Project 3: Expressivity of Neural Networks

Maxwell Patterson

11 May 2023

Introduction

The rise of neural networks as a tool has led to many technological advancements that have added significant value to society. Things such as voice and image recognition, medical diagnoses, and targeted marketing are a few examples of concepts that have seen significant improvements in performance and application from the application of neural networks. AI and deep learning unlock a whole new realm of what is possible within mathematics, as the computer is able to learn to differentiate among different representations in data that can reveal important trends and observations from some data set. In order to accomplish this, neural networks calculate, with the use of input and output data, some sort of pattern that can be applied to these situations such as voice recognition and medical diagnoses. In this project, we are tasked with exploring the nature in which neural networks can be applied to the approximation of functions. Overall goals of the project include developing an understanding of the training process dynamics, the ways in which the depth and width of the neural network influence the approximation, challenges and takeaways from this investigation.

Background

Perceptron

While modern neural networks can have millions of layers, the first neural network had, naturally, only one layer. In July of 1958, Frank Rosenblatt revealed his prized perceptron, which set the stage for which modern day AI was built upon. Rosenblatt coined the perceptron as being the "first machine which is capable of having an original idea" (Cornell). The perceptron is a type of single-layer neural network that is inspired from the collaborative nature of how neurons work in the brain. It operated by labeling inputs in two ways, such as left or right, or man or woman. In case of an incorrect prediction, the algorithm adjusts itself to improve the accuracy of future predictions. After the completion of thousands or millions of iterations, the neural networks gets more precise over time in order to obtain some valuable result.

Weights and bias play a critical role in establishing the relationship between inputs and outputs in the context of perceptrons. A perceptron computes the weighted sum of the input features,

$$f(x) = w_1x_1 + w_2x_2 + ... + w_nx_n + b$$

$x_1, x_2, ..., x_n$ represent the inputs, $w_1, w_2, ..., w_n$ represent the weights associated with these inputs, and b is the bias term. The weights and bias are the keys that drive the neural network, as they are flexible controls that establish the model and allow for optimal relationships to be understood during the learning process. Bias allows for the offsetting of any constant in the data, which enables the perceptron to generate more accurate result.

The result that the perceptron generates, known as the decision boundary, is calculated using the input data, the associated weights in the network, and the bias term. This is done through the linear combination of these factors. The decision boundary can be defined as the set of points x such that

$$f(x) = w_1x_1 + w_2x_2 + ... + w_nx_n + b = 0$$

In this case, the sign of the function $f(x)$ determines the significance of the output. For example, a positive value could be associated to the left direction and a negative value associated to the right direction. The flexibility of the perceptron is dictated by the weights and biases it possesses. The model is able to learn which input features are more important than others by assigning proper weight values to each input. Coupled with the bias term allow for certain constant to be offset in the data, the perceptron is able to pick up certain trends in the input data. Ultimately, the perceptron's power lies in its ability to find an adequate decision boundary through the manipulation of weights and biases, highlighting how important these concepts are in the process of neural networks.

Activation Functions In Play

Activation functions allow for neural networks to learn and understand complicated patterns between inputs and outputs by presenting the possibility of non-linearity into the network. Non-linear networks are much more powerful than linear networks as they are able to capture much deeper and more profound correlations in data. There are many usable activation functions today: the Sigmoid function, the tanh function, Reduced Linear Unit function, or ReLU, and LeakyReLU, and Exponential Linear Unit function, or ELU to name a few. The choice of activation function depends on the specific architecture of the network. It will be looked into later in the project as to how each of these activation functions is best suited for certain types of situations.
Sigmoid : a smooth, S-shaped curve that maps input values to values in the range of 0 to 1. Mathematically, the function is defined as

$$f(x) = \frac{1}{1 + e^{-x}}$$

While the sigmoid function is sufficiently utilized in the outputs of binary situations, it can struggle when dealing with more hidden layers due to vanishing gradient type of issues. In these erroneous scenarios, the gradient approaches zero or some large, positively or negatively, constant that hinders the learning ability of the network.

tanh: smooth like the sigmoid function, but maps input values to the range of -1 to 1 instead of 0 to 1. Mathematically, the function can be defined as

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

The tanh function is effective in that its output is centered around zero, which can help deal with the vanishing gradient issue to a degree but still deals with the issue of blow-up values.

ReLU/LeakyReLU: a piecewise linear function that returns the input value if it is positive, and zero, or a small negative value associated the input, if the value is negative. Mathematically, the function is defined as

$$ReLU(x) = max(0, x)$$

ReLU helps deal with the vanishing gradient issue since the gradient is constant for positive input values and cannot spiral in or out. The dying ReLU problem does exist however, in which the neurons deactivate for negative inputs which makes it difficult for the network to learn and adapt over time.

ELU: variation of the ReLU function that smooths the curve to the left of the x-axis. Mathematically, the function can be defined as

$$ELU(x) = x, if x > 00$$

$$ELU(x) = \sigma * (e^x - 1), if x \leq 0$$

Here, σ is some positive constant. ELU helps to address the vanishing gradient issue and upholds a stronger curve.

The differences in application and results obtained using these approximation functions will be discussed later.

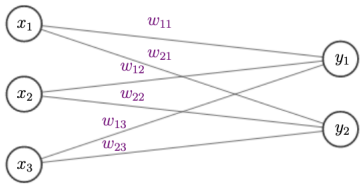
Backpropagation

Backpropagation is an essential tool that is used in the training of neural networks, especially ones that have a multitude of layers. Backpropagation calculates the loss function's gradient with respect to the weights and biases of the neural network, which then enables the network to update it's weights and biases in a productive manner that increases its accuracy. The process of backpropagation contains a forward and backward direction. In the forward direction,

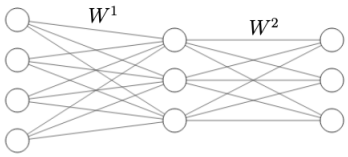
input data is fed into the neural network, which transmits the signal through the network for it to update itself. In the backwards direction, error is calculated by analyzing the difference between predicted and actual outcomes. Then, the error value is used to update the network weights via gradient descent. To accomplish this, a loss-function will be defined that calculates the error value. Since gradient descent is used to minimize the error, its derivative to the weight matrix is obtained and can be multiplied with some positive value, σ, and subtracted value to complete one step. σ is also known as the learning rate. It is essential to set an adequate learning rate, as one that is too high will help the model learn faster, but opens the possibility of a failure to converge so that the network does not learn anything. If the learning rate is too low, the training process may take too much time. Mathematically, using W as the weight matrices, this process looks like

$$W_{new} = W_{old} - \sigma * \frac{\delta E}{\delta W}$$

A single layer neural network is pictured below, with inputs x_1, x_2, x_3, weights $w_{11}, w_{21}, w_{22}, w_{13}, w_{23}$ and outputs y_1, y_2



Let's move on to neural networks with multiple layers. Consider a shallow neural network with two layers. Here is a visual representation of this kind of network.



Here, the output will depend on the W^1 and W^2 matrices, so

$$y = \sigma_2(W^2\sigma_1(W^1x))$$

Gradient descent is used to update W^2 in order to minimize the error gained from the loss function. To minimize the error, E, the formula in the last sentence is used and embedded into the error. To simplify things a bit, let's say that $y_1 = \sigma_1(W^1x)$ The derivative is equal to the following,

$$\frac{\delta}{\delta W^2}E(\sigma_2(W^2y_1)) = \frac{\delta}{\delta y}E(\sigma_2(W^2y_1)) * \frac{\delta}{\delta W^2}y(\sigma_2(W^2y_1))$$

This can be simplified by setting $e = (d - y)$ and $\theta = e. * \sigma_2'(W^2 y_1)$ so that

$$W^2 = W^2 + \sigma * \theta * y_1^t$$

The values obtained from the vector θ are used to assign output values. Next, these values are backpropagated to the hidden layer using the weights in W^2 in order to obtain error values. This error value is used to obtain the updated form of W_1,

$$W^1 = W^1 + \sigma * \theta^{(1)} * x^t$$

The process can be replicated over however many layers in the neural network, leading to the following,

$$y = \sigma_n(W^n(...(\sigma_1 W^1 x)$$

Cybenko's Theorem

Cybenko's theorem states that any continuous function on a subset of $[0, 1]^n$ is able to be approximated by a feed-forward neural network with a single hidden layer and a finite amount of neurons. Under the right circumstances and by establishing proper parameters for the network, a neural network can accurately approximate any continuous function based on some sort of predetermined accuracy level. The theorem reveals the power of neural networks, validating that they have the potential to approximate any continuous function to some degree of accuracy. This is an key insight into neural networks, as it shows that these networks can learn to model any input-output relationship in a given domain assuming the function is continuous. Furthermore, Cybenko's theorem reveals the validity of a single hidden layer in neural networks in function approximation. The theorem establishes the fact that adding more hidden layers to the network won't necessarily increase the capacity of the network. However, it has come about that deeper neural networks are able to learn more complex functions and analyze more nuanced relationships in data, which will require fewer neurons than the single-layered network.

Hanin and Sellke

Boris Hanin and Mark Sellke investigated the theoretical bounds on the minimum width of a neural network needed to approximate certain types of functions. The work they did aids in the understanding of the expressive power of neural networks and how hidden layers factor in. The duo revealed that a neural network with one hidden layer and ReLU activation functions can approximate continuous functions with a certain degree of accuracy, as long as the width of the hidden layer is at least double the number of linear components in the target function. This established a lower bound on the minimum width needed for approximations. Their work also shined light on the understanding of the relationship between changing the network's width and depth. For certain types of function, a lack of depth in the neural network can be made up for by increasing the width of the hidden layers.

Methodology

In order to understand how neural networks approximate functions, the functions must be generated and selected. We are working with Fourier series and polynomial functions since these are ones that humans are able to understand. By approximating different continuous functions and by altering the depth and width of the neural network, it is possible to explore the manner in which the networks approximate these functions and investigate the dynamics of the training process. We are given data that gives us 5 functions to work with. To investigate neural networks, I will play around with different depths, widths, and activation functions and explain methods used to assess the quality of the approximations generated.

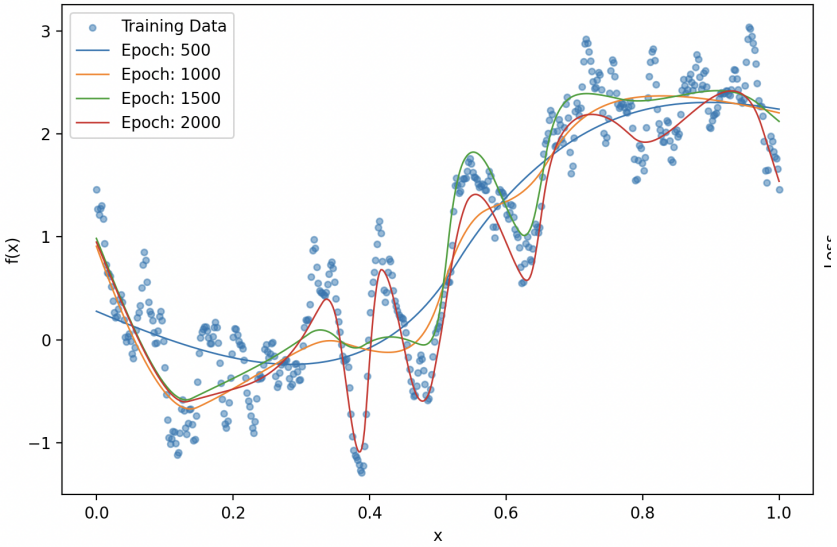
Results and Discussion

Let's consider some obtained results from the code written, which came with the help of GPT-4. I changed the code up a bit as I was working on this project to produce different graphs, so this code attached is more of a template of what I was using to play around with different adjustments based on things I wanted to investigate.

Time Dynamics

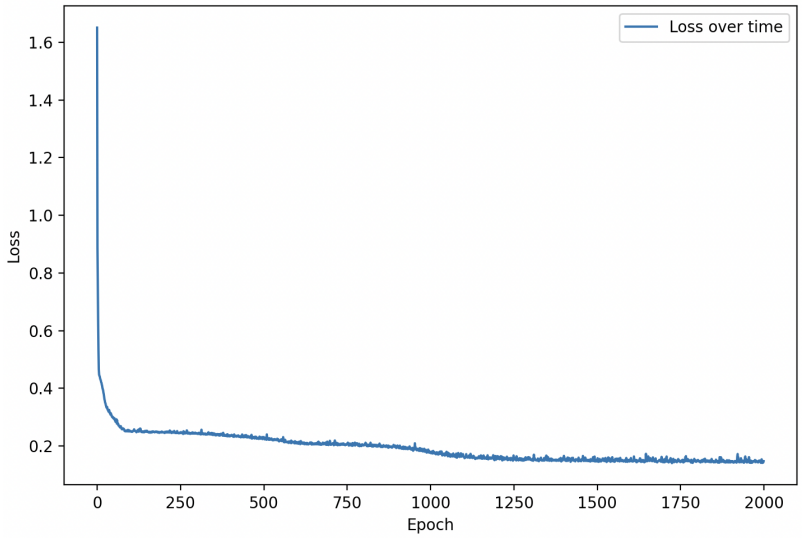
First, let's look into the time dynamics of training the system. In order to observe the time dynamics of the training process, I plotted the output at different time intervals during training. This is done by outputting the curves at certain epochs and overlaying them. To isolate the effect of time, I set the depth of the network to 5 and the width to 20 and overlaid curves at different epochs unless stated otherwise. This network structure allows for solid approximations of the functions without overfitting, which helps visualize the time dynamics of training the system.

Looking at the first function:



Time dynamics of the training process for function one

This graph shows that the neural network training process fits the function better and better over time, or over each epoch. This demonstrates the neural network's ability to learn and adapt to the training data. This is an fundamental yet key takeaway of neural networks: they get smarter. As the network trains over time, the loss over each epoch decreases:

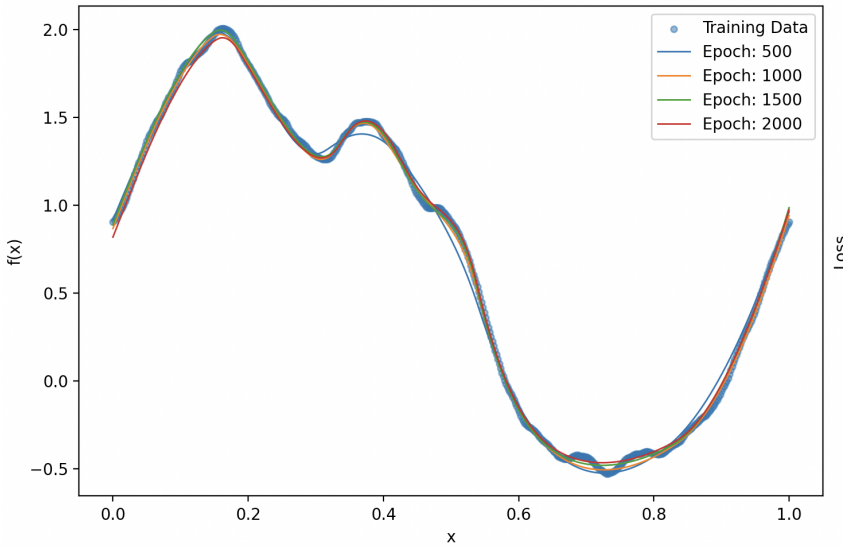


Function one loss over time

The slope of the loss over epochs shows the rate at which the loss is diminishing over time. The decrease in loss slows over time until it reaches nearly zero,

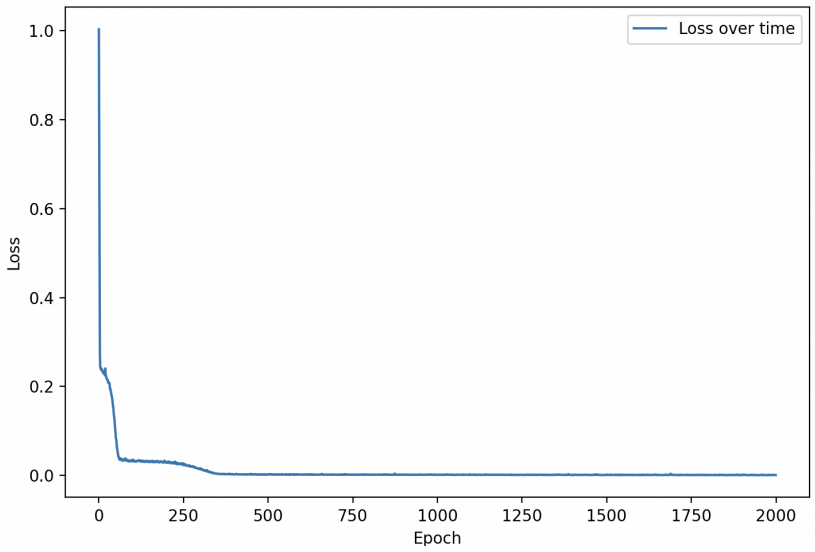
exhibiting the improvement of the training process. The slope of the losses over epochs for this function is less negative than other functions, which means that it is taking more time to converge to a good fit. This could mean that the learning rate or optimization algorithm used could be improved and changing these can result in a model that learn more quickly. It is also possible that the model underfits the data. Changing the depth, width and overall architecture of the model can improve this. Finally, it is also possible that the slope becomes less negative over time because the model is not able to improve itself with the data provided.

Moving on to the second function:



Time dynamics of the training process for f2

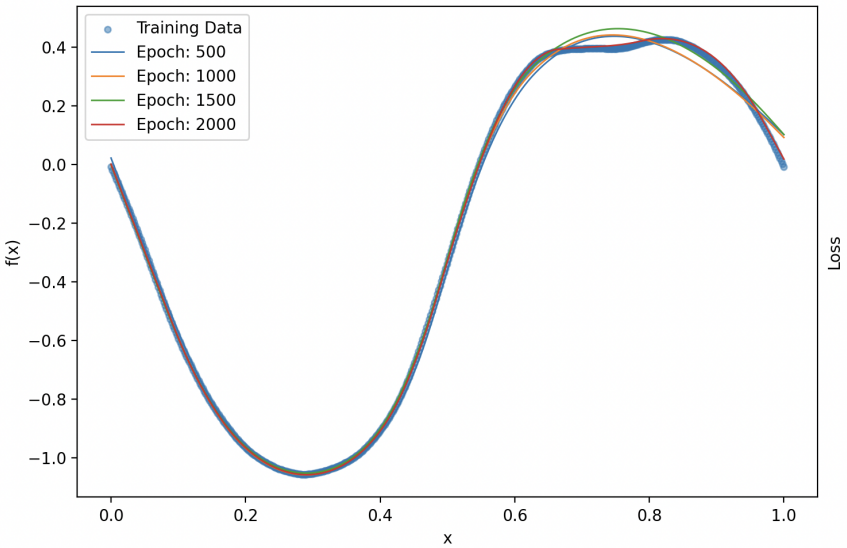
Compared to the first chart, the neural network learn this function much more efficiently as the network approximation at epoch 500 is almost identical to the training data. This is due to the fact that this second function itself is less complex than the first function in that it is smoother and more predictable in nature based on its looser shape with less troughs and dips. Since the network initializes itself with random weights, the more predictable nature of the curve makes it easier for the network to generate its next most optimal move. This is apparent by looking at the loss over time, which levels out around epoch 100:



Function two loss over time

It is also possible that the optimization or model architecture used in more suitable for the second function rather than the first, however this is unlikely to make a difference due to the overwhelming difference in complexity between curves.

Looking at the third function:

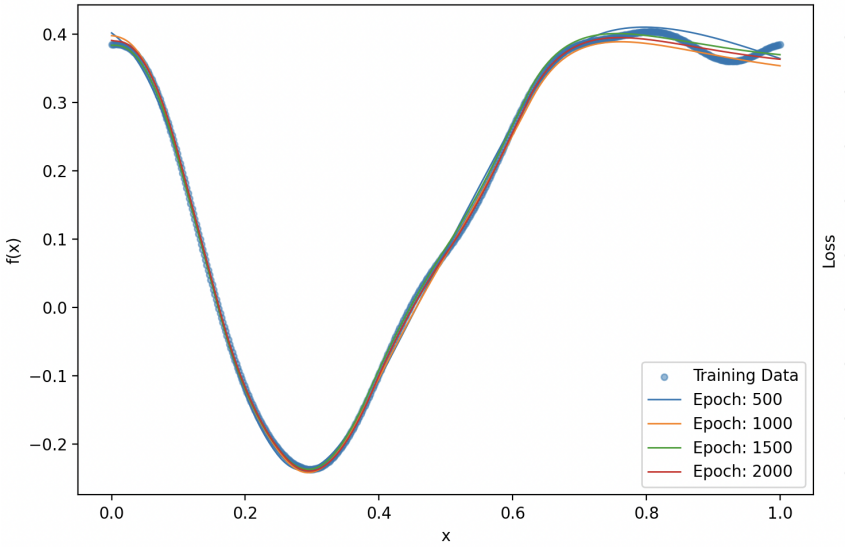


Time dynamics of the training process for f3

This function is similar in nature to the second, so the same reasoning applies

here.

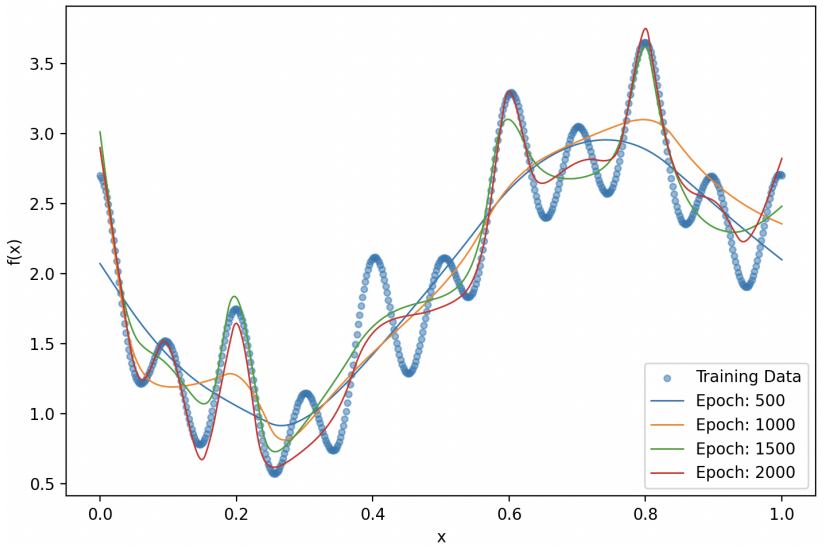
Moving on to the fourth function:



Time dynamics of the training process for f4

This function is similar in nature to the second and third, so the same reasoning applies here.

Finally, the fifth function:



Time dynamics of the training process for f5

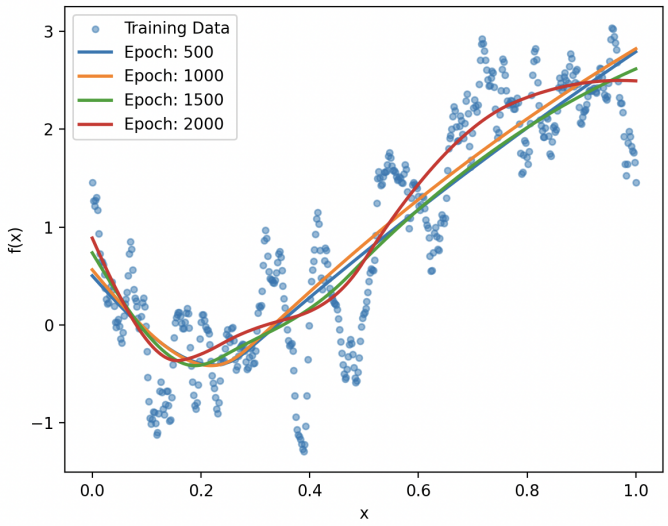
Like the first function, this one is working with a more complex Fourier series and therefore it takes the network more time to understand the nature of the function.

Overall, visualizing the dynamics of how the networks learn over time provides insight into the nature of neural networks. By observing how the loss changes over time, we can get insight into potential complications in the learning process, such as the existence of local minima that halt the learning process or vanishing gradients.

Depth and Width

Neural networks are very flexible systems that can adapt by changing the width and depth of the network. A deeper network can better capture complex relationships in the data, which can be powerful in approximating complex tasks. A wider network can be beneficial in that it establishes more nuanced relationships of input data. However, it is key to understand how networks that are too deep or too wide exist. Overfitting and vanishing and exploding gradients can occur when the model becomes too complex relative to the available training data, which makes the training ineffective. In the next section, the work done by Hannin and Sellke is referenced in regards to how changing the depth of the network.

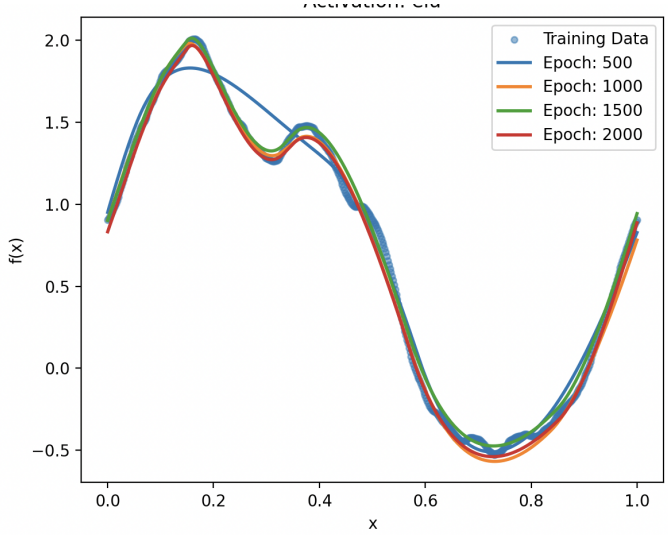
The concept of shallow networks with one hidden layer with an arbitrarily large width, which was introduced in Cybenko's theorem, shows the powers of changing the width of the network. The relationship between width and epochs is not linear or straightforward. Increasing width in a network can compensate for having a shallower network or a model with less epochs.



Training f1 with d=3, width=250

This chart shows that, despite it being very shallow, the network is able to somewhat discern the direction of the function. I ran into run-time issues for widths over about 500, so I am using this width of 250 to get at the capabilities of a large width. It is clear that increasing the width to a large value can only do so much in the fitting however, even at a large value such as 500. A large width is not able to pick up nuances in the inflection points in this function, and needs to have a deeper network in order to capture the relationships in the data. Stronger computing power would be needed to help speed up this run-time issue when working with a network with a single hidden layer.

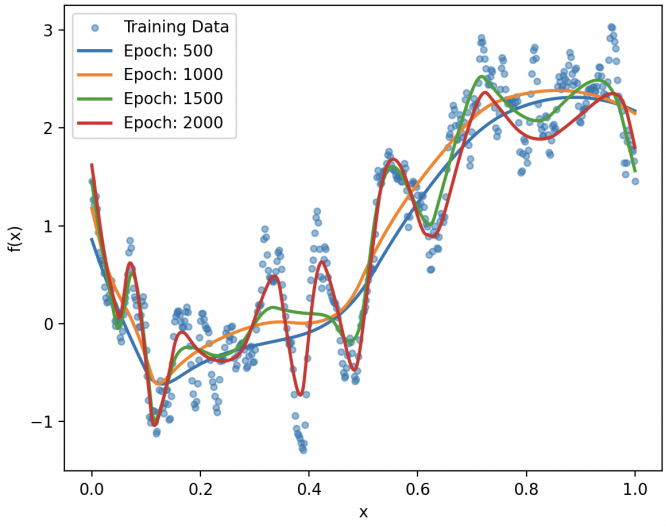
Let's look at the second function:



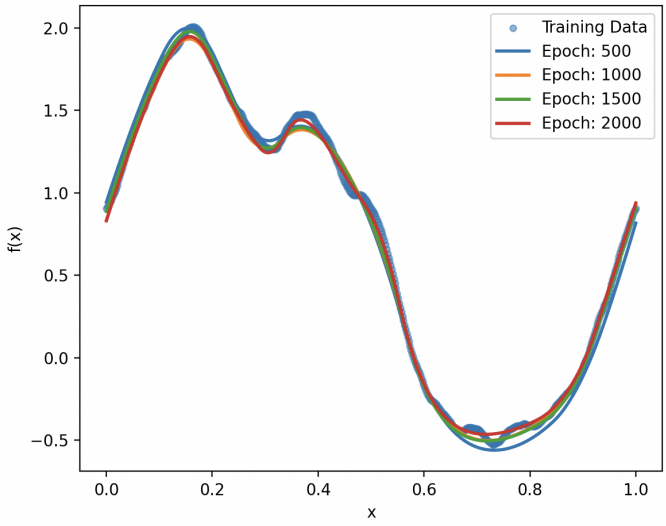
Training f2 with d=3, width=250

The approximation of this function is much more accurate than the first, as the network curve is almost exactly overlaying the function. This reveals a strength of this extremely wide and shallow network architecture: it can capture more simple relationships in the data very well. But as we saw in the graph above, it struggles with handling more complex relationships. Functions three and four follow similar logic based on there being fewer inflection points in the curve.

To demonstrate the capacity of increasing the depth in order to obtain a better trained network, I added an additional hidden layer and reduce the width value to 75:

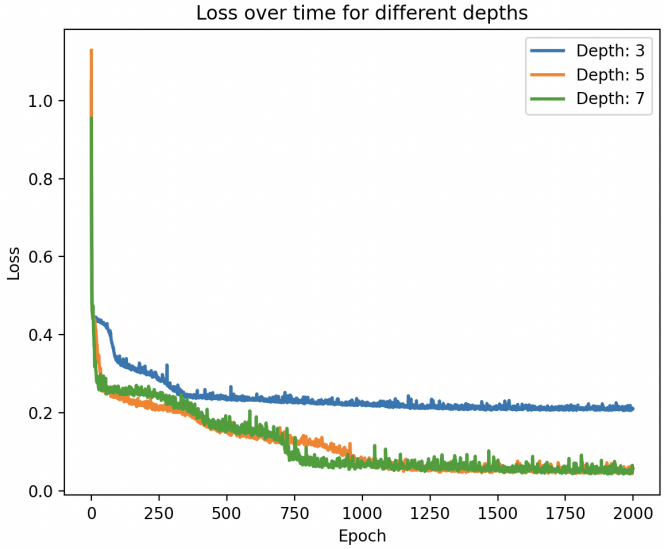


Function one with d=4, w=75



Function two with d=4, w=75

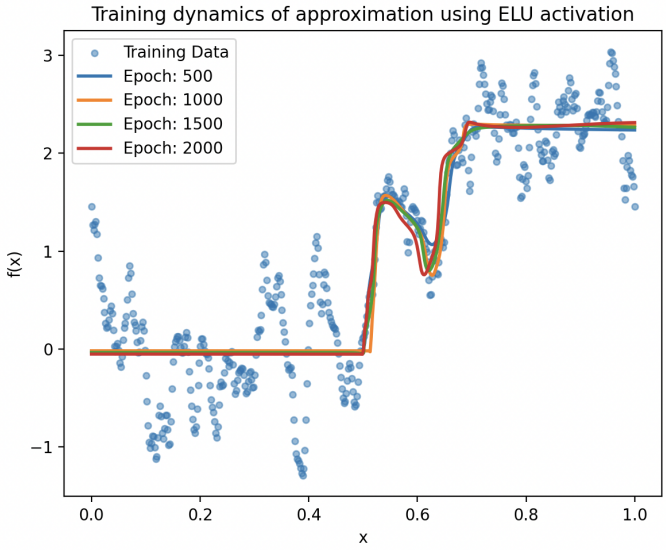
Compared to the graph of function one with one hidden layer and a width of 200, the neural network is significantly more effective at approximating function one, which is the big takeaway here since it is far more nuanced than figure 2 as has been addressed. Adding one layer while cutting over half of the width reveals the capacity for increasing the depth of the network to find adequate fits, and in terms of the relationship with changing width as well. To look at this more closely, let's consider the losses over time with respect to different relationships in width and depth.



Losses for different depths f1

The figure above overlays the different losses over time for different network depths. Comparing the slopes of each loss curve shows that increasing the depth of the network makes it much more efficient as the network with a depth of 7 achieves a proper learning rate much faster and more accurately than the network with depth 3. Furthermore, the loss curve of the depth 3 network tails off around a loss of 0.2, exposing the inefficiency of the training as this will lead to less powerful relationships in the network based on training data. Formula five deals with this same issue.

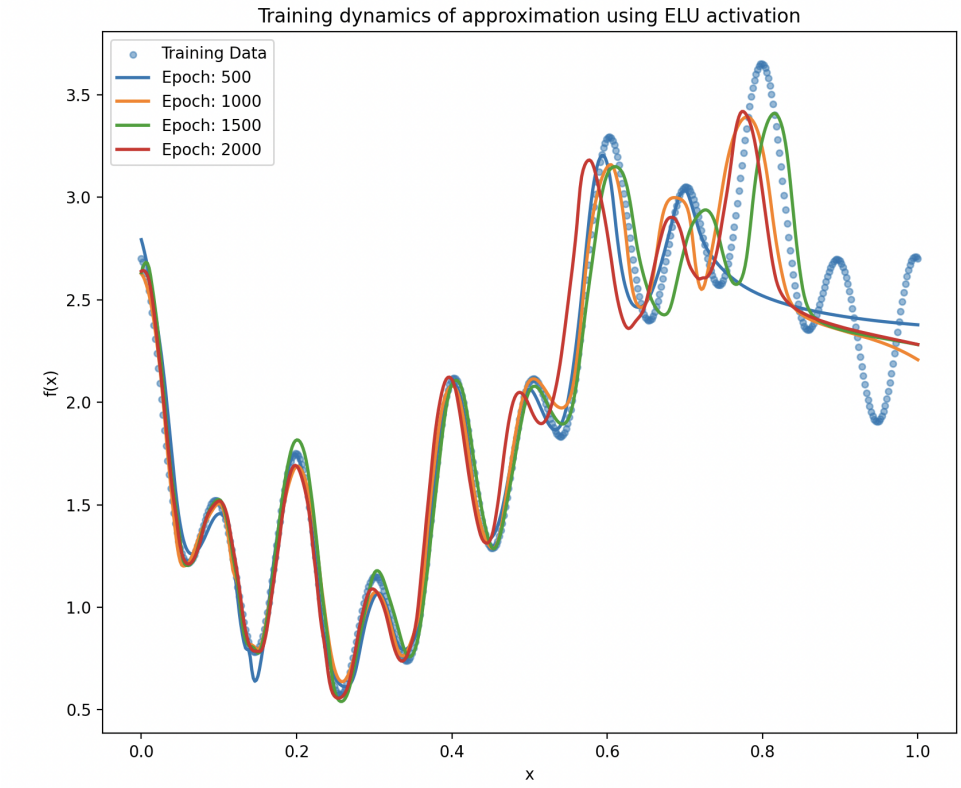
Now, let's investigate what happens when the depth is made arbitrarily large. In a similar way as to how increasing the width of the neural network can make it more effective, increasing depth can do this but even more powerfully. However, issues can arise such as vanishing and exploding gradients, overfitting and degradation. Consider function one with a depth of 100 and width of 5:



Training f1 with d=50, w=10

The fact that the network approximation of the function has a slope of zero from 0 to 0.5 x values shows that this network is currently too deep to train based on the depth and other architectural factors involved. The model would be better off by adjusting the width, and potentially activation function, in order to achieve better results. Increasing the number of layers also has significant implications on run time, which is key to factor in when dealing with larger networks.

Based on the existence of overfitting and vanishing gradient alone, which establish a sort of range for what to work under in terms of building network structure. There must exist some optimal combination that produces the best neural network. Looking at the approximation of function five gives a closer, more accurate model but slips up a bit at the end due to slight misalignment of architectural pieces.



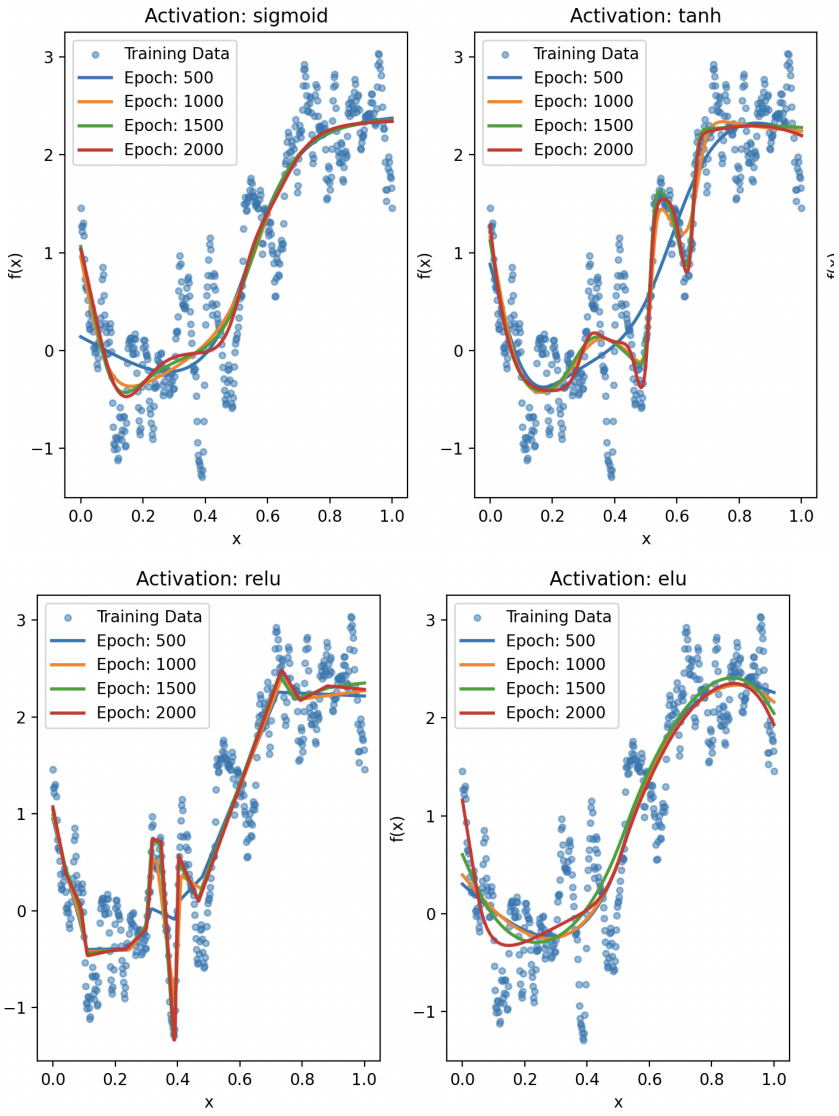
Training f5 with d=50, w=10

This close approximation, despite the imbalance in depth versus width, points to the fact that very accurate, and sometimes almost perfect, networks can be produced through training.

Activation Functions

In this section, I will be looking into how using different activation functions affects the performance of the neural network model. Activation introduces nonlinearity and allows networks to understand much richer relationships in the data. I will be considering the activations introduced earlier: sigmoid, tanh, ReLU and ELU. Activation functions allow the output signal to be non-linear, as a non-activated network acts as a linear regression with capped learning power. The sigmoid function can be used if the model can be learned more slowly due to a wider range of activating functions. If the network is too deep, sigmoid may not be able to pick up complex relationships in the data, so a more involved activation such as ELU can be used instead. For this section, I set the depth to 4 and the width to 16 to more easily observe distinctions in the activation functions. Sacrificing accuracy for comparison is valuable in understanding the nature of each activation.

Below are charts showing the training process for each activation function for function one:

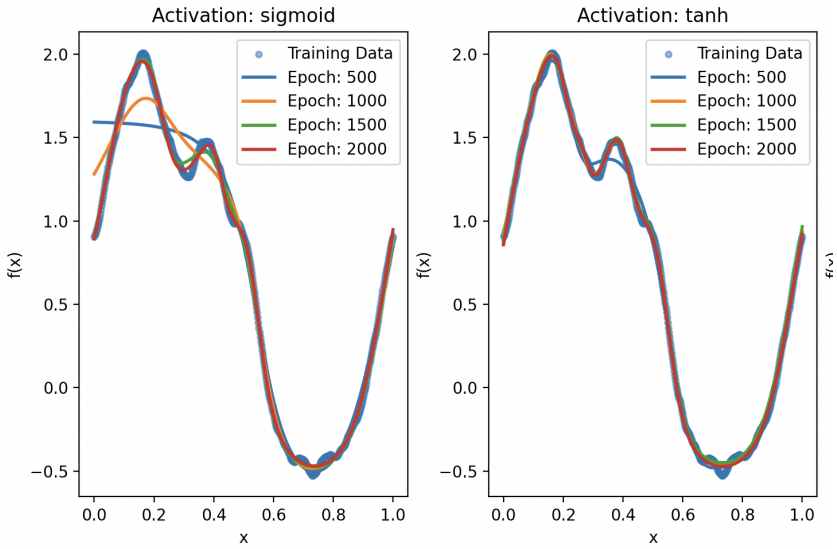


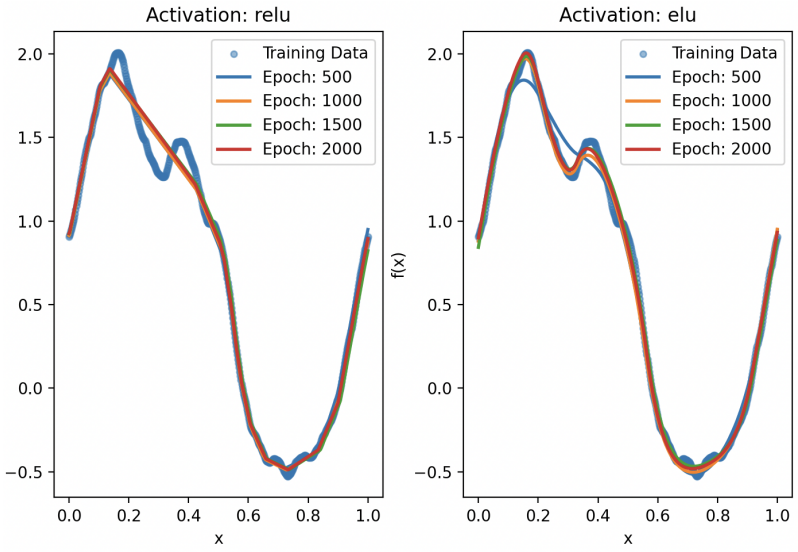
Activation training f1

We can understand the nature of each activation by observing the shape of each approximation. Based on the mathematical structure of the sigmoid function and tanh, it can be observed that both functions are S-shaped and push input values towards the bounded range of 0 to 1. The tanh function is like a stretched out, shifted version of the sigmoid, which check out based on the visual comparison. Both functions are also centered around zero on the x-axis.The training on

x from 0 to 0.5 follows a very similar trajectory, while the tanh function is better suited to pick up on more extreme data points. Tanh is able to recognize big gradients and learning steps in the training process, and is therefore almost always superior to the sigmoid as seen in this function approximation. These functions can be effective in avoiding the exploding gradient issue which arises when the gradient becomes large. However, they can suffer from the vanishing gradient issue, especially when dealing with very deep neural networks. The amount of error decreases significantly as the data is backpropagated through each hidden layer. Next, let's consider ReLU and ELU. As made obvious through the names, the two activations are very similar to one another. Both functions are in identity form for non-negative inputs. However, ELU smooths over time while ReLU is sharper in nature. This can be observed in the graphs above, as the jumps in the change in slope are much more dramatic using ReLU. The same trends can be observed in function five, as like function one it is much more complex than the other three given to us.

Below are charts showing the training process for each activation function for function two:

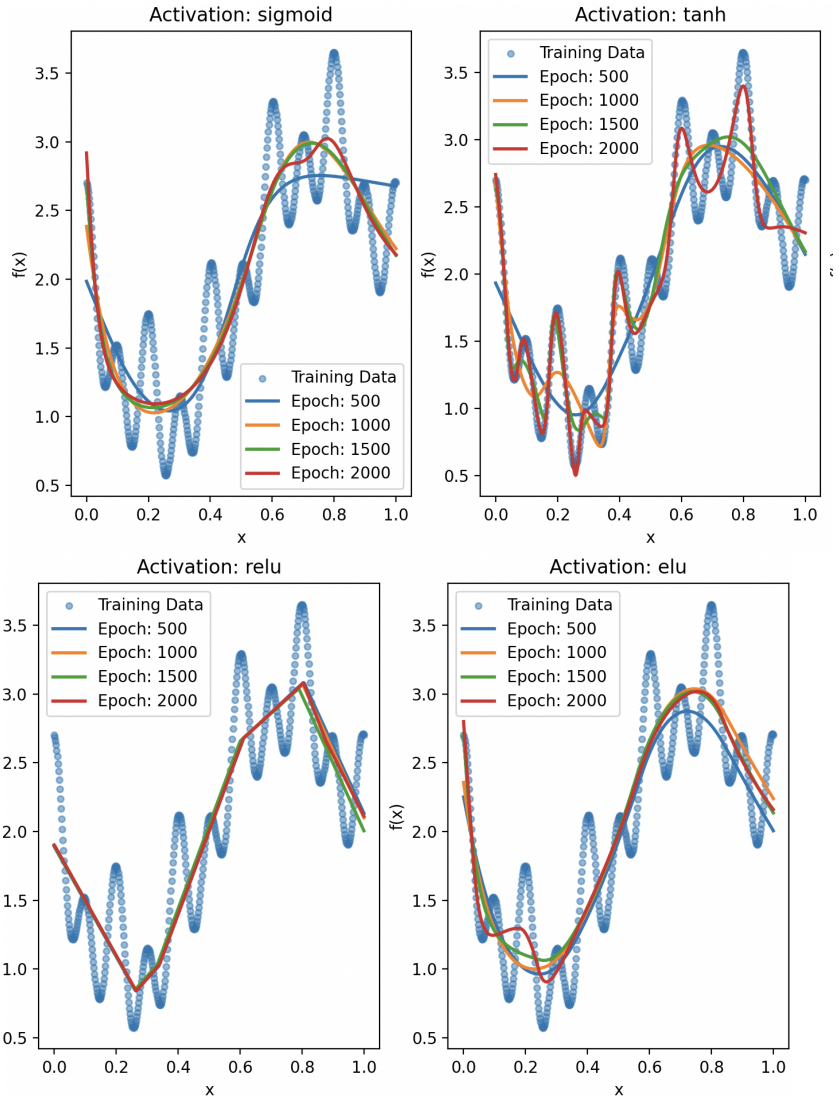




Activation training f2

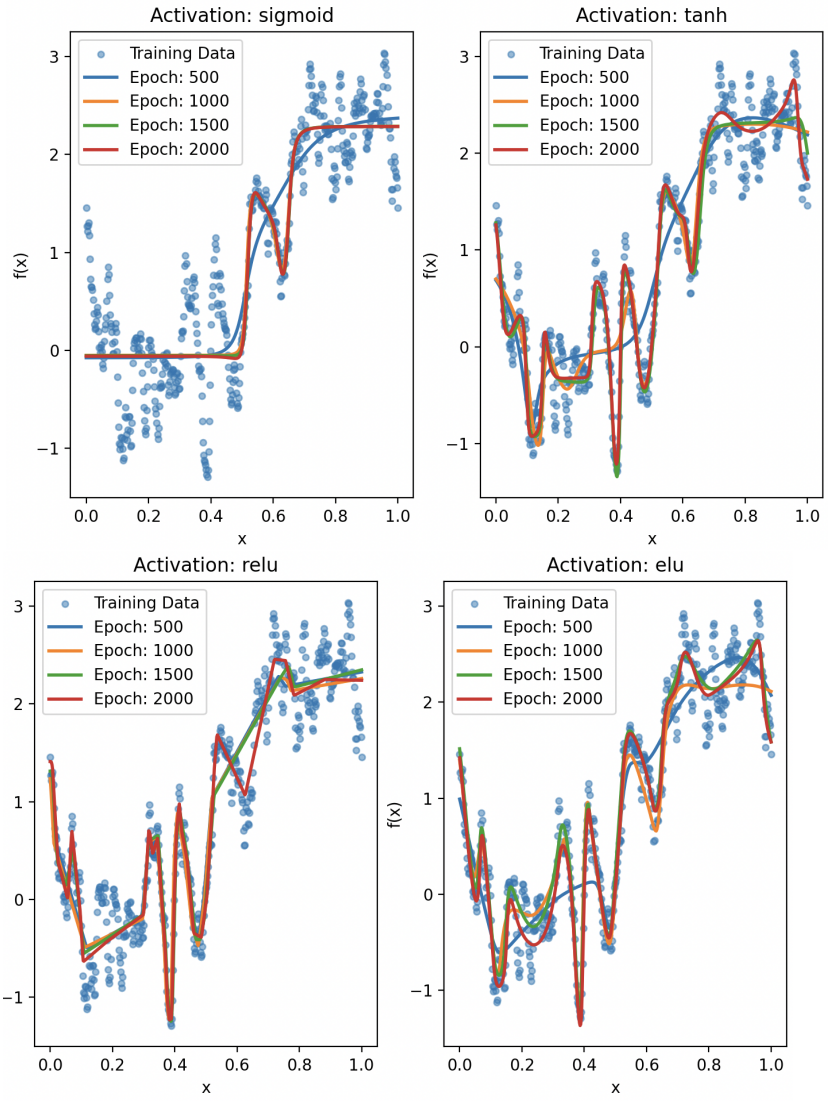
Since the function being approximated is more predictable based on there being fewer inflection points, sigmoid and tanh perform better in modeling the approximation as they are better suited to capture the more predictable changes in the slope. ReLU struggles to understand the fluctuations of the curve for the smaller x-values, as the curve from roughly x=0.15 to x=0.5 is essentially a straight line (yes, all lines are straight). After 2000 epochs, ELU performs just as well as sigmoid and tanh. The third and fourth functions perform similarly for each activation.

Let's look at the activation training for function 5, as it provides insight into how altering the depth of the network benefits and harms certain activations.



Activation training f5 with depth = 4

When we have depth equal to 4 and width equal to 16 under 1000 epochs, the ELU activation does not perform as well as sigmoid and tanh. However, altering the depth of the network will allow ELU to make better use of its capabilities. Setting the depth to 8 while leaving the width at 16 produces the following visuals:



Activation training f5 with depth = 8

Hannin and Sellke showed that there exists a neural network with a width of two and some arbitrary depth that will approximate the function. While I have the depth set to 15 with 2000 epochs, this outcome is similar to increasing the epochs to some large amount, say 100,000, and having a width of 2. This is the framework for which Hannin and Sellke discussed the power of the depth of the network. Tanh and ELU do a very sufficient job approximating function 5, RelU does a decent job, and sigmoid runs into big trouble. The power of ELU to perform well in deeper neural networks comes through here, as doubling the depth of the neural network significantly decreases the loss over time, making

the network very close to the approximation function. ReLU performs decently but struggles with some of the data. Sigmoid struggles the most as it deals with the vanishing gradient problem. During the backpropagation process, the gradient becomes very small and makes it impossible for the neural network to learn.

Discussion

We live in a world where people are always looking for better a better way to do things. After all, there is no situation I can think of where someone would not rather be better at something (assuming this thing adds positive value to the world). This logic applies directly to the study of neural networks approximating functions and the real-world applications. Whether it be the accuracy of voice recognition or precision of medical diagnoses, the improvement in accuracy of neural networks can continue to leave a positive impact on the world. While the types of networks investigated in this project are minute in comparison to the networks used by supercomputers that drive the application of neural networks, there are several key insights that we can take away from the study of these smaller networks that can be applied to any sized network.

Supercomputers are able to handle neural networks with thousands and even millions of layers and extremely large widths. The challenge these supercomputers face is finding the ideal network architecture based on training data it has access to. The key to obtaining an ideal network lies in the balance of the network's width, depth, activation and overall structure, all of which are connected. The endogenous nature of neural networks leaves neural networkers on the hunt for more dynamic networks that reveal trends in input data. The inevitable progression of this desire for stronger algorithms and tools to better understand the world around us are the reason neural networks are such an exciting topic today as the increased capacity of computers, larger data sets, and the collective effort to channel the powers of AI leave no end game in sight.

Code

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.callbacks import Callback

class ModelSnapshot(Callback):
```

```
    def __init__(self, epochs_to_snapshot, train_data):
        self.epochs_to_snapshot = epochs_to_snapshot
        self.train_data = train_data
        self.predictions_history = []

    def on_epoch_end(self, epoch, logs=None):
        if (epoch + 1) in self.epochs_to_snapshot:
            predictions = self.model.predict(self.
                train_data)
            self.predictions_history.append((epoch + 1,
                predictions))

def create_model(depth, width, activation):
    layers = [Dense(width, activation=activation,
        input_shape=(1,))]
    for _ in range(depth - 2):
        layers.append(Dense(width, activation=activation)
            )
    layers.append(Dense(1))

    model = Sequential(layers)
    model.compile(optimizer='adam', loss='mse')
    return model

def plot_graphs_from_csv(file_path):
    data = pd.read_csv(file_path, sep='\t')

    for i in range(len(data)):
        x = data.columns.values.astype(np.float32)
        y = data.iloc[i].values.astype(np.float32)

        train_data = x.reshape(-1, 1)
        train_labels = y.reshape(-1, 1)

        depth = 50
        width = 10
        activation = 'elu'

        model = create_model(depth, width, activation)

        epochs_to_snapshot = [500,1000,1500,2000]
        model_snapshot = ModelSnapshot(epochs_to_snapshot
            , train_data)

        model.fit(train_data, train_labels, epochs=2000,
            verbose=0, callbacks=[model_snapshot])
```

```
        fig, ax = plt.subplots(1, 1, figsize=(6, 5))

        ax.scatter(x, y, s=15, label="Training Data",
            alpha=0.5)

        for epoch, predictions in model_snapshot.
            predictions_history:
             ax.plot(x, predictions, label=f'Epoch: {epoch
                 }', linewidth=2)

        ax.set_xlabel("x")
        ax.set_ylabel("f(x)")
        ax.set_title(f"Training dynamics of approximation
            using ELU activation")
        ax.legend()

        plt.tight_layout()
        plt.show()

plot_graphs_from_csv('data.csv')
```

References

Expressivity of neural network texts provided on Canvas
Cornell: https://news.cornell.edu/stories/2019/09/professors-perceptron-paved-way-ai-60-years-too-soon