FYS-STK 4155 Project 2

Espen Lønes

November 20, 2021

In this project we expand on project 1 by using new machine learning methods. These include Stochastic Gradient Descent (SGD), feed forward neural network (FFNN) with back-propagation and logistic regression using the neural network. We will use these methods on data generated from the Franke function or the Wisconsin Breast cancer data set.

1 Theory

Gradient descent

The gradient of a function f for a given variable x tells us how much the function changes with changes in x. Usually the function is depended on multiple variables. It's gradient is therefore found by finding its partial derivatives.

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \cdots \right)$$

Gradient decent is an iterative method where given a starting point, usually random but may be purposely chosen. The method goes along the gradient until it reaches a local minimum in the search space (hopefully the global minimum). For us this search space is the chosen cost function, telling us how far from the correct solution our model is. Since our goal is to reach the minimum we move in the opposite direction of the gradient such that we move in the direction of steepest decent. This direction is found by calculating the gradient of the search space at the current point. We then move a given length in that direction to get our new point. This repeats until we get to the minimum where the gradient is zero. If we are at point w_i with costfunction $C(w)$ we get the iterative scheme.

$$w_{i+1} = w_i - \eta_k \nabla_w C(w_i)$$

Where η_k is the step length also called learning rate. The point w corresponds to a set of values we call weights.

As stated previously stated our goal is to reach the global minimum but depending on cost function, leaning rate and initial guess (w_0) we may (and for complex problems usually will) end up in a local minimum instead. A problem with the learning rate is that having a small learning rate makes the training slow as we need many points until we reach a minimum. On the other hand if it is to large we may overshoot the optimal solution. A popular method for mitigating this is by having a decreasing learning rate, then as we hopefully get close to the global minimum the

learning rate will decrease making it less likely we overshoot the solution.

Stochastic gradient descent

The calculation of the cost functions gradient may be computationally heavy. We may instead use a version of gradient descent called stochastic gradient descent to speed this process up. SGD works by dividing the points into groups called mini-batches. We then choose a random batch and compute its gradient. If the sample size is large enough the average of these gradients will approximate the gradient calculated for all data points. The flow of the SGD method is therefore as follows. The SDG picks a random mini-batch of training points and uses these for training. Then the next mini-batch is chosen until all batches are used. This is called one epoch. We then use this gradient as in gradient descent and may also then start another epoch. The fact that this method only approximates the gradients means that some randomness is introduced in the gradient. This has the added benefit of possibly getting us out of a local minima.

As a step to further increase the algorithms speed, SGD is often used with a momentum factor. The momentum increases the acceleration of the gradient giving a higher convergence rate. Having a momentum $0 \leq \gamma \leq 1$ we get this scheme for our weights.

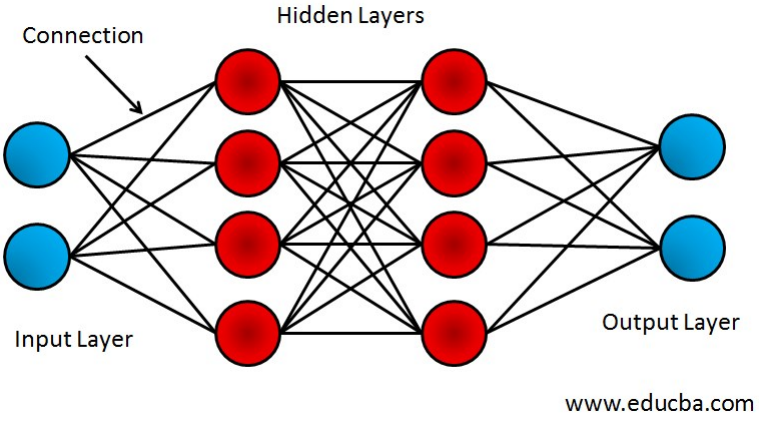
$$v_{i+1} = \gamma v_i + \eta_k \nabla C(w_i)$$

$$w_{i+1} = w_i - v_{i+1}$$

(Usualy having $v_{-1} = 0$)
High γ values gives large acceleration. While $\gamma = 0$ gives ordinary SGD.

Neural networks

A neural network consists of layers, one input layer, one output layer and a any number of hidden layers (may have no hidden layers). Each layer has a set of neurons, for the input layer these are the input neurons where the number of neurons are determined by our problem and our representation of it. The output layer has the output neurons. There is no need to have the same amount of input a s output neurons.



Feed forward neural network (FFNN)

A neural network where the output from one layer is used as input to the next. This implies that the network does not contain loops. These are the simplest and easiest forms of neural network and use algorithms like back propagation for learning.

Activation functions

In neural networks the sum of the weighted inputs to a node, is used as input to that nodes activation function. Activation functions are divided in two main groups, linear and non-linear. The linear function are lines/linear and are not bounded by any range, the identity function is such a function $\sigma(z) = z$. The non-linear functions are more often used as they usually make the model better at generalizing and adapting to different data.

Sigmoid functions are a set of s-shaped functions often used as activation functions in neural networks. One specific sigmoid that has been used frequently in NNs is the logistic function, given by.

$$\sigma(z) = \frac{1}{1 + e^{-z}} = \frac{e^z}{e^z + 1}$$

With the derivative

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

Its advantage lies in that it's values fall between 0 and 1. This is very useful when the output is a probability. Unfortunately the logistic function can halt the NN as the derivative is often very close to zero. We now often use functions like ReLU and leaky ReLU instead.

The ReLU and (rectified linear unit) function is variant of the sigmoid.

$$\sigma(z) = \begin{cases} 0 & \text{if } z \leq 0 \\ z & \text{if } z > 0 \end{cases} = max\{0, z\}$$

Derivative

$$\sigma'(z) = \begin{cases} 0 & \text{if } z \leq 0 \\ 1 & \text{if } z > 0 \end{cases}$$

We see that for ReLU increasing the weights does not saturate and stop the learning. But it does it the weights are negative. ReLU is often used in the hidden layers.

Another variation is the Leaky ReLU that does not saturate that instead of giving zero for negative input it multiples with 0.01.

$$\sigma(z) = \begin{cases} 0.01z & \text{if } z \leq 0 \\ z & \text{if } z > 0 \end{cases}$$

$$\sigma'(z) = \begin{cases} 0.01 & \text{if } z \leq 0 \\ 1 & \text{if } z > 0 \end{cases}$$

As the gradient of leaky ReLU newer goes to zero the neurons never stop learning. Making this activation function very robust.

Cost functions

Cost functions tell us how right or wrong a prediction is. It uses the predicted value and a true value in order to quantify the quality of our current model. In neural networks this value is used to update the weights and biases. For liner regression we use the mean squared error (MSE), while for logistic regression cross entropy is used.

The MSE cost function.

$$C(w) = (Xw - z)^T(Xw - z) + \lambda w^T w$$

Where X is the design matrix, z is the true labels, w is the weights and λ is the regularization parameter.

The MSE takes the averages of the squares of the errors. This works well when fitting to a line. Since we do gradient descent we also needs its derivative.

$$\nabla C(w) = 2X^T(X^T w - z) + 2\lambda w$$

When categorizing on the cancer data we look at probabilities. As we are no longer looking at points, and also wanting the probabilities to lie in $[0, 1]$ with a total sum of 1. MSE is not a god choice of cost function. Instead we use cross entropy.

$$C(\hat{w}) = -logP(D|\hat{w})$$

Logistic regression

Logistic regression is a special use case of neural networks often used when doing binary classification, like we do on the cancer data. It may also be used with multiple categories. It uses the logistic function as activation function and cross entropy as cost function. It also has no hidden layers. This method does not suffer from the same problems that general neural networks suffer from. Like to small or to large gradients. But its simplicity also limits its usefulness. Like leaning the fine details of a dataset, especially feature rich data.

Backpropagation

Backpropagation is an algorithm for updating the weights and biases of a neural network. It computes the gradient of the cost function in respect to the weights and biases. The aim of backpropagation is to calculate the partial derivatives.
$\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$
This is done with the chain rule. Where the cost function depends on the activation function a, which in turn depends on the weighted sum z, which again depends on the weights and biases. Giving us.

$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial a}\frac{\partial C}{\partial z}\frac{\partial C}{\partial w}$$

$$\frac{\partial C}{\partial b} = \frac{\partial C}{\partial a}\frac{\partial C}{\partial z}\frac{\partial C}{\partial b}$$

Doing this individually for all weights and biases is very slow. Instead we calculate the gradient for each layer in reverse order. By doing this we avoid unnecessary calculations and duplicate values. It is also easier to see how changing the weights and biases change the output when starting on the last layer. The gradient of the weighted input(s) to a given hidden layer is δ^l from the back to the front layer (often called the error). We also use k to denote the k-th node in the $(l-1)$-th layer. And j for the j-th node in the l-t layer.

Backpropagation consists of a set of four equations. The first being for the error δ^l in the output layer.

$$\delta_k^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L)$$

Thereafter the error for the hidden layers $L-1, L-2, \cdots, 2$ are computed recursively.

$$\delta_j^l = \sum_k \delta_j^{l+1} w_j^{l+1} \sigma'(z_j^l)$$

These errors are then used to calculate the cost functions partial derivatives.

$$\frac{\partial C}{\partial w_{jk}^l} = \delta_j^l a_k^{l-1}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l$$

Lastly the partial derivatives are used to update the weights and biases for layers $L-1, L-2, \cdots, 2$ using gradient decent.

$$w_{jk}^l \leftarrow w_{jk}^l - \eta \frac{\partial C}{\partial w_{jk}^l} = w_{jk}^l - \eta \delta_j^l a_k^{l-1}$$

$$b_j^l \leftarrow b_j^l - \eta \frac{\partial C}{\partial b_j^l} = b_j^l - \eta \delta_j^l$$

In order to avoid overfitting a regularization parameter λ may be added. This penalises the weight matrices. A common regularization parameter is the L2. The penalty is added to the cost function. The regularization forces the weights towards zero but not all the way to zero. This way a simpler and more generalizable solution with the same error will likely be selected. Thus reducing overfitting. The cost function with L2 is defined by.

$$C(\theta) = L(\theta) + \lambda ||w||_2^2$$

With gradient.

$$\nabla C(\theta) = \nabla L(\theta) + 2\lambda w$$

2 Method

SGD

Below is the pseudo code for the SGD. When making the mini-batches we have an array containing the indexes of the data. We then shuffle this and use it to index the data making the data shuffled. Then the data is spilt into the given number of mini-batches.

```
1. Initialize the weights, learning rate and momentum
2. v = 0
3. for epoch in epochs do
4.     Randomly divide data into given number of mini-batches
5.     for mini-batch in mini-batches do
6.         v <- gamma*v + eta*gratient_cost_function
7.         w <- w - v
8.    end for
9. end for
```

Neural Network

For the neural network two classes are used. A neural network is an instance of the NN class and a layers are instances of the Layer class. The layer elements are appended to a list as they created (front to back). This makes it easier to change the number of hidden layers, their activation functions and the number of nodes in a given layer.

Backpropagation

When doing the backpropagation it is easiest to use matrix operations. Rewriting the backpropagation equations for element-wise multiplication using the Hadamard product ($\circ$) we get.

$$\delta^L = \nabla_a C \circ \sigma'(z^L)$$

$$\delta^l = \delta^{l+1}(w^{l+1})^T \circ \sigma'(z^L)$$

$$\frac{\partial C}{\partial w^l_{jk}} = (a^{l-1}_k)^T \delta^l_j$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j$$

Pseudo code for the backpropagation.

1. `Initialize network and do feed forward.`
2. `Compute error of output layer`
3. $\delta^L = \nabla_a C \circ \sigma'(z^L)$
4. `for l in L-1 : 1 do`
5. $\delta^l = \delta^{l+1}(w^{l+1})^T \circ \sigma'(z^L)$
6. $w^l \leftarrow w^l - \eta(a^{l-1})^T \delta^l - 2\eta\lambda w^l$
7. $b^l \leftarrow b^l - \eta\delta^l$
8. `end for`
9. $\delta^0 = \delta^2(w^1)^T \circ \sigma(z^0)$
10. $w^0 \leftarrow w^0 - \eta(X^T\delta^0) - 2\eta\lambda w^0$
11. $b^0 \leftarrow b^0 - \eta\delta^0$

Dataset

Accuracy

To know if out trained model is good or not we need a measure for how goof the model is. The metric we use is the accuracy score, which counts the number of correct classifications in relation to the number of observations to classify.

$$Accuracy = \frac{\sum_{i=0}^{n} I(t_i = y_i)}{n}$$

Where I is the indicator function for binary classification.

3 Results / Discussion

SGD for OLS and Ridge

After running tests to determine best hyper-parameters, these where found to be the best for both OLS and Ridge:
$\eta = 0.01$
$\gamma = 0.7$
$number\ of\ epochs = 1500$
$Batch\ size = 8$
And ridges regularization parameter:
$\lambda = 0.01$

(For the testing of hyper-parameters see 'run_sgd.py')
Using these values on the test data we got these values:

```
OLS Best:
MSE: 0.1477030399345097
Best Ridge:
MSE: 0.18235018557921942
```

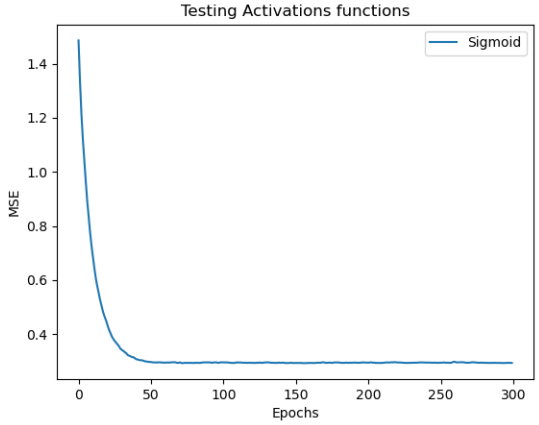
This shows that OLS outperforms Ridge in this task, meaning the actual best λ value should be 0.

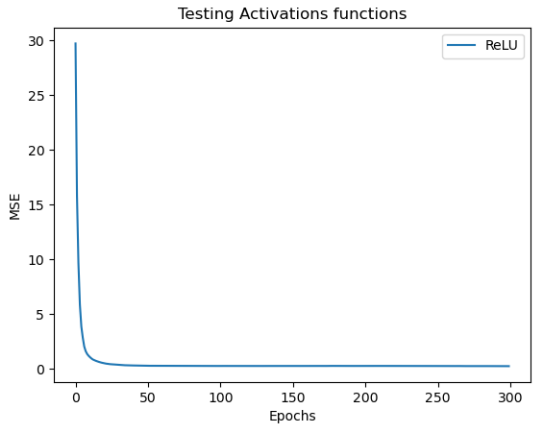
Neural network on Franke data

(see 'run_NN.py' for source code)

Tested with sigmoid, ReLU and LeakyReLU in the hidden layers, with Identity at the output layer.
Had problems when Using LeakyLeLU in the hidden layers so i have no data for this.

First the different activation functions for used in the hidden layers where tested with a variety of hidden layer combinations.
First, one hidden layer with 50 nodes:

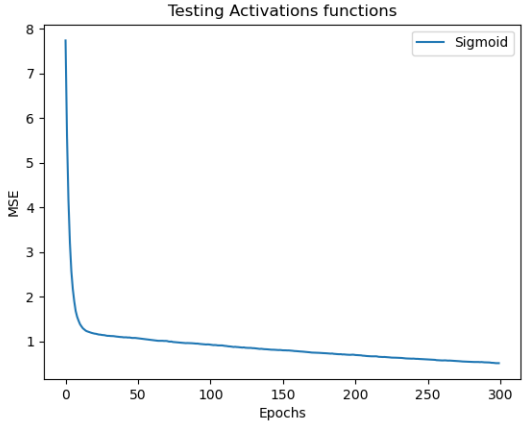


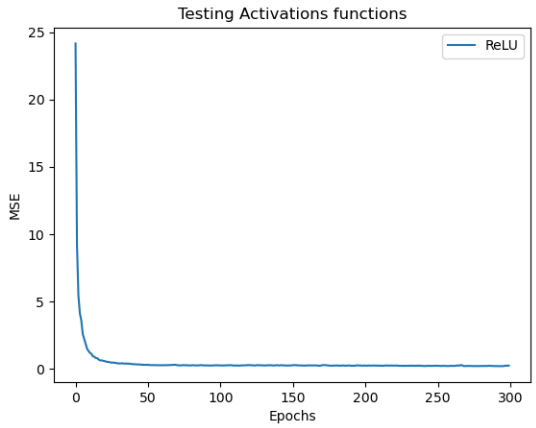



```
One layer with 50 nodes
MSE OLS:  0.09455542520459001
MSE before training (Sigmoid) :  1.6971725263855568
MSE train (Sigmoid): 0.3977370923679221
MSE test (Sigmoid): 0.2932577278959031
MSE before training (ReLU) :  60.94677831588465
MSE train (ReLU): 0.31137113354947654
MSE test (ReLU): 0.2559470451224469
```

We see that ReLU preforms slightly better than Sigmoid, but both much worse than OLS.

Then for two layers with 20 nodes each.

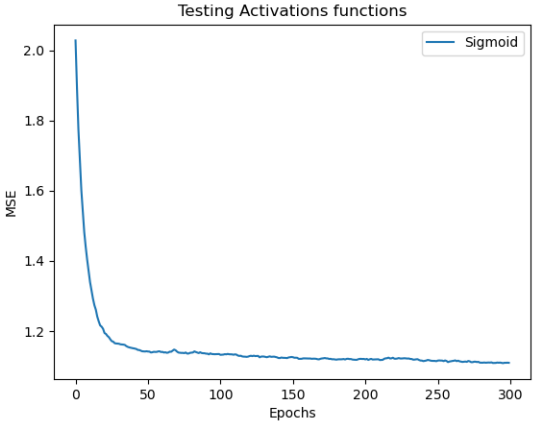


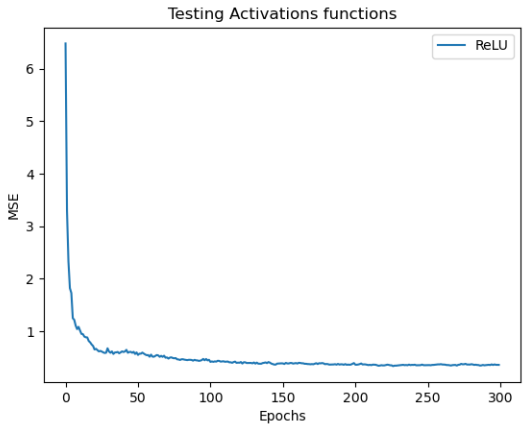



```
Two layers with 20 nodes
MSE OLS:  0.09455542520459001
MSE before training (Sigmoid) :  10.84097261409264
MSE train (Sigmoid): 0.560181226948035
MSE test (Sigmoid): 0.5072626231608858
MSE before training (ReLU) :  1832.6634227775553
MSE train (ReLU): 0.30307922577730323
MSE test (ReLU): 0.23704229828658846
```

OLS and ReLU preforms about the same as for the previous case. But Sigmoid preforms much worse.

Lastly, three layers with 10 nodes each.



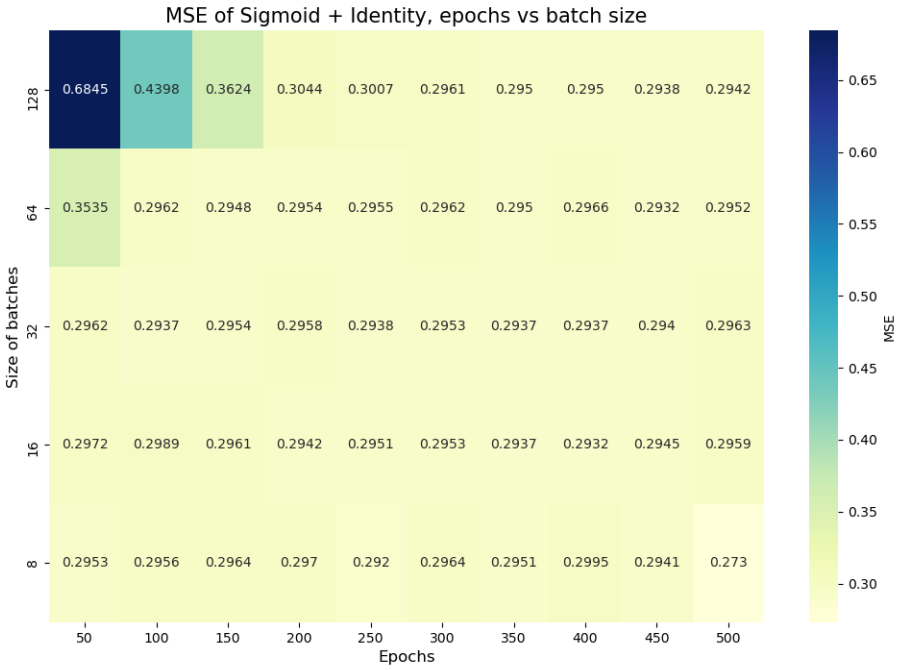


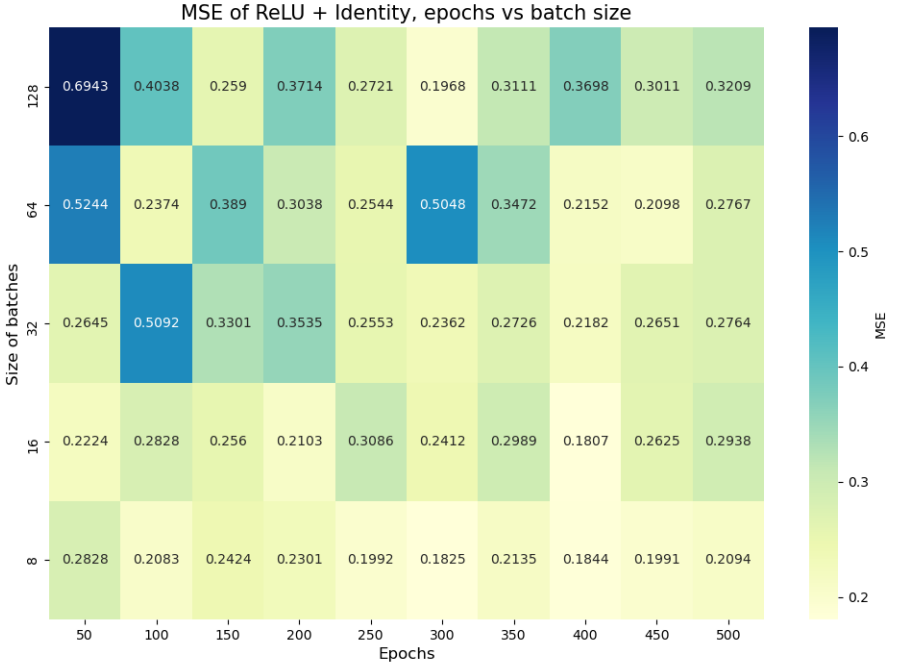

```
three layers with 20 nodes
MSE OLS:  0.09455542520459001
MSE before training (Sigmoid) :  2.1879519921608233
MSE train (Sigmoid): 1.0047682434994045
MSE test (Sigmoid): 1.1098615334781325
MSE before training (ReLU) :  53.92304399545829
MSE train (ReLU): 0.2885547102020297
MSE test (ReLU): 0.3608360554583901
```

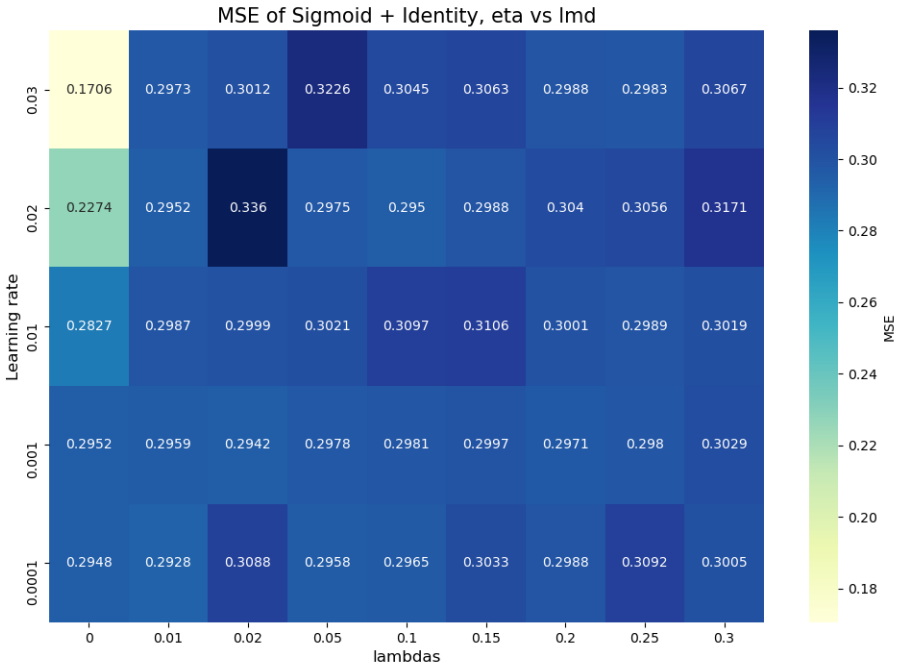
Now ReLU is a little worse, Sigmoid even worse than last time again and OLS still the same.

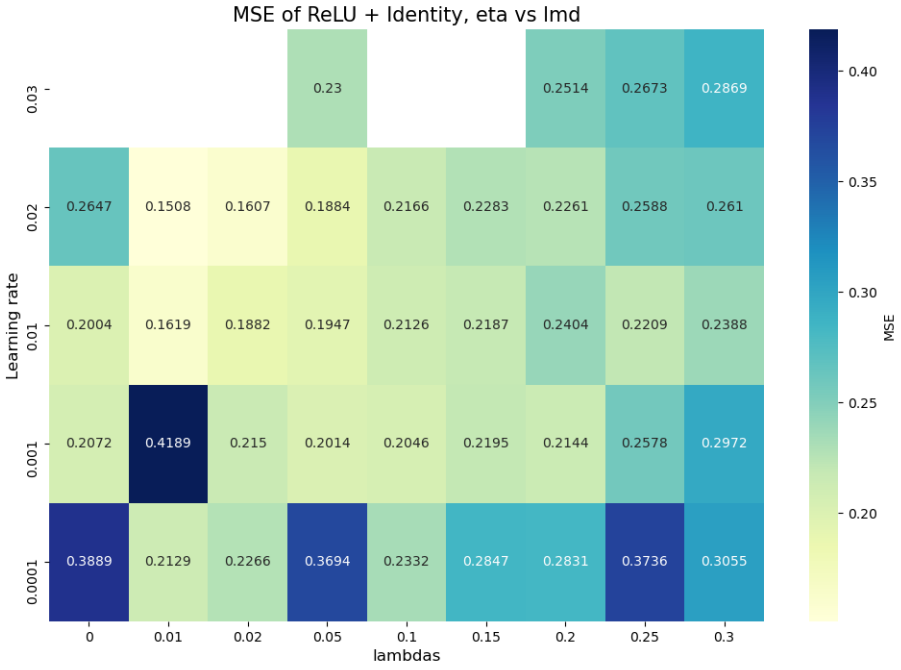
From these observations we see that if a neural network is to be used. Of the tested variants, one layer with 50 nodes was best.

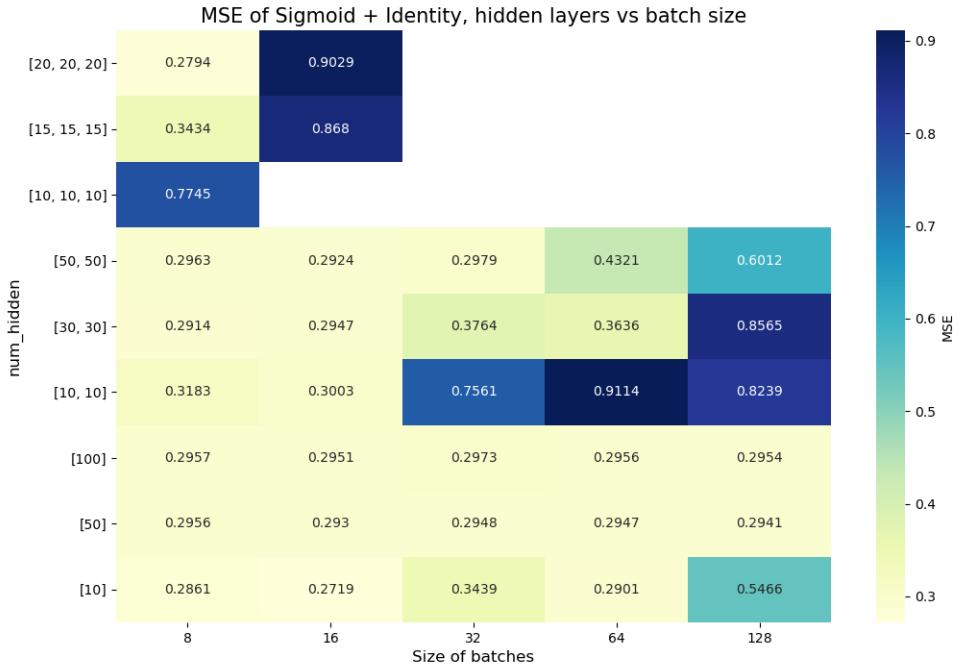
These runs where preformed with, 32 mini-batches, $\eta = 0.001$ and up to 300 epochs.
These values where chosen on grounds of the flowing results.

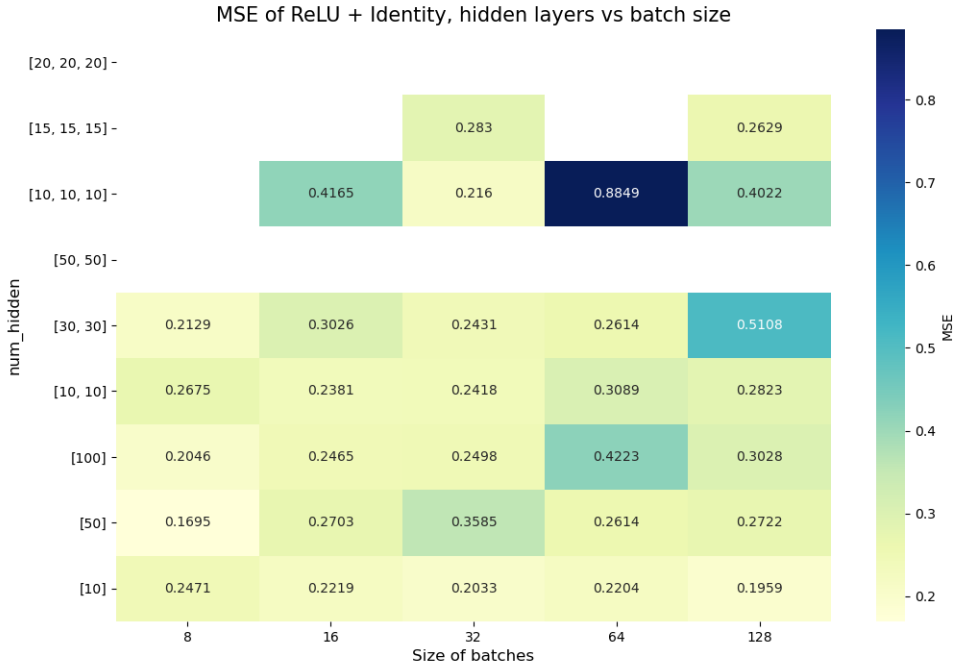

MSE of Sigmoid + Identity, epochs vs batch size
Size of batches
128
64
32
16
8
0.6845 0.4398 0.3624 0.3044 0.3007 0.2961 0.295 0.295 0.2938 0.2942
0.3535 0.2962 0.2948 0.2954 0.2955 0.2962 0.295 0.2966 0.2932 0.2952
0.2962 0.2937 0.2954 0.2958 0.2938 0.2953 0.2937 0.2937 0.294 0.2963
0.2972 0.2989 0.2961 0.2942 0.2951 0.2953 0.2937 0.2932 0.2945 0.2959
0.2953 0.2956 0.2964 0.297 0.292 0.2964 0.2951 0.2995 0.2941 0.273
50 100 150 200 250 300 350 400 450 500
Epochs
MSE
0.65 0.60 0.55 0.50 0.45 0.40 0.35 0.30


MSE of ReLU + Identity, epochs vs batch size
Size of batches
128
64
32
16
8
0.6943 0.4038 0.259 0.3714 0.2721 0.1968 0.3111 0.3698 0.3011 0.3209
0.5244 0.2374 0.389 0.3038 0.2544 0.5048 0.3472 0.2152 0.2098 0.2767
0.2645 0.5092 0.3301 0.3535 0.2553 0.2362 0.2726 0.2182 0.2651 0.2764
0.2224 0.2828 0.256 0.2103 0.3086 0.2412 0.2989 0.1807 0.2625 0.2938
0.2828 0.2083 0.2424 0.2301 0.1992 0.1825 0.2135 0.1844 0.1991 0.2094
50 100 150 200 250 300 350 400 450 500
Epochs
MSE
0.6 0.5 0.4 0.3 0.2


MSE of Sigmoid + Identity, eta vs lmd
Learning rate
0.03
0.02
0.01
0.001
0.0001
0.1706 0.2973 0.3012 0.3226 0.3045 0.3063 0.2988 0.2983 0.3067
0.2274 0.2952 0.336 0.2975 0.295 0.2988 0.304 0.3056 0.3171
0.2827 0.2987 0.2999 0.3021 0.3097 0.3106 0.3001 0.2989 0.3019
0.2952 0.2959 0.2942 0.2978 0.2981 0.2997 0.2971 0.298 0.3029
0.2948 0.2928 0.3088 0.2958 0.2965 0.3033 0.2988 0.3092 0.3005
0 0.01 0.02 0.05 0.1 0.15 0.2 0.25 0.3
lambdas
MSE
0.32 0.30 0.28 0.26 0.24 0.22 0.20 0.18


MSE of ReLU + Identity, eta vs lmd
Learning rate
0.03
0.02
0.01
0.001
0.0001
0.23 0.2514 0.2673 0.2869
0.2647 0.1508 0.1607 0.1884 0.2166 0.2283 0.2261 0.2588 0.261
0.2004 0.1619 0.1882 0.1947 0.2126 0.2187 0.2404 0.2209 0.2388
0.2072 0.4189 0.215 0.2014 0.2046 0.2195 0.2144 0.2578 0.2972
0.3889 0.2129 0.2266 0.3694 0.2332 0.2847 0.2831 0.3736 0.3055
0 0.01 0.02 0.05 0.1 0.15 0.2 0.25 0.3
lambdas
MSE
0.40 0.35 0.30 0.25 0.20





I did unfortunately not have the time to implement the use of the cancer data. Though much of the code is done (make_dc_data_ready_for_NN adn rain_NN_classifier in analasis_functions.py.)