Fys-Stk4155 Prosjekt2

Thomas Bergheim
Vidar Lunde Olaisen

September 2021

Contents

1 Abstract

In this project we will sstudy both regression and classification problems by developing our own feed-forward neural network (FFNN) and Stochastic Gradient Descent code. Our data is generated by the Frankefunction and MNIST-data (handwritten numbers). In this project we studied both regression and classification problems by trying to develop our own feed-forward Neural Network(FFNN) and by combining it with Stochastic Gradient Descent(SGD) we learned how to approximate models based on learning up models. We've found out how Learning rate affects the speed of approximation and how randomness in the model as both a good and negative aspect of FFNN with back-propagation. We also introduced a parameter called regularization(λ) that worked to limit the growth or shrinkade of the weights. With these in mind we worked on a graph created with Franke Function, for then afterwards to go over to data about Breast Cancer from Winsconsin.

2 Introduction

As technology improves day by day new problems occurs that requires more and more computational power to be solved. Therefore we need to find new methods to solve these problem effectively. These new problems most likely differ a lot from each other and it is therefore difficult to find one common specific model or solution which solves all of the problems. Our approach to the problems will therefore have be to solve each of them individually to find a optimal solution for the specific problem. However, the path to the right solution and method are not always as easy as we want, and there are most likely some considerations to take. These can be when we're working with huge amounts of data, or if the data has several outliners, or if it contains many random variables that doesn't affect the model in any way that matters. These are just a few that we have to solve for each time, we will therefore try a new blueprint for algorithm that will solve these problems while at the same time doing it effectively.

So in this project the main target is to solve a problem concerning big data, and we will try to solve it by implementing Stochastic Gradient Descent(SGD) into a Neural Network. SGD is becoming more and more usual in the algorithm world as an regression model, but the one we've learned up til now has been Ordinary Least Squares(OLS). So why don't we keep using OLS instead of SDD?

OLS is an regression method is a good way to approximate the correct model, but only while the data is limited. At once we're working with big data, as in this project, OLS meets a barrier when it comes to computational time. The reason for this is that OLS takes all the data, compiles it into memory, start doing calculations with the whole set several times and in the end gives a good model. This won't work if the matrices are too large for the computer to handle.

SGD works in a different way, it starts with random variables, but instead of using the entire dataset to approximate the optimal solution, it takes one and one point to improve. This does however bear the risk of importing randomness over accuracy to save computational power.

And it's here the Neural Network comes into the picture, while SGD on it's own has negative effects thanks to randomness, in Neural Networks it can actually be a positive aspect. We in this project try to see how these two interact with eachother, and if it works at all on both regression and classifications.

3 Theory

3.1 Gradient Descent

Gradient descent is a first order iterative optimization algorithm. As it is a first order system the algorithm is best at solving first order derivatives. From this knowledge the more data provided the better accuracy should be obtained.

As the algorithm deals with iteration it means that we start with a model with some values, which can be random, and from there it hopefully approximates closer and closer to the best value, or the closest descent for each loop. The algorithm is also an optimization algorithm, and it simply infers that the object is to find the best element available given some criterion.

There are 3 different types of gradient descent, where they differ by the number of data they take to compute each step.

The first type is Batch Gradient Descent. This method take use of all the data for each new step. The next type is mini-Batch Gradient Descent. Instead of using all the data points, we split the data into subset which are then used for each iteration. The last type is Stochastic Gradient Descent. This method takes one single data point for each step.

Gradient Descent is a machine learning algorithm that tries to fit hidden variables based on the input and output in a way that it can predict future input as accurately as possible. The model is built up by the input in form of data points in a matrix X, a model g(β) where β are parameters and a cost function C(X, g(β)). The cost function makes it possible to compare the model g(β) against other models.

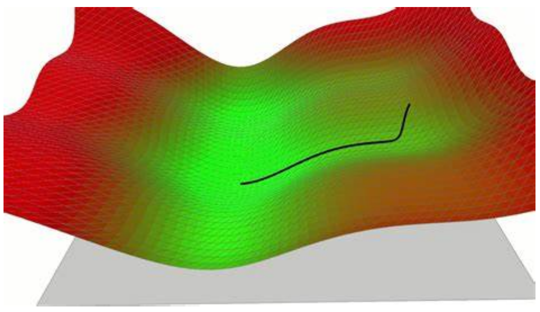


Figure 1: 3D plot of terrain data with gradient descent. The greener the better results

Figure 1 shows us a typical grid of terrain, where the colors describe the cost function based of the x and y axis. It illustrates nicely that the lower the terrain, the greener, and therefore the lower cost function. The point where the model is greenest is the global minimum, which means that if we select the x and y variable that corresponds with that cost value, we get the best results when accurately predicting future input.m.

The main purpose of Gradient Descent is to find a local/global minimum from a differential function, or in other words to find the lowest cost function value.

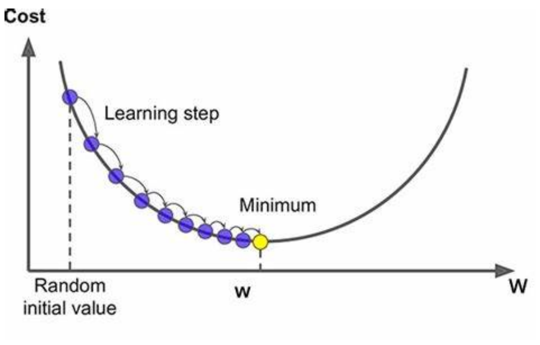
$$C(\beta) = \sum_{i=1}^{n} c_i(\mathbf{x}_i, \beta)$$

This can be achieved with the help of the gradient.The main idea is to first select an element, either chosen or random. From this point we want to know which direction the greatest change exists. This is achieved by computing the gradient from the element to any nearby elements. On the opposite side we can also use this to find the lowest value. We then subtract the gradient from the element. This might result in reaching a either local or global minimum. When the gradient is found we now choose to keep moving in the direction we found the result we wanted. one problem occurs and it is, how far do we need to search for other elements to include the important data we else might miss.

$$\bar{x}_{k+1} = \bar{x}_k - \gamma_k \nabla F(\bar{x}_k)$$

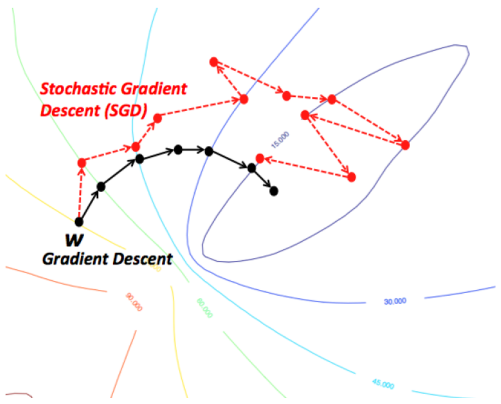
Here $\bar{x}_k$ is the element while $-\nabla F(\bar{x}_k)$ is the equation for the negative gradient.

A solution to the problem is to introduce a variable that controls the search rate. This variable is referred to as γ and is called learning rate. The learning rate controls how far each search step should be. The choice of learning rate is important as choosing a too large step will increase our pace towards a minimum, but it might overshoot and miss the lowest point and instead jump back and forward indefinitely until the loop ends. This also introduces a few other factor one might want or not, the first one is that it includes more randomness into the selection of local/global minimum, as it can overshoot greedily and end up in a different grove, which can lead to better or worse results. It also requires less cycles to approximate a good result. On the other hand it can also be a good idea to choose small steps. This has positive effects that the large learning steps don't have, most essential it can actually reach the minimum point in a groove thanks to the small steps. It does however not have the positive effects large steps have. A good approach to get the best of both worlds is to have a dynamic learning rate that decreases during the computation, this leads to both having less cycles and getting more accurate results.



3.1.1 Stochastic Gradient Descent

As we work with a large amount of data the Gradient Descent might still run into problems with time usage and computational power as OLS would. We can then use a version of Gradient Descent called Stochastic Gradient Descent (SGD) instead. SGD differs from GD in that while GD takes the whole the whole dataset for each step in approximating the global minimum cost, SGD only takes one and one data point to approximate. Progressing forward this way makes the computational time of the algorithm much quicker, but it introduces randomness to the results. Because while the GD know what way to go because it has all the data from the dataset at it's disposal, the SGD only have one datapoint to go from. This means that the randomly selected datapoint has all the power to decide what way for the gradient to go, and therefore it will quite rarely go the best way for the certain point it's standing on.



In this figure we can see that the GD always goes the way that the descent is highest, and it will therefore go straight to the global minimum in the plot. SGD will on the other hand go all over that place, but given enough time it will approximate towards the global minimum.

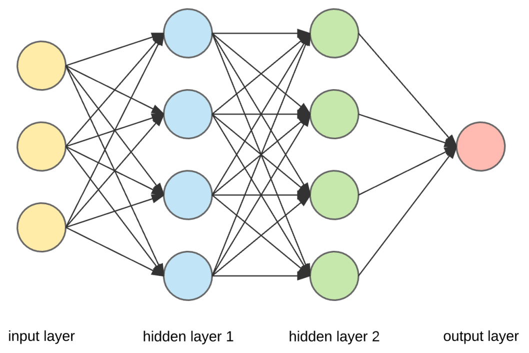
3.2 Feed Forward Neural Networks

Feed Forward Neural Networks(FFNN) with back propagation is a machine learning algorithm that vaguely tries to reassemble how the neurons in the brain work together. By working together in the same way as neurons the algorithm is trying to evolve into the best fitting model.

The structure of the FFNN contains several layers and it's algorithm is built up of 2 processes.

For the structure of the FFNN is the input layer, output layer and in between one or several hidden layers depending on the complexity the programmer wants. For the hidden layers and output layers we add weights and bias to tune the network.

The first processes is forward propagation, which calculates the values according to the different weights, biases and the input. While the second process is backward propagation, here we calculate the error of the model, in other words how correctly it is in guessing the output of the input. We also update the weights and the biases in an effort of trying to decrease the error of the model.



The image shows a neural network that contains two hidden layers. All FFNN has input and output layers, and most have hidden layers. The paths or connections between the layers are the weights. The circles are refereed to as nodes. The nodes contains the values and bias of the input, middle calculations and output.

Our input layer contains the data points we want to evaluate. In our project the input is the xy-grid which means we need two nodes in our input layer.

The nodes in the hidden layers contain values computed from the input layers and the belonging weights. The amount of nodes are chosen and can be as many as wanted or needed. But as the amount of nodes increase, so does the computation time as well. When adding more nodes than necessary there might emerge calculation errors. The hidden layers are often referred to as black boxes as there are no limits to how many layers and nodes we can add to the system. The nickname is given as it will be difficult to have a good overview over each layer and data point.

The output layer is the most interesting part of the algorithm. The result from the output can tell us how accurate the model is. In this layer we can differ error in the input from correct input and also retrieve the calculated result of the algorithm.

One of the most important part of the system are the weights. The weights are the connecting link between the different layers, and they combine the layers together. For each connection there is a weight affecting the calculation of the next value. The weights are the only changeable part in the system. For a new made network we do not know what values the weights should have.They are therefore chosen by random.

Bias are added to each node in the network to control the influence from several weights onto one node. In our FFNN we add bias on every node, but it's also possible to add a single node in every layer except output, and just have the unique bias there where there are only weights from the bias to the next layer and not into the bias.

The activation function is a function that scales the calculated values for each layer. The reason for scaling is to decrease the amount of computing power at the same time as the data becomes more readable and understandable for the user. We also want scaled values when we are looking at a classification problem. The output layer will also directly be the result we are looking for.

3.3 Logistic Regression

Logistic regression is a process where we want to calculate the probability of output given a certain input. The problems are often classification problems where the output is binary, either true or false, 1 or 0. The Logistic Regression is a linear model transformed into a non-linear model using the sigmoid function. The advantage of using the sigmoid function is because the function smooth out the output when there are changes in the weights and bias. By using the sigmoid function our outpus is no longer just binary, but any real number between 0 and 1. This is useful if we want to look at the output as average probability of some sort. On the other hand it is difficult to process unscaled data as the input could be any number higher and lower than 0 and 1 but the output will be in between.

4 Methods

4.1 Forward Propagation

In FFNN the main effort is just calculating the different values of the different nodes according to the input, weights, biases and activation functions.

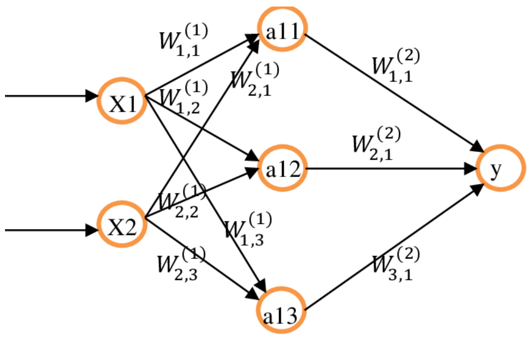


Figure 2: A typical FFNN Forward Propagation

Figure 2 describes how a a typical FFNN Forward Propagation might look like visualized, where we begin from the left, also called input node, and move us right through the hidden nodes until we reach the output node. In more mathematical terms it can be described with the following equations.

$$
\begin{aligned}
z_1 &= X * W_1 + b_1 \\
a_1 &= f_1(z_1) \\
o &= a_2 * W_2 + b_2 \\
y &= f_2(o)
\end{aligned}
$$

In a more programming sense we can calculate each node for itself, first we can show how to calculate the hidden layers nodes values, and then use the activation function on the value

$$
\begin{aligned}
z_{1,1} &= X_1 * W_{1,1}^{(1)} + X_2 * W_{2,1}^{(1)} + b_{2,1} \\
a_{1,1} &= f_1(z_{1,1}) \\
z_{1,2} &= X_1 * W_{1,2}^{(1)} + X_2 * W_{2,2}^{(1)} + b_{2,2} \\
a_{1,2} &= f_1(z_{1,2}) \\
z_{1,3} &= X_1 * W_{1,3}^{(1)} + X_2 * W_{2,3}^{(1)} + b_{2,3} \\
a_{1,3} &= f_1(z_{1,3})
\end{aligned}
$$

The number of calculations will always be size of the input layer and the size of the next layer. When we've calculated the values for the hidden layers nodes, we will use them to find the value of the output layer.

$$
\begin{aligned}
z_{2,1} &= a_{1,1} * W_{1,1}^{(2)} + a_{1,2} * W_{2,1}^{(2)} + a_{1,3} * W_{3,1}^{(2)} + b_{3,1} \\
y &= f_2(z_{2,1})
\end{aligned}
$$

We now have a model containing layers filled up with values that have been activated, weighted and biased. The output on the other hand might not be what we want yet. This is because the first run through is made by chosen or random chosen variables for weights and biases with no reasoning behind them.

4.2 Backward propagation

To solve the bad output from the first run we will do backward propagation. Backward propagation works in the opposite way of forward, as it goes from the output layer towards the input layer.

It starts of by finding out how good the model has progressed this far by finding the cost function. The higher the cost function, the worse the model is, and the more we have to change the different variables. But how do we know what variables to change to effect the model in a positive way? For this we use the following function for each layer's weight's to see how much each weight affect the cost function

$$
\frac{\partial C_0}{\partial w_{jk}^{(L)}} = \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial C_0}{\partial a_j^{(L)}}
$$

Figure 3: How to see how much each weight($\partial w_{jk}^{(L)}$) affects the cost(∂C_0)

We will then use SGD on the variable to try make it move in a direction that lowers the cost function. This will be done for each and every weight and bias in every layer. When we've run through the backward propagation we should have a different set of weights and biases than the previous round and a lower cost function. The size of these changes is decided by the learning rate.

4.3 Regression vs Classification

In regression the object is to make a model that fits and predicts certain data or objects. These methods are based on mathematical problems of approximation. A classification model splits the problem into two or more classes. The model then predicts what class the input should belong to. The output is usually a predicted probability which can be interpreted as a belonging to a certain class.

4.4 Error rate

But how does one know how good the model is compared to other models? This is where the selection of error rate is important

To decide how well our method is working we use specific error functions for the layers. These function compares the calculated values with exact values. Our error function for the hidden layer L is

$$\delta_f^L = f'(z_j^L)\frac{\partial C}{\partial(a_j^L)}$$

for the error in the output layer where

$$f'(z_j^L) = (a_L * (1 - a_L))$$
$$\frac{\partial C}{\partial(a_j^L)} = (a_L - z)$$

where z is the exact results while a_L is the predicted results.

One thing to take into consideration is the way that the weights and biases might operate when given enough leisure. If we let the approximation go it's way the weights might become reach incredible high or low values as it compensates for a low or high value in the node. While this would work on the training set, it will not work on the test set as it will then be over fitted.

4.5 Activation functions

We chose Sigmoid, Rectified Linear Unit (RELU) and Leaky Reactified Linear Unit (Leaky RELU) as different activation functions on the hidden layers.

Using the sigmoid function as a activation function is a good way to divide the values between 0 and 1, as the most extreme values won't have to much weight on the results. The derivative of the sigmoid function can also be used, but instead this function limits the values between 0 and 0.4 and therefore treating extremes in both end the same way.

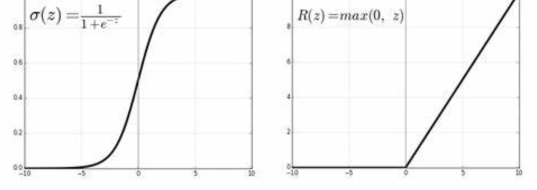


Figure 4: Sigmoid vs ReLU

The Rectifier Linear Unit (RELU) is one of the most common activation function in deep learning. The function returns 0 for any negative input, but for any positive input value x it returns that value back. Leaky RELU is a variant of the RELY where instead of returning 0 for negative values, it has a constant slope less than 1.

$$Leaky\ ReLU(z) = \begin{cases} \alpha(exp(z) - 1) : z < 0 \\ z \qquad\qquad\qquad : \text{else} \end{cases}$$

4.6 ETA and Regularization

In Gradient Descent the goal is to reach the minimum by searching. When we are searching through a data set we want to find the next data point that decrease most as possible. When the direction of the decrease is located we want to make some kind of steps in that direction. This is where the variable ETA (Learning Rate) is introduced. The ETA is a variable controlling the length of the next search step. As he model is ab out to reach a minimum there might be a good idea to change the learning rate to hit the minimum value.

Regularization is a way to avoid over fitting. It is a part of a more complex cost function where the regularization is a term added to the function. Usually the term are the L1 and L2. Due to the added term, the values in the weight matrices decrease and therefore also help to prevent over fitting.

5 Results

The first thing was to actually see if the model was improving for each time we trained the model

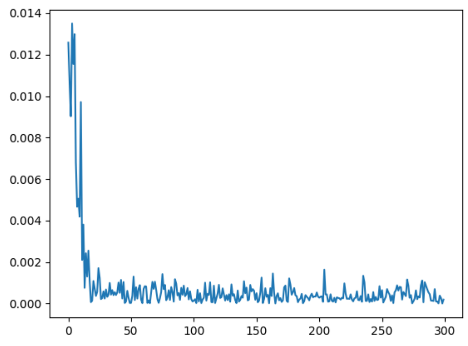


Figure 5: MSE of Franke Function map, with learning rate(eta) = 0.1. x-bar = epoch

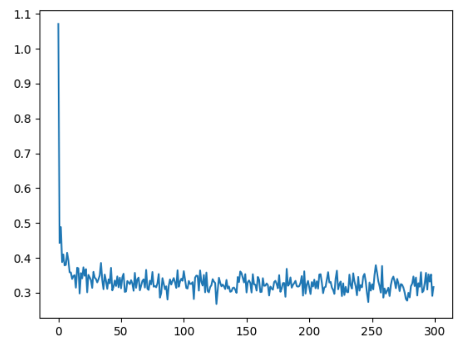


Figure 6: MSE of Franke Function map, with learning rate(eta) = 0.1. x-bar = epoch

These two figures showed exactly what we wanted to get, as it first started at a random spot in the grid, for then to go towards a local go global minimum. As both of them are from the same graph, we know that Figure 5, is stuck at a local minimum of a value around 0.3. We know this is just a local minimum as in Figure 4 we get even lower MSE, Figure 4 is most likely a global minimum. The graph also never rests at one value, which means that it tries to jump around to a better value, which is exactly what we want to see.

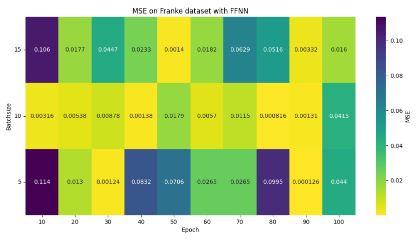


Figure 7: Shows how batch size and size of epoch affects the MSE

Figure 6 shows us how epoch and theoretical batch size affect the cost function MSE. When it comes to outcome of the figure we got what we wanted for the epoch, or times training the model on the training set, and a bit for the batch size.

Batch size affects the stability and progress of the model. The larger the batch size the less randomness should be introduced into each step, and it should therefore improve for each increase. The main reason for keeping it low was as mentioned that it is faster to compute. Therefore the cost values should become lower the higher the batch size. But when we take the randomness into consideration we could get a jump from a local minimum to a global minimum or another lower minimum, so it can get better results with low batch size.

The figure should get lower value for each increase in epoch, this all comes down to the fact that it only symbolises the number of times it trains on the training data. It therefore can't affect the result in any way than improving it or jumping back and forth between 2 values around the local minimum after a certain amount of epochs.

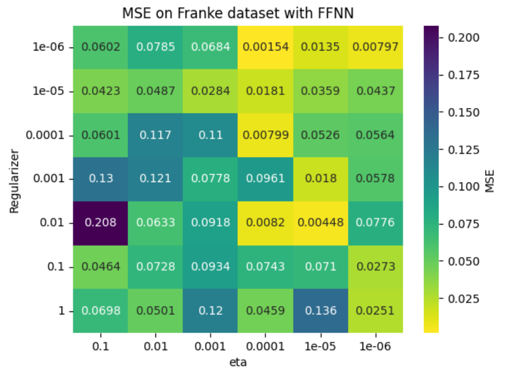


Figure 8: Regularization vs eta, sigmoid

The figure shows the results as expected with each variable, eta also called learning rate and regularization, also called penalty.

Regularization works as intended on the figure, as the lower the regularization, the less the weights get changed for each time, and therefore the less biased and over fitted the model becomes. So when we test the figure on test set, the results should be smaller when it's low. It shouldn't however change to much when we're training the test a huge amount of times.

Eta is also giving the expected results on the model, as it can give more precise values towards the end than the larger eta values can because of the step size.

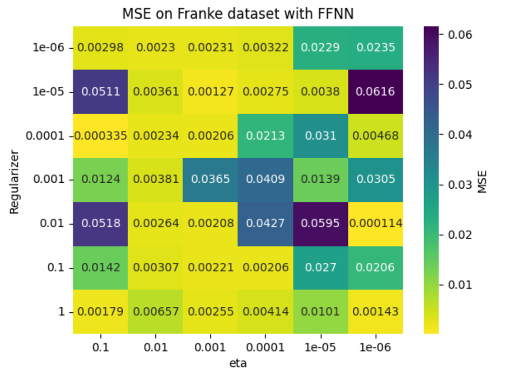


Figure 9: Regularization vs eta, sigmoid

6 Conclusions

Even though there is a gap in the report as a result of not being able to get to code to work for most of the project we've been having a lot of progress when it comes to the theoretical side of Feed forward neural network and gradient descent.

We've learned that as a result of big data, SGD has become more and more prevalent over OLS and GD because of how it works, and how when we combine it with FFNN we at the same time make up for the randomness SGD provides.

The error functions has also a way of affecting the way the model improves, while sigmoid gives a value between and not including 0 and 1 while ReLU gives values between 1 and 0. Sigmoid is better when it comes to values to keep until later as it's values are never 0 and 1 and can therefore be kept and used without breaking any code. It does however take longer time to compute, so the third option of Leaky ReLU which doesn't have the problem with losing values but at the same time it is quite quick is the best option.

The ETA and the Regularization are two variables who have both good and negative side when increasing or decreasing them. Therefore we've found out that it's best to find a middleway of value and try not to go to the extreme on either of them.

If we found the problem in our code i believe we could've found out much more as there is so much more to these algorithms, but at the same time i think we've learned a lot.