COMPARE SHALLOW AND DEEP NEURAL NETWORKS FOR FUNCTION APPROXIMATION

Yingqi Ma UNI: ym2926
Zhan Shu UNI: zs2584
Yuan Dou UNI: yd2676
Qingyang Gu UNI: qg2208
Columbia University, Department of Electrical Engineering

ABSTRACT

Based on the materials provided in the class, we are interested in figuring out why deep neural networks are preferable to shallow networks in approximation problems. Based on the paper titled "*Why Deep Neural Networks for Function Approximation*?" by Shiyu Liang and R. Srikant [1], we design experiments to prove the theorems and results given in the paper. First, we consider univariate functions on a bounded interval and use a neural network to achieve an approximation error of ε uniformly over the interval. We show that for a given structure in the paper, the mean absolute error is less than or equal to ε. And with the same number of neurons, we compare the mean absolute error of deep networks and shallow networks and show that the performance of deep networks is better than shallow networks. We then extend these results to multivariate functions. Our results are derived from neural networks using fully-connected rectifier linear units (ReLUs) networks as our architecture.

1. Introduction

In recent years, neural networks have become an increasingly popular tool for function approximation in various domains such as computer vision, natural language processing, and finance [2]. The architecture of neural networks can be divided into two categories: shallow and deep. Shallow neural networks typically consist of one hidden layer, while deep neural networks have multiple hidden layers. While deep neural networks have shown impressive performance in many applications, the question remains whether deeper architectures are always superior to shallower ones for function approximation tasks.

This report aims to compare the performance of shallow and deep neural networks for function approximation. The reference paper focuses mainly on the size limit of multilayer neural networks versus a sufficient number of neurons to guarantee an ε-approximation and shows that shallow networks (i.e., networks whose depth does not depend on ε) with $o(\log(1/\varepsilon))$ layers require $\Omega(\text{poly}(1/\varepsilon))$ neurons, whereas deep networks (i.e., networks whose depth grows with $1/\varepsilon$) with $\Theta(\log(1/\varepsilon))$ layers only needs $O(\text{poly}\log(1/\varepsilon))$ neurons. That is to say, shallow networks require exponentially more neurons than deep networks to achieve the same mean absolute error for function approximation. Through experiments, we prove that for the given structure, the mean absolute error is less than ε. We choose fully connected ReLU networks that satisfy the upper bound for ReLU neurons stated in the theorem of the reference paper as our architecture for the deep neural network. Then we compare the mean absolute error of deep networks and shallow networks and show that the performance of deep networks is better than shallow networks.

The outline of this paper is as follows. In Section 2, we present detailed definitions of Binary Step Unit (BSU) and ReLU, as well as theorems to be tested. In Section 3, we present the problem statement. In Section 4, we present the progress and results of the experiments. Conclusions and discussions are presented in Section 5.

2. Preliminaries and Theorems

In this section, we explain in detail the mathematical theorems that are implemented and tested in our deep neural network (DNN) experiments.

The theoretical explanation is primarily based on "*Why Deep Neural Networks for Function Approximation*?" by Shiyu Liang and R. Srikant, where the authors discuss the use of DNN in function approximation tasks. The authors point out that using a single hidden layer network to approximate a function may require a large number of hidden units, which can lead to overfitting and poor generalization performance. In contrast, DNNs with multiple hidden layers have been shown to require fewer hidden units to achieve the same level of accuracy, due to their ability to learn hierarchical representations of the input data. The paper then proves the advantages of using DNNs in function approximation by producing theoretical and empirical results.

The authors list eleven theorems, each of which expands a mathematical function using feedforward neural networks with two types of activation functions: binary step unit and rectifier linear unit. In this paper, we focus on Theorem 1, Theorem 2, and Theorem 8 proposed by Liang & Srikant. Each activation function and theorem used in our experiment is explained in the sections below.

2.1 Binary Step Unit and Binary Expansion

The binary step unit is a function that takes in a scalar input and returns either 0 or 1, depending on whether the input is less than or greater than a threshold value. In general, the binary step unit can be defined as the function below where 0 is the threshold value.

$$f(x) = \begin{cases} 0, & if\ x < 0 \\ 1, & if\ x \geq 0 \end{cases}$$

or formally,

$$\sigma(x) = \mathbb{I}\{x \geq 0\}, \qquad x \in R$$

In other words, the binary step unit "activates" or outputs a 1 if the input is greater than 0, and "deactivates" or outputs a 0 if the input is less than or equal to 0. The threshold value of 0 can be adjusted to control the sensitivity of the function.

In the context of this paper, this activation function is used for the binary expansion of decimal numbers with a limited number of bits. The expanded decimal number x is between 0 and 1.

The expansion is performed with n binary step units, where n is the number of layers in the neural network to find the binary expansion of n bits. Each binary step unit produces a single bit by evaluation if the input is less than or greater than ½, setting ½ as the threshold value. The input to each binary step unit depends on the weight of the binary bit. The input to the binary step unit decreases by ½ for each less significant bit. This can be expressed by the equation:

$$f(x_i) = \begin{cases} 0, & if\ x_i < \frac{1}{2} \\ 1, & if\ x_i \geq \frac{1}{2} \end{cases}$$

where

$$x_i = \frac{x}{2^i}$$

The result of the binary expansion is the sum of the output of each binary step unit. The operation of the string of n+1 binary step units can therefore be summarized in Figure 1 below.

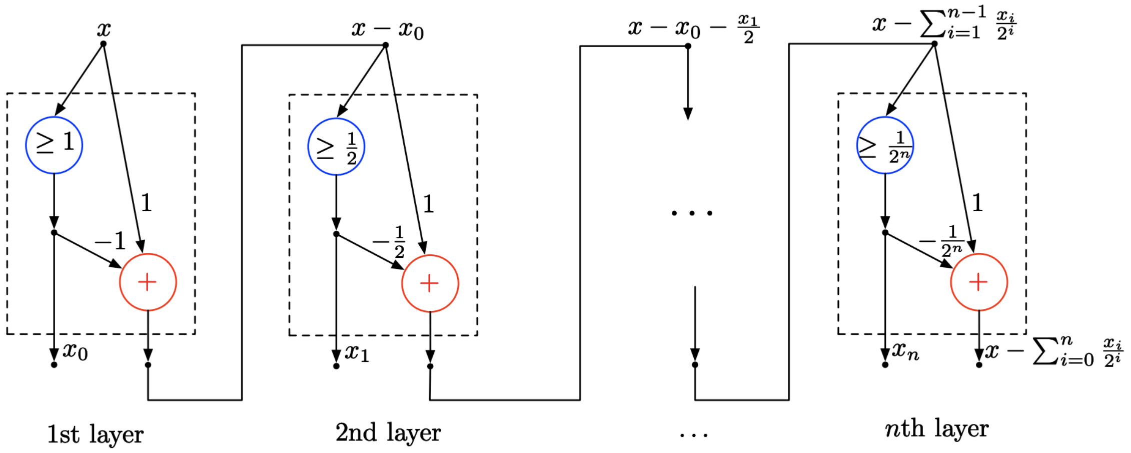


Figure 1: n-layer neural network to find the linear expansion of input x ∈ [0, 1] [1].

The complete expansion can be expressed as the equation below, where x is the original decimal number, and the right-hand side is the result after expansion.

$$x \approx \sum_{i=0}^{n} \frac{x_i}{2^i}$$

Since the precision is bounded by the limited number of layers of the neural network, the right-hand side is not exactly equal to the left-hand side, producing an error Ɛ. The number of layers can be adjusted based on the desired accuracy. A larger n results in a more accurate binary expansion output.

The implementation of this activation function with coding will be explained in more detail in the Experiment section.

2.2 Rectifier Linear Unit

The rectified linear unit (ReLU) is a piecewise linear function that returns the input if it is positive, and returns zero if it is negative. In other words, the ReLU "rectifies" or clips off the negative part of the input signal. Formally, the ReLU can be defined as:

$$\sigma(x) = max\{0, x\}, \qquad x \in R$$

The advantage of using ReLU is suitable for large-scale deep learning models because of its simplicity, requiring a single comparison and a single multiplication. The function also preserves the nonlinearity of the output into the network, making it useful in complex models for it alleviates the vanishing gradient problem in DNNs [3].

The use of ReLU in the context of this paper will be explained separately for each theorem below.

2.3 Approximation of Square Function

Theorem 1 in Liang and Srikant introduces the method of using a multilayer neural network to approximate the square function where the input value is between 0 and 1. The function before approximation is expressed as

$$f(x) = x^2, \qquad x \in [0,1]$$

This multilayer neural network uses both binary step units and ReLUs. The number of layers in the neural network, the number of binary step units, and the number of ReLUs are all n, a natural number that is decided based on the acceptable accuracy Ɛ. The relationship is expressed as the equation below.

$$n = O\left(log\frac{1}{\varepsilon}\right)$$

The approximation of the original square function f(x) is denoted as $\tilde{f}$(x), and is based on the n-bit binary expansion explained in Section 2.1. The expression of $\tilde{f}$(x) is therefore

$$\tilde{f}(x) = f\left(\sum_{i=0}^{n}\frac{x_i}{2^i}\right)$$

The operation of the neural network then rewrites $\tilde{f}$(x) as

$$\tilde{f}(x) = \left(\sum_{i=0}^{n} \frac{x_i}{2^i}\right)^2 = \sum_{i=0}^{n}\left[x_i \cdot \left(\frac{1}{2^i}\sum_{j=0}^{n}\frac{x_j}{2^j}\right)\right] = \sum_{i=0}^{n} max\left(0, 2(x_i - 1) + \frac{1}{2^i}\sum_{j=0}^{n}\frac{x_j}{2^j}\right)$$

The first equality directly substitutes the input x in the original square function *f*(x) with the binary expansion of x. The second equality uses the distributive law of multiplication to convert the product of sums (since binary expansion is a sum of n terms) on the left-hand side to a sum of products by multiplying each term (or bit) in the first binary expansion, x_i, with each term (or bit) in the second binary expansion, x_j.

The third equality in the equation above uses ReLUs to only add products into the sun when both multipliers, x_i and x_j, equal 1. This is because all bits in the binary expansion can only either be 0 or 1, and therefore the product is 0 unless both multipliers are 1. Neglecting terms with value of zero does not affect the value of the sum.

2.4 Polynomial Function Approximation

Theorem 2 in Liang and Srikant uses a similar approach as explained in Section 2.3 to approximate polynomial functions using finite layers of neural networks. The function before the expansion is expressed as

$$f(x) = \sum_{i=0}^{p} a_i x^i, \qquad x \in [0,1]$$

where p is the maximum degree of the polynomial, and i is the degree of each term.

Similar to the previous section, this method uses both binary step units to obtain the n-bit binary expansion of the function and ReLUs to construct a neural network of n layers. An additional function *g* is defined as

$$g_i = x^i, \qquad where \;\; i \in [0, p], i \in N$$

to substitute input-relevant components in the polynomial terms.

The number of layers is designed to get the desired accuracy Ɛ. In contrast to the neural network for square functions, the number of layers for the polynomial function, n_{DNN}, is different from the number of binary step units, n, and the number of ReLUs, n_{ReLU}. They also depend on the degree of the polynomial instead of only on the accuracy, as shown in the equations below.

$$n = \mathcal{O}\left(log\frac{p}{\varepsilon}\right)$$

$$n_{ReLU} = \mathcal{O}\left(p \cdot log\frac{p}{\varepsilon}\right)$$

$$n_{DNN} = \mathcal{O}\left(p + log\frac{p}{\varepsilon}\right)$$

The approximation of each g is achieved by implementing the equation below that is similar to the approach in Section 2.3.

$$g_{m+1}\left(\sum_{i=0}^{n}\frac{x_i}{2^i}\right) = \sum_{j=0}^{n}\left[x_j \cdot \frac{1}{2^j}g_m\left(\sum_{i=0}^{n}\frac{x_i}{2^i}\right)\right] = \sum_{j=0}^{n} max\left[0, 2(x_j - 1) + \frac{1}{2^j}g_m\left(\sum_{i=0}^{n}\frac{x_i}{2^i}\right)\right]$$

The proposed structure of the neural network to implement a polynomial function this approach is illustrated in Figure 2.

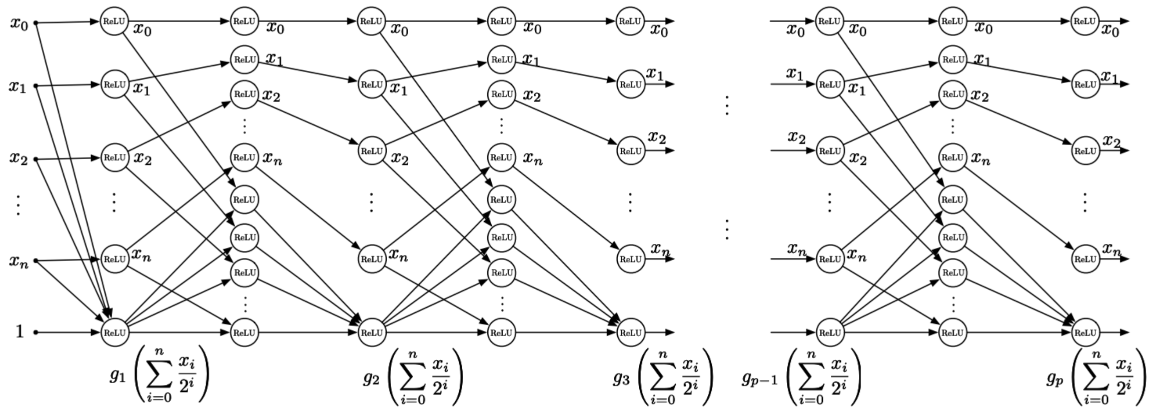

x_0
x_1
x_2
x_n
1
ReLU
x_0
x_1
x_2
x_n
$g_1\left(\sum_{i=0}^{n}\frac{x_i}{2^i}\right)$
$g_2\left(\sum_{i=0}^{n}\frac{x_i}{2^i}\right)$
$g_3\left(\sum_{i=0}^{n}\frac{x_i}{2^i}\right)$
$g_{p-1}\left(\sum_{i=0}^{n}\frac{x_i}{2^i}\right)$
$g_p\left(\sum_{i=0}^{n}\frac{x_i}{2^i}\right)$

Figure 2: Neutral network structure to approximate polynomial function [1].

A problem encountered when using this structure is that the number of nodes exceeds the maximum allowed size of the neural network [4]. Our proposes an alternative method using a fully-connected network, which will be explained in Section 3.

2.5 Product of Multivariate Linear Function Approximation

Theorem 8 in Liang and Srikant shows an upper bound on the network size for ε-approximation of a product of multivariate linear function. The function before the expansion is expressed as:

$$f(\boldsymbol{x}) = \prod_{i=1}^{p}\left(\boldsymbol{w}_i^T\boldsymbol{x}\right),\ \boldsymbol{x} \in [0,1]^d$$

$$W = \{\boldsymbol{w} \in \mathbb{R}^d : \|\boldsymbol{w}\|_1 = 1\} \quad \boldsymbol{w}_i \in W,\ i = 1, \ldots, p$$

First, use the deep structure shown in Figure 1 to find the binary expansion for each $x^{(k)},\ k \in [d]$, the binary expansion of $x^{(k)}$ is $\tilde{x}^{(k)} = \sum_{r=0}^{n} \frac{x_r^{(k)}}{2^r}$

Use a multilayer neural network with *n* layers and *dn* binary units in total to decode all the *n*-bit binary expansions and define

$$\tilde{f}(\boldsymbol{x}) = f(\tilde{\boldsymbol{x}}) = \prod_{i=1}^{p}\left(\sum_{k=1}^{d} w_{ik}\tilde{x}^{(k)}\right) \quad g_l(\tilde{\boldsymbol{x}}) = \prod_{i=1}^{l}\left(\sum_{k=1}^{d} w_{ik}\tilde{x}^{(k)}\right)$$

Rewrite $g_{l+1}(\tilde{x}),\ l = 1, \ldots, p-1$ into

$$g_{l+1}(\tilde{\boldsymbol{x}}) = \prod_{i=1}^{l+1}\left(\sum_{k=1}^{d} w_{ik}\tilde{x}^{(k)}\right) = \sum_{k=1}^{d}\left[w_{(l+1)k}\tilde{x}^{(k)} \cdot g_l(\tilde{\boldsymbol{x}})\right] = \sum_{k=1}^{d}\left\{w_{(l+1)k}\sum_{r=0}^{n}\left[x_r^{(k)} \cdot \frac{g_l(\tilde{\boldsymbol{x}})}{2^r}\right]\right\}$$
$$= \sum_{k=1}^{d}\left\{w_{(l+1)k}\sum_{r=0}^{n}\max\left[2(x_r^{(k)}-1) + \frac{g_l(\tilde{\boldsymbol{x}})}{2^r}, 0\right]\right\}$$

For $k = 1, \ldots, d$ *and* $\forall x \in [0, 1]^d$,

$$\left|\frac{\partial f(\boldsymbol{x})}{\partial x^{(k)}}\right| = \left|\sum_{j=1}^{p}\left[w_{jk} \cdot \prod_{i=1, i\neq j}^{p}(\boldsymbol{w}_i^T\boldsymbol{x})\right]\right| \leq \sum_{j=1}^{p}|w_{jk}| \leq p.$$

$$|f(\boldsymbol{x}) - \tilde{f}(\boldsymbol{x})| = |f(\boldsymbol{x}) - f(\tilde{\boldsymbol{x}})| \leq \|\nabla f\|_2 \cdot \|\boldsymbol{x} - \tilde{\boldsymbol{x}}\|_2 \leq \frac{pd}{2^n}$$

By choosing $n = \lceil log_2 \frac{pd}{\varepsilon}\rceil$, we can find the upper bound of $O(d\ log\frac{pd}{\varepsilon})$ binary step units, $O(pd\ log\frac{pd}{\varepsilon})$ rectifier linear units and $O(p + log\frac{pd}{\varepsilon})$ layers.

3. Problem Statement

In the paper we referenced, their problem statement is to find a neural network with bounded depth L and size N such that for any function $f(x)$ and the approximation function $\widehat{f}(x)$satisfied:

$$\min_{\tilde{f}\in\mathcal{F}(N,L)} \|f - \tilde{f}\| \leq \varepsilon.$$

Our project focuses on reproducing the proof of the theorems in Python code and testing the neural network performance. To verify the statement above, we aim to build the architecture of the neural network described in the paper and adjust its hyperparameters to match those specified in the theorem. We will then use this network to approximate various polynomial functions and compare the results to the theoretical guarantees provided by the theorem. By doing so, we hope to provide empirical evidence that supports the claims made in the paper and sheds light on the practical implications of the proposed theorem.

To set up our experiment configuration, we choose TensorFlow keras library[5] and sklearn library. Keras is a high-level neural networks API, written in Python and capable of running on top of several lower-level deep learning frameworks, such as TensorFlow. Keras makes it easy to define, train, and evaluate neural networks. It is used in our project for tasks of building the neural network architecture model. Scikit-learn is a machine-learning library for Python that provides simple and efficient tools for data mining and data analysis. It is built on top of NumPy, SciPy, and matplotlib[6]. In our project, we use sklearn for tasks involving data processes such as train/test data split and error function selection.

As stated in section 2, the paper provides an architecture that states the ReLU connection in Figure 2. However, we observe that this architecture will cause a use of more than the upper bound of the number of rectifier linear units of $O(pn)$. To our understanding of the architecture which they claimed in the paper, it will use $O(3 * p * (n + 1))$ ReLU and layer of $O(n + 1)$. This is inconsistent with the theorem's initial setting.

Therefore, according to Universal Approximation Theorem[7]: For any Bochner–Lebesgue p-integrable function f, where $f : \mathbb{R}^n \to \mathbb{R}^m$ and $\epsilon > 0$ there exists a fully connected ReLU network F with width $d_m = \max\{n + 1, m\}$ that satisfies:

$$\int_{\mathbb{R}^n} \|f(x) - F(x)\|^p \mathrm{d}x < \epsilon.$$

This is used for our experiment such that the univariate and multivariate function approximation both satisfy the condition above. Also, the fully connected ReLU network satisfies the upper bound for ReLU neurons that states in the theorem of the reference paper. Therefore, we chose the fully connected ReLU network as our architecture for the deep neural network.

4. Experiment

4.1 Square Function Approximation

First, we implement the binary expansion function structure using python. This function performs binary expansion of a decimal number x up to a given number of bits n. The binary expansion represents the decimal number as a sum of powers of 2. We initialized a numpy array result with n zeros, and then the function iterates over the bits of the binary expansion from the most significant bit (leftmost bit) to the least significant bit (rightmost bit). For each bit position, the function checks whether x is greater than or equal to 1 divided by 2 raised to the power of the current bit position plus 1. If it is, then the current bit is set to 1, and a temporary variable tmp is set to 1, otherwise, the current bit is set to 0 and tmp remains 0. Finally, the function returns the binary expansion as a numpy array.

Then we defined the function create_deep_network which takes two arguments, layers, and neurons to create a network that is suitable for our experiment. It creates a Sequential model with a specified number of neurons per hidden layer and the specified number of layers. The activation function used in the hidden layers is the rectified linear unit (ReLU) and a linear activation function is used for the output layer.

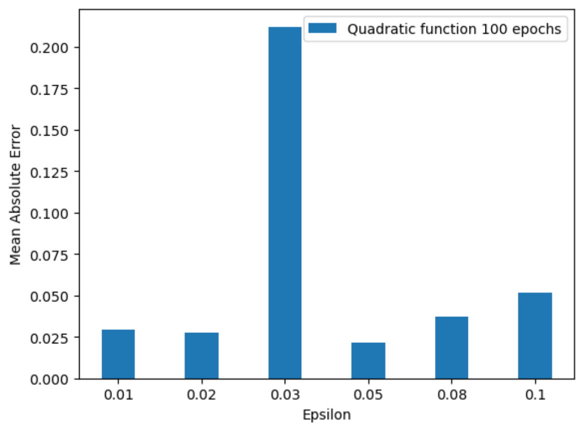
The train_and_evaluate function takes five arguments: model, X_train, X_test, y_train, and y_test. It compiles the model using the Adam optimizer with a learning rate of 0.001 and the mean absolute error (MAE) loss function which we use to calculate the error described in the paper. It then fits the model on the X_train and y_train data for a specified number of epochs and batch size. After training, the model makes predictions on the X_test data, and the MAE between the predicted values and y_test is returned. This function is used to evaluate the performance of the trained model on unseen data.

The data input for the above function is 1000 randomly generated numerical data. Note that we set here the input data x is between the range [0,1].

To test our model's error for each epsilon, we test six different epsilon situations. For each value of epsilon in the list "epsilon", and for each set of polynomial coefficients in the list "polynomial_coefficients", we generate training and testing data using the function "generate_data". Then we apply binary expansion to the input values using the function "binary_expansion" and train a neural network with an architecture defined by the function "create_deep_network".

The trained neural network is evaluated on the test data using the function "train_and_evaluate" which returns the mean absolute error between the predicted and actual values. The resulting mean absolute errors for each set of polynomial coefficients are stored in the list "deep_results".

Figure 3 shows the result of our quadratic function approximation. We test the data in six epsilons of [0.01, 0.02, 0.03, 0.05, 0.08, 0.1] with three different epochs. We can clearly observe from the graph below that the model did not converge at 100 and 200 epochs. The right-most graph is the result of the mean square error training at 500 epochs, 0.001 learning rate. The result is showing a linearity as the epsilon increases the error increase. This is reasonable since the error should have an upper bound of epsilon. As we increase epsilon, the layer of the model is decreasing and hence the error will show an increasing trend. The neuron and layers of the model are consistent with the theorem and we discover the fully connected ReLU model is suitable for the experiment and performs as expected.



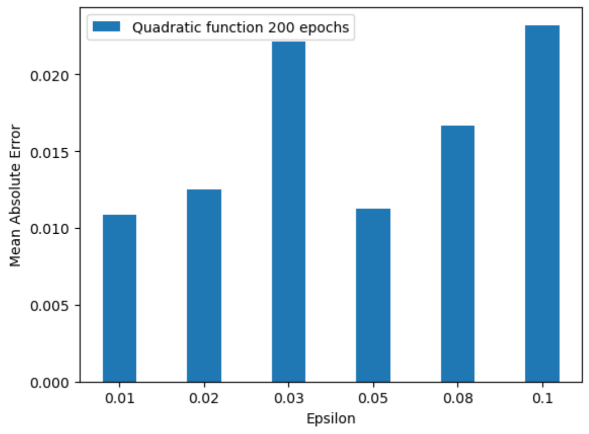


Figure 3 a(left) and b(right): Approximation error quadratic function running 100 and 200 epochs.

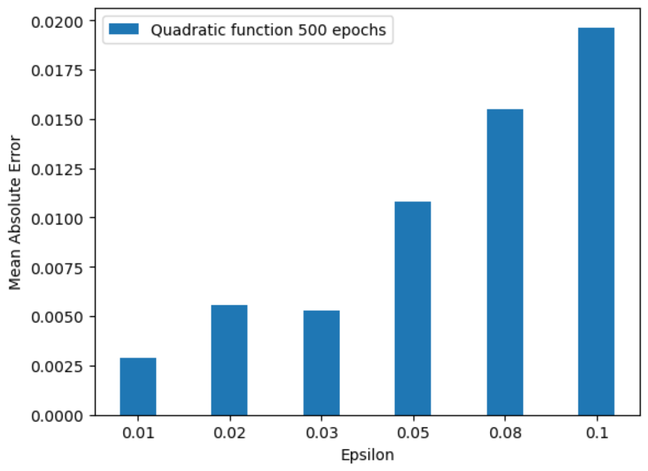


Figure 3 c: Approximation error quadratic function running 500 epochs.

4.2 Polynomial Function Approximation

According to Theorem 2, given an error bound ε and a polynomial function

$$f(x) = \sum_{i=0}^{p} a_i x^i,\ x \in [0,1]\ \textit{and}\ \sum_{i=1}^{p} |a_i| \le 1$$

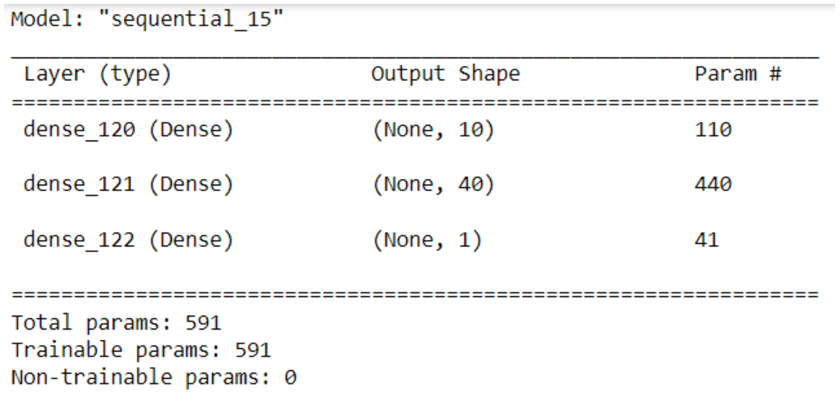
where p represents the highest degree of the polynomial, there exists a multilayer neural network with $O(p + log_2 \frac{p}{\varepsilon})$ layers, $O(log_2 \frac{p}{\varepsilon})$ binary step units and $O(plog_2 \frac{p}{\varepsilon})$ rectifier linear units such that the mean approximation error is upper bounded by ε.

To compare the performance of shallow and deep neural networks on approximating the polynomial function, we randomly generated 1000 data points for each given p and ε. The p polynomial coefficients are also randomly generated and normalized so that their sum of absolute values is equal to 1, which satisfies the condition in the theorem. Given x and the polynomial coefficients, $f(x)$ is calculated accordingly using the polynomial function. These 1000 data points are then split into 80% for training and 20% for testing.

Each x will then goes through the binary step unit function to get the n-bit binary expansion, where

$$n = \lceil \log \tfrac{p}{\varepsilon} \rceil + 1$$

For deep neural networks, we use a fully connected neural network structure instead of the multilayer neural network described in the paper. In this case, a deep neural network has one input layer, $p - 1$ hidden layers, and one output layer, with the input layer and each hidden layer having $log_2 \frac{p}{\varepsilon}$ rectifier linear units. The output layer has a single linear unit, representing the output $\widehat{f}(x)$.



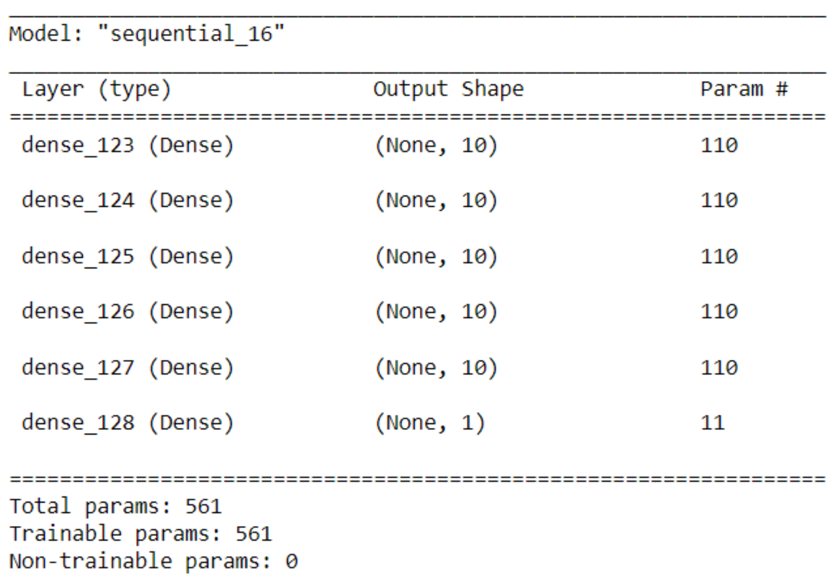
```
Model: "sequential_15"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_120 (Dense)           (None, 10)                110

 dense_121 (Dense)           (None, 40)                440

 dense_122 (Dense)           (None, 1)                 41

=================================================================
Total params: 591
Trainable params: 591
Non-trainable params: 0
```

Figure 4: Structure of a shallow neural network.



```
_________________________________________________________________
Model: "sequential_16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_123 (Dense)           (None, 10)                110

 dense_124 (Dense)           (None, 10)                110

 dense_125 (Dense)           (None, 10)                110

 dense_126 (Dense)           (None, 10)                110

 dense_127 (Dense)           (None, 10)                110

 dense_128 (Dense)           (None, 1)                 11

=================================================================
Total params: 561
Trainable params: 561
Non-trainable params: 0
```

Figure 5: Structure of a deep neural network.

The above two figures show the structures of a shallow and deep neural network when $p = 5$ and $\varepsilon = 0.01$. In this case, a deep neural network would have 5 layers and an output layer. Each layer would have 10 rectifier linear units. A shallow neural network has an input layer, a hidden layer, and an output layer, with the total number of neurons equal to the deep neural network. From our observations, the total number of trainable parameters is equal for shallow and deep neural networks if the total number of neurons is fixed as p and ε varies.

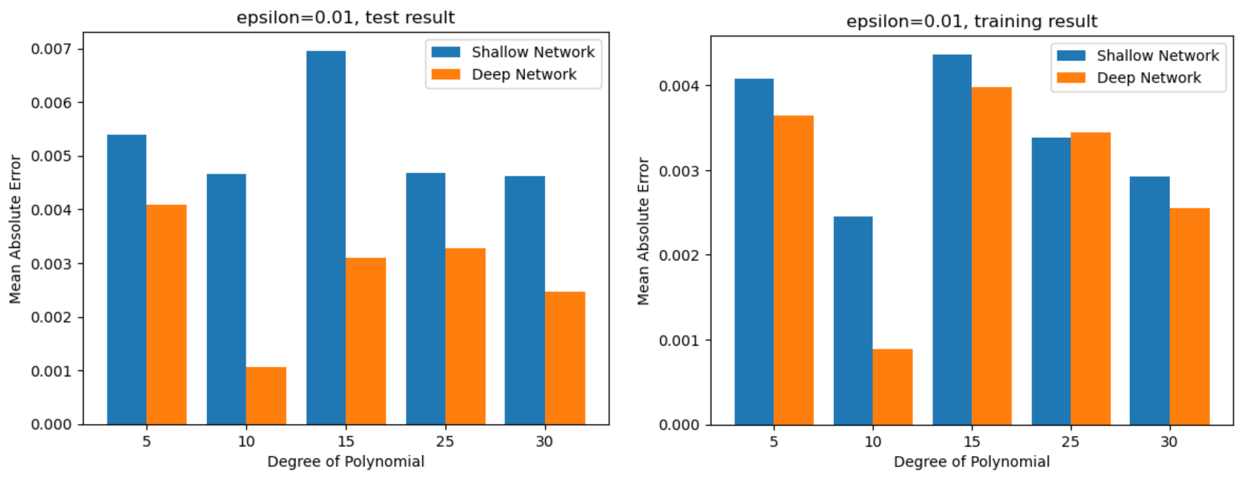


Figure 6: Comparison of shallow and deep NN when $\varepsilon = 0.01$.

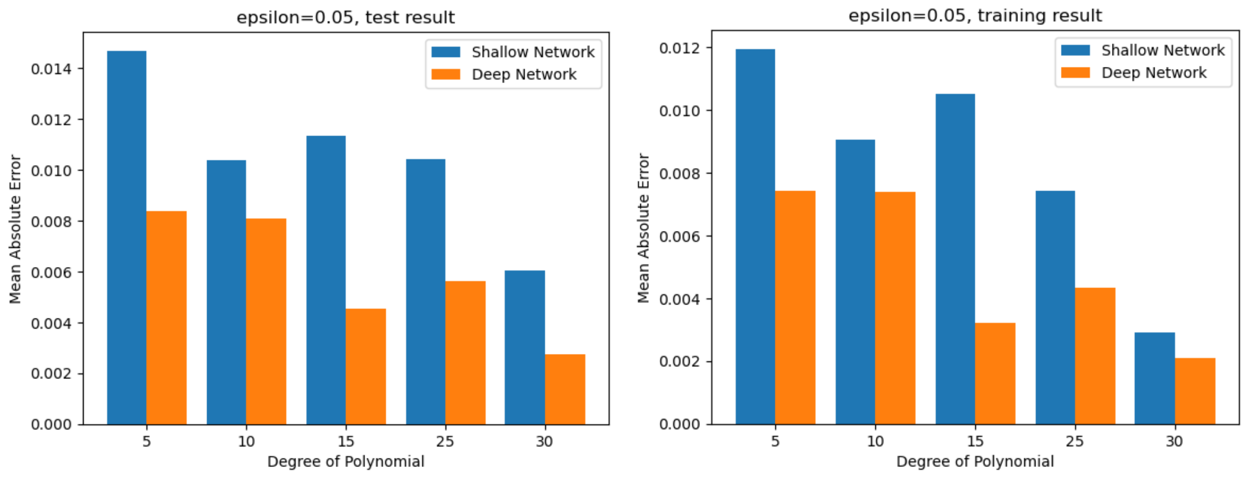


Figure 7: Comparison of shallow and deep NN when $\varepsilon = 0.05$.

We conducted two sets of experiments, with $\varepsilon = 0.01$ and $\varepsilon = 0.05$. For each experiment, we gradually increase p, the highest degree of the polynomial, and compare both the mean absolute error on the training dataset and testing dataset. As shown in figure 6, when $\varepsilon = 0.01$, the mean absolute error on the training dataset is similar for shallow and deep neural networks, with deep neural networks achieving a slightly lower mean absolute error except when $p = 25$. However, deep neural networks achieve a much lower mean absolute error on the testing dataset compared to shallow neural networks with the same number of rectifier linear units and a similar number of trainable parameters. We can also observe that the mean absolute error of both shallow and deep neural networks is smaller than ε, which experimentally proved the theorem.

Figure 7 shows the mean absolute error on the training and testing dataset for shallow and deep neural networks when $\varepsilon = 0.05$. Compared to the previous results, the mean absolute error of in both shallow and deep neural networks increases, since as ε increases, n is smaller, and there are fewer rectifier linear units in each layer, which means fewer trainable parameters.

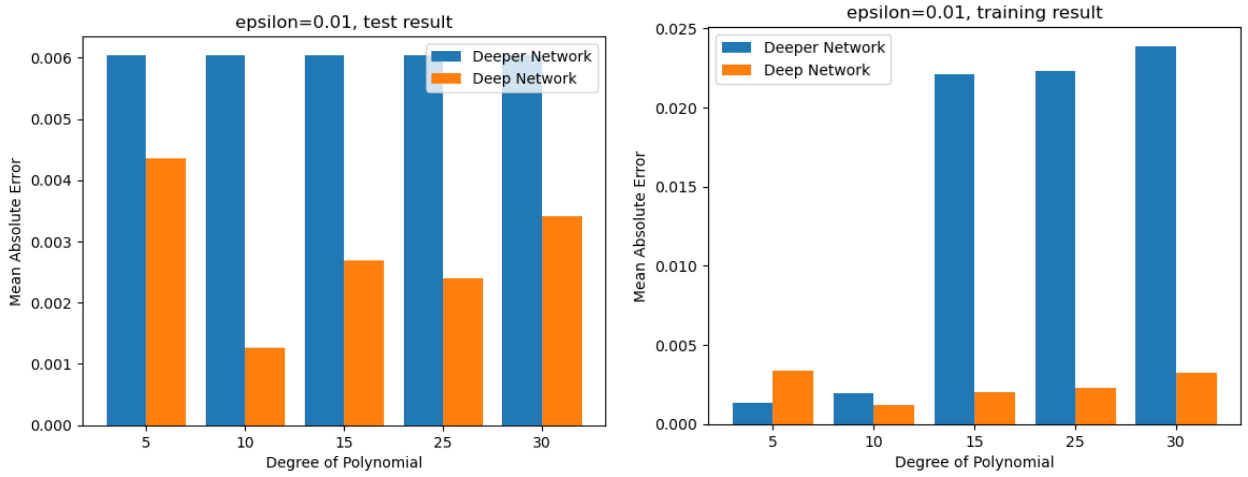


Figure 8: Comparison of deep NN and deeper NN with 2 times the number of hidden layers.

From the previous experiments, we find that deep neural networks generally achieve a lower mean absolute error compared to shallow neural networks no matter what p is. Thus, we conducted another experiment to see if a deep neural network structure with more hidden layers, more rectifier linear units, and more trainable parameters can achieve a lower mean absolute error. As shown in Figure 8, we find that deep neural networks with 2 times the number of hidden layers actually result in a much higher mean absolute error on the training dataset and testing dataset, though the mean absolute error is still smaller than the upper bound ε.

4.3 Multivariate Function Approximation

According to Theorem 8, given an error bound ε and a multivariate polynomial function

$$f(\boldsymbol{x}) = \prod_{i=1}^{p} (\boldsymbol{w}_i^T \boldsymbol{x}),\ \boldsymbol{x} \in [0,1]^d$$

where p represents the highest degree of the polynomial, d is the dimension of x, and

$$W = \{\boldsymbol{w} \in \mathbb{R}^d : \|\boldsymbol{w}\|_1 = 1\}$$

there exists a multilayer neural network with $O(p + log_2 \frac{pd}{\varepsilon})$ layers, $O(log_2 \frac{pd}{\varepsilon})$ binary step units and $O(p \cdot d \cdot log_2 \frac{pd}{\varepsilon})$ rectifier linear units such that the mean approximation error is upper bounded by ε.

To compare the performance of shallow and deep neural networks on approximating the multivariate polynomial function, we randomly generated 1000 data points for each given p, d and ε. The p weights are also randomly generated and normalized so that their L1 norm is equal to 1, as stated in the theorem. Given x and the p weights w, $f(x)$ is calculated accordingly using the multivariable polynomial function. These 1000 data points are then split into 80% for training and 20% for testing.

Different from the previous two experiments, we directly feed x into the neural network instead of letting it pass through the binary step units because x has a shape of (d,). An n-bit binary expansion of x would result in a shape of (d, n), which in this case, we'll need to use convolution layers instead of dense layers to build the neural network. Since the paper does not specify using a convolution neural network structure to approximate multivariate functions, and for the simplicity and effectiveness of training, we decide to stick to dense neural networks and not transform x.

Following the upper bounds for the number of layers and rectifier linear units in the theorem, we construct deep neural networks with one input layer, $p - 1$ hidden layer, and one output layer, with the input layer, and each hidden layer having $dlog_2 \frac{pd}{\varepsilon}$ rectifier linear units. The output layer has a single linear unit, representing the output $\widehat{f}(x)$. Shallow neural networks have one input layer with $dlog_2 \frac{pd}{\varepsilon}$ rectifier linear units, one hidden layer with $(p - 1) \cdot dlog_2 \frac{pd}{\varepsilon}$ rectifier linear units, and an output layer with a single linear unit, representing the output $\widehat{f}(x)$.

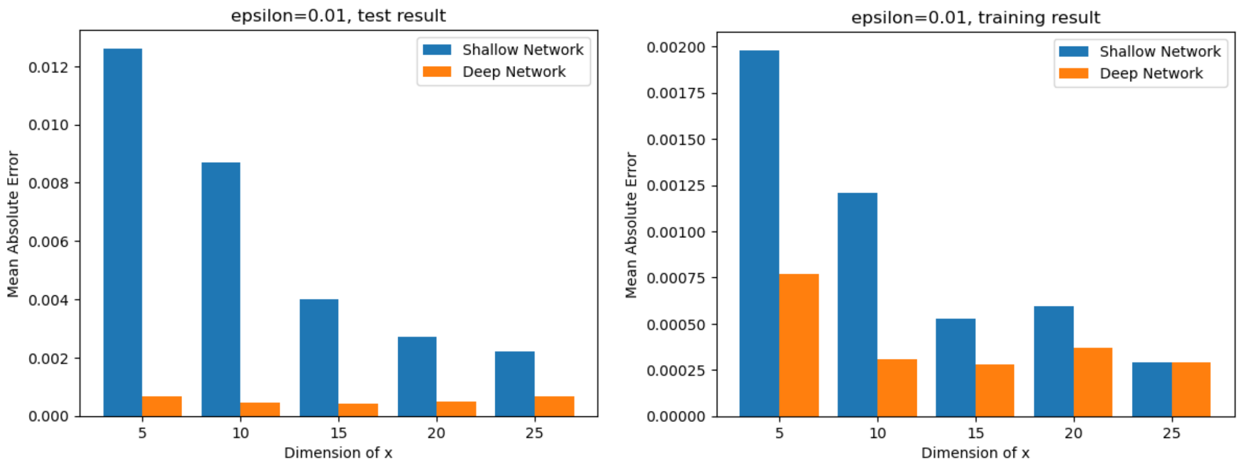


Figure 9: Comparison of shallow and deep NN as the dimension of x varies.

Figure 9 shows the training and test result of shallow and deep neural networks on multivariate function approximation as d, dimension of x increases, while p, the highest degree of polynomial, remains constant at 10. For the training result, deep neural networks achieve lower mean absolute errors compared to shallow neural networks except when d = 25, where the mean absolute errors achieved by shallow and deep neural networks are almost the same. However, when looking at the test result, we can easily observe a large difference between deep and shallow neural networks, with deep neural networks achieving a much lower mean absolute error, and it is also much smaller than ε. Furthermore, we can observe that as the dimension of x increases, mean absolute errors achieved by deep neural networks remain constant at about 0.001, while mean absolute error achieved by shallow neural networks gradually decreases. Since p remains constant at 10, the number of layers of deep neural networks remains constant, while the number of rectifier linear units in each layer, as well as the total number of rectifier linear units increases. With an increase in the total number of neurons, the number of trainable parameters also increases, and that is probably why the performance of shallow neural networks gets better as d increases.

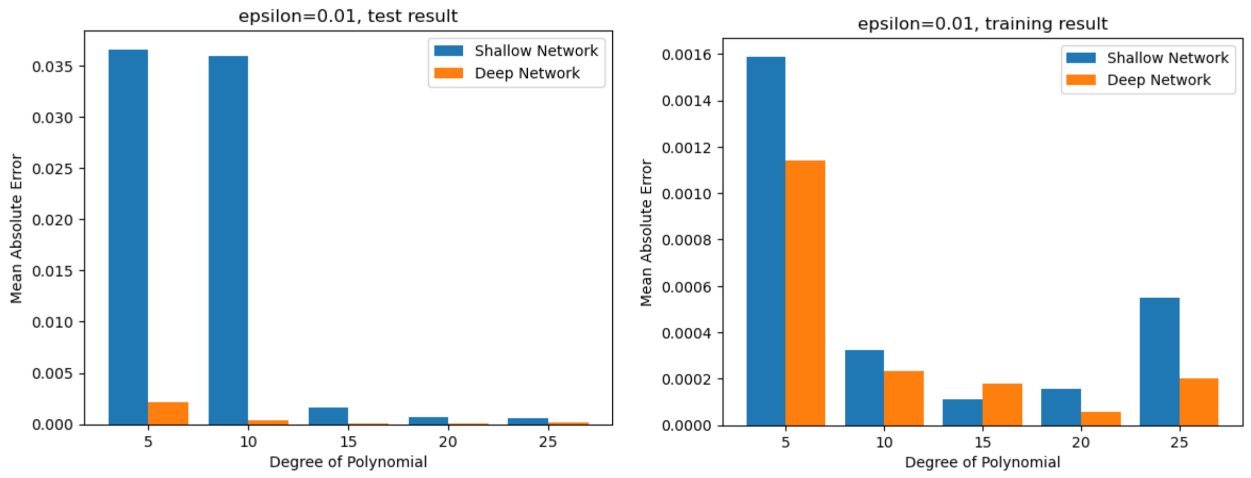


Figure 10: Comparison of shallow and deep NN as the highest degree of polynomial varies.

We then set d, the dimension of x, to a constant of 10, and increase p, the highest degree of polynomial to see the effect of p on the performance of shallow and deep neural networks. As shown in Figure 10, deep and shallow neural networks achieve similar mean absolute errors as p increases. However, deep neural networks perform much better than shallow neural networks on the testing dataset, since deep neural networks can provide more generalization. Furthermore, as p increases, mean absolute errors achieve by both shallow and deep neural networks decrease, since as p increases, there are more layers in the deep neural network model, and a larger number of rectifier linear units in both shallow and deep neural networks, which leads to a larger number of trainable parameters.

5. Conclusion and Discussion

In summary, we successfully implemented the multilayer neural network that reproduces the proof of the theorem from the paper "*Why Deep Neural Networks for Function Approximation*?" by Shiyu Liang and R. Srikant. The paper suggested that for the same bounded neuron network, the model with deeper architecture has better performance, thus lower error than the shallower model. In our experiments, we observed that when increasing the model to a deeper network architecture, the performance of the model decreases drastically. Therefore, there exists an upper bound of the model that is thus proved. We also explored the relationship between the number of parameters and the effect on the error. We observed that in the shallower model, there are more parameters than the deeper model, however, the performance is still better for the deep neural network model. This is also an interesting finding as we expected approximately the same performance if parameter numbers are approximately the same. Furthermore, we also observe that if we use more layers than required for a deep neural network model, the performance actually gets worse, since we don't need that many trainable parameters for function approximation. Another set of experiments we did is to compare shallow and deep neural networks on approximating products of multivariate polynomial functions. We observed that deep neural networks perform much better than shallow neural networks, and the mean absolute error is much smaller than the error bound.

6. Future Work

In this paper, we focus on the comparison of fully-connected deep neural networks with shallow neural networks with the same number of neurons on function approximation. However, for each function, there may be structures other than fully-connected neural networks that can achieve better performance. In the future, we will be working on modifying structure, tuning parameters, adjusting learning rate, optimizers, etc. to find lower mean absolute errors for different function approximation tasks.

7. Reference

[1] Liang, Shiyu & Srikant, R. (2017). Why Deep Neural Networks for Function Approximation?. ICLR.
[2]Tran, D. T., Kanniainen, J., Gabbouj, M., & Iosifidis, A. (2021). Bilinear Input Normalization for Neural Networks in Financial Forecasting. ArXiv.
[3]Glorot, X., Bordes, A., & Bengio, Y. (2011). Deep sparse rectifier neural networks. In Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (pp. 315-323). PMLR.
[4]Andoni, Panigrahy, R., Valiant, G., & Zhang, L. (2014). Learning polynomials with neural networks. ICML.
[5]https://www.tensorflow.org/learn
[6]https://keras.io/about/
[7]https://en.wikipedia.org/wiki/Universal_approximation_theorem