Project 2

Emil Engelstad Moghaddam

November 14, 2020

Abstract

In this study, we compared effectiveness and ability of a self developed feed forward neural network (FFNN), in problems of classification and regression, against logistic and linear regression. We also compared FFNN without and with an implementation using Keras - Tensorflow. The Wisconsin breast cancer data set, an 8x8 handwritten digits and data points from Franke function were in turn used for binomial, multinomial and regression analysis, respectively. Comparing ridge regression, our own FFNN and an implementation using Keras, we were able to predict the franke function data points with a R - squared values of .940, .990 and .997, respectively. We demonstrated that adding momentum was beneficial to stochastic gradient descent applied to both linear regression and FFNN.

1 Introduction

The aim of this project is to demystify the basic structure of neural networks, optimization of parameters and some simple methods for calibrating weights and biases. We apply the network to both discrete and continuous data. The results are compared to the standard ways of doing classification and regression which are logistic regression and linear regression.

The architecture of neural networks are inspired by how neurons in the human brain are assumed to work, a result of millions of years of evolution and our best example of intelligens to date. The architecture of neural networks make the method an effective procedure to find hidden patterns. Generally neural networks don't need to rely on as many assumptions as linear or logistic regression. In an age with data abundance and ever increasing computational power, neural networks are a great tool for humans.

Gradient descent based methods for optimizing neural networks is an active field of research. Before applying the method to neural networks we first use gradient descent on simpler problems such as linear regression, ridge regression and logistic regression. The goal here is to gain a deeper understanding of the method, which is very important for the computational efficiency of neural networks. For ridge and lasso we have an analytical solution for the weights, which lets us compare our results with ideal values.

We also study the variables connected with the architecture of the neural network, such as number of layers and number of nodes, and type of activation functions. Here there are no established solutions, but finding a great combination of variables can massively improve the quality of the predictions produced by the network. Then we study the effect of regularization on the network. The main properties of regularization is letting the network extract a pattern without reacting to the noise and improving the predictive quality of the network on new data.

1.1 Structure

The theoretical part of the report starts with the architecture of neural networks. We look at the cost functions characteristic of linear and logistic regression, and of neural networks. Then we look at the gradients, which are the derivatives of the cost function. After deriving the gradients we look at the applied usage of gradients in the stochastic gradient descent method. We finish of the theoretical part by analyzing regularization in all the methods we employ.
In the results and discussion section we first show results for linear regression, before moving to neural networks, first for regression, subsequently for classification.

2 Theoretical model and technicalities

2.1 Arcitecture of a neural network

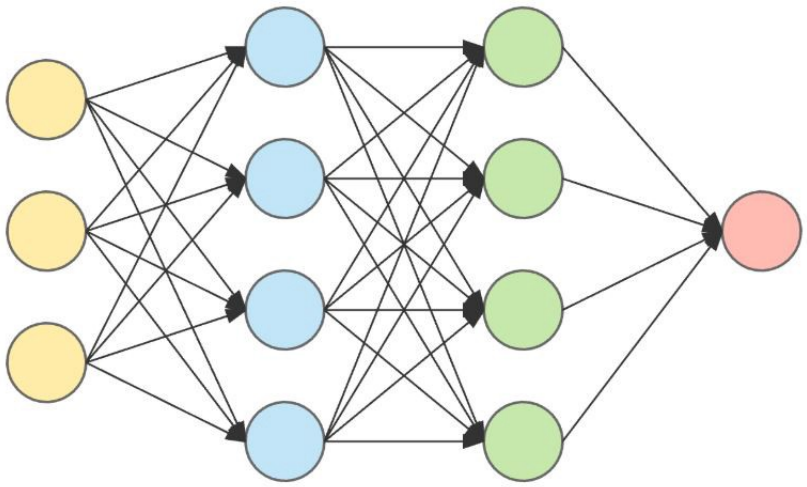

Figure 1: Simple scretch of a neural network

A neural network consists of a set number of nodes organised in several layers. The layers are separated into three types. The input layer which is marked in yellow in figure 1; the hidden layers, marked in blue and green; and the output layer, marked with red. The number of nodes in the first and final layer is often decided by the type of data we are studying and the type of regression we are doing. The number of hidden layers and the number of nodes that are placed in each of the hidden layers, are variables we can change to improve the quality of our network. In the input layer there is one node for each variables used for prediction. That means 2 nodes, one for x and one for y, for the franke function, 30 nodes one for each variable of the Wisconsin dataset, and 64 nodes one for each pixel of the 8x8 handwritten digits.

When performing regression, which means that the data is continuous, we have one output node that contains the final predicted value of the model. On the contrast, when working with k discrete classes, like the 10 digits dataset, we use k output nodes each containing the probability calculated by the network that that data belongs to that class [1] . When the data is binary, k equals 2, we also have the choice of using just 1 output node, with a chosen threshold t that determines which of the two classes the model predicts.

We designate the number of nodes in layer $l-1$ as n, and the number of nodes as in the preceding layer l as m. Each of the n nodes in layer $l-1$ will have m unique weights, one for each of the nodes in the next layer. Between

two layers there will be a [N,M] matrix of weights. Each node of layer l receives input from all the nodes of the previous layer. In addition each node of layer l receives a bias value. Every layer has its own function H(x), which transforms the weighted sum of the values of the previous layer. The value of node p of layer l can be written as $z_{l,p} = H(\sum w_{i,p} \times z_{l-1,i} + bias_p)$. In the input layer it is often usual to keep the function, H(x) just equal to x.

With the mathematics of how the values of one layer are transformed, weighted and sent to the preceding layer, we can write an algorithm that calculates how starting values at the input layer are changed at each of the layers of the network until we have the models prediction at the final layer. We keep the weighted sums in one array, and the transformed sums in another array.

$a \leftarrow$ [input]
$z \leftarrow$ [ ]
for hidden and output layers **do**
  z.append($a[-1] \times weights + bias$)
  a.append($h(z[-1])$)
end for

2.2 Cost function

All the methods used in this project are based on minimizing a chosen function of numerical distance between the actual data points and the predictions made by the model. The function we are trying to minimize is called either the "cost function" or the "loss function". Different choices for the cost function, changes the quality of the network and how we find an optimal solution.

For linear regression, the cost function is the squared distances between the model, $X\beta$ and the data points. For the Feed Forward Neural Network we use the same cost function, but instead use the squared distance between the data points and the values at the final node. Another possible choice for cost function is the absolute value of the distance between data points and model predictions. Equation 1 shows the cost function for linear regression, while equation 2 shows the cost function for the neural network.

$$Costfunction_{linear} = |Y - X\beta|^2 = [Y - X\beta]^T[Y - X\beta] = \sum (y_i - \sum x_{ij}\beta_j)^2 \tag{1}$$

$$Costfunction_{FFNN} = |Y - \tilde{y}|^2 = [Y - \tilde{y}]^T[Y - \tilde{y}] = \sum (y_i - \tilde{y}_i)^2 \tag{2}$$

When working with classification problems we use mathematical functions, like sigmoid and softmax, to map our predictions into values ranging from zero to one. The data points are always zero or one. One signalizes that the case is true, eg person has cancer or data belongs to class j, while zero signalizes that the case is false, person does not have cancer and the data does not belong to

class j. For logistic regression, the cost function is minus log of the numerical distance between the data and the model. When working with classification in a neural networks, we employ use the same negative logarithm, but only for the class that is true.

Equations 3 and 4 show two equivalent ways of writing the cost function for logistic regression, and equation 5 shows the sigmoid function.

$$Costfunction_{logistic} = \begin{cases} -log(h(x)) & \text{for} \quad y = 1 \\ -log(1 - h(x)) & \text{for} \quad y = 0 \end{cases} \tag{3}$$

$$= -\sum y_i \times log(h(x_i)) + (1 - y_i) \times log(1 - h(x_i)) \tag{4}$$

$$h(x) = \frac{1}{1 + e^{-\beta x}} = \frac{e^{\beta x}}{1 + e^{\beta x}} \tag{5}$$

2.3 Gradient theory

The gradient is a vector that at each point, points in the direction of maximum growth. Going in the opposite direction of this value would result in reaching a local minimum. Ideally we would like to reach the absolute minimum of the cost function. However we can't always know whether we have reached a local minimum or a global minimum. Calculating the gradient of the cost function with respect to the weights of the model, shows in what direction and ratio the weights of the models ought to be changed to improve the model.

Mathematically if we have a function $f(x_1, x_2, x_3, ...x_n)$ the gradient becomes:
$\Delta f = (\frac{\delta f}{\delta x_1}, ..., \frac{\delta f}{\delta x_n})$

For linear regression we use the chain rule:
$\frac{\delta Cost}{\delta \beta_j} = 2 \times \sum(y_i - \sum x_{ij}\beta_j) \times \frac{\delta \sum x_{ij}\beta_j}{\delta \beta_j} = 2 \times \sum((y_i - \sum x_{ij}\beta_j) \times xij)$

For logistic regression we rewrite the two compoents : [2]

$log(h(x_i)) = -log(1 + e^{-\beta x_i})$

$log(1 - h(x_i)) = log(\frac{e^{-\beta x_i}}{1+e^{-\beta x_i}}) = -\beta x_i - log(1 + e^{-\beta x_i})$

This lets us rewrite the costfuntion as

$cost = \sum y_i \times \beta x_i - \beta x_i - log(1 + e^{-\beta x_i})$

$cost = \sum y_i \times \beta x_i - log(e^{\beta x_i} \times (1 + e^{-\beta x_i}))$

$cost = \sum y_i \beta x_i - log(1 + e^{\beta x_i})$

Then we take the derivative

$\Delta cost/\delta\beta_j = \sum y_i x_{ij} - \frac{1}{1+e^{\beta x_i}} \times x_{ij} e^{\beta x_i})$

$\Delta cost/\delta\beta_j = \sum x_{ij} \times (y_i - \frac{e^{\beta x_i}}{1+e^{\beta x_i}})$

$\Delta cost/\delta\beta_j = \sum x_{ij} \times (y_i - h(x_i))$

The results looks very similar to that of linear regression.

For neural networks we want to derive the gradient of the cost function for both the biases and for the weights. Between an arbitrary layer l and the previous layer l-1, there are $m \times n$ weights. For layer l we can make m derivatives, one for how a change of value in each of the nodes changes the cost function. $\big(\frac{\delta Cost}{\delta node_1}, ..., \frac{\delta cost}{\delta node_m}\big)$.
For layer l-1 we can make m derivatives for how the biases changes the values of the next layer and we can make n derivaties for how the weights of layer l-1 change the values of the next layer. Note l stands for layer l not loss.
$\big(\frac{\delta l_1}{\delta bias_1}, ..., \frac{\delta l_m}{\delta bias_m}\big)$, $\big(\frac{\delta l}{\delta w_1}, ..., \frac{\delta l}{\delta w_n}\big)$

The derivative of how a change in $bias_j$ changes the values of the next layer is simply one.
The derivative of values of the next layer with respect to w_j is the value at node number j - $\frac{\delta l}{\delta w_j} = h(x_j)$

With n derivaties for layer l-1 and m derivatives for layer l we can make a $n \times m$ matrix of derivatives for the weights between the two layers. Since the derivative of the beta values is 1 the derivative with respect to beta is just the derivatives of the functions at the next layer

We now have derived how the weights of layer l-1 change the values of layer l. We now want to figure out how a change in the arbitrary layer l changes the cost function. The first step is seeing that a change in layer l changes the values of the next layer by the derivative of characteristic function of layer l times the weights of layer l. $h'(x) \times W_l$. We can start at the cost function and use the chain rule to work us backwards until we get to layer l. [3]

$$\frac{\Delta Cost}{\delta \vec{x_l}} = \frac{\Delta Cost}{\delta h(x)_{-1}} \times h'(x)_{-1} \times \prod W_i \times h'(x)_i \tag{6}$$

Note that the algorithm starts at the last layer and goes back to include layer L. The function $h(x)_{-1}$ is our prediction $\tilde{y}$, minus 1 indicating the final layer. We calclutate the derivatives for all layers at the same time, saving the values for each step. For each layer we we make a matrix by matrix multiplying the

derivatives for weights of the preceding layer, with the derivatives calculated in the formula above.

Some extra notes on the first term of the gradient of neural networks

When the network is doing regression with squared error as the loss function, the derivative of the first term is simply $2 \times (Y - h(x))$. However, when we are doing classification with k classes, the gradient can become very complex, unless we use the softmax as the final function of the layer. We now look at the product of the 2 last terms of the equation. While only the value at the node where the data is equal to one is used in the cost function, a change of value in one of the other output nodes will change the value in the correct node.

Softmax: $\tilde{y_i} = \frac{e_i^x}{\sum^k e_j^x}$

Equation 7 shows how a change in the value of the same node x_i changes our prediction $\tilde{y} = h(x_i)$, where h is the softmax. Equation 8 shows how a change in the values of one of the other nodes in the output layer changes our prediction in the correct class.

$$\frac{\delta y_i}{\delta x_i} = \frac{e_i^x \sum e^x - e_i^x e_i^x}{(\sum e^x)^2} = \tilde{y_i} - \tilde{y_i}^2 \tag{7}$$

$$\frac{\delta y_i}{\delta x_j} = \frac{0 \times \sum e^x - e_i^x e_j^x}{(\sum e^x)^2} = \tilde{y_i}\tilde{y_j} \tag{8}$$

Here we have the costfunction $L = -\sum y_i \times log(\tilde{y_i})$. Equations 9 and 10 show the derivative of the costfunction with respect to the same node, and other nodes. It is a simple use of the chain rule. We insert equation (7) and (8).

$$\frac{\delta L}{\delta x_i} = -\sum y_i \frac{1}{\tilde{y_i}} \times \frac{\delta y_i}{\delta x_i} = -y_i(1 - \tilde{y_i}) \tag{9}$$

$$\frac{\delta L}{\delta x_j} = -\sum y_i \frac{1}{\tilde{y_i}} \times \frac{\delta y_i}{\delta x_j} = -\sum y_i \tilde{y_j} \tag{10}$$

When i is the correct class y_i is equal to 1. When we add equation 7 and 8 we get $1 - \tilde{y_i} + \sum \tilde{y_j}$, very simply it is the data minus the model, or the error of each node.

2.4 Stochastic gradient descent

The method of stochastic gradient descent is based upon taking steps in the direction of the gradient to minimize the loss function.

Nonlinear networks usually have multiple local minima of differing depths. [3](ch. 1) To use the method of gradient descent requires that we initialize the network using some initial weights and biases. Weight initialization is also required for logistic and linear regression. How we choose to initialize the weights influence how fast we reach a minima, which minima we reach and therefore also the depth of the minima. For logistic regression the problem is convex, so every weight initialization will eventually reach the global minimum.[4]

Initial weight initialization will also influence the learning speed of the network. When using the sigmoid function it is recommended that the weights are initialized in such a way that the values are in the linear region of the sigmoid. [3]. In my implementation, when initializing the weights and bias, I calculated the values at the next layer, before changing the weights and bias so that the values would be in the linear region. The sigmoid is linear in an area around 0.5. Another way of initializing weights is by using a normal distribution with a specific mu and sigma values.

When doing stochastic gradient descent methods we usually don't use the whole dataset to calculate the gradient. Instead we split the data into several batches, and use the batches to calculate the gradient. This makes the calculation of the gradient more efficient, as the number of calculations scale exponentially with the number of data points. There are two methods for splitting data into batches. The first method is performed by just picking random samples from the dataset, each time a new gradient is calculated. The second method is by splitting the dataset into equally sized samples. This ensures that the whole dataset is represented when an entire iteration(epoch) of each batch has been used in calculations of the gradient. I have implemented both methods in the project. The first method has been implemented in the neural network, the second batching method was implemented in gradient descent method on linear regression. Increasing the number of batches makes calculations faster, however the gradients will be less stable, eg the variance of the gradients become larger.
Mathematically, we say that the gradient $g(\theta^t)$ is replaced with $Z(\theta^t, \xi^t) = \frac{1}{\xi}\sum_{\xi} \Delta cost(\theta)$, where ξ is a random sample. The random component is the reason we use the word stochastic, as the gradient based on a sample of the data is a probabilistic measure of the true gradient.

The efficiency of the stochastic gradient descent is dependent on the learning rate used. If the learning rate is too high, the gradient will not converge against a desired minimum. Instead the costfunction will jump back and forth or might even diverge. If the learning rate is too low, optimization will be slow. A possibility for improving the stochastic gradient descent method is by adding "momentum" to the gradient. This is done by saving a percentage of the previous gradients and adding the previous grade to the next change in weights and biases. The percentage is represented by an alpha value between zero and one. The intuition behind momentum is that adding a percentage of previous gra-

dients will prevent the gradient from being stuck in minor gropes and increase stability in the gradient.

2.5 Regularization

In this project we have utilized L2 norm regularization, which is also known as weight decay, ridge regularization or Tikhonov regularization. This means that we are trying to optimize the cost function while trying to keep the weights/betas as small as possible. We add a $\lambda \times \beta^2$ to the cost function. The derivative of the regularization, which we add to the gradient, is $2 \times \lambda \times \beta$. In project 1 i show how using l2 norm regularization is equivalent with optimizing the cost function while only using weights form a constraint set.

Goodfellow et al.,(2016) defines regularization as any modication we make to a learning algorithm that is intended to reduce its generalization error but not it's training error. The goal of regularization is to prevent the model from overfitting. We want the network to find the pattern of the data, without reacting to the noise. If we were to make several models based on differing subset of the data points, an indication that we are overfitting would be that the mean of these models would predict test data with high accuracy, but the individual models would not predict the test data accurately. Another indication of overfitting is large discrepancies between the test and training errors. When using a gradient approach, instead of an analytic solution for the weights, overfitting is also a product of the number of gradient steps used in optimization of the model

3 Results and discussion

3.1 Gradient decent for linear regression

Figure 2 and figure 3 show the effects of learning rate and batch size when using the gradient descent method, for both linear regression and ridge regression, respectively. The graphs are calculated after 500 epochs and show disparity in how regression with and without regularization react to batch size. For linear regression without regularization, the results of the gradient descent method are stable and might even improve with increasing the number of batches. Increasing number of batches is in theory more computationally efficient. In contrast, when we include a regularization parameter, increasing the number of batches radically decreases the performance of the model produced by the gradient descent method. Note that the scales on the plots are different for ridge and linear regression, and that gradient descent without regularization achieves better results than with regularization for regression. There is an outlier in the data, in the bottom left corner of linear regression, which may be due to the magnitude of the step size being too large. Figures 2 and 3 indicate that it might be faster to initially use stochastic gradient descent to find the ideal weights for the model

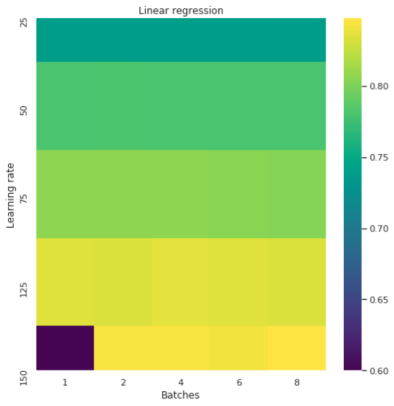

Figure 2: An illustration of the effect batch size and learning rate on R - squared for normal linear regression - z scale indicates R - squared

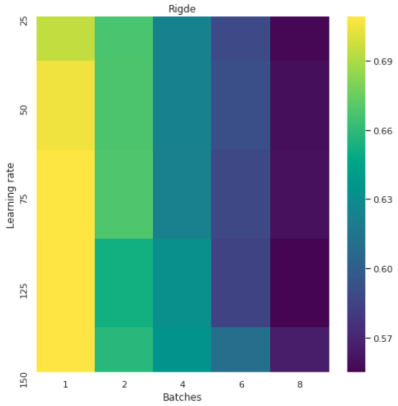

Figure 3: An illustration of the effect batch size and learning rate on R - squared for rigde regression - z scale indicates R - squared

without regularization.

Figure 4: An illustration of how momentum decreases number of itterations to reach the optimal value

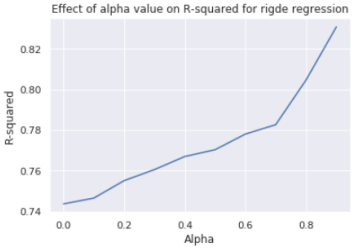

Figure 5: An illustration of momentum improving the results of gradient decent for rigde regresstion

Figures 4 and 5 illustrate how momentum improves stochastic gradient descent implemented in regression without and with regularization (ridge), respectively. The number of epochs needed to reach within 1 percent of optimal values varies with the parameter alpha (Figure 4). The optimal values were precalculated using the analytical expression derived in project 1. Here we have capped epochs at three thousand and consequently we do not see an improvement before alpha reaches .5. It should be noted that learning rate was set at 250 with 2 batches and further tuning of these parameters was not pursued. Stochastic gradient descent implemented on ridge regression did not lead to optimal beta values, at least within a reasonable number of iterations (data not shown). The optimal beta values were calculated based on analytical expres-

sion. Consequently, I used R - squared and sixty thousand iterations to measure improvement with gradient descent (figure 5).

For both methods described in figures 4 and 5, regression without and with regularization, respectively, momentum improved the results. In regression without regularization, momentum decreased the number of epochs needed to reach the optimal values from above three hundred thousand to fifty thousand. For ridge regression, momentum increased R-squared from .74 to .83. With many batches, the mean values of the gradients decreased. For ridge regression, the regularization component became as large as the rest of the gradient and pointed in the opposite direction. Consequently the model stopped improving. The magnitude of the gradient tended to spike in the early iterations.

3.2 Neural networks for regression

To determine whether FFNN can outperform the previous described regression models I developed a FFNN model with and without keras. Optimization of the parameters for these models are shown in figures 6 and 7. Figure 6 shows the FFNN model derived model derived R - squared as a function of batch size and the momentum parameter alpha. Smaller batch sizes and a small momentum parameter of .2 was the ideal combination.

This implied that stochastic (probabilistic) gradient calculated from a small batch was a good approximation of the true gradient. Consequently, we can take several steps in the correct direction for optimization at each epoch. This optimization would lead to moving faster towards the minimum and increasing computational efficiency.

An alternative analysis of figure 6 where small batch sizes and low alpha gives optimal values is that the magnitude of the gradient is scaling badly with batch size and we are looking at an artefact. I have not taken further steps to determine this possibility.

Figure 7 shows optimization of lambda. Whilst the high value of lambda seemed to be ideal, it came at the expense of the learning speed of the network.

The output from the FFNN algorithm implemented with and without keras, using some of the parameters described above is shown in the github file.

FFNN outperformed linear regression and ridge regression on the franke function, but not for all combinations of parameters. Using 2 hidden layers of 30 and 15 nodes, and with sigmoid as activation function we get a r - squared of .94 which is equal to that of ridge regression. When switching to 3 hidden layers of 30, 15 and 8 nodes with tan hyperbolic as activation function in the hidden layers, sigmoid in the output function of the last layer, and lambda equal to 0.00007 we get an r -squared of .99, for the test data.

Despite FFNN improvement regression, the results of FFNN tended to be somewhat inconsistent. This inconsistency made it difficult to know exactly

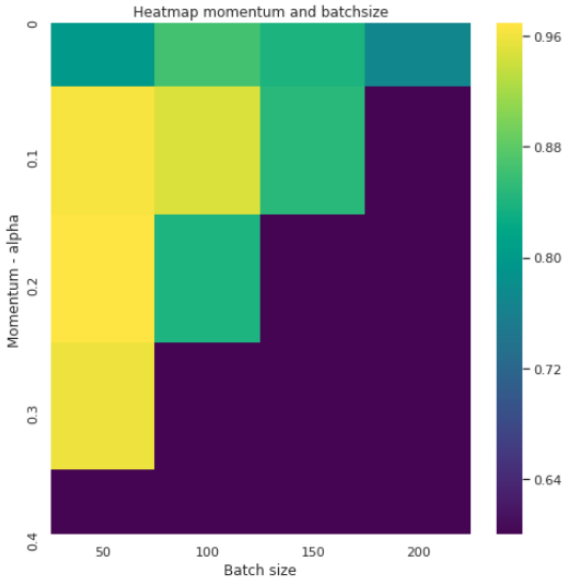

Figure 6: An illustration of R - squared as a function of batch size and the momentum parameter alpha

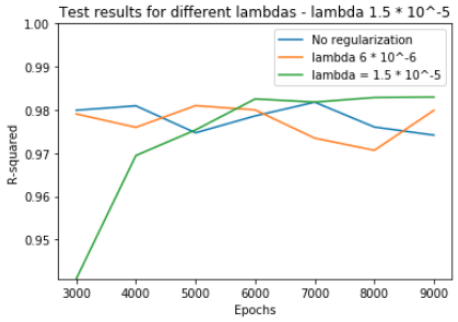

Figure 7: R - squared as a function of lambda and number of epochs

which combinations architecture parameters, learning rate and regularization were optimal. The tan hyperbolic and sigmoid function however gave more stable results than the relu, leaky relu and linear activation function.

FFNN with keras outperformed my implementation and at significantly faster learning rate. It predicted the franke function with a test R - squared of .997 using the relu function as activation function. Without Keras R-squared was calculated as .990. One possible reason that the implemented neural network without keras is slower is that the magnitude of the gradient without keras spikes in the early iterations. Spikes at early iterations prevent the network from using large step size which can improve learning speed.

3.3 Neural networks for classification

FFNN classified the image data with an accuracy score of .85 while the FFNN with keras achieved accuracy of .96. No further comparative analysis was performed on these models.

For the Wisconsin Cancer data set, both logistic regression and FFNN with keras had an accuracy score of .9370629 implying that the variables were weighted equally in both models. For the Wisconsin cancer data set, FFNN predicted an accuracy score of .916.

In the cancer dataset, there is an important trade off. One can lower the threshold for categorization, which will decrease the number of false negatives, but increase the number of false positives. This means that fewer cancers will go undetected, but positive results will be more unreliable.

4 Conclusions

In this study we show that learning speed for models of binomial classification, multinomial classification and regression, that make use of stochastic gradient descent, both linear and non-linear, with and without regularization, can be improved by adding momentum. It seams that if the stochastic gradient is representative of the true gradient, higher number of batches might be a good method for efficently optimizing the neural network. Linear regression with stochastic gradient descent was outperformed by FFNN.
FFNN implemented with keras, out-performed my network in prediction and at higher learning speed. Intializing the weights so that they are in the linear region of the sigmoid, greatly improved the speed at which my network learned.

4.1 Recomended future work on the topic

There are several issues that have not been addressed in this study. Most importantly, the methods used by keras in FFNN are unknown and comparative analysis becomes somewhat superficial and based on prediction alone. Secondly,

a systematic approach to optimizing parameters would make model development a lot more efficient. It would be of interest to know what makes a stochastic gradient representative of the true gradient and whether adding momentum would indeed make stochastic gradients more representative of the true gradient. It may be valuable to be able to calculate confidence intervals for the true gradient based on batch sizes. Lastly, I would be interested in attempting to introduce regularization parameters later in the stochastic gradient process, after a set of epochs, to improve learning time.

5 References

References

[1] T. Hastie, R. Tibshirani, and J. Friedman, *Neural Networks*. Springer New York, 2009.

[2] H.-J. Morten, "Machine learning leacture september 24," 2020.

[3] G. Montavon, G. Orr, and K.-R. Mller, *Neural Networks: Tricks of the Trade*. Springer Publishing Company, Incorporated, 2nd ed., 2012.

[4] J. H. M. Daniel Jurafsky, *Speech and Language Processing*. 2019. Chapter 5.4.