Neural Network Optimization for Function Approximation

Sharjeel Ahmad (24100083) - *Physics*

Supervision

– Muhammad Faryad (muhammad.faryad@lums.edu.pk)

1 Abstract

This research project aims to answer a fundamental question about neural networks "how many neurons and layers would be required in order to create the most efficient and accurate architecture for a given class of functions?". To answer this question, the project utilizes an experimental approach where a select function is mapped using an unsupervised neural network and its loss is calculated under varying number of layers and neurons. Finally an attempt is made to establish a semi-formal correlation between the loss, number of layers and number of neurons in the system.

2 Introduction

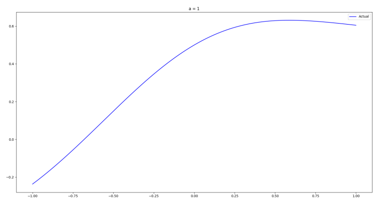
Machine learning software have been essential in today's digital age. Its ability to create statistically accurate models using millions of data points has led to new advances in scientific and economic analysis. Nonetheless, Machine Learning algorithms are not self sufficient. Aside from the input data, the algorithm requires the operator to set up multiple parameters, activation functions and so on in order for the algorithm to run smoothly and give a statistically accurate output. This has led to the rise of data scientists and analysts who often specialize in machine learning. Normally, the set up of these machine learning parameters is done through the users experience of dealing with similar data sets and knowing which algorithm would be best suited. If that is not the case then through a hit and trial method is followed. Theoretically speaking it is possible to accurately map any function/data set through an arbitrarily large neural network, but doing so would require enormous amount of time and energy. Instead data scientists use hit and trial to approximate the least number of layers and neurons needed for the algorithm to functionally converge onto the data set.

3 Function

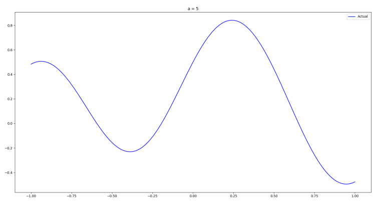
For this experimental a linear combination of a sin and exponential function were used inside a domain of $[-1, 1]$. The exponential function was particularly introduced to add a layer of randomness inside the function and prevent it from being symmetric at $x = 0$. A frequency parameter 'a' was also introduced to increase or decrease the variation in the function and test the machine learning algorithm across varying data sets. The selected function was:

$$f(x,a) = \frac{sin(ax) + exp(-ax^2)}{2}. \tag{1}$$

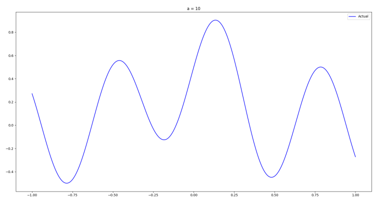
Three values of 'a' were used: 1, 5, 10. The graphical representation of these functions can be seen below.



(a) $a = 1$



(b) $a = 5$



(c) $a = 10$

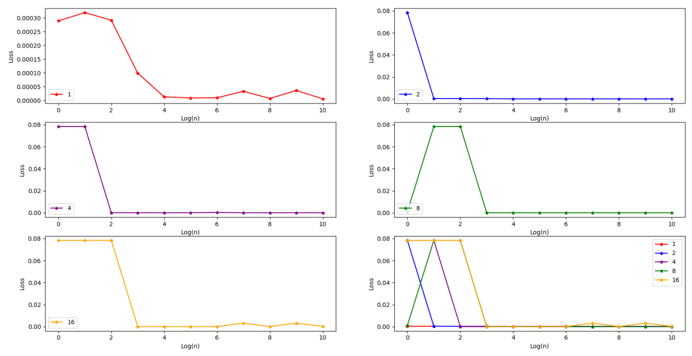
Figure 1: Graphical showcasing of the functions

4 Experimental Setup

The TensorFlow library was used for the machine learning algorithm. The number layers and number of neurons per layer were adjusted in powers of 2 i.e. 2^i where i went from 0 to n. The activation functions used was 'Relu' followed by a single linear activation for the output. The optimizer used was 'Adam'. The number of epochs until convergence was set to 100 by default, however, changes were made to the epoch counts of different functions to ensure there was a convergence. The loss was calculated as a mean square error. The input data was divided into 1000 data points on the x-axis going from -1 to 1 as mentioned earlier. The experiment was conducted in two phases. In phase 1, the number of neurons per layer were taken as a variable and plotted against the loss, for different values of the layer count. In phase 2, the number of layers were taken as a variable and plotted against the loss for different values of the neurons per layer count. The activation function of each layer was kept as 'Relu' and every layer had the same number of neurons. The whole process was repeated three times for each value of 'a'. Log plots were used given the large range of the variables.

5 Neuron Variable

For phase 1 of the experiment the number of neurons were increased from 1 to 1024 in steps of 2^i. The loss was then observed for the number of layers going from 1 to 16 in steps of 2^j. For $a = 1$, the following graphs were obtained:



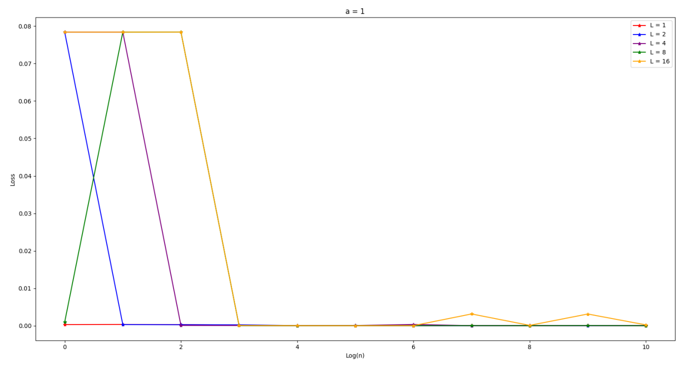


Figure 2: Loss against Log(number of neurons per layer) for $a = 1$

From the results we can see that for a relatively smooth graph like $f(x, 1)$ the loss converges to a minimum mean if we keep increasing the number of neurons per layer. However, in general, for a greater number of layer we require a greater amount of neurons for the algorithm to converge. This means that after a certain point the number of layers may be over-fitting the data set or creating too many parameters.

For $a = 5$, the following graphs were obtained:

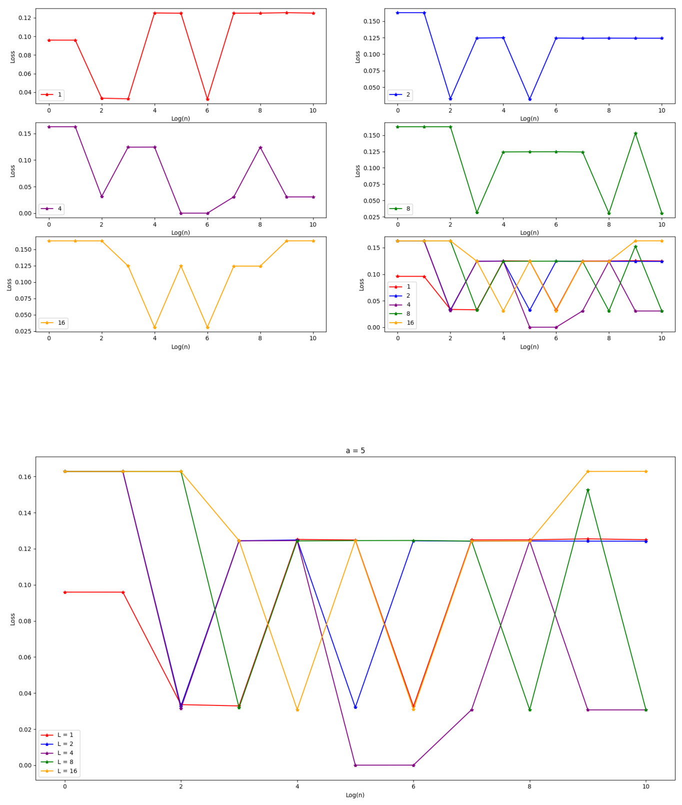
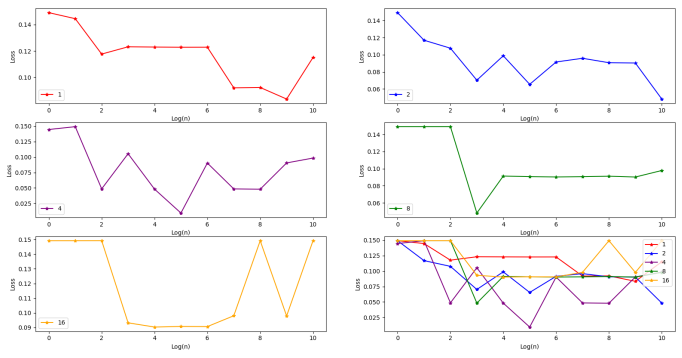


Figure 3: Loss against Log(number of neurons per layer) for $a = 5$

For $a = 5$ our convergence in loss becomes much more randomized with no discernible patterns on how the layers and neurons are affecting the overall loss. Nonetheless there are some “sweet spots” where the loss has minimized for a particular layer and neuron count such as in the case of 16 layers with 16 neurons per layer and 16 layers with 64 neurons per layer. This randomization is of course expected given the fluctuating nature of $f(x, 5)$, making it difficult for the algorithm to reliably converge. These “sweet spots” may not be indicative of a particular pattern rather the function just converging to a low loss by chance.

Finally for $a = 10$, the following graphs were seen:



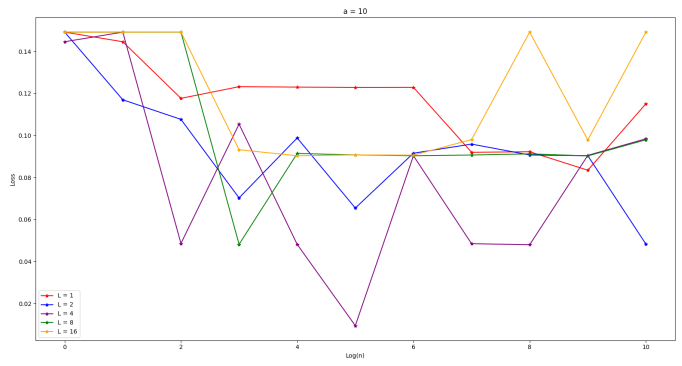


Figure 4: Loss against Log(number of neurons per layer) for $a = 10$

Just as for $a = 5$, at $a = 10$ we observe a randomized variation in loss with increasing neurons and layer counts. However, there is some loss stabilization at 8 layers for after 16 neurons or more. Similarly for 16 layers the loss is relatively stable between 8 neurons and 128 neurons. The range of neurons for which the loss has remained constant does make it unlikely that it was due to chance, but rather the fact that the algorithm is able to map the data set in a particularly unique way given these set of parameters.

6 Layer Variable

For phase 2 of the experiment the number of layers were increased from 1 to 16 in steps of 2^i. The loss was then observed for the number of layers going from 1 to 16 in steps of 2^j. For $a = 1$, the following graphs were obtained:

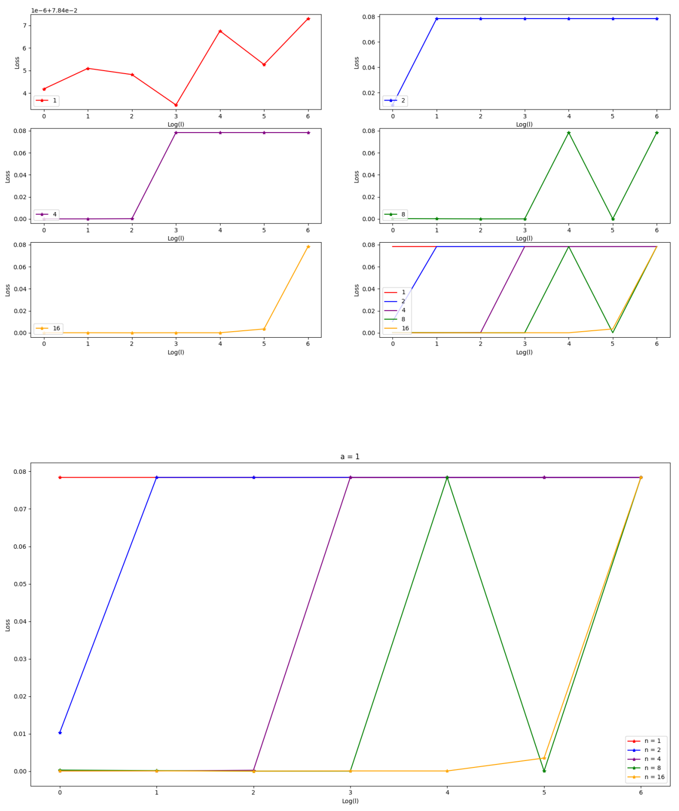


Figure 5: Loss against Log(number of layers) for $a = 1$

Just as we had seen in Phase 1 of the experiment, for a larger value of the number of layers the loss increases after a certain threshold. However, this threshold can be offset to a higher value of 'L' if we use a higher neuron count per layer. This could mean that the number of neurons may have a more general positive contribution towards decreasing the loss of the model.

For $a = 5$, the following graphs were obtained:

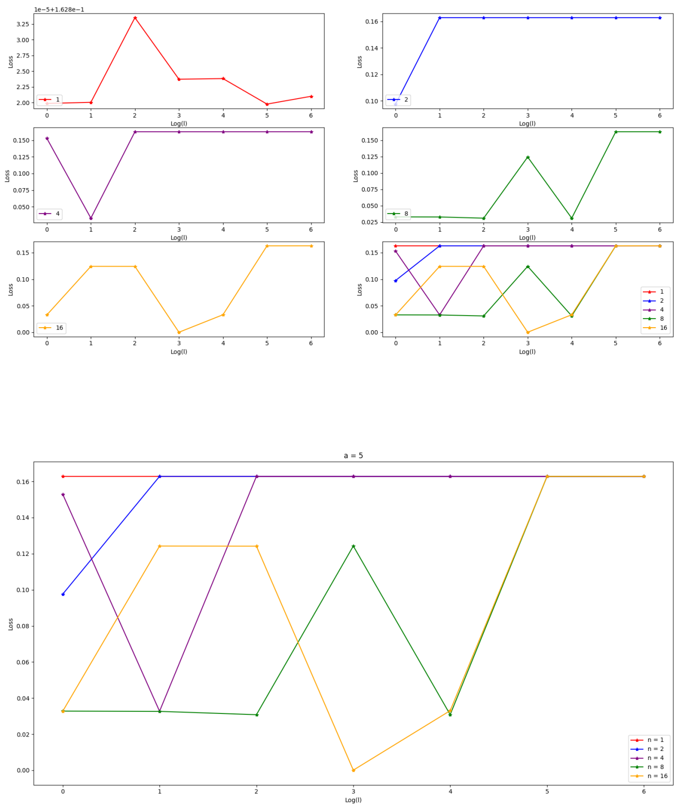
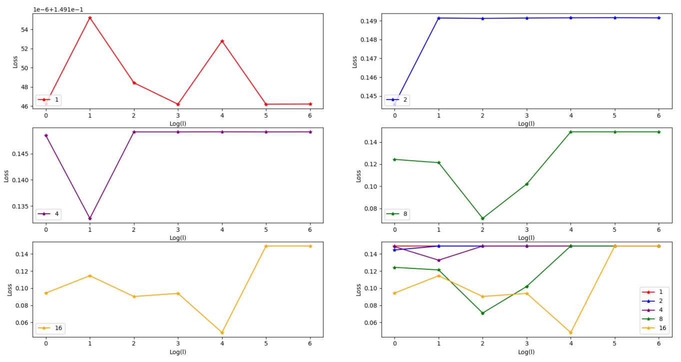


Figure 6: Loss against Log(number of layers) for $a = 5$

In this case we see a more stabilized relationship between between the loss and the layer count for 2 neurons per layer and 4 neurons per layer. It is also important to note that as a general trend all the graphs show an eventual increase in the loss as the number of layers are kept on increasing, this alludes to the fact that layers beyond a certain value result in hyper paramertization of the algorithm which makes it extremely volatile to even small changes in the highly varying $f(x, 5)$ function.

In the case of $a = 10$, we are outputted the following data:



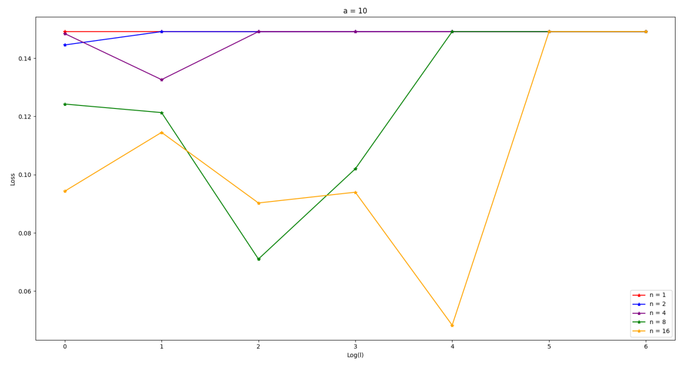
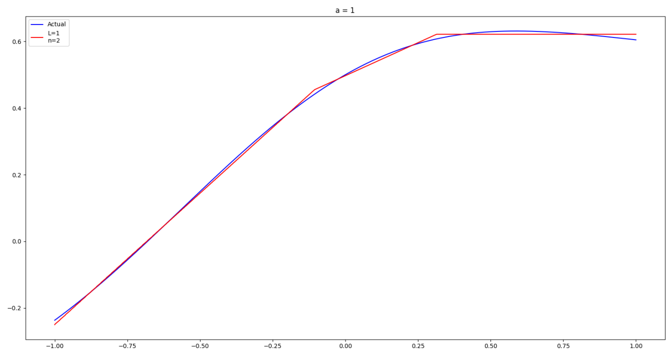


Figure 7: Loss against Log(number of layers) for $a = 10$

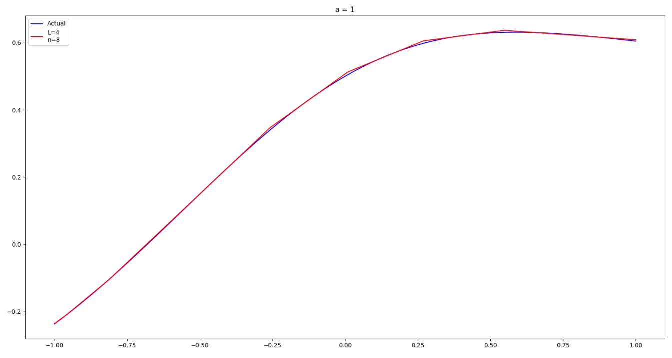
Just like with $a = 5$, there is a trend in $a = 10$ where the loss eventually increases to a maximum for increasing number of neurons. Once again as observed before this loss-layer count threshold can be offset by increasing the neuron count per layer.

7 Curve Fitting

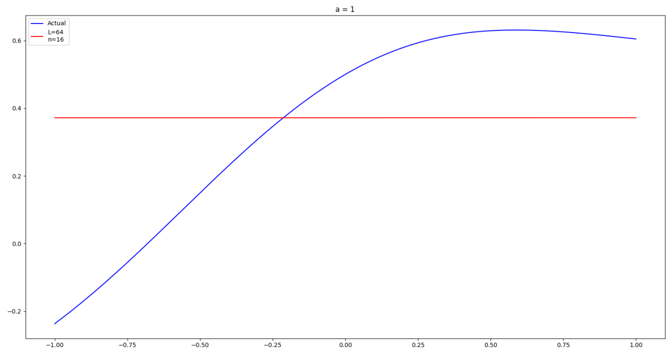
To better illustrate our results, we constructed models with different layer and neuron counts and fit them onto the actual functions. For $a = 1$, we get the following fits:



(a) L = 1, n = 2



(b) L = 4, n = 8

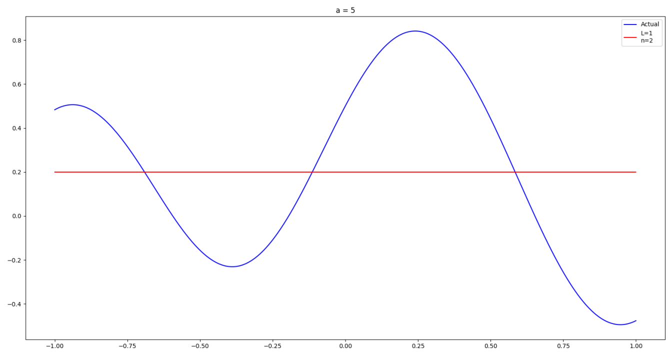


(c) L = 64, n = 16

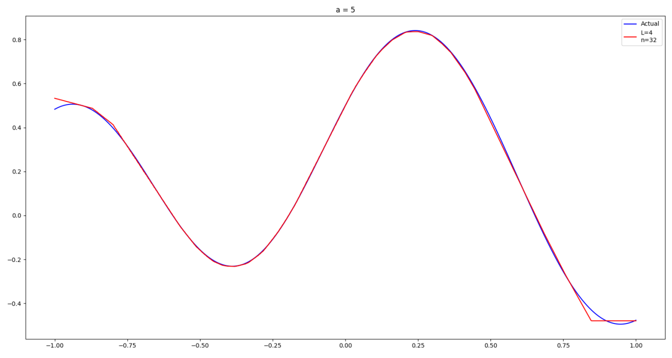
Figure 8: Curve Fitting for $a = 1$

Given the simple and semi-linear nature of $f(x, 1)$, it is easy to see why a low layer and low neuron count neural network would be able to fit the curve as shown in figure 8(a). Taking a look at figure 8(b), we see that for a moderate layer and neuron count we are able to obtain a curve with even lower loss. However, if we keep on increasing the layer and neuron count, we end up with a straight line through the curve: over-parametrization has made it difficult for the model to follow along the curve.

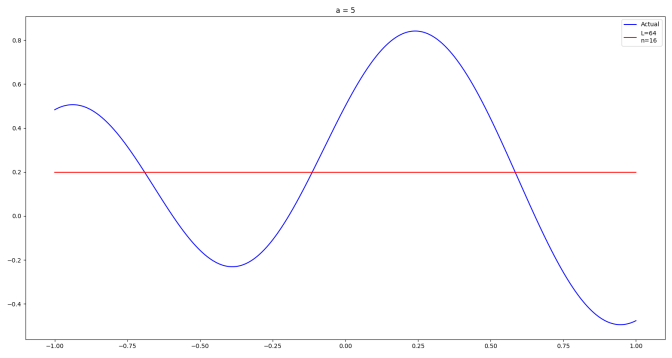
For $a = 5$, we get the following fits:



(a) L = 1, n = 2



(b) L = 4, n = 32

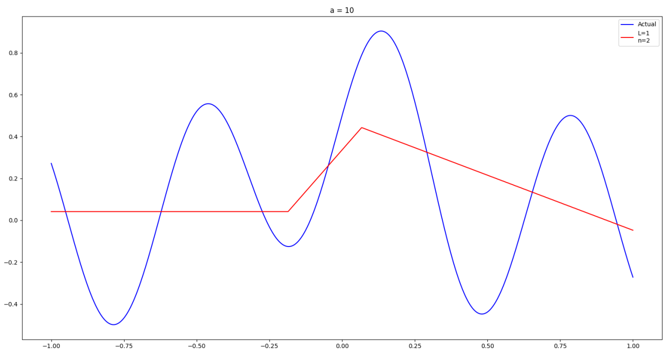


(c) L = 64, n = 16

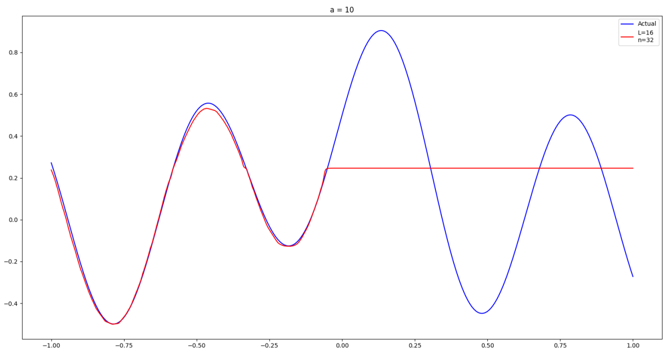
Figure 9: Curve Fitting for $a = 5$

Unlike in the case for $f(x, 1)$, a low neuron and layer count algorithm can no longer parameterize a curve with high variance. Similarly in the case for n = 16 and L = 64, there is over-fitting and the model relegates itself to a straight line through the curve. It is only at a moderate layer and appropriate neuron count that we are able to see the model accurate predict the data set.

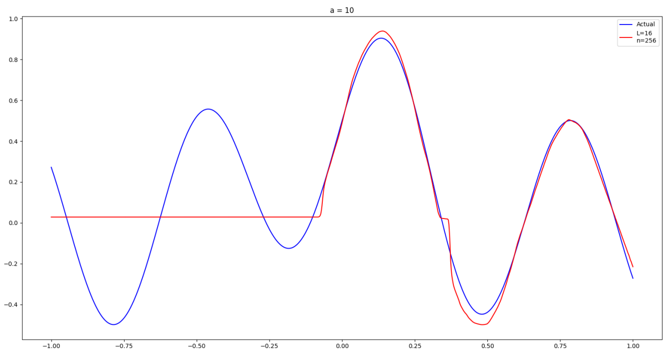
Lastly for $a = 10$, we get the following predictor models:



(a) L = 1, n = 2



(b) L = 16, n = 32



(c) L = 16, n = 256

Figure 10: Curve Fitting for $a = 10$

In this case the, the function is too complex for any model to accurate fit it, this may be due to the lack of symmetry at $x = 0$ which s why the curve is forced to fit either the left hand side or the right hand side of the graph. However, it should be noted that in figure 10(a) we are able to see that the model has a good bias predictor even if it can not correctly allocate the weights. This is indicated by the models attempt to minimize the MSE in figure 10(a) through the use of 3 Linear Functions.

8 Conclusions

It is difficult to predict an equation between the loss and the parameters due to the randomized changes in the loss function for a given number of layers and neuron. Repeating the whole code to reconstruct the neural model and subsequently the code showcases graphs which are different from the ones we have showcased, indicating that the low loss 'sweet spots' may also be a random occurrence. However, in the case of $a = 1$ and somewhat in the case of $a = 5$ we can see that the loss is minimized between 4 to 8 layers and 8 to 16 neurons per layer. This correlation becomes increasingly weaker with an increase in function complexity. Hence for relatively straight-forward function it is safe to assume that:

$$\epsilon \propto \Gamma_L \Gamma_n \tag{2}$$

where ϵ represents the loss, Γ_L represent an inverted bell curve with a Gaussian mean distribution between 4 and 8, and Γ_n represents an inverted bell curve with a Gaussian mean distribution between 8 and 16.

9 Recommendations

To further enhance the efficacy of the results, the experiment should be modified in the following ways. Firstly, this experiment should initially be done on simple sine and cosine functions which do not vary from a set pattern with changing x-axis value to get a better baselines results. Secondly, alternative Machine Learning Libraries should also be explored as well as various activation functions (such as Tanh and Sigmoid) which were not used in this set up. Thirdly, different optimizers like SGD should also be explored and their impact on the convergence should be taken into account. Fourthly, the layer count for these experiments should not be greater than 16 or 32 as it makes for an unrealistic deep learning model with very little practical applications. Lastly for further optimization, the model should be assessed without the restriction of a constant number of neurons in every layer.