NeuralNetworks & Logistic regression binary classification breast cancer

Arne Elias Tidemand Ruud & Bjørn Magnus Hoddevik
University of Oslo
(Dated: November 22, 2022)

Breast cancer is the most common type of cancer that's diagnosed. About 1 in 8 of all cancer types. Using features such as mean radius, mean compactness and mean symmetry we are able to predict if the cancer is malignant or not. The goal is to build build a feed forward neural network (artificial neural network, loosely modeled the neurons in the brain) and logistic regression model able to determine that with determine with high accuracy (greater than 95%). Testing different activation functions with the network to improve network. Additionally we explore how the neural network can produce smaller mean squared error for a classic regression problem using the FrankieFunction. Briefly explore different optimizer which updates the learning rate for Stochastic Gradient Descent. For SGD with optimizers we reach the same value OLS(3.35e-5) and with the network we reach 1.1e-5. For the breast cancer classification the models are able to predict with about 99% accuracy which is outstanding considering the complexity of diagnosing such a disease as a human can be very difficult.

I. INTRODUCTION

In this report we wish to study the advantages and disadvantages of regression and feed forward neural networks (FFNN). We do this by comparing the results of a linear regression model compared to FFNN trained to handle regression. To begin with, we will start with simple equations which we use to verify if our models work. Next, we compare them with more advance functions. When comparing the two, we not only wish to look at which one does better, but also relate it to their complexity. FFNN's are more complex, not only when it comes to computation time, but also the amount of hyper parameters. We seek to see if the added complexity is justified by the improvement we see in our results.

To answer this question, we will not only look at linear regression problems, but also classification problems. We want to find out if a FFNN is the better choice for modelling in either, neither or just one of the cases.

To answer these questions, we will first develop a method for generating models for different data. We will start by coding a function which generates a model using various different linear regression learning rate schedulers, such as: Adagrad, RMSprop and Adam. Once verified that these work properly, we move onto constructing flexible FFNN code, which can easily be repurposed for classification analysis. Lastly, we code a logistical regression program. We can then compare the generated models.

When discussing our results, we will take into account complexity and computation time. We will also look at potential ceilings for possible improvement and diminishing returns.

The data sets mainly being explored is the FrankieFunction dataset which we generate with self defined function. After verifying the model is working the final dataset and main goal is to explore the breast cancer dataset to see if its possible to build an accurate model which predicts if the cancer is malignant or not. Therefore reducing the burden on doctors and potential for more human error in the life threatening genetic disease which account for 1 in 8 of all cancer types.

II. METHOD

A. Splitting and Scaling data

When it comes to any type of machine learning its important to prepossess the data. As the regression data is self generated and the breast cancer data is seemingly already devoid of any extreme outliers, misclassified or NaN values we just have to deal with splitting and scaling. For any When training a model its important to have independent data. Therefore we split the data, 75% percent for training and 25% for testing. Its crucial that the model training or the use of scaling is not affected at all by the test data as it would skew the way the weights are updated. Therefore no longer being able to used to test if the model works on new data. Similarly when scaling the features and/or targets its important we scale according to the training data. When scaling we subtract the mean and divide by the standard deviation.

B. Stochastic Gradient Descent

An additional level of complexity added to our models will be stochastic gradient descent(SGD). SGD differs from gradient descent (GD) by using mini-batches with a batch size lower than the size of the data set. So, instead of working with the whole data set at the same time, you instead work with a subset. This subset is chosen randomly without replacement to ensure a smaller bias from related consecutive data points, a problem which can arise for example from several samples being from the same patient.

To optimize the SGD algorithm there's many alternate version to change the learning rate used instead of

the scalar. The different methods that's explored further with the Frankie Function dataset is: RMSprop, Adagrad and Adam. These are adaptive learning rate methods. Will not derive too deep into how each algorithm works since it's not used throughout the project, however if deeper exploration is desired i refer to [9] [10] and [11] respectively. ADAM algorithm is pulled from the original paper published in 2014 [12]. Momentum is also a way to improve exploration by keeping a history of previous gradient, and can be compared to in physics a ball keeping momentum rolling down a hill and therefore avoiding a local minimum. The various optimizer is explored in first section of the paper.

C. Feed Forward Neural Network

To start, we look at how to construct FFNN code, specifically for regression analysis. Early on already, we must mach choices for how to structure our code. Using object oriented programming will make our code readable, easily modified and easier to develop new features. Conceptually, it makes sense to make one class for the neural network and another for initializing the individual layers within the neural network. Having a class object which can easily store variables specific to that layer and its nodes is crucial for developing our neural network class methods. It does not however make sense to make the neural network a subclass of our Layer class, since a neural network is only made up of layers and not a type of layer.

In order to get a better idea of what kind of parameters the classes will take when initializing, we must first discuss how FFNNs work. As all other types of machine learning, FFNNs are heavily dependent on the quality and quantity of our data, but also how we utilize it. In this report we choose to train our network using a SGD method. This comes with a major upside: reduction to computation time. As previously mentioned, SGD lets us work with a smaller subset of data at a time, which a neural network prospers from especially.

Additionally, having too much data to work with for each hidden layer means the cache in the computers memory would be forced to fetch data from the main memory several times when updating values within just one hidden layer. Meaning, that even with proper vectorization of our code we still rely heavily on the size of our cache and the bandwidth within our memory.

A feed forward neural network consists of two phases, the feed forward phase and the back propagation phase. The feed forward phase, which gives the method its name, is where the input data is fed through the network. To begin with, the input is given to the first layer which has a given activation function and a specified amount of nodes; when we initialize our layers, these must be given as parameters. Then, each layer has weights equal the amount of inputs per data point and a given bias. In the next layer, the number of weights is equal to the previous layers number of nodes; when initializing each layer, passing the previous layer as a parameter will make later use easier. This means the initial hidden layer must be initialized differently than the later ones. This feed forward phase can be written as:

$$\begin{aligned} Z^0 &= X\vec{w^0} + b^0 \\ A^0 &= \sigma^1(Z^0) \\ Z^1 &= A^0\vec{w^1} + b^1 \\ A^1 &= \sigma^1(Z^1) \\ &\dots \\ Z^i &= A^{i-1}\vec{w^i} + b^i \\ A^i &= \sigma^i(Z^i) \end{aligned} \tag{1}$$

Where $\vec{w^i}$ is the weights, bi is the bias and σ^i is the activation function of layer i. There are a variety of different activation functions. We will be experimenting with a variety of these depending on the problem which is to solved. The different activation functions on the hidden layers explored in this paper is: sigmoid, tanh and relu. The shape of the different functions is shown in figure below. One common issue for sigmoid and tanh is the vanishing gradient problem. Which is what happens due to the derivative decreases until at some point the partial derivative of the loss function approaches a value close to zero, and the partial derivative vanishes when doing back propagation. This becomes worse the more complex the network is. The RELU functions solves this as the derivative is constant equal 1 for inputs greater than 0. A potential problem however is if the input is less than 0, in which case the node becomes dead. This is fixed if you adjust the activation function a leaky RELU.[7] However this method is not explored in this paper.

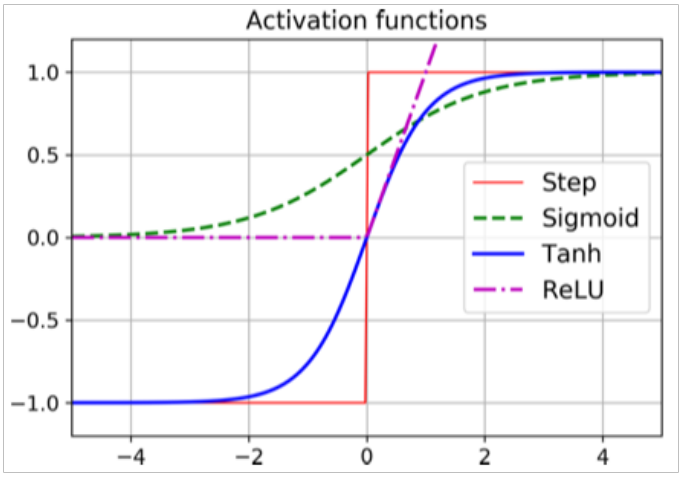

Figure 1. Shows the different activation functions tested in the hidden layers. [6]

The dimensionality of the values A^i and Z^i are given by $(n_{data}, n_{weights})$ where n_{data} are the amount of data

points and $n_{weights} = n_{nodes} \cdot n_{features}$, where n_{nodes} are the amount of nodes and $n_features$ is either the amount of nodes in the previous layer or the amount of features of the matrix X. The activation function in the output layer depends on the type of problem the FFNN is solving, in the case of a linear regression problem $\sigma^L(z) = z$ (Ronaghan, 2018). Updating these weights and biases is what happens when we teach our network, which takes place during the back propagation phase.

An important discussion to be had is how we choose to initialize our weights and biases. Biases are useful for preserving the gradient so that it does not vanish, while weights is a fundamental part of how the FFNN works. When it comes to weights, there is the matter of choosing their distribution and also their scale. We will be distributing the weights with a normal distribution. Then try to scale the distribution by trial and error. As for the weights, we will try both with, with different values, and without them for different activation functions.

The back propagation phase relies on the chain rule. It works by first taking the gradient of the OLS cost function with respect to both the weight w and bias b:

$$\begin{aligned} \frac{\partial c}{\partial w} &= \delta^L A^{L-1}, \quad \delta^L = y - t \\ \frac{\partial c}{\partial b} &= \delta^L \end{aligned} \tag{2}$$

Where δ^L is the error of our model, A^L is our current prediction and t is the target from our data. Intuitively, this error can be thought of as the a measure of how much we need to adjust our vectors. From the feed forward phase it is evident that the output of each layer is dependent on the the output of the previous layer. So by applying the chain rule, we can propagate this error backwards such that we can measure how much the weights and biases in each layer also needs to be changed. This is where the upside of smaller data sets from SGD comes in, These matrix multiplications can become exponentially worse for larger data sets.

These gradients can be found for every layer by the equation 2, only $\delta^l = \delta^{l+1} w^T \cdot \sigma'(Z^l)$. So starting with the output layer you can find δ^L which can in turn be used to find δ^{L-1} and so on until the first hidden layer δ^1. The weights and biases are then updated as in equation 3 after first having calculated all the gradients.

$$\begin{aligned} w^i &= w^i - \eta \delta^i A^{i-1} + 2\lambda w^i \\ b^i &= b^i - \eta b^i + 2\lambda b\hat{i} \end{aligned} \tag{3}$$

Where η and λ are both hyper parameters, the learning rate and regularization parameter respectively. The η is a scalar which is multiplied with the gradient when updating weights in the network. A larger η means the network is trying to take bigger steps towards the optimal parameter. However if too large it can overshoot and not be able to reach the bottom. If too small it will take however take too long. Therefore we wish to find a middle ground[8]. These are two more parameters our neural network code needs to take in when initializing. In order to find the best combination of these two parameters, we will be performing a grid search.

A grid search is when you try different combinations of values of two parameters in order to search for the lowest error. This can be a calculation heavy task, and in order to alleviate some of the trouble, we will not be doing as many mini-batches. A relatively small amount of back propagation rounds should be enough to give us a clear winning combination. This will be a center point of our analysis.

So far, we have concluded that our neural network class object needs to be initialized with the parameters: n_{layers}, n_{nodes} per layer, σ and σ', learning rate η and lastly a regularization parameter λ. From discussing how stochastic gradient descent works, we will also need two additional parameters: number of epochs and batch size. Finally, from equation 1 and 1, it is apparent we also need to take in the data as a parameter. Our final initialization function should look something like figure 2.

```
def __init__(self, X_data, Y_data, n_layers, n_nodes, sigma,
             sigma_d, epochs=100, batch_size=100, etaVal=0.001,
             lmbd=0, showruninfo=False):
```

Figure 2. Shows the parameters of our neural network class.

While our layer class object only needs the previous layer, *nnodes*, σ and σ'. However, this will not be enough. How we initialize our weights and biases will also have an effect on how well our models do. The bias is just a simple scalar, which can easily be included in the class' initialization and is set to 0. Choosing how the weights are initialized is more difficult. The weights is initialized randomly using numpys rand which returns samples from the "standard normal" distribution. To help the network the weights is divided by the number of nodes in the layer.

Next we need to discuss class methods, starting with our NN class. For our linear regression, there should be two methods used by the user. One to train and one to predict. However, there is also an argument for letting the user choose the activation function of the output layer, so this should also be easily possible when using the class. The train function should not take any parameters, as the parameters needed for SGD have already been passed in the initialization. However as we want to evaluate on the test data we send in X test and Y test for each epoch. The train function iterates through the amount of epochs and within each epoch through all the data points with batch size number of data points at a time, meaning $iterations = n_{datapoints}/batchsize$. For each iteration, a mini-batch is selected and used in the feed forward phase and then the back propagation phase. These phases should have their own private class methods. The predict method needs to be able to take in a new data set, as it will mostly be used on the testing

data. The prediction method then relies on a specialized feed forward method. The structure of the class then ends up looking like figure 3. The network class stores all the layers in an array.

```
class NeuralNetwork:
    def __init__(self, X_data, Y_data, n_layers, n_nodes, sigma, sigma_d,
                 epochs=100, batch_size=100, etaVal=0.001, lmbd=0, type="Regression", showruninfo=False):

    #feeds the input data forward
    def feedForward(self): # (batch_size, no_of_output_nodes=1), for regression

    #does the same as feedForward, only it takes a input and returns a value instead
    def feedForwardOut(self, X):

    #performs the back propagation
    def backProp(self):

    def train(self, X_test = None, Y_test = None, calcMSE = False, calcAcc= False):

    #Predicts
    def predict(self, X):

    def get_MSEtest(self):

    def get_accTrain(self):

    def get_accTest(self):
```

Figure 3. Shows the different methods of our Neural Network class.

The layer class does not have many methods, only property and setter methods for easily storing data within the objects. The layer object stores the weights and bias which is updated in the NeuralNetWork class. As well as the output value of the different nodes which depends on the activation function chosen.

In our analysis we will first use a simple polynomial (4).

$$f(x) = 1 + 5x + 3x^2 \tag{4}$$

Then move on to a more complex function, the Franke function which is used for all the SGD and regression using neurtal network calculations (5).

$$\begin{aligned} Frankie(x,y) = &\frac{3}{4}\exp\left(-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}\right) \\ &+ \frac{3}{4}\exp\left(-\frac{(9x+1)^2}{49} - \frac{(9y+1)}{10}\right) \\ &+ \frac{1}{2}\exp\left(-\frac{(9x-7)^2}{4} - \frac{(9y-3)^2}{4}\right) \\ &- \frac{1}{5}\exp\left(-(9x-4)^2 - (9y-7)^2\right) \ . \end{aligned} \tag{5}$$

D. Logistic regression & Classification

As we move from a regression problem where we predict a specific scalar, we now instead are trying to learn a binary classification problem. The goal is now for the network to either predict 1 meaning True or 0 meaning False. The data-set which is now being explored is the breast cancer pulled from sklearn library. The data is then scaled using sklearn StandardScaler producing better results due to big differences in the scalar of the features. The data-set consists of 30 features and 569 samples where 212 is malignant [3]. The output produced for one sample will simply be a 0 or 1.

There is no changes done structurally to the way network works. However a step function is introduced to the output for the network where if the value is less than 0.5 the value is set to 0 and if equal or greater than 0.5 its set to 1. Then using sklearn accuracy score we calculate the accuracy on the train and test data and store it in a array in the class.

Logistic regression makes use of the neural network architecture. However it is simplified, consisting just of an input and an output layer. A separate class called LogRegClass is made looking similar to the NeuralNetwork, just excluding hidden layers and the step-wise function being added directly. Then just adding couple functions to get access to the lists containing the accuracy for train and test data for every epoch.

III. RESULTS & DISCUSSION

A. Regression: SGD & Neural Network

Firstly we explore how different adaptive learning rate optimizers with SGD affects the MSE using the FrankieFunction data. Using gridsearch we explore the best parameters within a range and use those to compare how MSE evolves over the number of epochs that's calculated.

As we can see from the gridsearch for the default SGD with a eta over 1, overflow occurs (MSE is set to 10 if error in mse function due to infinity/NaN). 4 A learning rate less than 0.001 seems to not be able to reach minimum fast enough. The lambda has just a negative effect as well. For Adaprop, seemingly the eta value is again the most important as for anything lower than 0.01 it fails to reach a minimum. RMS however performs quite well across the board as long as lambda remains below 0.001. Adam performs better with a larger eta value and a low lambda value.

We see that the for all except deafult SGD we reach about the same MSE as calculated with SKlearn OLS regression model of 3.39e-05. The best parameters corresponding to find the best MSE value is pulled and the MSE over number epochs is then plotted to show how fast the methods converge. S 5 6 Reading the gridsearch we see that there's several paramters for certain methods that all converge to a similar loss, however the speed it reaches it varies. One for mini batch size equals 70 and one for 120 is shown. The major difference being how many epochs is needed. With a larger batch size requiring more epochs. However this is likely due to the algorithm looping over m=number_of_epochs/batch_size. Meaning more calculations per epoch. Additionally if the batch size is too large e.g 1000 it lead to no learning. This however might be an error in the algorithm, as you'd expect as the batch size get bigger the gradient would be

more precise, but computation time drastically increase. For both 70 and 120 there's a clear winner. ADAM performs best and RMSprop and Ada seems to follow each other very closely. While the default lags far behind and requiring a lot more epochs to arrive at a similar MSE. Default lacking behind makes sense as it does not incorporate any memory or speed up when it comes to the change of eta. One downside of ADAM is for a lot of epochs it seems to get unstable for an epsilon with 1e-8. This might be due to the way ADAM algorithm if v_hat become too small. To fix this we set the epsilon to a larger number which seemed to help.[12]

To begin with the neural network, we produced a plot 7 showing how the MSE evolves for every mini-batch for four different activation functions: Sigmoid, RELU, Tanh and leakyRELU. with learning rate $\lambda = 10^{-2}$, $\lambda = 10^{-3}$, $\lambda = 10^{-3}$ and $\lambda = 10^{-4}$ respectively. The simple polynomial 4 has been used.

Now for the FrankieFunction data we want to explore best eta and lambda for the sigmoid function. Do a gridsearch, which shows regardless of epochs the best value for eta seems to be 0.01 and lambda equals zero. For 30 epochs the MSE for these parameters is 6.3e-3, 300 epochs 1.9e-4 and 10 000 epochs the value is 1.1e-5. Comparing values with OLS from 5 of 3.4e-5 it seems the network performs better after 10 000 epochs with specific parameters by a power of 10. However the computation needed is obviously much more and time consuming. Also reading the gridsearch its clearly quite sensitive for the different parameters. Its also worth mentioning in our data set, we create as many samples as we wish. These methods are quite data hungry, so these results where we have generated 7000 samples in both x and the y space might be unrealistic depending on the data and relationship we study.

B. Classification: Network & Logistic Regression

Moving onto binary classification we use our existing network, but only modify the output with a step-wise function to calculate accuracy. We set layers to 2 and nodes to 16 as they seem to be a good baseline with a relativly simple architecture. Then search for lambda and eta over a wide range of values.10 We let the model train over 300 epochs and manually read off the best parameters to pick for each type of activation function we test: sigmoid, tanh and RELU. Then we do the same thing and search for best combination of layers and nodes using those values. We observe several of the combinations perform extremely well. Producing an accuracy of 99 percent. The models generally performs worse for higher complexity (4-5 layers). This might be due to the number of weights needing to be train. For the best models we are able to reach 99 percent with just one layer and either 4 (sigmoid and relu) or 32 nodes(RELU) which is impressive. Generally the model performs poor with lambda greater than 0.1. As a too high regularization parameter prevents the model will ignore the data and not able to update the weights correctly. Also with a too little learning rate will not be able to reach the minimum after 300 epochs. 11

To display the problem of overfitting the accuracy for both train and test data is plotted for an eta value for 0.001 over 250 epochs.12 We here can clearly see but its also marked with an X symbol where the accuracy peaks. After about 100 epochs the test data performs worse due to the model being overfitted to the training data. Eventually test data accuracy reaches close to or equal to 1(100%). At best the basic model predicts with about 98% accuracy which is quite well, considering the parameters is not searched for in detail.

Next the Logistic Regression model is considered. To have something to compare with the model the LinearRegression from sklearn library is used. The figure shows the accuracy for different amount of epochs as well as the effect scaling the data has on the result. 13 As expected the unscaled data performs very poorly as the features varies in scalar which introduces imbalances in how the features are weighted. Also by scaling the network can learn faster.

Then as with the neural network we do gridsearch to find the best eta and lambda combination.15 However since there's no longer hidden layers, there's no need to search for those parameters. Leading to less computation time and effort. For both sigmoid and tanh we improve on sk learn model from 97.9 percent to 98.6 percent while RELU matches it. The plot for 30 epochs is also included to highlight that for certain parameters its able to match sk learn. However its not able to improve upon the score until we reach a certain threshold for number of epochs. This could be due to sklearn model not extensively searching for the best parameters.

Generally judging from the gridsearch for this dataset the sigmoid generally performs better across the board with the different parameters. While tanh performing worse poor lambdas over 10. RELU however is dependent mainly on the eta value. As it performs quite poorly generally for a value less than 0.01.

The tradeoff between using the custom model and sklearn being the computation time needed for the increase in accuracy. However when building a model which potentially could be saved and used for extended amount of time, the computation time being e.g hours/-days (it's not in this case) is worth the small increase.

IV. CONCLUSION

Conclusion Discuss pros cons of algorithms. OLS, NeuralNetwork, LogisticRegression Include main results from MSE and Accuracy.

The stochastic gradient descent to solve regression problem is more general and works for any objective function and OLS mean squared error is not always the best way to fit the line to the data. As computers got

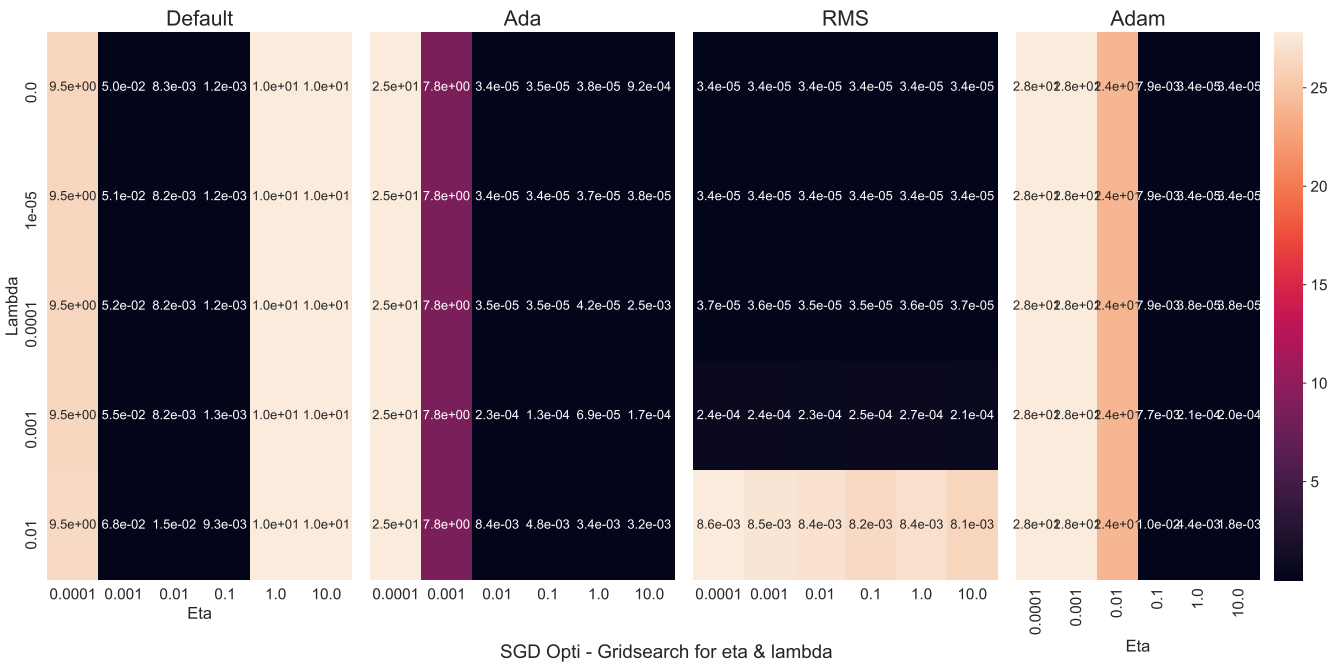

Figure 4. Example of how gridsearch looks for a run with 400 epochs with mini batch size=70. Learning rates spanning from 0.0001 to 10 and lambda from 0, 1e-5 to 0.01. Each grid for a different type of learning rate optimizer.

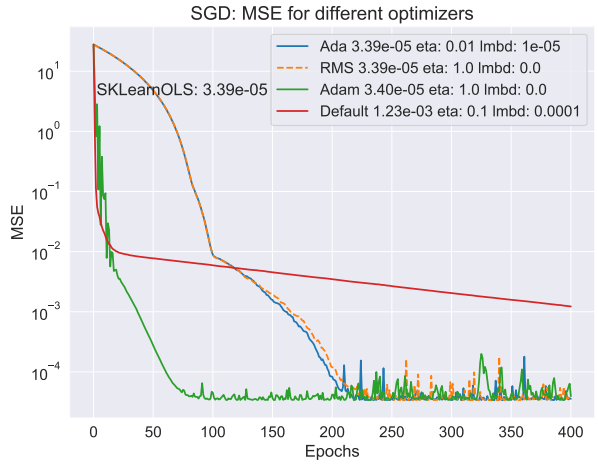

Figure 5. MSE score for 400 epochs with a mini batch=70. Parameters found using gridsearch.

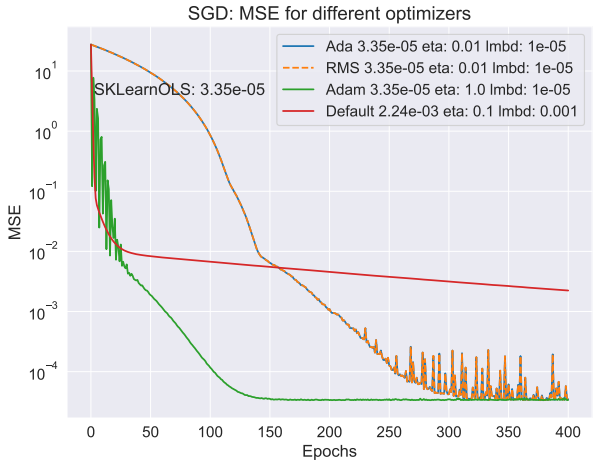

Figure 6. MSE score for 400 epochs with a mini batch=120. Parameters found using gridsearch. Hard to draw definite conclusions. Not done a lot of search for these parameters.

faster computing the gradient in a multidimensional feature plane is a viable solution. When the feature matrix is complex inverting is also no longer possible, therefore we must use a search algorithm such as SGD. SGD is also able to handle big data sets. Main downside is time spent calculating. Not only all the back-propagation calculating gradients, but one might also need to do more extensive search to find the best parameters resulting in the best results. Also ways to optimize the SGD by using optimizers such as Adam, Adagrad and RMSprop. We also divide the data set into mini batches, such as instead of calculating gradient over whole data set, we pick out a portion and calculate on that. In which due improves speed. Judging by the results done with Frankie data set the ADAM seemingly reached not only same value as SKlearn but also at a much faster speed than others, just requiring about 100 epochs with a batch size=70. For regression with the Frankie function generated data the neural network, with 10 000 epochs ,was able to reach

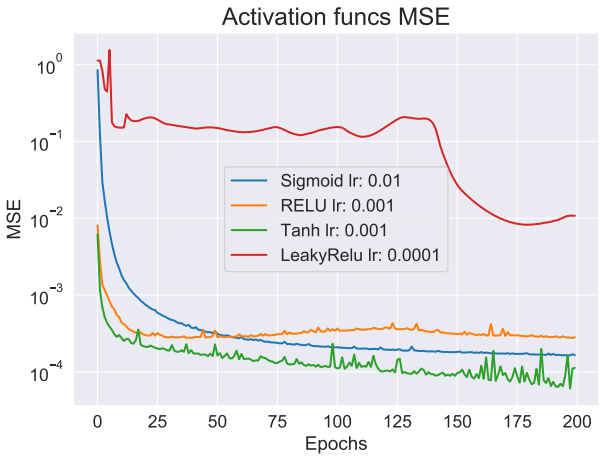

Figure 7. Activation functions for a simple 2nd degree polynomial. Sigmoid, RELU, tanh and LeakyRelu. Learning rate set to 0.001 for RELU and tanh. For sigmoid its 0.01 amd leakyRelu its 0.0001.

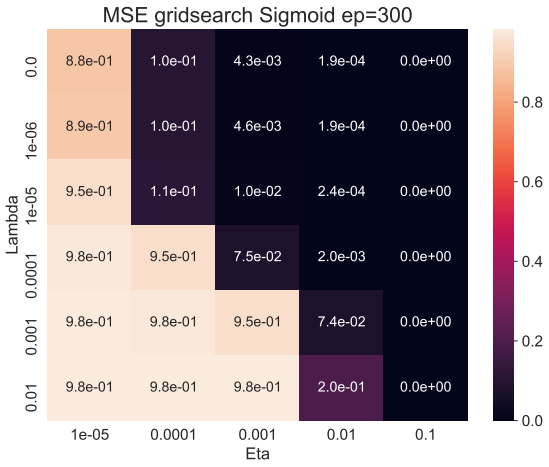

Figure 8. Activation functions for a simple 2nd degree polynomial. Sigmoid, RELU, tanh and LeakyRelu. Learning rate set to 0.001 for RELU and tanh. For sigmoid its 0.01 amd leakyRelu its 0.0001.

a MSE of 1.1e-05 compared to the SK learn OLS model which resulted in an MSE of 3.35e-05.

For the main goal which was to look if we could predict if a the breast cancer was malignant or not with high certainty (¿95%) it definitely succeeded. As for the neural network with both using sigmoid and tanh was able to reach 99% (rounded off) accuracy after extensive gridsearch was performed to find the best combination of eta, lambda, layer and nodes. A simplified neural network with no hidden layers, Logistic Regression, was also able to reach high certainty. Compared to sk learn regression model, the custom model after another gridsearch with the three different activation functions; sigmoid, tanh and RELU was performed. After 300 epochs it was able to reach 98.6% accuracy on new test data fed into the model. Seemingly as the more complex network and logistic regression performed similar, the network required more time. This is due to more gradients and layers need to be computed. Logistic regression also requires less paramters to be tuned as its architecture is very basic.

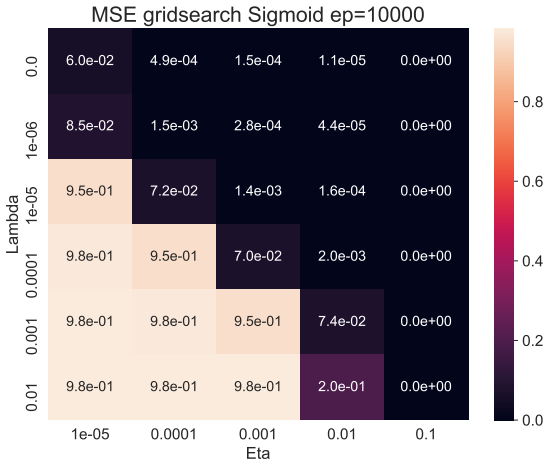

Figure 9. Activation functions for a simple 2nd degree polynomial. Sigmoid, RELU, tanh and LeakyRelu. Learning rate set to 0.001 for RELU and tanh. For sigmoid its 0.01 amd leakyRelu its 0.0001.

A. To improve/what to explore

- Different optimizer were explored initially using stochastic gradient descent, however were not implemented into the network. Doing this would definitely improve the speed of likely the performance of the network in general for both regression and classification problems. As seen in 5 ADAM seems to perform best, and to avoid to many gridsearch it would be best to implement that.
- As it stands the weights are randomized from the standard normal distribution divided by number of nodes. However different activation functions perform better with different intialized weights. For example tanh and RELU benefit from “glorot” or “xavier” initialization. [13]
- Another way to both store model for later use and improve speed when training network with different epochs is to store the weights/network into a text file to then later load them in anywhere else. This would increase the ease of use anywhere and any point in the future when training the model. As well as enabling faster search for parameters, in turn improving the model.

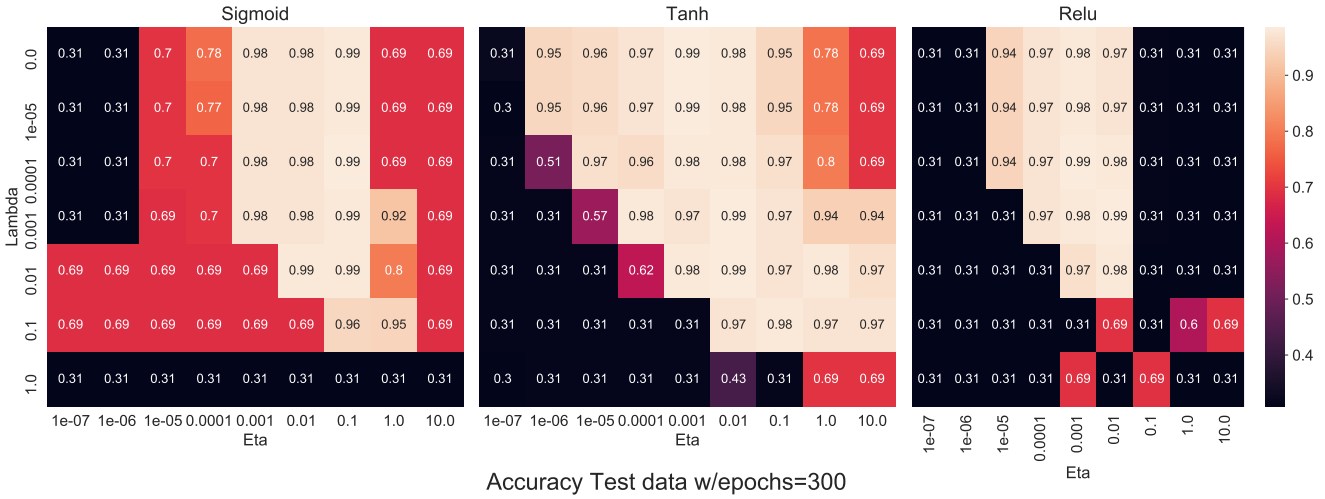

Figure 10. Gridsearch for breast cancer dataset searching for best eta and lambda value using layers=2 and nodes=16. Three different activation functions: sigmoid, tanh and RELU

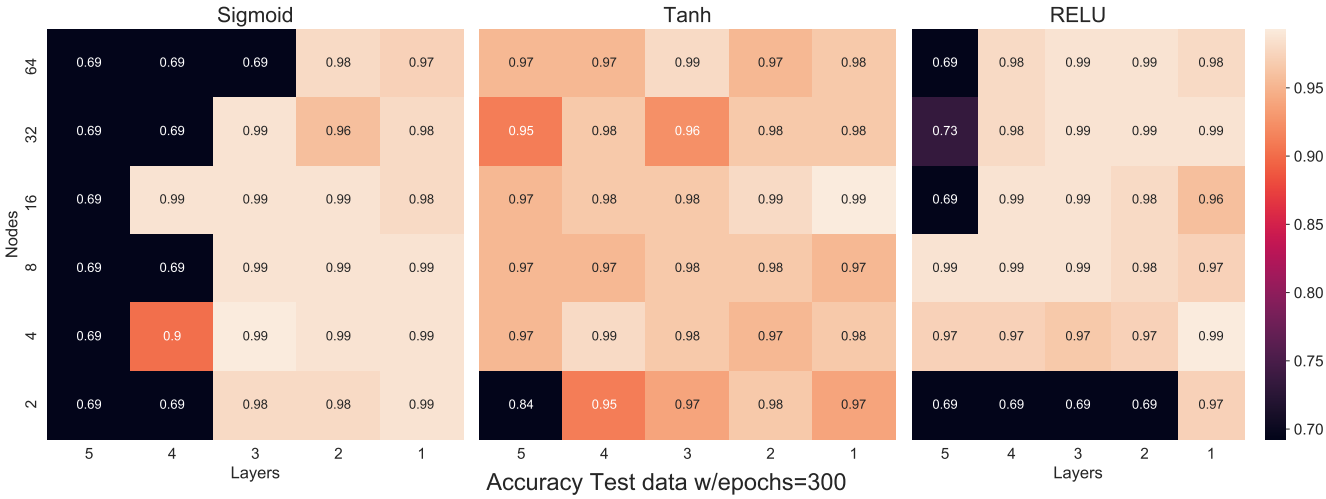

Figure 11. Gridsearch for breast cancer dataset searching for best layer and nodes value using layers=2 and nodes=16. Three different activation functions: sigmoid, tanh and RELU. Chosen best value for eta and lambda by manually reading them off.

- Lastly the code can be better generalized such a way to run any type of problem which the fast forward neural network could solve. This includes perhaps more auto detection and simpler way to input the data into the model.

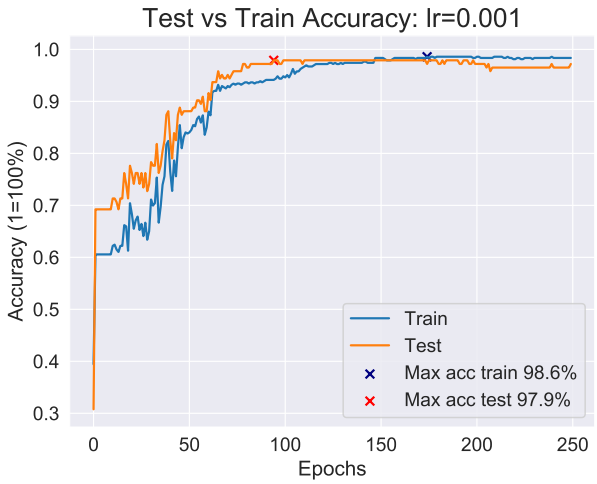

Figure 12. Sigmoid. Accuracy for test and train data plotted vs epochs. Shows clearly concept of overfitting where after a certain amount of training the model performs worse with the test data.

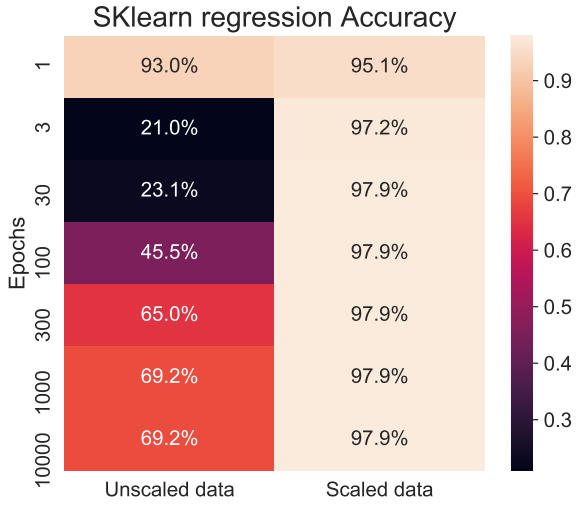

Figure 13. Using SK_learn linear model: LogisticRegression to predict with the breast cancer data set. Scaled vs unscaled data. Over epochs from 1 up to 10 000. 97.9 percent being the best value which is reached after 30 epochs.

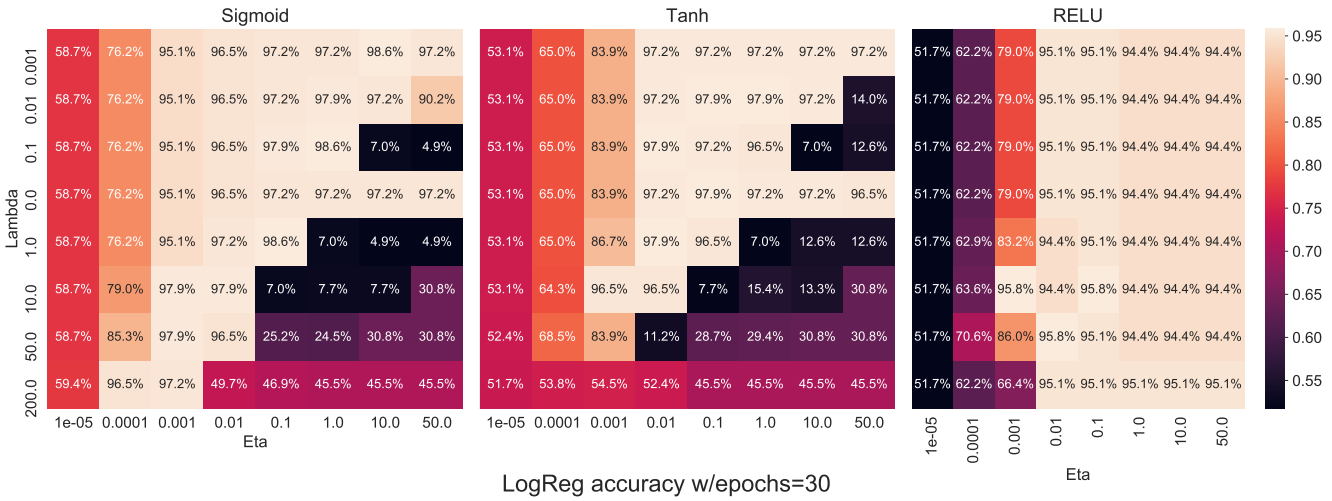

Figure 14. Search for eta and lambda value to optimize the accuracy using logistic regression class. Run is done with 30 epochs and displays best value as 97.9% for sigmoid and tanh, and 95.8% for RELU. Using the breast cancer dataset.

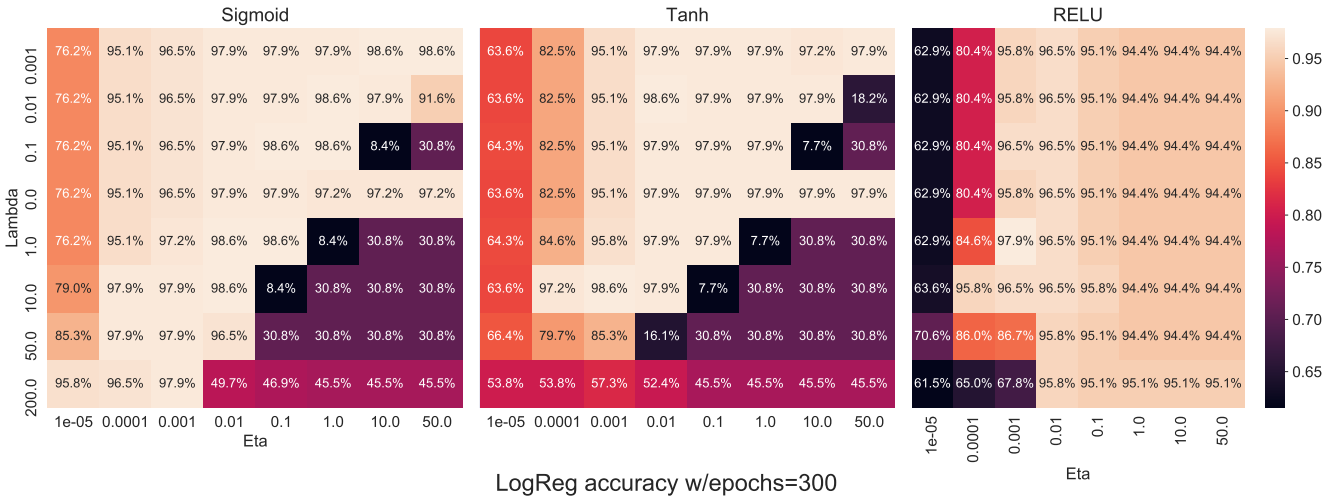

Figure 15. Search for eta and lambda value to optimize the accuracy using logistic regression class. Run is done with 300 epochs and displays best value as 98.6% for sigmoid and tanh, and 97.9% for RELU. Using the breast cancer dataset.

[1] O.L. Mangasarian, W.N. Street and W.H. Wolberg, University of Wisconsin, *Breast Cancer Wisconsin (Diagnostic) Data Set*. 1995.

[2] Stacey Ronaghan (Data Scientist at Amazon Studios), *Deep Learning: Which Loss and Activation Functions should I use?*, TowardsDataScience, 2018.

[3] Amita Dhainje (Machine learning engineer), *Breast Cancer Dataset Classification, exploration of dataset*, Kaggle, 2018.

[4] Attyuttam Saha (Software Engineer, MCA from National Institute of Technology Warnagal), *Comparison between Logistic Regression and Neural networks in classifying digits*, Artificial Intelligence in Plain English, 2020.

[5] Michael Nielsen (author and researcher at Astera Institute), *How the backpropagation algorithm works*, Book: Neural Networks and Deep Learning, 2015.

[6] Kinder Chen, *Hidden Layer Activation Functions*, Medium, 2021.

[7] Tina Jacob, *Vanishing Gradient Problem, Explained*, kdnuggets, 2022.

[8] Jeremy Jordan, *Setting the learning rate of your neural network.*, jeremyjordan, 2018.

[9] Jason Huang, Cornell University *RMSProp*, Cornell University Computational Optimization Open Textbook, 2020.

[10] Daniel Villarraga, Cornell University *AdaGrad*, Cornell University Computational Optimization Open Textbook, 2021.

[11] Akash Ajagekar, Cornell University *Adam*, Cornell University Computational Optimization Open Textbook, 2021.

[12] Diederik P. Kingma, University of Amsterdam & OpenAI, Jimmy Lei Ba, University of Toronto *ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION*, Published as a conference paper at ICLR 2015, Page 2 2014.

[13] Jason Brownlee, *Weight Initialization for Deep Learning Neural Networks*, Machine Learning Mastery, 2021.

[14] Elias T.Ruud and Magnus Hoddevik, *GitHub Code - Project2Final*, GitHub Repo, 2022.