

Provided that the $(\mathbf{X}^T\mathbf{X})$ is non-singular such that its inverse exists, there exists a unique, analytical solution for the coefficients

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{3}$$

Many machine learning problems follow the same recipe as linear regression. Given a dataset $\mathbf{X}$, a model $g(\boldsymbol{\beta})$ and a cost function $C(\mathbf{X}, g(\boldsymbol{\beta}))$ the model is fit to the observed data by choosing the parameters $\boldsymbol{\beta}$ that minimize the cost function [3]. For linear regression there exists an analytical solution. This is however not always the case. For such problems we can fit the model to data by using numerical optimization methods, which are the topic of the following subsection.

2.2 Optimization methods

Fitting a machine learning model to data boils down to optimization of the cost function, i.e. choosing the model parameters that minimize the cost for the observed data.

2.2.1 Gradient descent

The gradient of a function can be interpreted as the direction which gives the greatest increase in the function. For a cost function $C(\boldsymbol{\theta})$ the gradient is written $\nabla_\theta C(\boldsymbol{\theta})$ and is a vector with one component for each parameter θ_j. As the gradient gives the direction of the greatest increase, the idea of gradient descent is that we can most efficiently decrease the error by moving in the opposite direction of the greatest increase. For each iteration of gradient descent, each coefficient is updated according to [3]

$$\theta_{k+1} = \theta_k - \eta \nabla_\theta C(\boldsymbol{\theta}_k),\ k = 0, 1, \cdots \tag{4}$$

where η is the learning rate.

While gradient descent can perform very well when fitting the parameters of a model, it has some limitations. Firstly, gradient descent will always make the biggest change in the direction of the highest gradient. If one parameter has a sporadic high gradient, gradient descent will prioritize change in that parameter and ignore smaller but more consistent gradients in other directions. This slows down the convergence of gradient descent. A possible improvement is adding momentum to the algorithm. This is the topic of section 2.2.2.

Additionally the algorithm is deterministic and will always converge to the closest local minimum. Thus gradient descent is sensitive to the choice of initial parameter values. Stochastic gradient descent, the topic of section 2.2.3, improves the performance by adding randomness to the algorithm.

Finally gradient descent is sensitive to the choice of the learning rate parameter η. If η is too large the algorithm will become unstable and never converge, while too small a learning rate makes the algorithm converge very slowly.

2.2.2 Gradient descent with momentum

In momentum based gradient descent the parameters are updated as follows

$$\begin{aligned} \mathbf{v}_k &= \gamma \mathbf{v}_{k-1} + \eta \nabla_\theta C(\boldsymbol{\theta}_k) \\ \boldsymbol{\theta}_{k+1} &= \boldsymbol{\theta}_k - \mathbf{v}_k \end{aligned} \tag{5}$$

where γ is the momentum parameter.

As previously mentioned, regular gradient descent will always make the biggest change in the parameter that currently has the highest gradient. This can slow down convergence. By adding momentum we introduce some history to the algorithm. Previous gradients are used in the update,

taking into account persistent smaller gradients. The result is an algorithm that may converge faster than regular gradient descent.

2.2.3 Stochastic gradient descent

Stochastic gradient descent adds randomness to the algorithm. The data is split into k subsets, called minibatches. For each iteration (commonly called an epoch when using stochastic gradient descent) a minibatch is chosen on random, and the data in this minibatch is used for gradient descent. As only a subsection of the total data is used, stochastic gradient descent is not as computationally expensive as gradient descent. The stochastic nature of the algorithm also makes it less sensitive to initial conditions.

2.3 Logistic Regression

Logistic regression is used for classification tasks. In classification tasks the output is categorical, and the goal is to assign a class label to the data. The idea is similar to linear regression, but whereas the output of linear regression is a continuous value the output of logistic regression is the probability that the input belongs to a certain class. We will consider a problem with two possible output classes. In a two-class problem it is normal to code the output $\mathbf{y}$ such that a value $y_i = 0$ corresponds to the first class and a value $y_i = 1$ corresponds to the second class. As in linear regression we have input data $\mathbf{X}$ and coefficients $\boldsymbol{\beta}$. To get the probability that the input belongs to a certain category logistic regression uses the sigmoid function, which outputs values between 0 and 1:

$$\sigma(z) = \frac{1}{1 + \exp(-z)} \tag{6}$$

where $\mathbf{z} = \mathbf{X}\boldsymbol{\beta}$. The output class is the one with the highest probability.

The negative log-likelihood is used as the cost function [2]

$$C(\beta) = \sum_{i=1}^{N} \{y_i \beta^T x_i - \log(1 + \exp(\beta^T x_i)\} \tag{7}$$

The cost function is often referred to as the cross-entropy. The coefficients $\boldsymbol{\beta}$ that are the best fit to the data can be found by using an optimization method such as stochastic gradient descent to minimize the cost function.

For classification a useful error metric is the accuracy score

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} I(t_i = y_i)}{n} \tag{8}$$

where $I(t_i = y_i)$ is 1 when $t_i = y_i$ and 0 otherwise.

2.4 Neural Networks

As the name suggests, a neural network is built up of a network of neurons. Neural network were initially inspired by the neurons in our brains. A neuron is an excitable cell that will activate if it gets a certain output, and will transform and transmit the signal to other parts of the brain.

The behavior of neurons inspired a mathematical model for the artificial neuron, or node [3]

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b_i\right) \tag{9}$$

where y is the output of the neuron, $\mathbf{x}$ are the input data, and $\mathbf{w}$ and $\mathbf{b}$ are the weights and biases of the model respectively. The function f is known as the activation function. Its purpose is to transform the input in some way, i.e. by outputting 0 if the input to the function is below 0. Such an activation function is known as a Rectified Linear Unit, or ReLU, and is defined as:

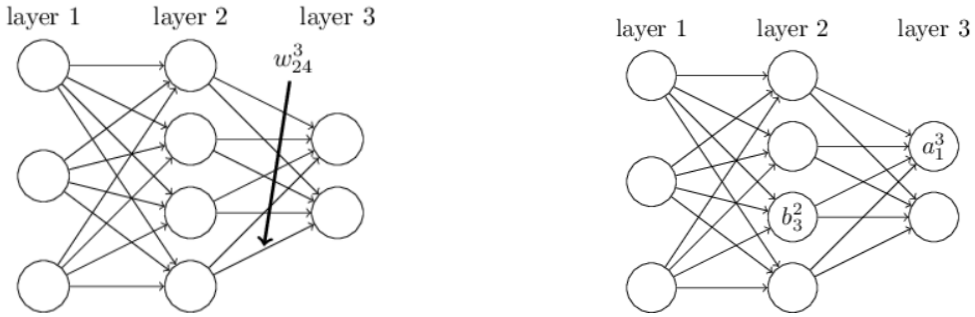
$$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{10}$$

In certain cases it can be useful to not cap input values less than 0 completely, but rather shrink them. In that case the activation function is called the leaky ReLU, and is defined as

$$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \alpha x, & \text{otherwise} \end{cases} \tag{11}$$

where α is the slope and is typically a number of order 0.01.
Another popular activation function is the sigmoid function, defined in section 2.3.

In deep learning one often refers to the neurons as nodes. Figure 2.1 shows an example of what a neural network can look like. Each circle is a node. The first layer is the input data.



(a) A subfigure (b) A subfigure

Figure 2.1: Notation and example layout of a neural network, from [4]

The input data is propagated through the network by calculating the activation for each node based on the input data and weights of the previous layer:

$$a_j^l = f\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right) \tag{12}$$

2.4.1 Backward propagation

We will not go into the gritty details of backpropagation here. A good resource is [4]. In the final layer the activation function should transform the output to For classification a good choice is the sigmoid function. For regression one can use a linear activation function.

3 Data

We will perform both regression and classification. For all regression tasks a simple polynomial

$$y = 2 - 5x + x^2 + \epsilon \tag{13}$$

is used, where ϵ is a random error drawn from a gaussian distribution with standard deviation 0.1.

For classification we use the Wisconsin Breast Cancer data set [5], which contains information about whether or not a patient has breast cancer. The output is 0 for False and 1 for True. There are 30 predictors in the data set, and 569 observations.

For both tasks the data in split into a train (60%), a validation (20%) and a test set /20%). As optimization methods such as gradient descent are sensitive to the scale of the input features, all input is scaled with sklearn's `StandardScaler`.

4 Results and Discussion

4.1 Optimization methods

First we implemented gradient descent and tested it on the simple polynomial defined in equation 13. As discussed in section 2.2.3 gradient descent is sensitive to the choice of learning rate, and a badly chosen learning rate can slow down convergence. In figure 4.1 we see the mean squared error as a function of the number of iterations of gradient descent. The error is calculated using the train set. Three different learning rates are used for the algorithm. As we can see in the figure, a learning rate of 0.01 seems too small for this task. The mean squared error is steadily decreasing, but it has not converged even after 1000 iterations. A better choice seems to be either $\alpha = 0.1$ or $\alpha = 0.3$. In both cases the error converges at a mean squared error of about 0.01. The larger learning rate does however converge faster, so $\alpha = 0.3$ appears to be the best choice. Larger learning rates were also tested, and although a learning rate of e.g. 0.5 converged faster, it was unstable for low number of iterations.

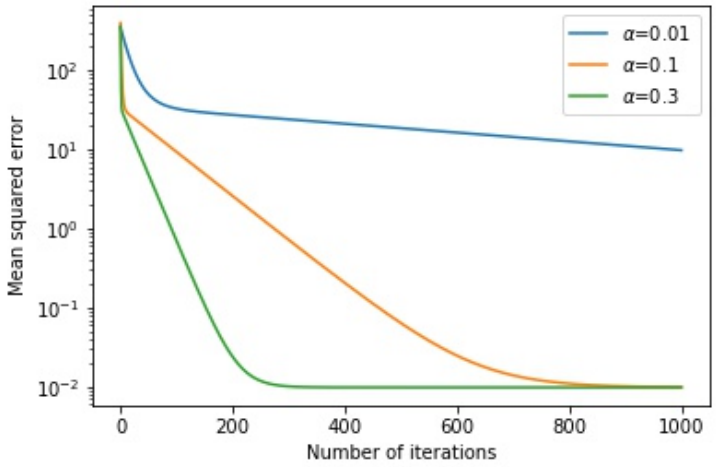


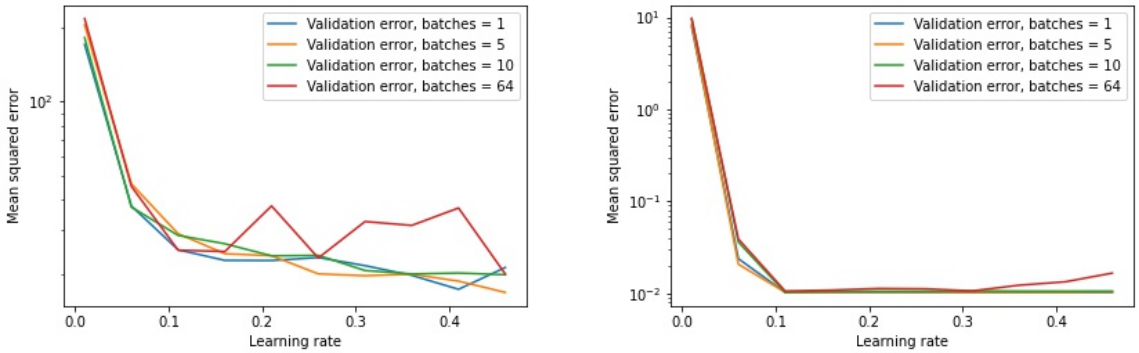
Figure 4.1: Mean squared error as a function of the number of iterations of gradient descent for different values of the learning rate, α

Adding momentum to the gradient descent can speed up convergence. In table 4.1 we can see the number of iterations of gradient descent needed to achieve a train error less than 0.01 for different momentum parameters.

μ	Iterations
0	313
0.1	283
0.3	216
0.7	77

Table 4.1: Table showing the number of iterations of gradient descent needed to achieve a train error of less than 0.01 for different momentum parameters, μ. A learning rate of $\alpha = 0.3$ is used.

Gradient descent can be computationally expensive as all input data is used for each iteration. An alternative is the stochastic gradient descent (SGD), where the data in split into minibatches. In SGD we no longer talk about iterations, but rather epochs. For each epoch the data in shuffled, split into batches and gradient descent is used on a randomly chosen batch.



(a) 10 epochs

(b) 1000 epochs

Figure 4.2: Validation error as a function of the learning rate when using stochastic gradient descent. The error is plotted for different numbers of minibatches.

In figure 4.3 we can see the mean squared error for different number of batches. The errors are shown for SGD using 10 epochs and 1000 epochs. When using SGD the number of minibatches is important. If the number is too high, the algorithm only sees a very small ratio of the data on each epoch. We see in the figure that when using 10 epochs the results are unstable when we have 64 minibatches. This makes sense, as the number of minibatches is higher than the number of epochs so that not all the data is used for training. For smaller number of minibatches the error is not significantly worse than when using all data. For more complictaed data sets with more noise, we would however expect that the performance could be worse when we have a large number of minibatches and few epochs.
When we increase the number of epochs to 1000 we see that the results are very stable for all choices of minibatches, except that the one with 64 minibatches seems to overfit for large learning rates. The results are similar to the ones achieved with regular gradient descent with 1000 iterations, and the smallest error is about 0.01. But as 1000 epochs of SGD will be faster than 1000

iterations of GD, we see that SGD can very useful for computationally expensive training.

Lastly we used SGD with L_2 regularization. The results are shown in figure 4.3:

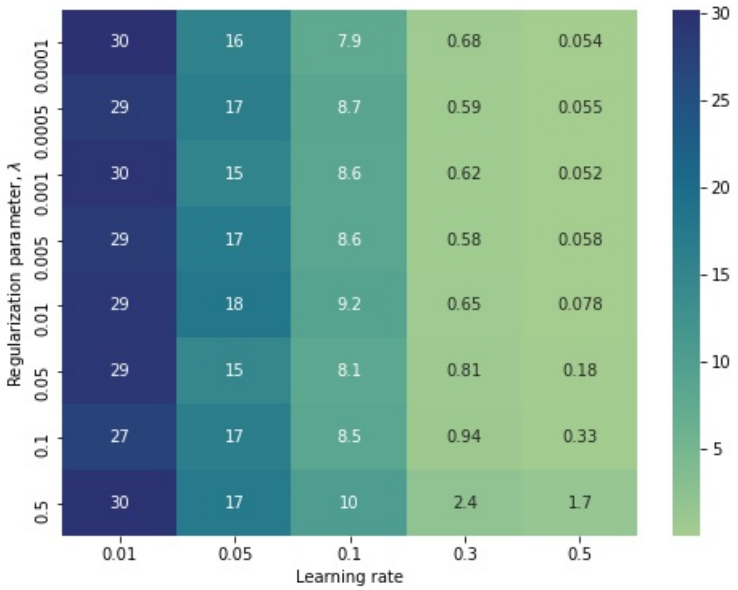


Figure 4.3: Validation error as a function of regularization parameter and the learning rate for SGD with L_2 regularization.

Stochastic gradient descent is used for training of the neural network. With our gradient descent methods implemented, tested and understood, we are ready to move on to our neural network.

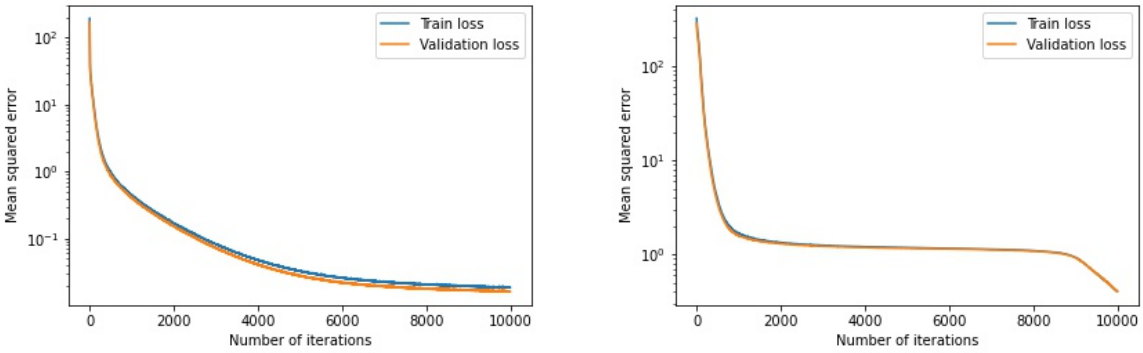
4.2 Regression with Neural Network

We will now use our neural network for regression. As this is a regression task we use the mean squared error as the cost function, as a linear activation as the final activation function. The linear activation function ensures that the output lives in the right space, that is the space of all real numbers. The weights are initialized by drawing from a gaussian distributiion, and the biases are set to a random number between 0 and 1.

A neural network is extremely flexible, and there is no really no limit to what networks you can build. First we investigate how the number of hidden layers and the number of nodes in each layer affects the result. Figure 4.4 shows the train and validation error as a function of the number of epochs (called iterations in the plots). The plot to the left has one hidden layer with 100 nodes, while the plot on the left has two hidden layers with eight nodes in each layers. For both networks the sigmoid function is used as activation in the hidden layers.

What is most apparent is that the network does not overfit, as the validation and train errors are almost indistinguishable for both networks. We also see that the network with one hidden layer has a smaller error. The network with two layers does have a decreasing error for the last

iterations though, so it is possible that this network will outperform the smaller network if we increase the number of iterations. Finally none of the networks have a smaller error than the one we achieved with regular gradient descent. This may be an indication that we have not chosen the hyperparameters and architecture well enough for the network to truly live up to its potential.



(a) 1 hidden layer, 100 nodes (b) 2 hidden layers, 8 nodes in each layer

Figure 4.4: Mean squarer error as a function of the number of epochs (iterations) for a neural network with a sigmoid activation function. The learning rate is 0.0001, 5 minibatches are used for stochastic gradient descent and the regularization parameter is 0.

The choice of activation function is also essential when training neural networks. Figure 4.5 shows the error for different activation functions. It is clear from the figure that the sigmoid is not the best choice of activation function for this task,

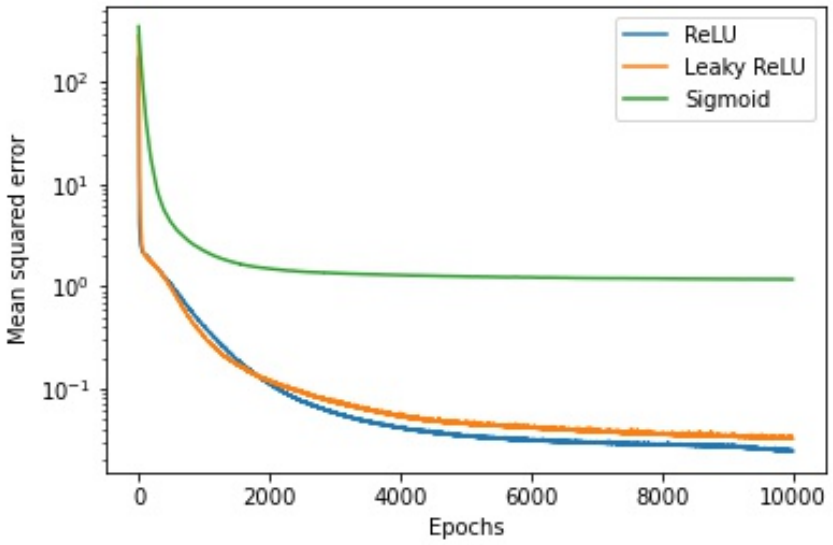


Figure 4.5: Validation error for different activation functions. The learning rate is 0.0001, 5 minibatches are used for stochastic gradient descent and the regularization parameter is 0.

Finally we want to find the learning rate and the regularization parameter that gives the highest performing network. In figure 4.6 there is a heatmap showing error for different regularization

parameters and learning rates.

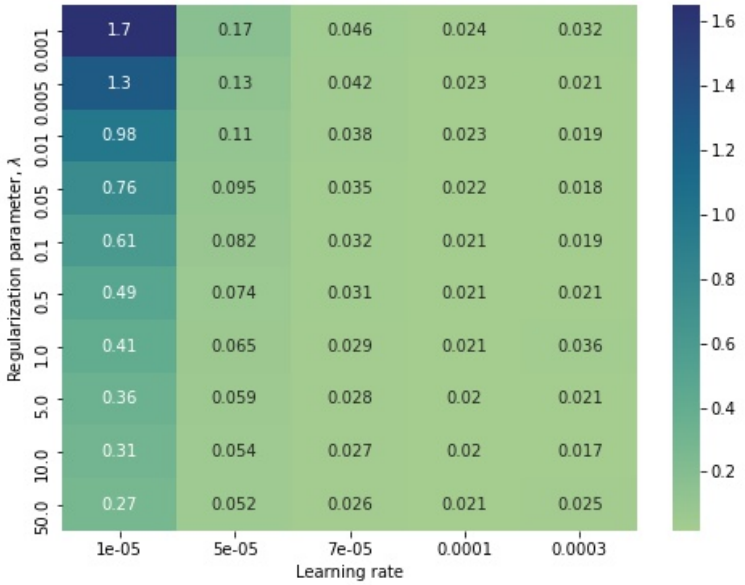


Figure 4.6: Validation error as a function of regularization parameter and the learning rate for a neural network with one hidden layer with eight nodes.

The most consistently well-performing learning rate is 0.0001. The best regularization parameter is 10. As we have seen the best activation function appears to be the ReLU. Using this activation function and these hyperparameters, we get a test error of 0.033. This error is higher than the one we got using regular gradient descent.

4.3 Classification using Neural Networks and Logistic Regression

We now move on to classification, with the Wisconsin Breast Cancer data. With logistic regression we get an accuracy of 1.0 for the train set, and 0.99 for the test set. This is without any parameter tuning, and shows the power of logistic regression when the classification task is not that complicated. Figure 4.7 shows the accuracy when using a neural network. The highest accuracy is about 0.99. The sigmoid function was used as activation when calculating the errors. As for regression, the ReLU performs very well, but for classification the sigmoid function is a very close second.

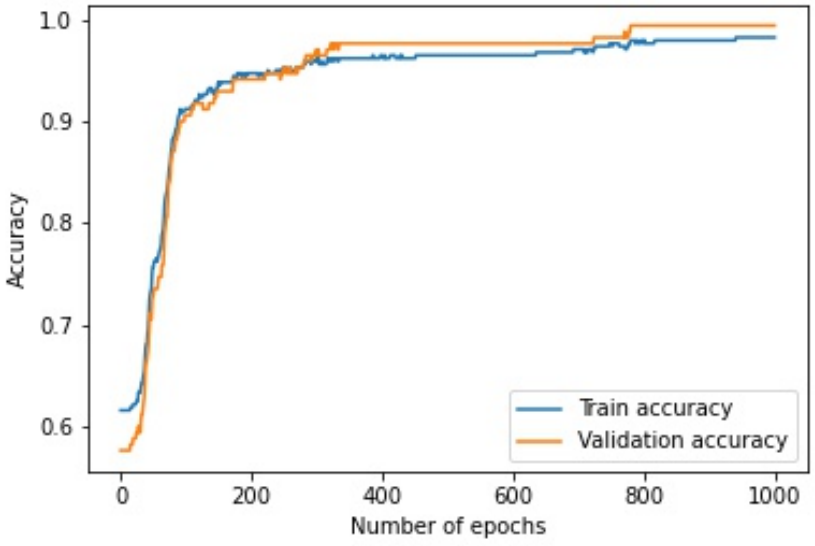


Figure 4.7: Accuracy as a function of the number of epochs for a classification neural network with one hidden layer with 10 nodes. The learning rate is 0.01.

5 Conclusion

We have implemented stochastic gradient descent, logistic regression and neural networks. The methods were used for both a regression and a classification task. In both cases we investigated how the performance depended on the hyperparameters of the model, and we compared the performance of neural networks to that of simpler models.

We discovered that linear regression and logistic regression perform well on the data sets we have used. For the classification task we achieved an accuracy of 1.0 on the train set, and 0.99 for the test set using the simple and straight-forward logistic regression. Training a neural network can feel overwhelming as there are so many hyperparameters to tune, architectures to choose, ways to initialize weights etc. For simple data sets we conclude that it might be best to try simpler models first, and then progress to neural networks if simple models do not perform satisfactory.

Neural networks will only truly shine when used on complicated data sets. The data sets I chose to use for this project were not so complicated and good results were achieved with simpler methods. Considering that deep neural networks are known to outperform simple algorithms when the data is complicated, it would be very interesting to compare a neural network and logistic regression on a more complicated data set.

With the small data sets and shallow networks, training was almost instantaneous even on my old laptop. With bigger data sets and/or deeper networks training of neural networks can grow very computationally expensive, which motivates more considerate use of momentum and minibatches for gradient descent to speed up convergence. An interesting extension to this project would be to use the methods implemented to make predictions on a complicated and large data set. For such a data set we would surely see the advantages of the methods that make gradient

descent less computationally expensive, and we would probably also get to really appreciate the power of neural networks.

References

[1] O. Langrekken, "Project 1." https://github.com/OdaLangrekken/FYS-STK4155/blob/main/Project1/report/FYS_STK4155___Project_1.pdf, 2022.

[2] J. F. Trevor Hastie, Robert Tibshirani, *The Elements of Statistical Learning*. Springer New York, NY, 2 ed., 2009.

[3] M. Hjorth-Jensen, "Applied data analysis and machine learning." https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html, 2021.

[4] M. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015.

[5] D. Dua and C. Graff, "UCI Machine Learning Repository, Wisconsin Breast Cancer," 2017. https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(diagnostic).