Implementing Optimization Techniques on Neural Network Backpropagation Learning

- Aditya Kumar

I. Introduction:

This final semester project had a three-fold agenda:

a. To design 100% custom codes using NumPy for neural network backpropagation training that will be able to generalize training to any architecture (# of Layers, Activation functions).
b. To be able to successfully implement learned non-linear optimization algorithms (Stochastic/Batch Gradient Descent, Variable Learning rate Gradient Descent, Moment Gradient Descent, Conjugate Gradient Descent, and Levenberg-Marquardt Algorithm) while studying heuristic variations and numerical optimization variations of backpropagation performance learning for neural networks through detailed pseudo-codes present in the Neural Network Design course textbook.
c. To effectively perform a comparison between the computational efficiency and performance metric (SSE for classification and MSE for regression) of the above-mentioned algorithms across pre-determined criteria such as:
    i. Varying learning rates
    ii. Varying number of input features
    iii. Varying number of layers
    iv. Varying activation functions

A brief outline of my shared work is:

a. Designed a simple 1-S-1 network with Stochastic Gradient Descent training. This code was used as a starting point for the entire project. Therefore, this can be found in the folder: Preliminary Code.
b. Implemented batch training / mini-batch training for 1-S-1 network.
c. Generalized the 1-S-1 neural network training code with Stochastic / batch training to the user-defined architecture of format R-S1-S2-…-SM with custom activation functions in layers.
d. Added functionality of user-defined initialized weights and biases of the neural network during debugging.
e. Implementation of the Levenberg-Marquardt Algorithm for neural network backpropagation learning.
f. Final integration of all developed optimizers into a single train function.
g. Attempted modeling of datasets with custom neural network codes.

II. Individual Work Description:

Background Information

a. Implemented 1-S-1 neural network backpropagation training with the following pseudo-code:

Forward propagation:

$$\mathbf{a}^0 = \mathbf{p},$$

$$\mathbf{a}^{m+1} = \mathbf{f}^{m+1}(\mathbf{W}^{m+1}\mathbf{a}^m + \mathbf{b}^{m+1}) \text{ for } m = 0, 1, \dots, M-1,$$

$$\mathbf{a} = \mathbf{a}^M.$$

Backpropagating sensitivity:

$$\mathbf{s}^M = -2\dot{\mathbf{F}}^M(\mathbf{n}^M)(\mathbf{t} - \mathbf{a}),$$

$$\mathbf{s}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T\mathbf{s}^{m+1}, \text{ for } m = M-1, \dots, 2, 1.$$

Weight updates:

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \alpha\mathbf{s}^m(\mathbf{a}^{m-1})^T,$$

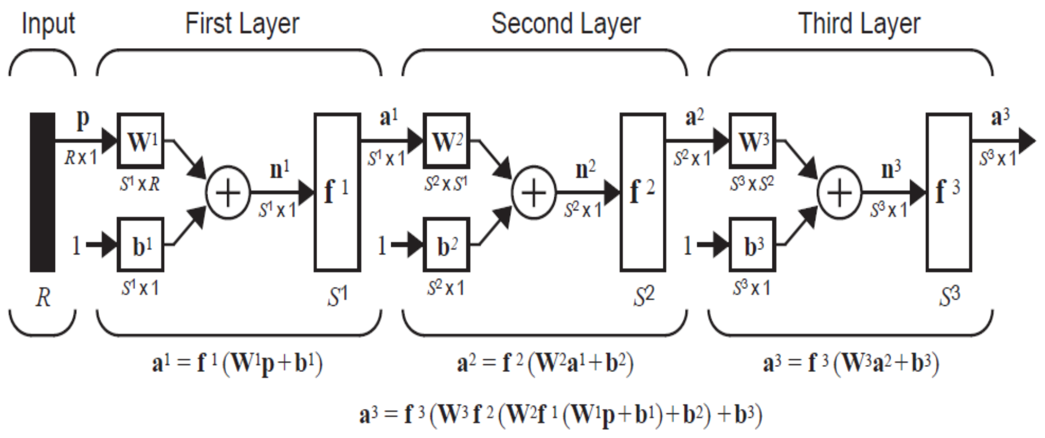
$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \alpha\mathbf{s}^m.$$

b. Implemented batch-training/mini-batch with enhancement in the above code through the following pseudo-code:

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \frac{\alpha}{Q}\sum_{q=1}^{Q}\mathbf{s}_q^m(\mathbf{a}_q^{m-1})^T,$$

$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \frac{\alpha}{Q}\sum_{q=1}^{Q}\mathbf{s}_q^m.$$

c. Generalized the 1-S-1 architecture to user-defined architecture with the help of the following diagram:



Along with this diagram, the pseudo-code mentioned above played an equally important role in writing the final code.

d. Levenberg-Marquardt algorithm implementation: All the necessary equations and diagrams that helped me develop this algorithm are provided below:

Gradient form:

$$\nabla F(\mathbf{x}) = 2\mathbf{J}^T(\mathbf{x})\mathbf{v}(\mathbf{x}),$$

Calculation of Jacobian dimensions:

$$N = Q \times S^M \text{ and } n = S^1(R+1) + S^2(S^1+1) + \cdots + S^M(S^{M-1}+1).$$

Key theoretical Jacobian structure (helped in extracting the right number of sensitivities in a single row based on the number of columns in the weight matrix):

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial e_{1,1}}{\partial w^1_{1,1}} & \frac{\partial e_{1,1}}{\partial w^1_{1,2}} & \cdots & \frac{\partial e_{1,1}}{\partial w^1_{S^1,R}} & \frac{\partial e_{1,1}}{\partial b^1_1} & \cdots \\ \frac{\partial e_{2,1}}{\partial w^1_{1,1}} & \frac{\partial e_{2,1}}{\partial w^1_{1,2}} & \cdots & \frac{\partial e_{2,1}}{\partial w^1_{S^1,R}} & \frac{\partial e_{2,1}}{\partial b^1_1} & \cdots \\ \vdots & \vdots & & \vdots & \vdots & \\ \frac{\partial e_{S^M,1}}{\partial w^1_{1,1}} & \frac{\partial e_{S^M,1}}{\partial w^1_{1,2}} & \cdots & \frac{\partial e_{S^M,1}}{\partial w^1_{S^1,R}} & \frac{\partial e_{S^M,1}}{\partial b^1_1} & \cdots \\ \frac{\partial e_{1,2}}{\partial w^1_{1,1}} & \frac{\partial e_{1,2}}{\partial w^1_{1,2}} & \cdots & \frac{\partial e_{1,2}}{\partial w^1_{S^1,R}} & \frac{\partial e_{1,2}}{\partial b^1_1} & \cdots \\ \vdots & \vdots & & \vdots & \vdots & \end{bmatrix}.$$

Key practical Jacobian elements creation equations:

$$[\mathbf{J}]_{h,l} = \frac{\partial v_h}{\partial x_l} = \frac{\partial e_{k,q}}{\partial w^m_{i,j}} = \frac{\partial e_{k,q}}{\partial n^m_{i,q}} \times \frac{\partial n^m_{i,q}}{\partial w^m_{i,j}} = \tilde{s}^m_{i,h} \times \frac{\partial n^m_{i,q}}{\partial w^m_{i,j}} = \tilde{s}^m_{i,h} \times a^{m-1}_{j,q}$$

$$[\mathbf{J}]_{h,l} = \frac{\partial v_h}{\partial x_l} = \frac{\partial e_{k,q}}{\partial b^m_i} = \frac{\partial e_{k,q}}{\partial n^m_{i,q}} \times \frac{\partial n^m_{i,q}}{\partial b^m_i} = \tilde{s}^m_{i,h} \times \frac{\partial n^m_{i,q}}{\partial b^m_i} = \tilde{s}^m_{i,h}.$$

Sensitivity initialization equation (differ from SGD):

$$\tilde{\mathbf{S}}^M_q = -\dot{\mathbf{F}}^M(\mathbf{n}^M_q),$$

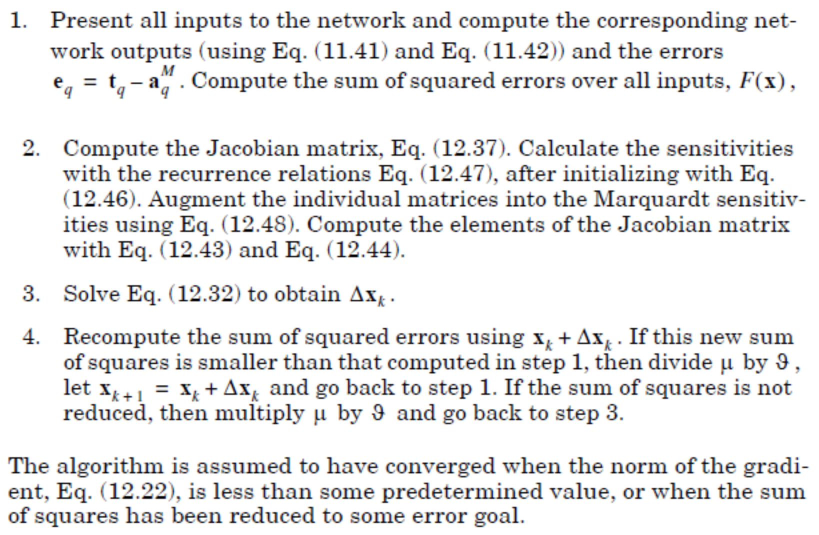
Sensitivity propagation equation (same as SGD):

$$\tilde{\mathbf{S}}^m_q = \dot{\mathbf{F}}^m(\mathbf{n}^m_q)(\mathbf{W}^{m+1})^T\tilde{\mathbf{S}}^{m+1}_q.$$

Final sensitivity creation through augmentation:

$$\tilde{\mathbf{S}}^m = \left[\tilde{\mathbf{S}}^m_1 \middle| \tilde{\mathbf{S}}^m_2 \middle| \ldots \middle| \tilde{\mathbf{S}}^m_Q\right].$$

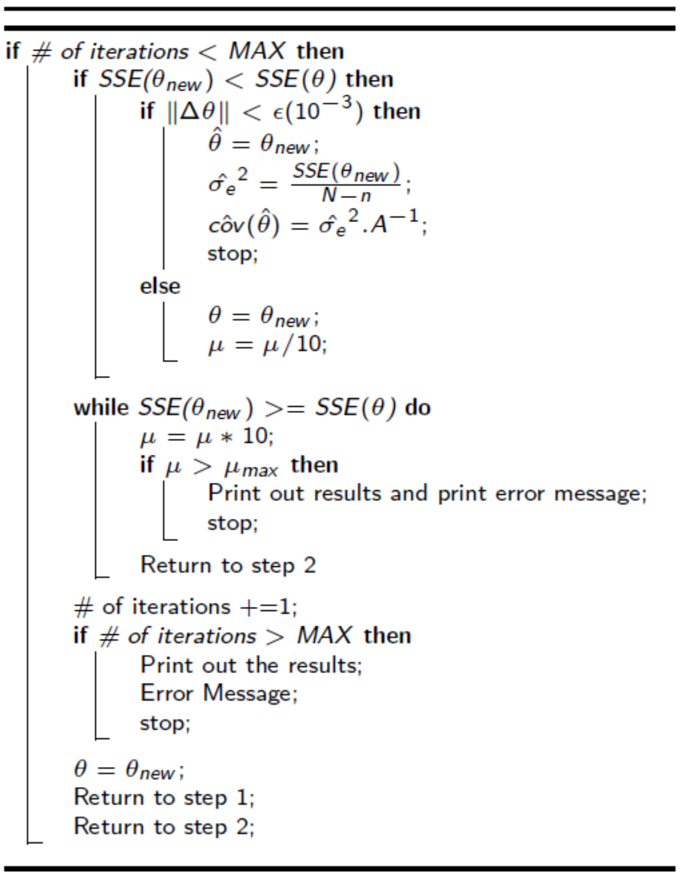
Theoretical pseudo-code used for reference:



1. Present all inputs to the network and compute the corresponding network outputs (using Eq. (11.41) and Eq. (11.42)) and the errors $\mathbf{e}_q = \mathbf{t}_q - \mathbf{a}_q^M$. Compute the sum of squared errors over all inputs, $F(\mathbf{x})$,
2. Compute the Jacobian matrix, Eq. (12.37). Calculate the sensitivities with the recurrence relations Eq. (12.47), after initializing with Eq. (12.46). Augment the individual matrices into the Marquardt sensitivities using Eq. (12.48). Compute the elements of the Jacobian matrix with Eq. (12.43) and Eq. (12.44).
3. Solve Eq. (12.32) to obtain $\Delta\mathbf{x}_k$.
4. Recompute the sum of squared errors using $\mathbf{x}_k + \Delta\mathbf{x}_k$. If this new sum of squares is smaller than that computed in step 1, then divide μ by ϑ, let $\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta\mathbf{x}_k$ and go back to step 1. If the sum of squares is not reduced, then multiply μ by ϑ and go back to step 3.

The algorithm is assumed to have converged when the norm of the gradient, Eq. (12.22), is less than some predetermined value, or when the sum of squares has been reduced to some error goal.

Actual implemented pseudo-code for entire algorithm:



```
if $\#$ of iterations $< MAX$ then
    if $SSE(\theta_{new}) < SSE(\theta)$ then
        if $\|\Delta\theta\| < \epsilon(10^{-3})$ then
            $\hat{\theta} = \theta_{new}$;
            $\hat{\sigma_e}^2 = \frac{SSE(\theta_{new})}{N-n}$;
            $\hat{cov}(\hat{\theta}) = \hat{\sigma_e}^2.A^{-1}$;
            stop;
        else
            $\theta = \theta_{new}$;
            $\mu = \mu/10$;

    while $SSE(\theta_{new}) >= SSE(\theta)$ do
        $\mu = \mu * 10$;
        if $\mu > \mu_{max}$ then
            Print out results and print error message;
            stop;
        Return to step 2

    $\#$ of iterations $+=1$;
    if $\#$ of iterations $> MAX$ then
        Print out the results;
        Error Message;
        stop;

    $\theta = \theta_{new}$;
    Return to step 1;
    Return to step 2;
```

Jacobian matrix referred from Problem12.5 for Jacobian creation check:

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} -4 & -4 & -1 & -1 \\ -16 & -8 & -4 & -1 \end{bmatrix}.$$

<u>Detailed contribution explanation</u>

a. A simple 1-S-1 network architecture training with stochastic gradient descent was designed by me as part of Homework 6. This implementation was fairly easy to perform as the homework had limited the scope of implementation to a fixed number of layers and fixed activation function of sigmoid and linear in hidden and output layers respectively, with the only variable being the number of neurons in the hidden layer. The key idea of this homework was to assess the performance of neural network models in the task of function approximation by varying the number of hidden neurons. More complex functions (high inflection points) required a greater number of sigmoid neurons in the hidden layer to be ANDed together.

b. This homework led to the motivation to implement something innovative as part of the final term project. Having created a 1-S-1 network, I asked myself why not just try to generalize to a custom neural network architecture with different activation function capabilities. To make this a substantive project, we decided to implement the above-listed optimization algorithms from scratch using only NumPy in an attempt to create a comparison between them in terms of performance and computation efficiency. After all, we had the pseudo-codes readily available.

c. The next thing I did was to implement a batch training / mini-batch training capability on top of the stochastic gradient descent on the 1-S-1 network. This was done prior to generalizing the network to custom architecture since replicating this to the general network would be easy once the logic was in place. The major component in this implementation was storing the gradients (in my case I decided to directly store the sum of gradients until the batch size was covered) as each input was presented. The weights and biases were then updated once the defined batch was covered after normalizing the final summed-up gradient with the batch size. This implementation was fairly straightforward as well. A couple of ambiguities I faced was whether an epoch was still considered as the entire train size, or whether it meant a batch size now (in the case of mini-batches). This conceptual question was cleared up after reading papers online. The entire trainset is considered an epoch, therefore while calculating the epoch error, it had to be after the entire trainset had passed. Observations made in the batch implementation will be recorded in the Results section.

d. The next and most crucial component of the project was to successfully generalize the neural network backpropagation training to a custom number of layers and custom activation functions from my end as the entire project was dependent on my implementations. The biggest initial challenge was to dynamically define class attribute declarations based on user input, since a custom number of layers meant a custom number of weight matrices which are the essence of neural networks. This is where I came across the functions, *setattr* and *getattr* which helped me implement this dynamic capability. Additionally, to be able to access the weight matrices I had them stored in a list object.
   The next immediate task I focused on was to create the feedforward function that performed the forward pass (called *prediction* in my code). This was given priority since if I figured out how to traverse these dynamic weights and biases in the forward direction, I could do the same while backpropagating the sensitivities. I realized I also had to preserve the state of each layer net input/output and decided to employ dictionaries to do this. As a matter of fact, most neural network implementations found online utilize dictionaries for state/architecture preservation as I found out later.
   With a couple of other tweaks and workarounds for epoch error calculation, sensitivity propagation, and weight updates, I was successfully able to create the network without syntax/shape errors. The Results section contains additional observations made here.

e. Finally came the implementation of the LM algorithm that I was tasked with. The first thing I did was read the text thoroughly and understand the nitty-gritty of the algorithm. The pseudo-code from the textbook was referenced to form the structure of the code. An observation was the lack of learning rate now, instead, we have the Hessian matrix. So, we didn’t need the learning rate as an input. I found the text to be quite confusing in terms of Marquardt’s sensitivity propagation. It is presented in a slightly different manner, by treating the error for every input in its component form comprising multiple elements (dependent on the output number of neurons) as opposed to a single vector representation. It took me a while to understand that Marquardt’s sensitivity is propagated in a similar manner as SGD, only initialization is different this time. After clearing this confusion, I also noticed the sensitivity augmentation. This led me to conclude that now I would need to store the sensitivity and output states for every input in a bigger dictionary set. The toughest part of figuring out the algorithm was the Jacobian matrix calculation. The text is extremely confusing with its multiple-letter notations. After successfully forming the total state representation of sensitivities (post-augmentation) and layer outputs, I was not able to figure out how I'm supposed to extract these sensitivities and outputs to form the Jacobian matrix. In terms of implementation, which variable to loop over, how many loops to have etc. was difficult in determining. However, after carefully referring to the equation for the Jacobian matrix and the representation, I was able to figure out the external loops and the way to extract the required sensitivity and outputs. Another major challenge that I faced here was constructing each row. Incorporating the bias term associated with a neuron along with its weight was difficult and required some help from ChatGPT in terms of array manipulation. In hindsight, I could have appended my weight and bias in a single matrix and proceeded.

However, after successfully creating the Jacobian with the help of the textbook-solved problem, the next hurdle I faced was extracting the corresponding weight and bias updates from a delta update vector. This was successfully completed, and I finally had all my components for the LM algorithm. The next step was just to follow the pseudo-code which laid out the rules for this algorithm. However, for this part, I referenced the pseudo-code that was provided in the Time Series lecture which I had previously implemented as part of time series parameter estimation.

This completed the LM algorithm. Additional observations can be found in the Results section.

f. While cross-checking for correct Jacobian calculation, I added the capability into my neural network of custom user-entered weight initialization. This was necessary to enter the weights the textbook had selected.

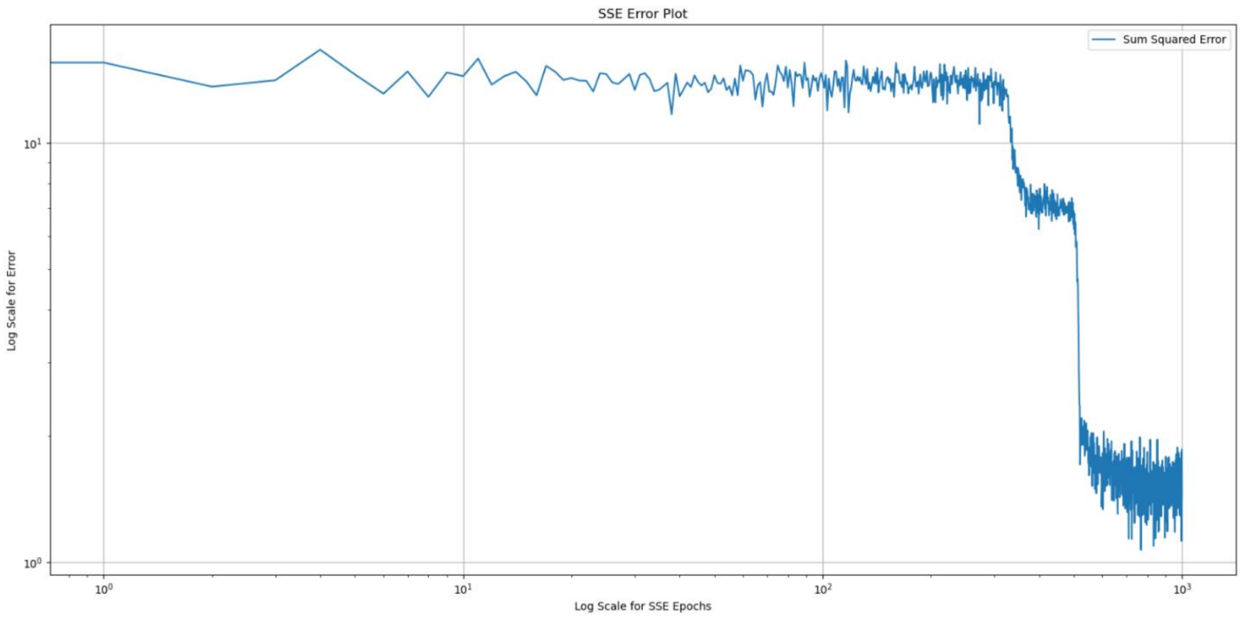
g. Finally, I integrated the conjugate gradient algorithm developed by my team member on top of my generalized network architecture into a final train function in the neural network class that selected the optimization technique based on user-entered optimization value.

h. Additionally, I initiated the actual dataset modeling from my end. I made further observations there are noted down in the Results section. To my unpleasant surprise, I had expected my team members to maintain a cleaned version of the datasets that I could make use of, however, after facing several implementation issues I was able to debug the issue to the presence of null values in the dataset I was using. I decided to halt progress here for the sake of submission and report creation. I will continue this project over the summer to lead to a final conclusion on the initial scope that was set.

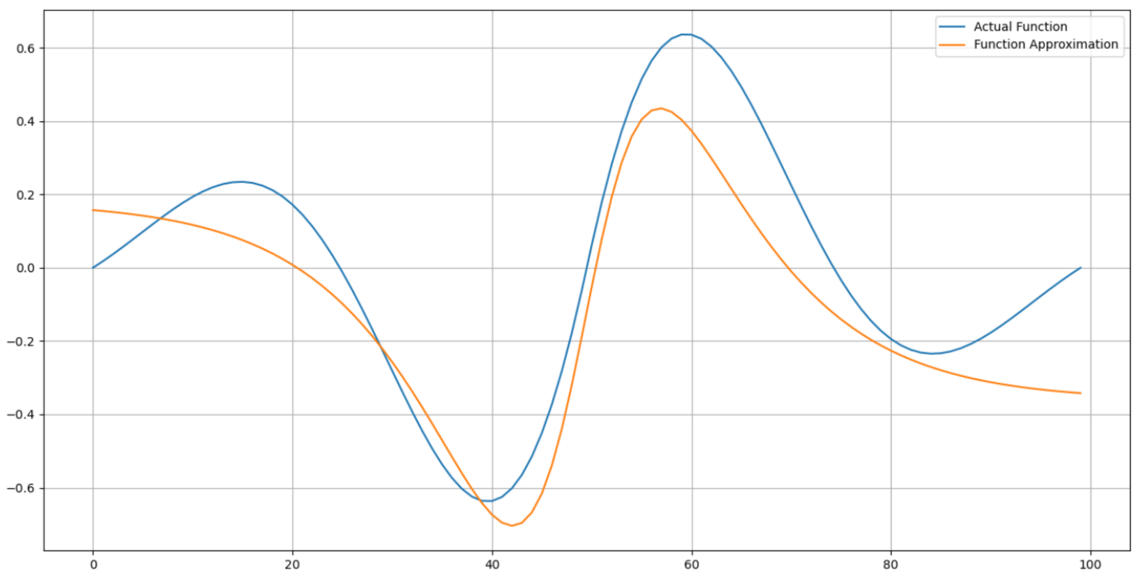
III. Results and Observations

a. Results from 1-S-1 network:

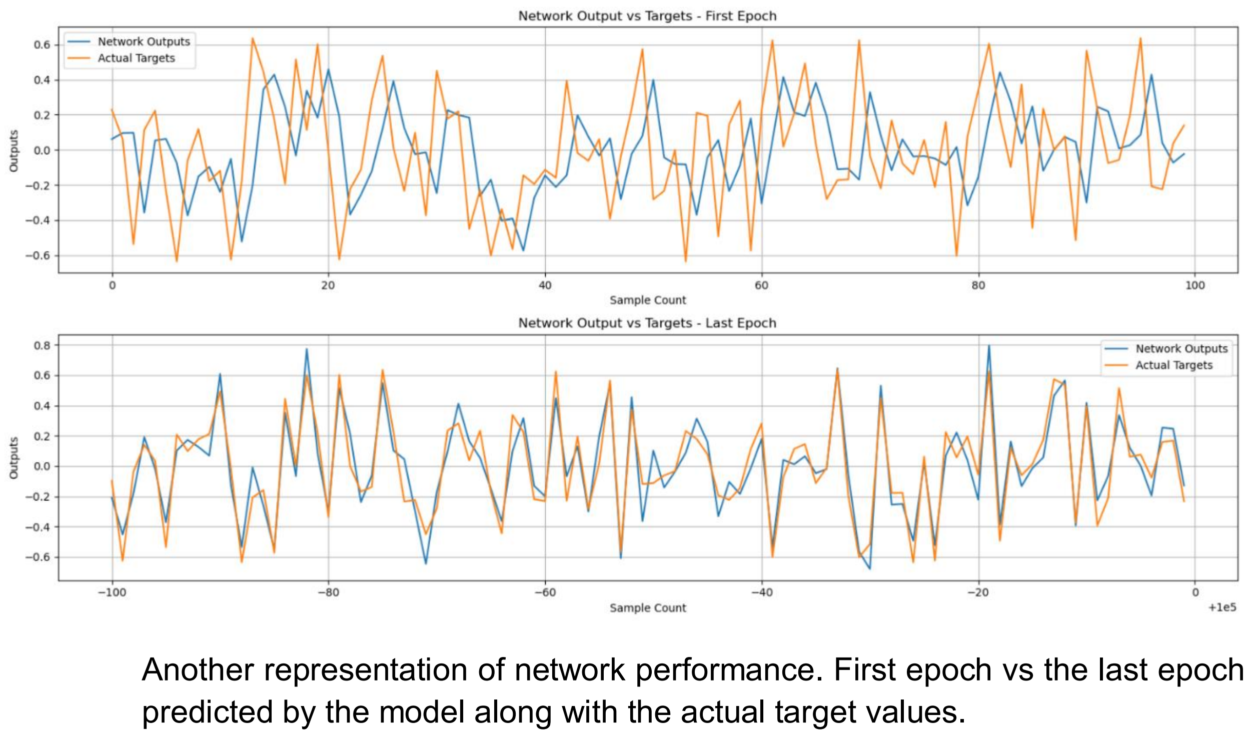
Task: Function approximation - $g(p) = e^{-\text{abs}(p)} \times \sin(\pi p)$ for $-2 \le p \le 2$



This is the SSE plot from a simple 1-S-1 network with a sigmoid activation function in the hidden layer and linear activation in the output layer. 2 neurons were used in the hidden layer and the network was trained for 1000 epochs with a learning rate of 0.2.

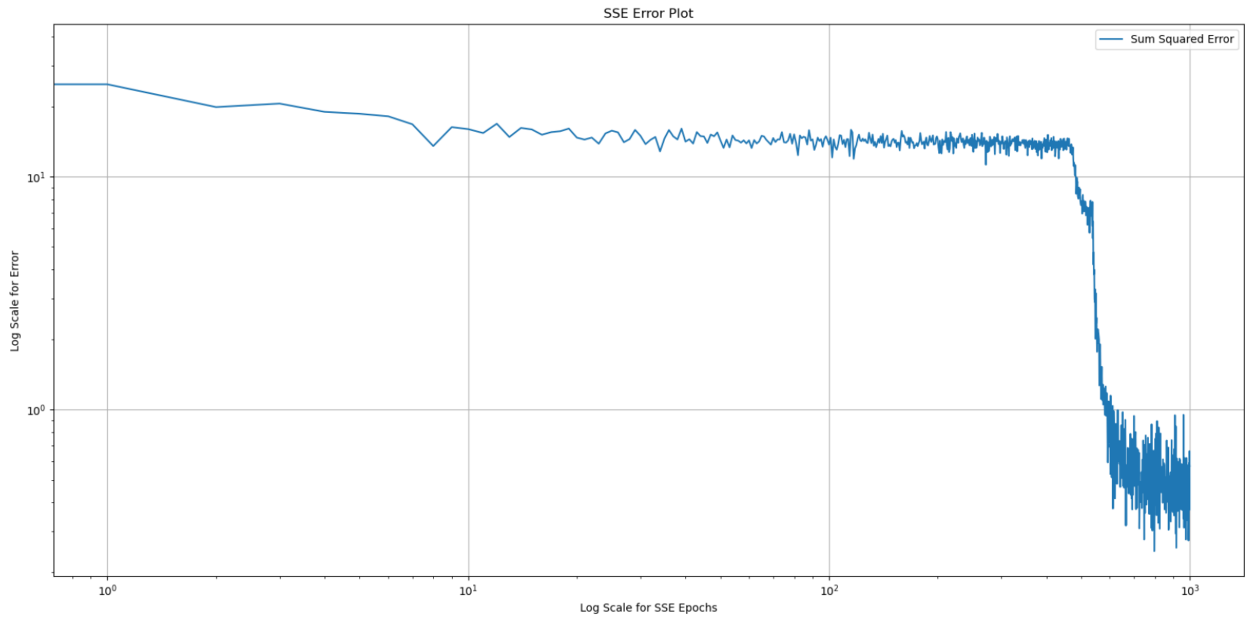


This is the final network output for the trainset vs the target set. It is to see how well the network is able to approximate the underlying function that generated the data. With 2 neurons, we can see that the inflection points are not captured effectively.

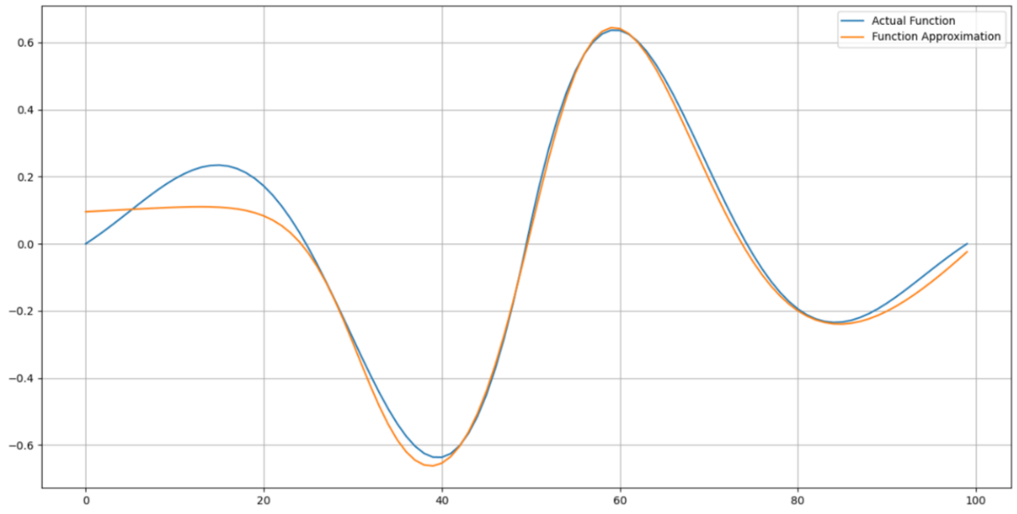


Another representation of network performance. First epoch vs the last epoch predicted by the model along with the actual target values.

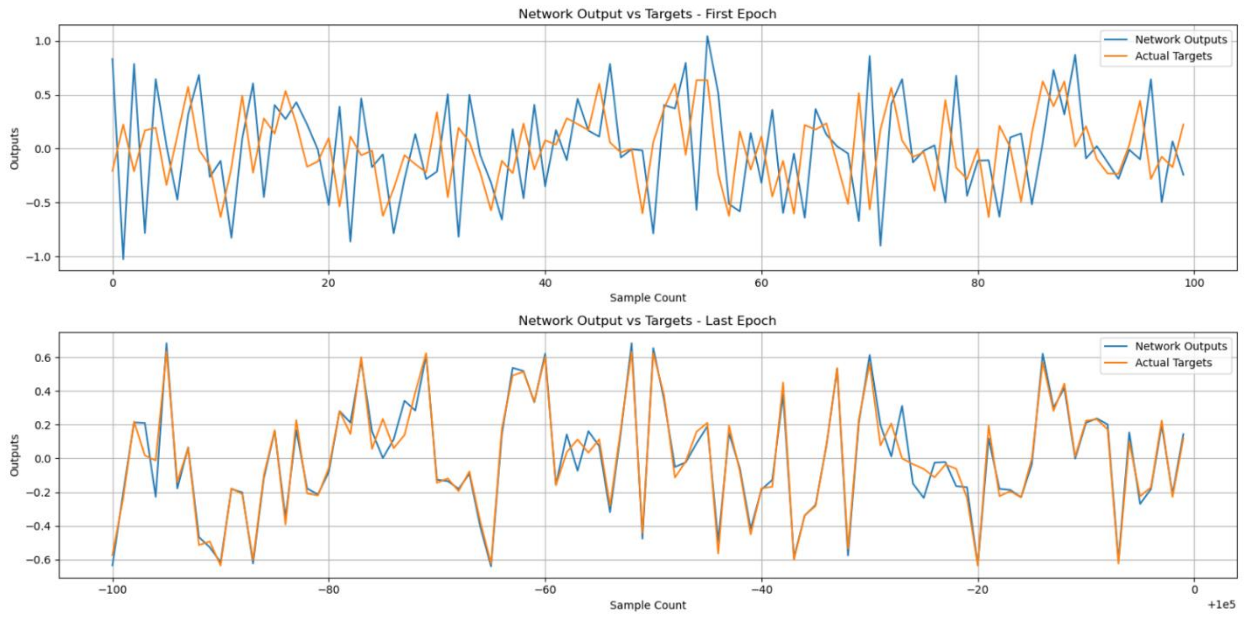
Now the same process was replicated for a network with 10 neurons in the hidden layer. Here are the outputs:



SSE is lower for the same number of epochs (1000) and learning rate (0.2) as compared to a network with 2 neurons.

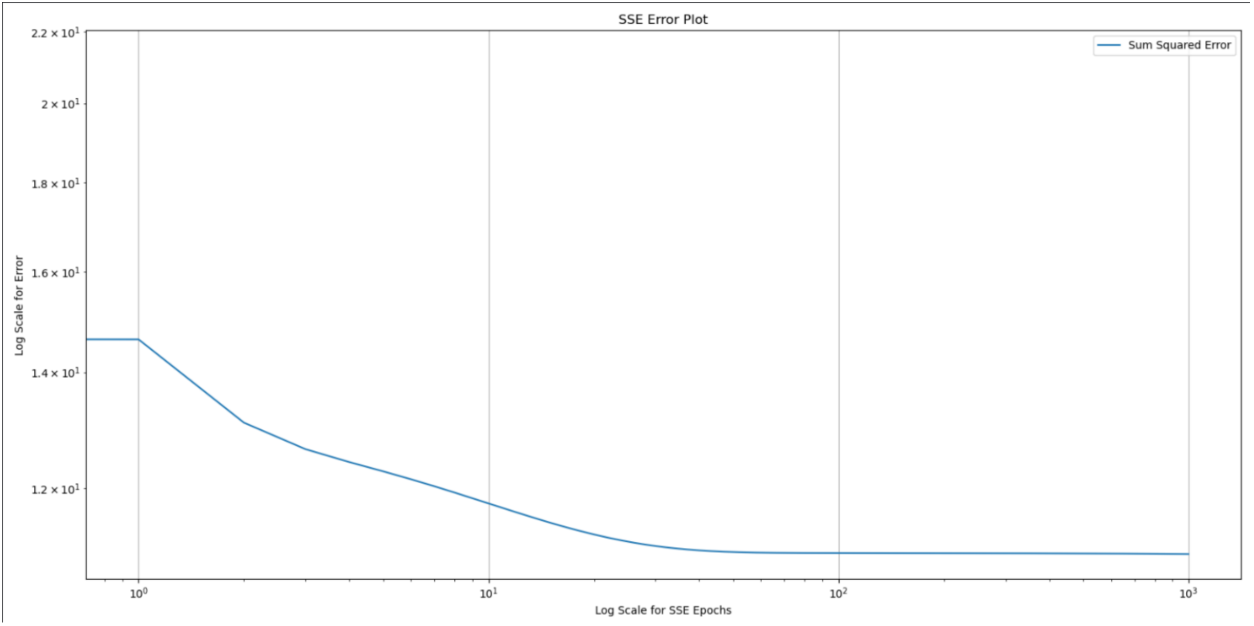


The function is approximated more accurately with a greater number of neurons. The inflection points are effectively captured in this output.

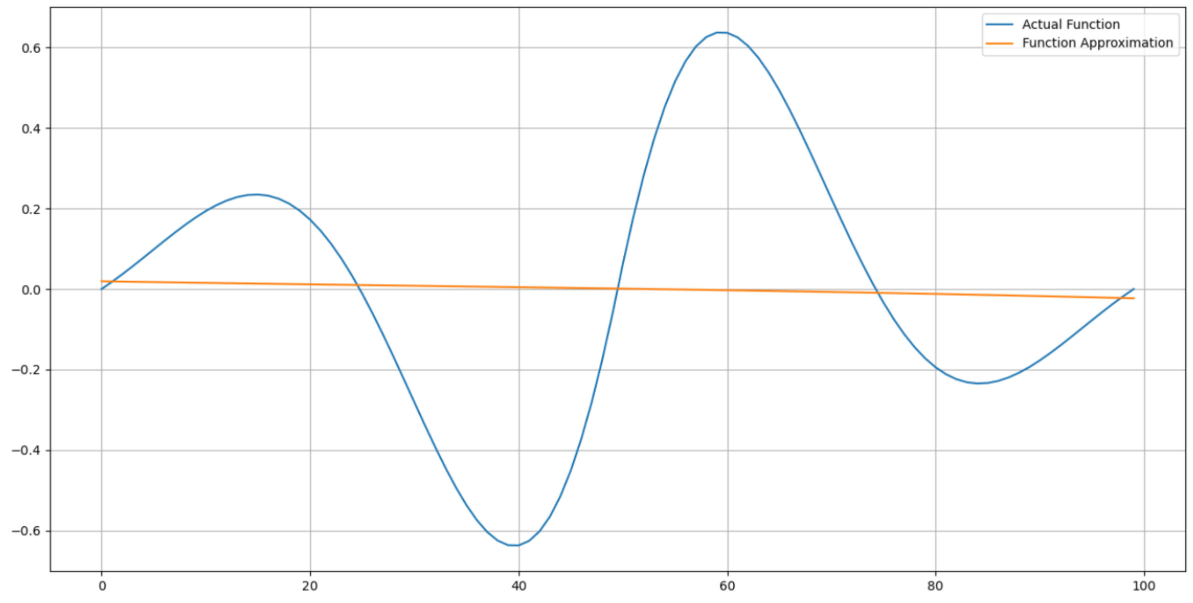


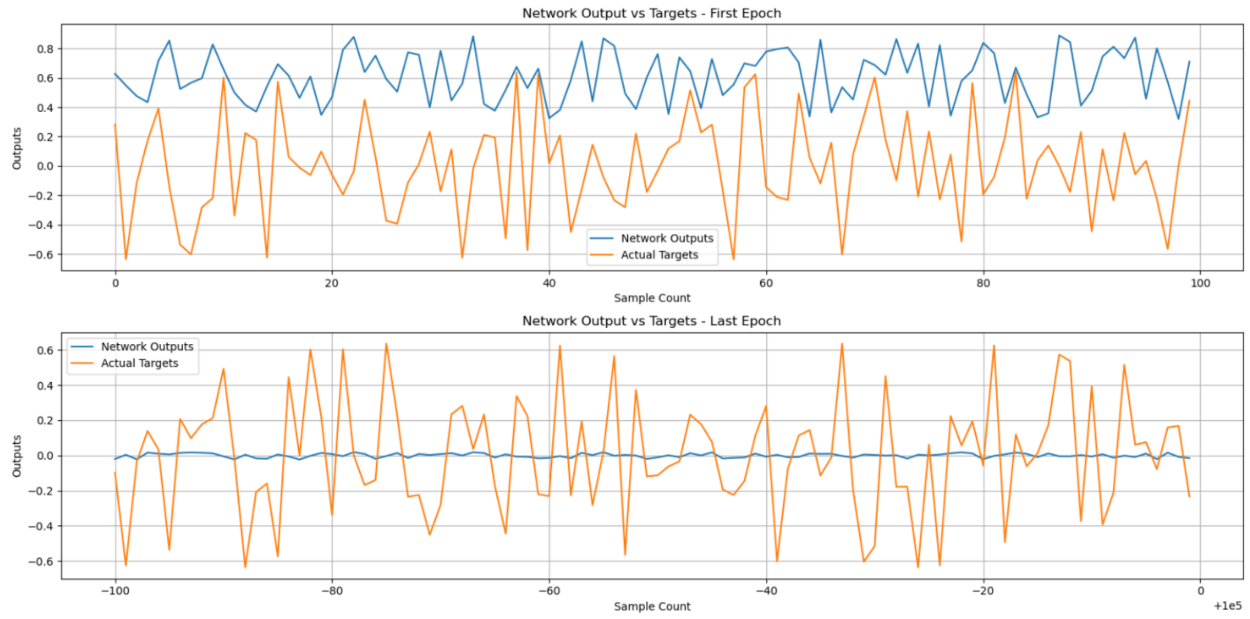
Alternate view of better network performance with 10 neurons. First epoch vs last epoch.

b. Batch training vs Incremental training:
The prior assumption with batch training was that batch training will take a lesser number of weight updates to converge to minima. This was assumed by me since the descent direction is guaranteed to converge. However, after successfully building the algorithm. When I tested it out on the same function approximation problem, with the number of neurons = 10, epoch = 1000, learning rate = 0.2, and batch size = 100 (full batch as 100 inputs points), these were the results:



The SSE appeared to be way higher than the incremental training results.



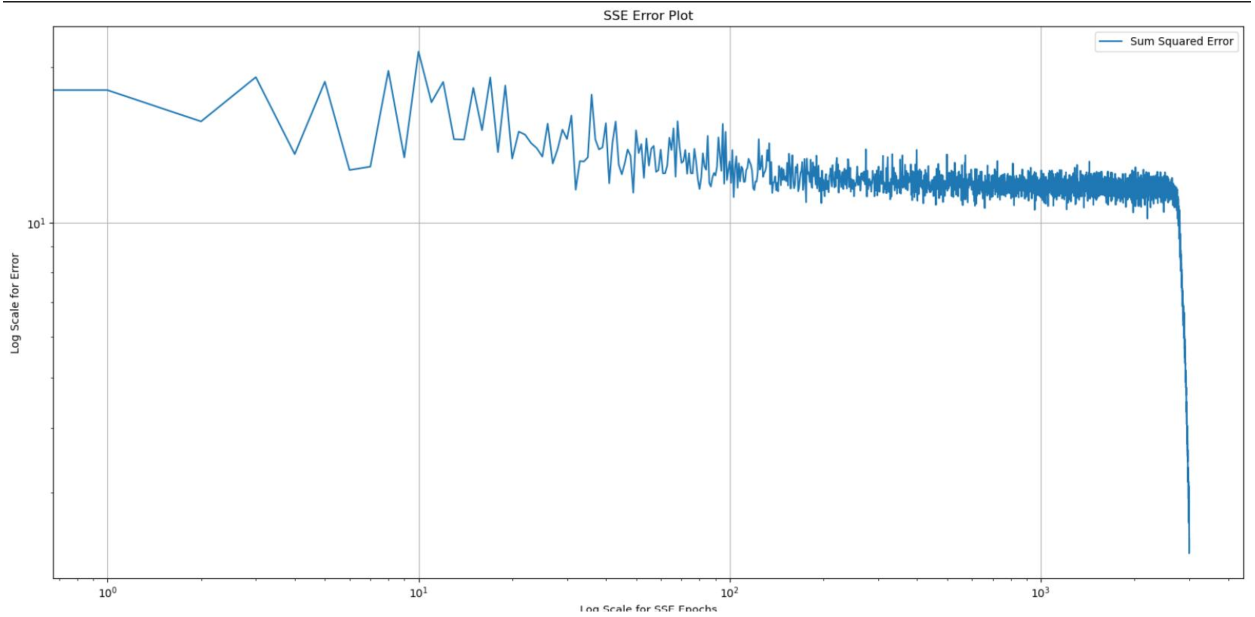


Very poor function approximation by the neural network. Additionally, the same 1000 epochs took a longer execution time as compared to stochastic. Therefore, not only were the results unsatisfactory, but the computation time was greater too.

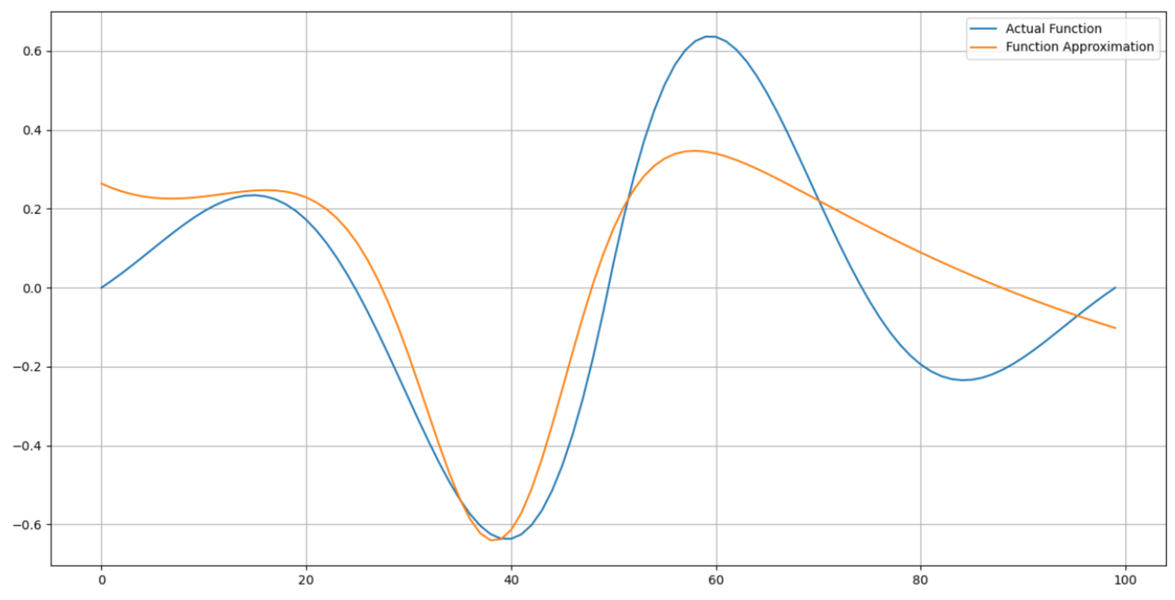
This phenomenon was a little confusing at first, however, I realized later that maybe the difference lay between the number of weight updates. The number of weight updates for a full batch for 1000 epochs was a mere 1000 as compared to the weight updates for stochastic for 1000 epochs, 1000 x 100 (train size) = 100,000 updates.

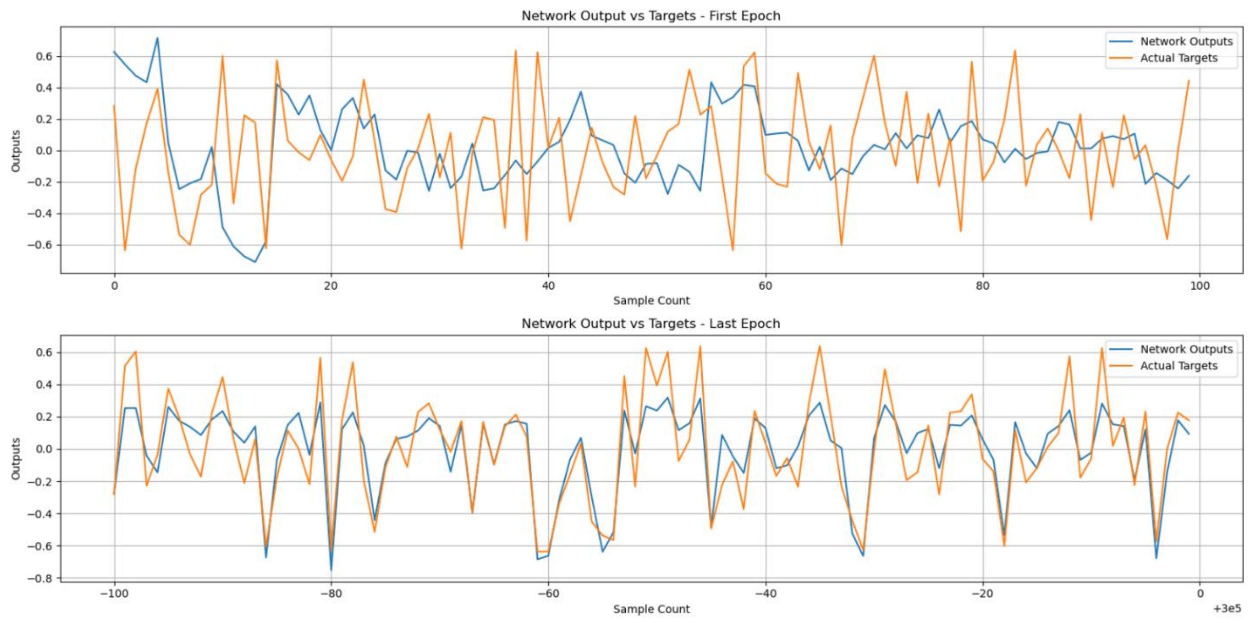
To test this hypothesis, I performed a comparative analysis between online and batch training.
I wanted to see whether batch training performed better for the same number of weight updates or not. To test this, these were the selections I made:
Incremental training: neurons = 10, epochs = 600, learning rate = 0.2
Batch training: neurons = 10, epochs = 3000, learning rate = 0.2 batch size = 5
These two selections would amount to the same number of weight updates.

Batch results:



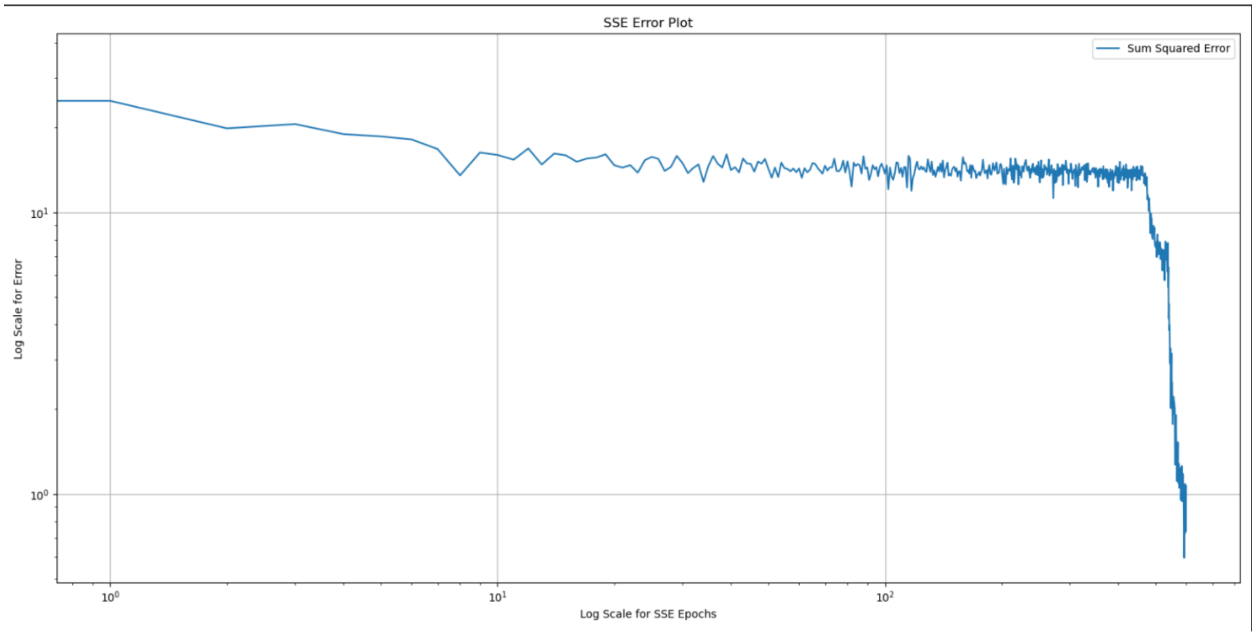
SSE plot for 3000 epochs.



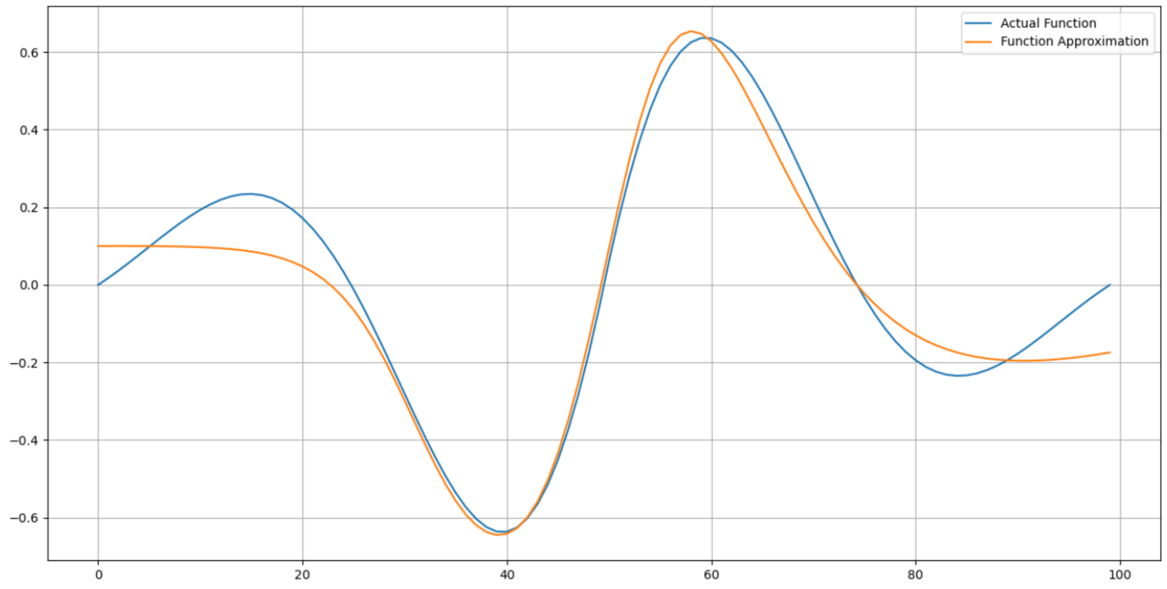


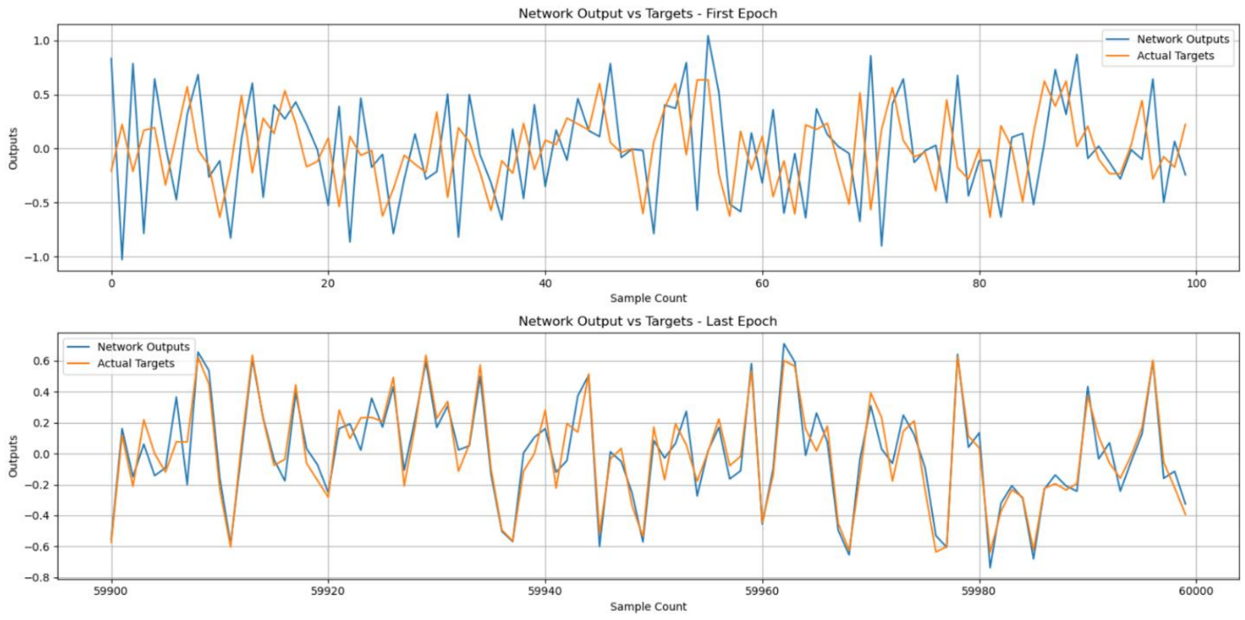
Function approximation plots.

For incremental training:



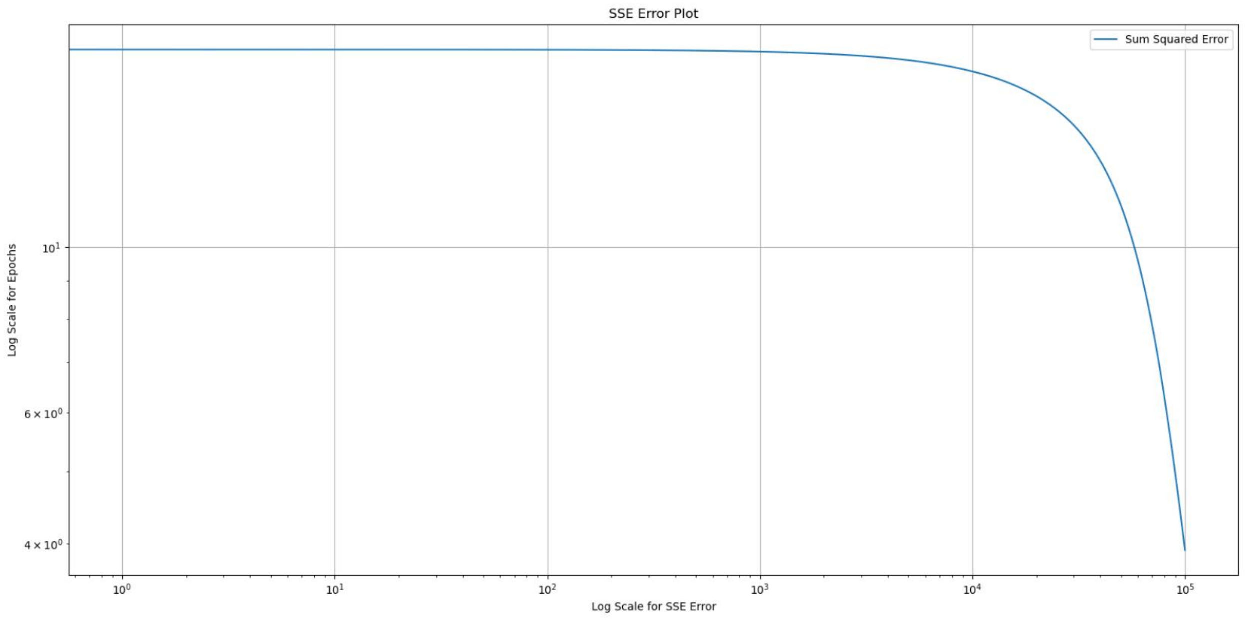
SSE plot for online training. Indicating a much better net SSE score achieved.





Better function approximation as well with incremental training.

Another test: Ran full batch training for 100,000 epochs (which took up to 20 minutes to finish executing) out of curiosity. Now I have given full batch training an ample number of weight updates to converge, as the incremental learning algorithm. Following is the SSE plot:

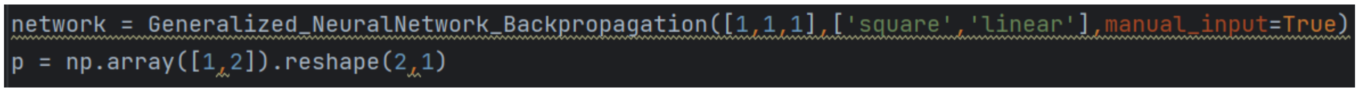


The graph is perfectly smooth and downhill as expected by batch training. However, a key observation is the net SSE achieved. Incremental learning was able to achieve SSE in the order of 0.01 whereas batch training has only reached in the order of 3-4.

Net verdict: Incremental learning seems to perform better than batch training. However, this is not the final conclusion I'm taking, and will not generalize the conclusion based on the above tests. More diverse sets of testing will be required to state this with more confidence.

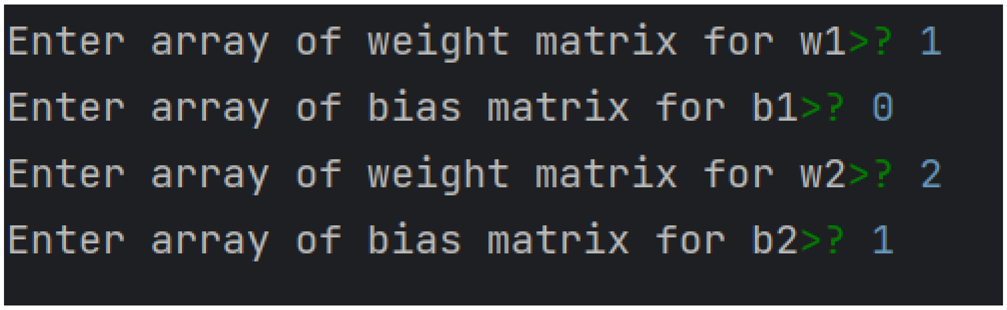
c. Neural Network Generalization observations:
After creating the general network with no syntax errors, I was facing an issue with overflow. Error calculation and hence sensitivity propagation were incorrect for the longest time which was causing my weights and biases to explode and activation output to overflow.
Helpful takeaway: If weights and biases seem to explode, there is a problem with sensitivity backpropagation. This should be checked first. Additionally, error calculation should be checked as the starting point.

d. LM algorithm and added neural network capability observations:
The general code for the added neural network capability is shown below:



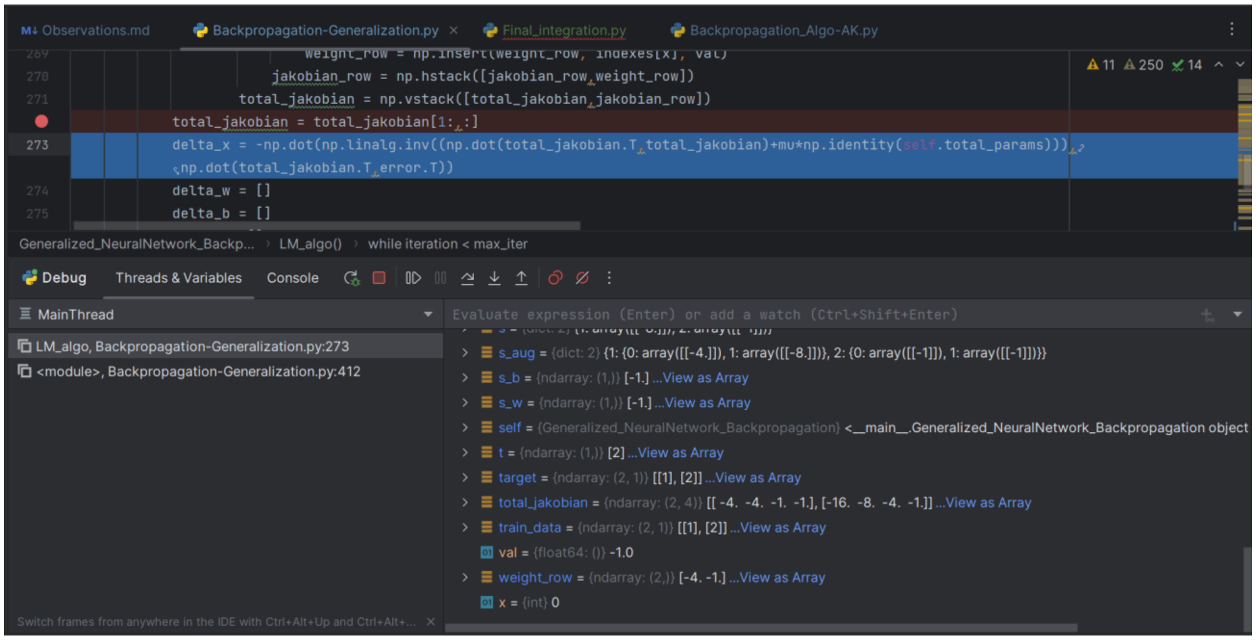
```
network = Generalized_NeuralNetwork_Backpropagation([1,1,1],['square','linear'],manual_input=True)
p = np.array([1,2]).reshape(2,1)
```

The inputs are presented as given in the Neural Network textbook.



```
Enter array of weight matrix for w1>? 1
Enter array of bias matrix for b1>? 0
Enter array of weight matrix for w2>? 2
Enter array of bias matrix for b2>? 1
```

Weights as entered as initialized in the textbook. The following capability was developed to check the development of the Jacobian matrix and match it with the textbook.



Observations.md | Backpropagation-Generalization.py | Final_integration.py | Backpropagation_Algo-AK.py

```
                    weight_row = np.insert(weight_row, indexes[x], val)
                jakobian_row = np.hstack([jakobian_row,weight_row])
            total_jakobian = np.vstack([total_jakobian,jakobian_row])
        total_jakobian = total_jakobian[1:,:]
        delta_x = -np.dot(np.linalg.inv((np.dot(total_jakobian.T,total_jakobian)+mu*np.identity(self.total_params))),
        np.dot(total_jakobian.T,error.T))
        delta_w = []
        delta_b = []
```

Generalized_NeuralNetwork_Backp... › LM_algo() › while iteration < max_iter

Debug | Threads & Variables | Console

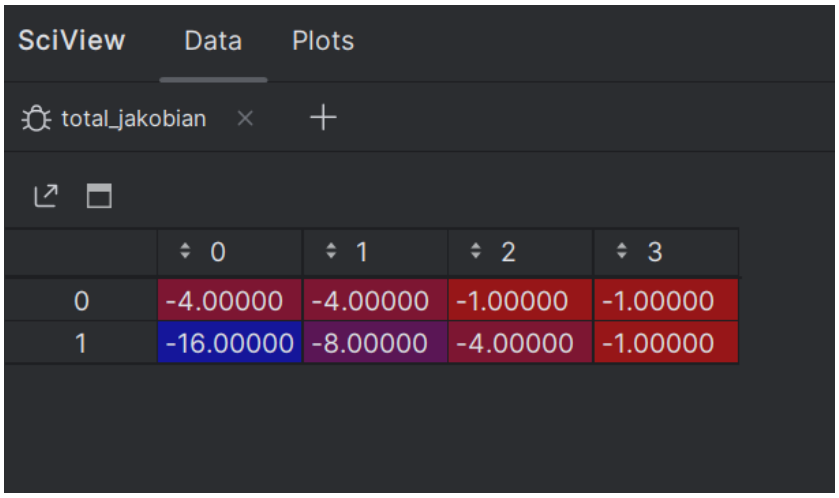
MainThread

LM_algo, Backpropagation-Generalization.py:273
<module>, Backpropagation-Generalization.py:412

```
s_aug = {dict: 2} {1: {0: array([[-4.]]), 1: array([[-8.]])}, 2: {0: array([[-1]]), 1: array([[-1]])}}
s_b = {ndarray: (1,)} [-1.] ...View as Array
s_w = {ndarray: (1,)} [-1.] ...View as Array
self = {Generalized_NeuralNetwork_Backpropagation} <__main__.Generalized_NeuralNetwork_Backpropagation object
t = {ndarray: (1,)} [2] ...View as Array
target = {ndarray: (2, 1)} [[1], [2]] ...View as Array
total_jakobian = {ndarray: (2, 4)} [[ -4.  -4.  -1.  -1.], [-16.  -8.  -4.  -1.]] ...View as Array
train_data = {ndarray: (2, 1)} [[1], [2]] ...View as Array
val = {float64: ()} -1.0
weight_row = {ndarray: (2,)} [-4. -1.] ...View as Array
x = {int} 0
```

Switch frames from anywhere in the IDE with Ctrl+Alt+Up and Ctrl+Alt+...

Utilized the debugging software to check the construction of the state variables. To match the sensitivities and outputs and most importantly, to match the selection of sensitivity and output as required for the Jacobian matrix calculation. This was a good way to test whether the loops I had written were conceptually valid or not.



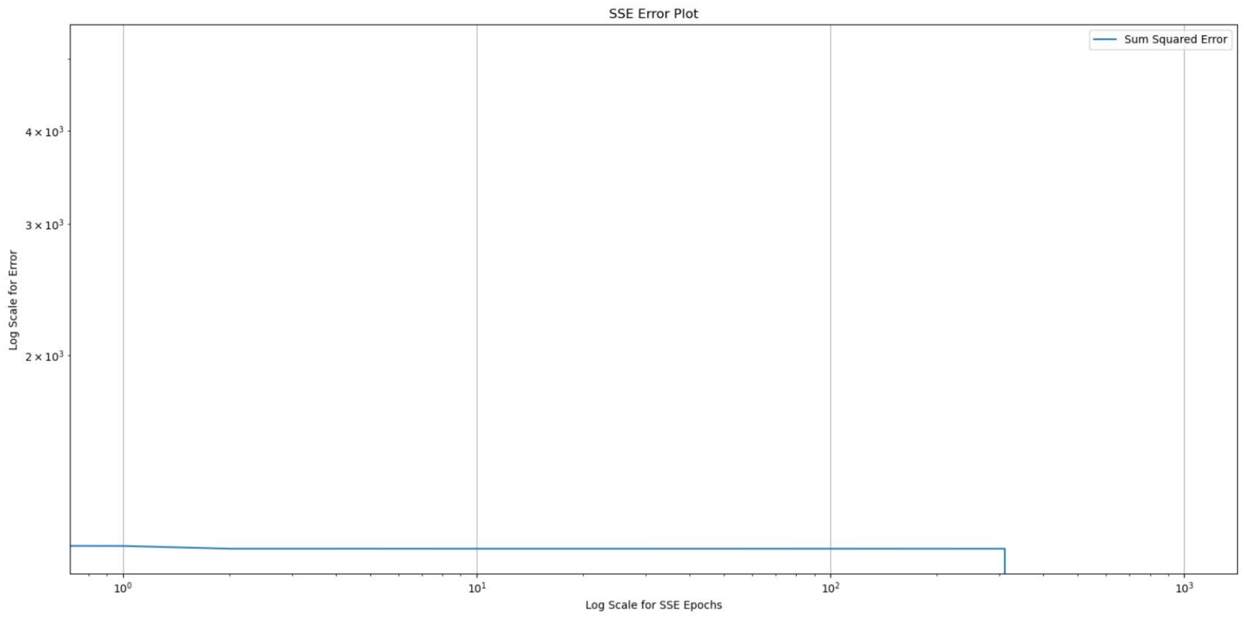
SciView | Data | Plots

total_jakobian

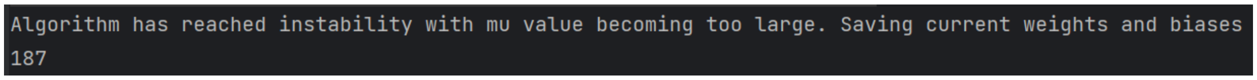
	0	1	2	3
0	-4.00000	-4.00000	-1.00000	-1.00000
1	-16.00000	-8.00000	-4.00000	-1.00000

Finally, through modifications in the code through observing the debugging, I was able to achieve the right Jacobian matrix for the given selection. The Jacobian matrix construction was successful.

Upon completion of my LM algorithm, there was yet another conceptual problem I encountered. For the function approximation problem, my algorithm was not performing well. The SSE results were not going down beyond a point and the plots looked like this:



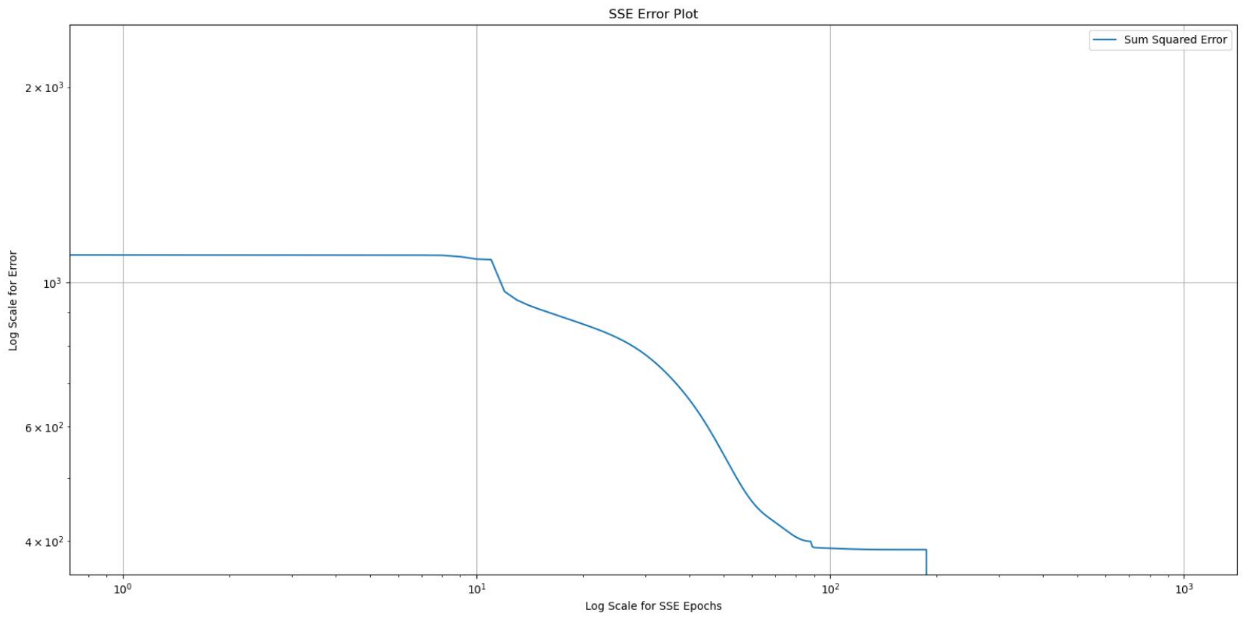
The SSE was not going below 10.99 for the same function approximation problem.



```
Algorithm has reached instability with mu value becoming too large. Saving current weights and biases
187
```

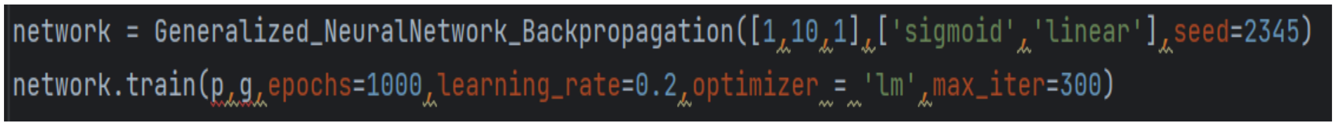
With this output on the console. Increasing the mu value was not helping, and the model wasn’t able to find the steepest descent direction.
I tried tweaking the various parameters such as max mu, convergence criteria, and number of iterations but to no avail.
However, I realized the problem was that the function was getting stuck in a local minimum/saddle point. With a different initialization seed and hence different weight and bias initializations I was able to solve this problem.

Resulting SSE curve:



The SSE was able to drop to 3-4 as compared to 11 before.

e. Final Integrated function call appearance:



```
network = Generalized_NeuralNetwork_Backpropagation([1,10,1],['sigmoid','linear'],seed=2345)
network.train(p,g,epochs=1000,learning_rate=0.2,optimizer = 'lm',max_iter=300)
```

f. Modeling observations:
Train data, features, and targets are not readily available to use after the train test split with my custom models. Additional modifications are required for compatibility.

IV. Summary:

After successfully modeling my own custom neural network architecture training for stochastic gradient decent algorithm, batch / mini-batch training algorithm, and implementing the Levenberg-Marquardt algorithm, I've got a deeper understanding of the workings of these algorithms, and I've effectively enhanced my skill in turning any pseudo-code to actual code which is an important skill to possess in this industry.

Few learnings:

a. Observing the Levenberg-Marquardt algorithm issue, it has become clear how it is important to initialize the network with different weights and biases to get optimum results and avoid saddle point/local minima issues.

b. Uncertainty regarding what stopping criteria to use for the LM algorithm. As of now, SSE value < 1 is the criteria but nothing seems to converge. Stopping criteria with the Norm of Delta vector < 0.1 also doesn't converge. Deciding these will be key steps in completing the algorithm. For the time being, provision has been made in making them custom-set thresholds.
c. Batch training doesn't necessarily perform better than the stochastic gradient. However, this needs to be further tested.

Some future work I would like to do in this project:

a. I would like to add a verbose type of feature in my custom training codes so that the status of training can be viewed easily.
b. Create an interactive dash enable dashboard that will display the findings of this project effectively.
c. After observing the LM algorithm issue, another added feature in my code could be multiple seed initializations and picking the lowest SSE score/scores model.
d. Time complexity comparison in backend execution.
e. Another added argument that takes in the metric argument - Typically SSE for classification and MSE for regression.
f. Perform in-depth comparison using initially defined criteria.
g. Comparison with conventional models.

Percent of code from internet = (8/777) * 100 = 1.029%

Only code for the softmax function and derivative of the tanh function were copy-pasted from the internet. However, for certain logics, I have used ChatGPT's help in implementing complex matrix manipulation. However, no code was copied, only logic was referred.

V. References:

a. "Neural Network Design" (2nd Ed), by Martin T Hagan, ISBN 0971732116
b. Henry P. Gavin (2022), "The Levenberg-Marquardt algorithm for nonlinear least squares curve-fitting problems", Duke University, Department of Civil & Environmental Engineering.
c. Rauf Bhat (2020), "Gradient Descent with Momentum", Towards Data Science.

d. I. Khan et al (2020), "Design of Neural Network with Levenberg-Marquardt and Bayesian Regularization Backpropagation for Solving Pantograph Delay Differential Equations" in IEEE Access, vol. 8, pp. 137918-137933, doi: 10.1109/ACCESS.2020.3011820.

e. Bruce, P., et. al.,(2020). Practical Statistics for Data Scientists, O'Reilly