ARTIFICIAL NEURAL NETWORKS FOR ROBUST CURVE FITTING

Developing a Versatile Algorithm for Curve Fitting of Numerical Data with Shallow Artificial Neural Networks

February 22, 2019

ABSTRACT

Curve fitting numerical data is a common process in any field of science where some phenomena is observed, however, the current methods of doing so that exist are insufficient when applied to more complex data sets. Therefore, the purpose of this project was to develop an accurate and versatile algorithm for curve fitting of numerical datasets. There were three main criteria for the model to be constructed to meet: high scalability, flexibility between accuracy and speed, and insensitivity to noise. Artificial Neural Networks (ANNs) were a prime framework for this due to inherent features in their mechanics which satisfy the above constraints and therefore they were used. The researcher tested ten three-layer ANN architectures on five mathematical functions which were selectively chosen to provide a wide variety of graph shapes. As a benchmark, polynomial regression was also tested on the same functions. Results were collected on speed and accuracy. It was found that the ANN model was generally slower, but on par with accuracy with polynomials. Future research will be on using multi-layer ANNs, different gradient descent algorithms, learning rate decay, and different activation functions. The ANN model will also be tested on real life datasets (stock price data, NASA astronomy data, etc). This project is really applicable to any field of study in which we observe some phenomena, collect numerical data, and perform analysis upon it. For instance, training an ANN with data obtained from Mass Spectroscopy could lead to predictions of the properties of certain molecules.

Keywords Curve Fitting · Artificial Neural Network · Dataset · Polynomial

1 Introduction

Curve Fitting is the process of constructing a function which well estimates the relationship between variables in a data set. It holds applications in any field of study in which we graph and analyze numerical data, and plays a particularly important role in engineering science and physical analysis. Usually it is accomplished by defining some cost (also error, loss, or objective) function and then using a method to minimize it.

A very famous approach to curve fitting is through **Polynomial Regression**: finding the optimal choices of the coefficients of a polynomial of a pre-specified degree by solving a matrix equation of its components to minimize the **Mean-Squared-Error (MSE)**. When applied to large or complex datasets, such methods are known to react strongly to outliers and succumb to the effects of overfitting. These problems are exacerbated when considering multivariate datasets as it is difficult to determine the data complexity and therefore the order of the optimal polynomial model. As the vast majority of real world phenomena are controlled by multiple factors, the data collected from them is indeed multivariate, and polynomials are thus poor predictors of real-world phenomena.

There do exist iterative algorithms for the minimization of the cost function aforementioned, but they are generally computationally intensive and consequently slow. In addition, they sometimes require human intervention for an initial guess to ensure that the model converges to a correct solution. This raises considerable interest in a versatile algorithm to automate curve fitting which can quickly process large amounts of data, or deal with data in real-time.

In this project we use **Artificial Neural Networks (ANNs)** as a new technique of determining functional parameters which minimize the value of a cost function, with respect to some dataset. This approach is superior in the sense that it can be applied to large or multivariate datasets, or process data in real time with minimal deviation from a predefined framework, and for that reason it has a remarkably high versatility, and therefore usability. Furthermore, for applications involving intensive computational work, ANNs can be incorporated into special hardware that can harness their hierarchical structures to achieve very fast processing speeds.

2 Background Research

2.1 A Brief Introduction To The Artificial Neural Network

Artificial Neural Networks (ANNs) are computational systems loosely modeled after the networks of neurons in the animal brain. Their versatility and ability to solve problems not easily able to be expressed algorithmically comes from the **fundamentally** different way they approach problems. Examples of these problems include, but are not limited to

- Image Recognition and Classification
- Function Modeling and Curve Fitting
- Image Restoration
- Language Translation
- Voice Synthesis and Recognition

A notable similarity among these five problems is that they are all non-trivial pattern recognition problems. In this project we investigate Artificial Neural Networks as they relate to the second point on this list, Function Modeling and Curve Fitting.

2.2 Artificial Neurons

An Artificial Neural Network is simply a network of **artificial neurons**, divided into various **layers**. There are two major types of neurons: **perceptrons** and **sigmoid neurons**.

2.2.1 Perceptrons

Perceptrons are the simplest types of neurons, they receive binary inputs and produce a binary output. To compute this output we introduce **weights**, real numbers signifying the strength of the effect of a particular input on the perceptron's output.

The output, or **activation**, of a perceptron is determined by whether the value of the **weighted sum** $\sum_{i=1}^{n} x_i w_i$ is greater than some **threshold** value. More explicitly,

$$\text{Output} = \begin{cases} 0 & \text{if} \quad \sum_{i=1}^{n} x_i w_i \leq \text{threshold} \\ 1 & \text{if} \quad \sum_{i=1}^{n} x_i w_i > \text{threshold} \end{cases} \tag{1}$$

To make the notation less cumbersome, let x and w be the vectors of inputs and their corresponding weights, respectively and define a **bias** value $b = -\text{threshold}$. Then we get

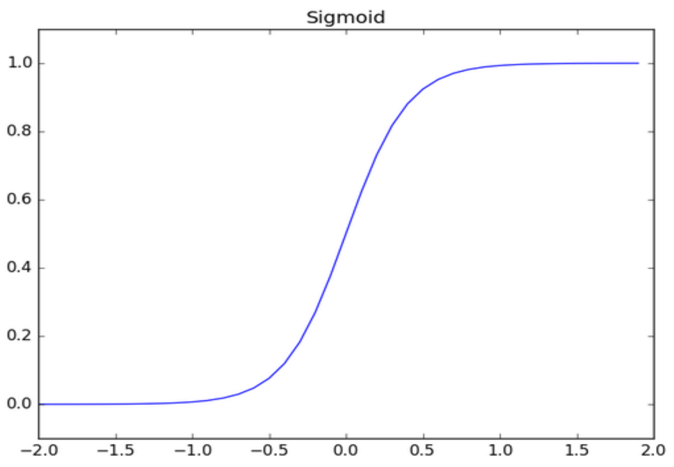
$$\text{Output} = \begin{cases} 0 & \text{if} \quad w \cdot x + b \leq 0 \\ 1 & \text{if} \quad w \cdot x + b > 0 \end{cases} \tag{2}$$

2.2.2 Sigmoid Neurons

For our network to be able to learn in a predictable manner, we need the output(s) of the networks to be continuous functions of the input(s), however this is not the case with the perceptron model. Hence, we introduce a new type of neuron, the sigmoid neuron.

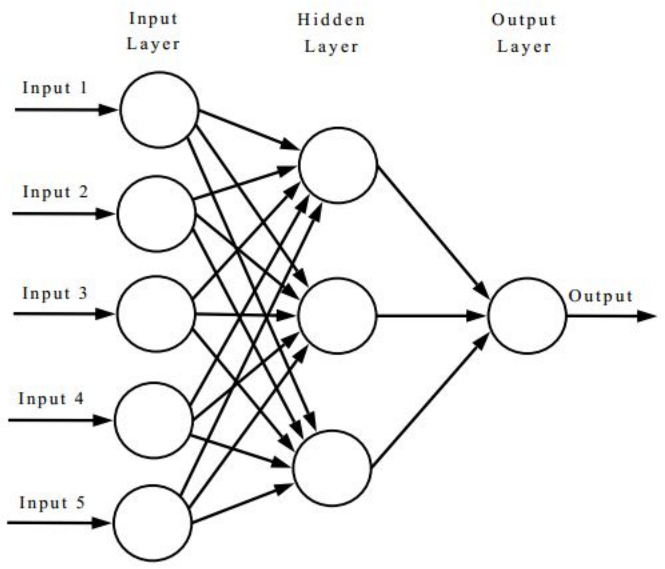
Sigmoid neurons function exactly the same as perceptrons, except the activation of the sigmoid neuron is not a piecewise function, it's rather defined by $\sigma(w \cdot x + b)$, where w and x are the vectors of the inputs and their corresponding outputs respectively, b is the bias, and $\sigma(z)$ (the sigmoid function) is defined as

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{3}$$



2.3 Artificial Neural Network Architecture

Artificial Neural Networks are divided in layers, with an arbitrary number of neurons in each layer. The first and last layers are the input and output layer respectively, and all layers in between are called **hidden layers**.



It is evident that the outputs from layer l are the inputs to layer $l+1$. This repeated process of taking the outputs from one layer and inputting them to the next is called **forward-propagation**. Neural networks requiring only a forward-propagation system to find the output are called **feed-forward** neural networks.

When numbers are inputted to the network, they're multiplied by their corresponding weights and inputted to the neurons of the second layer. The weighted sum of each of the neurons is then added to their bias and put into the sigmoid activation function. These values are now multiplied by the weights of the connections between the second and third layers and the process is repeated until the last layer.

If we let

- a^l_j denote the activation of the jth neuron in layer l
- $w^l_{j,k}$ denote the value of the weight connecting the jth neuron in layer l and the kth neuron in layer $l-1$
- b^l_j denote the bias of the jth neuron in layer l
- n_l denote the number of neurons in layer l

We can define a universal equation to find the activation of any neuron in our network

$$a^l_j = \sigma\left(\left[\sum_{k=1}^{n_{l-1}} w^l_{j,k} a^{l-1}_k\right] + b^l_j\right) \tag{4}$$

2.4 Training An Artificial Neural Network

The training process of an ANN is highly variable when it comes to the specifics, but the general structure is the following

1. Define a cost function (also error, loss, or objective function) of the network's parameters
2. Initialize the network with random parameters
3. Iterate a series of steps which changes the network parameters in such a way as to decrease the cost function

2.4.1 The Mean-Squared-Error Cost Function

To be able to train our network we must define a **cost function** (or **error function**), showing how well our network is performing. This function takes in the parameters (weights and biases) of the network and a set of **training data**, which is data from which our network must 'learn', or to be less ambiguous, it is the data from which our network must generalize. In short, it is, quite literally, the training data. One such cost function is the **Mean-Squared-Error (MSE)** cost function, or quadratic cost function. The MSE cost function is defined as

$$C(w,b) :\propto \sum_x ||y(x) - a^L(x)||^2 \tag{5}$$

where we write the desired output and actual output of the network as $y(x)$ and $a^L(x)$, respectively, as they are functions of the inputs, x. Usually the constant of proportionality is $1/2n$ but it does not matter as much as the general form of the equation. The MSE cost function is a good way of determining the error of a network as it gives a high value when even a single desired output is not close to the actual output since the raw difference is squared, and when the two outputs are close together, the raw difference (which is small) is squared to become even smaller, so that point is then basically ignored.

2.4.2 The Cross-Entropy Cost Function

2.4.3 Gradient Descent (GD)

Gradient Descent (GD) is the standard method by which we can train an ANN. It involves differentiating the cost function with respect to each weight and bias in the neural network, and using the resulting information to minutely alter each weight and bias to decrease the function. The extent to which each weight and bias is altered is controlled by a **hyper-parameter** called the **learning rate**, which is usually denoted by the lowercase greek letter η.

To make this more explicit, let us define the gradient descent mathematically.

Using the definition of a derivative, we can say that

$$\lim_{\Delta w_{j,k}, \Delta b^l_j \to 0} \left(\sum_{j,k} \frac{\partial C}{\partial w_{j,k}} \Delta w_{j,k} + \sum_{l,j} \frac{\partial C}{\partial b^l_j} \Delta b^l_j \right) = \Delta C \tag{6}$$

So that for small ∇w and ∇b we can approximate

$$\Delta C \approx \sum_{j,k} \frac{\partial C}{\partial w_{j,k}} \Delta w_{j,k} + \sum_{l,j} \frac{\partial C}{\partial b^l_j} \Delta b^l_j \tag{7}$$

And we can make this equation less cumbersome by defining the vectors Δw, Δb, ∇w, and ∇b:

$$< PUT\ DEFINITIONS\ HERE > \tag{8}$$

So that we can write:

$$\Delta C \approx \nabla w \cdot \Delta w + \nabla b \cdot \Delta w \tag{9}$$

Now, it is easy to compute the values of all of the partial derivatives in the vectors ∇w and ∇b through computer aided differentiation, and it becomes easier still with the fact that neural networks possess quasi fractal-like architectures. What we wish to compute are the values of the vectors Δw and Δb. To do this, we let

$$\Delta w = -\eta \nabla w \tag{10}$$

$$\Delta b = -\eta \nabla w \tag{11}$$

So that if we substitute these value into equation (9), we get

$$\Delta C \approx -\eta(||\nabla w||^2 + ||\nabla b||^2) \tag{12}$$

Where η is some small, positive parameter, which was aforementioned as the learning rate. Two things that become apparent here from equation (12):

- For every iteration of the changing of the weights and biases according to equations (10) and (11), ΔC is always negative
- The above statement is only true for sufficiently small η within the bounds of the approximation of equation (12)

The only way to ensure with very high certainty that ΔC will decrease with each iteration of the changing of w and b is to make the learning rate parameter incredibly small (like 1×10^{-30}), which would make training extremely slow. Thus, to sacrifice a bit of preciseness in favor of practicality is necessary in this situation. This can be done by making eta a moderately small number, such as 0.01.

Now let us define our weight-update and bias-update rules explicitly. Let $w_{j,k}$ be the initial value of the weight and $w'_{j,k}$ be the updated value. Similarly define b^l_j and b'^l_j. Then we get

$$w'_{j,k} = w_{j,k} - \eta \frac{\partial C}{\partial w_{j,k}} \tag{13}$$

$$b'^l_k = b^l_k - \eta \frac{\partial C}{\partial b^l_k} \tag{14}$$

This is essentially how GD works. We define a cost function, then repeatedly update the values of the weight and biases based on the learning rate and the derivative of the cost function with respect to each weight and bias until the cost of the network is acceptably low. In this sense, the training is never really *finished*. We rather abandon the training process once the accuracy of our network is sufficiently high. The only way we are forced to stop the training process is if accuracy on the training data reaches 100%, but in that case it is very likely that the network has overfit.

2.4.4 Stochastic Gradient Descent (SGD)

Although Gradient Descent may work perfectly in theory, there exists a hindrance when applying it practically, and that is the time required for training. To find ∇w and ∇b we need to average their individual values over all the training data inputs x, so learning occurs very slowly. To combat this, a variation of gradient descent is introducted, called **Stochastic Gradient Descent (SGD)**.

SGD works similarly in principle to normal GD with the distinction that the vectors ∇w and ∇b are not computed by averaging over entire training dataset, but by averaging a small, randomly selected sample of it, called a **minibatch**.

Notationally, let use denote a minibatch of size m as a set of training inputs $(X_1, X_2, X_3, ..., X_m)$. Usually, if m is large enough, then we can expect that

$$\frac{1}{m}\sum_{i=1}^{m} \nabla w_{X_i} \approx \frac{1}{n}\sum_{x} \nabla w_x = \nabla w \tag{15}$$

$$\frac{1}{m}\sum_{i=1}^{m} \nabla b_{X_i} \approx \frac{1}{n}\sum_{x} \nabla b_x = \nabla b \tag{16}$$

Where ∇w_{X_i} and ∇w_x denote the vectors of the partial derivatives of the cost function with respect to the training input X_i of the minibatch and a training input x, respectively. Notation for ∇b follows similarly.

3 Methodology

3.1 Engineering Goal

The main purpose of this project is to demonstrate the effectiveness of Artificial Neural Networks as a curve fitting model, and in particular, display its superiority over traditional methods such as polynomial regression.

There are three major criteria for this ANN model to satisfy:

1. Very High Scalability
2. Good Flexibility Between Accuracy and Speed
3. Insensitivity to Noise (An Avoidance of Overfitting)

Immediately from our preliminary research we can see that ANNs possess inherent properties which resolve much of the design criteria.

1. The issue of scalability is very easily avoided just by considering the structure of neural nets. If we want to curve fit a function $f : \mathbb{R}^n \to \mathbb{R}^k$ then we simply construct a network with n neurons in the input layer and k neurons in the output layer and train it with training data.
2. Since ANNs train by an iterative process, there is a clear speed-accuracy tradeoff, and thus training can be stopped at an arbitrary point in time to sacrifice accuracy for speed or vice versa. This ensures that a researcher can perform a quick approximation of lengthy analysis of any dataset.
3. There exist regularization methods to lower the responsiveness of the training algorithm to random noise in the training data, hence decreasing the effects of overfitting.

3.2 Virtual Materials

There were no physical materials in this project, but a multitude of virtual materials were used, which include

- Python 3.7.0 programming software
- Numpy Library for fast linear algebra
- Matplotlib library for easy graphing

3.3 Procedure

There were three programs developed during the duration of this project, which were for the purposes of

- Data Collection for the ANN model
- Data Collection for the Polynomial model
- Comparison Program which Graphs and a trained ANN and polynomial along with Training and Test data

Code for each program can be found in Appendix A

Procedure for the main programs (Data collection) consisted of four main steps

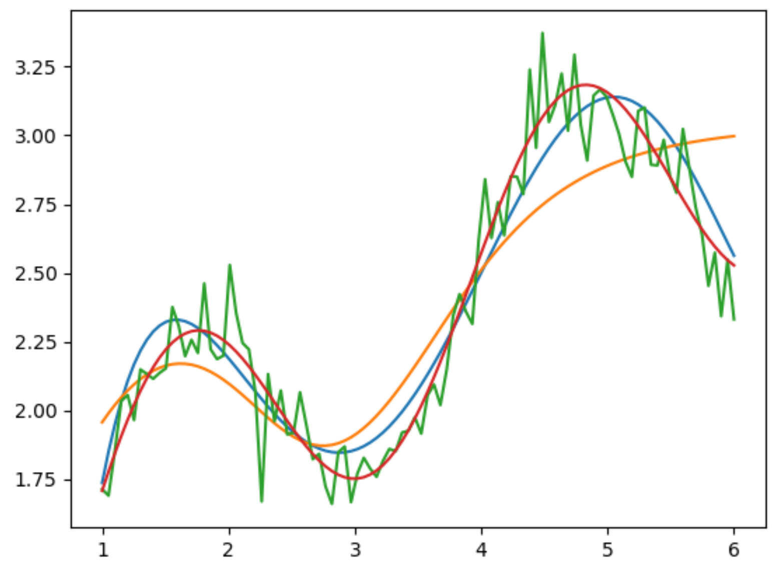
1. Select ANN architecture/polynomial degree
2. Generate training dataset
3. Train ANN/polynomial to training dataset
4. Compute data and save to external text file

Training datasets were generated from the following functions. In each graph the colors have the denotations:

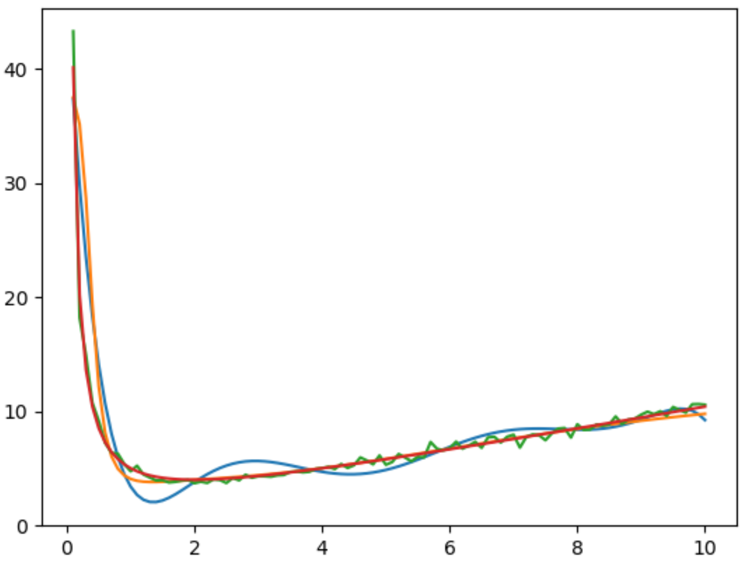
- Red - Pure Function ($f(x)$)
- Green - Noisy Function ($g(x)$)
- Orange - ANN Predictions
- Blue - Polynomial Predictions

In each graph the degree of the polynomial is 7 and the ANN has an architecture of $[1 - 15 - 1]$

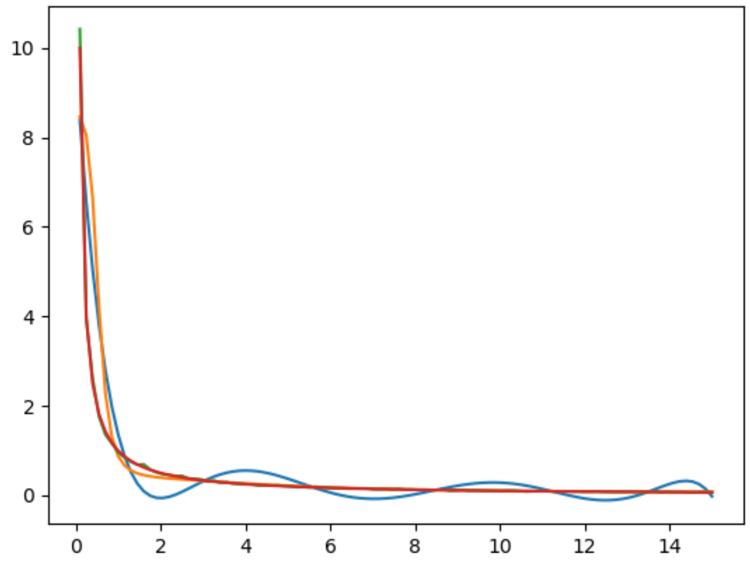
$$y = \sin^2 x + \sqrt{x}$$



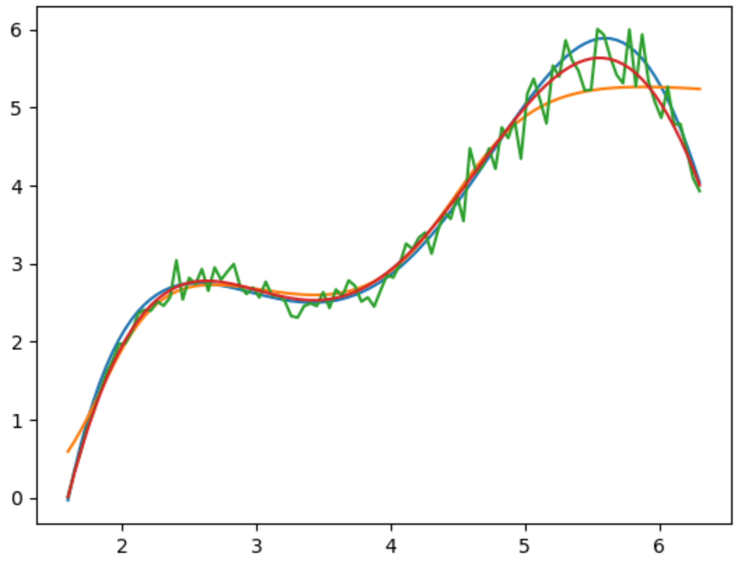
$$y = \frac{x^2+4}{x}$$



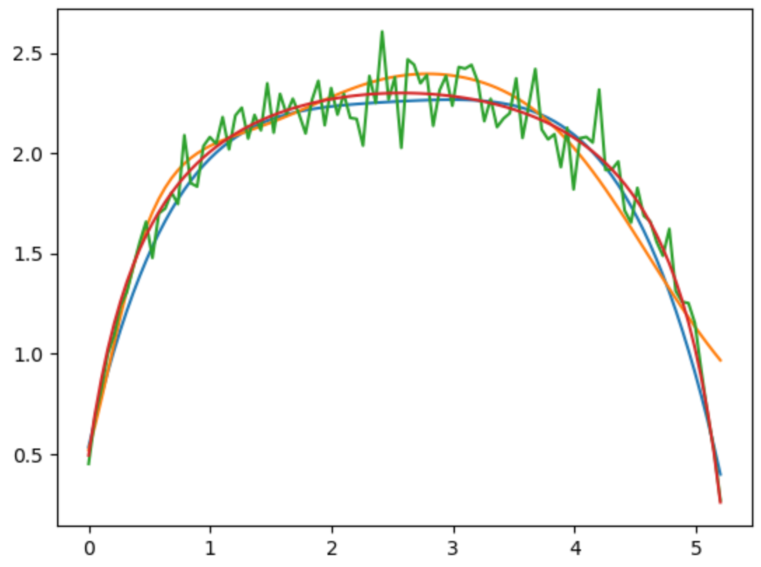
$$y = \frac{1}{x}$$



$$y = -(x-5)^2 + 2(x-5)\cos(x-5) + 5$$



$$y = \frac{-0.7}{\sin(x/2) - 6} + 3$$



These functions were chosen because they can simulate a wide range of possible shapes in their graphs. Therefore, by testing with data from these functions, the capabilities of the model were tested in a thorough and scientific manner

Now that miscellaneous information on the methodology has been provided, we may denote the entire procedure as the following:

1. Construct a three layer ANN of architecture $[1 - 2k - 1]$ with randomly initialized parameters/Select a polynomial order k.
2. Select a function, $f(x)$ from the above list from which to generate training data
3. Construct a new function $g(x) := f(x) + m$ where m denotes a random 10% error based on the normal distribution
4. Generate a training dataset of 15 sample points from $g(x)$
5. Train the ANN model/polynomial model with the training dataset
6. Once the model has been trained, obtain data on
   - Time Elapsed Training
   - Mean-Squared-Error
   - Average Percent Accuracy
   - Number of Epochs Elapsed (ANN only)

   where accuracies are measure with respect to the function $f(x)$.
7. Save the aforementioned data to an external text file
8. Perform 30 repetitions of steps 2-6 for each function
9. Perform one repetition of steps 1-7 for each $k \in \{1, 2, 3, ..., 10\}$
10. Summarize resulting data (take averages)

3.4 Major Documentation Of Progress

There emerged 4 major versions of the ANN program out of the continuous process of testing and bettering the design. Below is a comprehensive list of the version name and the changes associated with it.

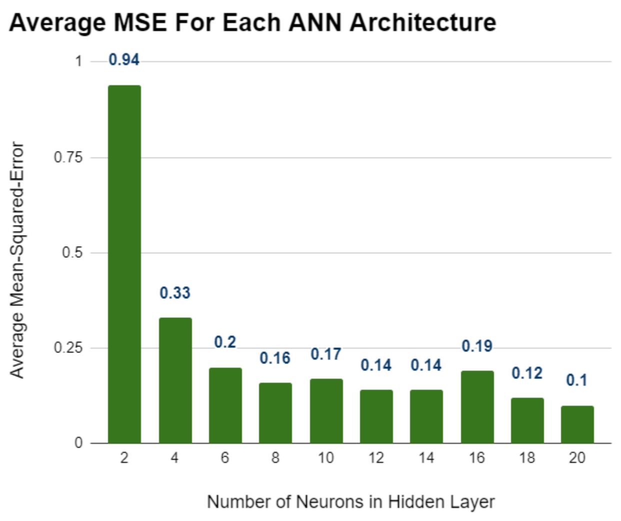
- Version 1 - Basic ANN Mechanics: Training and numerical output of ANN based on input after training.
- Version 2 - Introduction of Polynomial Regression: Polynomials as a comparison benchmark and graphing of predictions after training

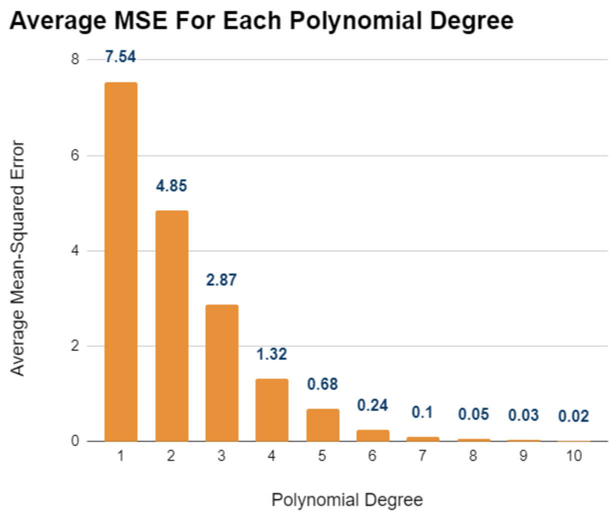
- Version 3 - Epoch Checking: An acceptable accuracy level is set and after each epoch of training, the ANN model's accuracy is compared to the acceptable accuracy. If it is greater, then training terminates.
- Version 4 - Update of Accuracy Algorithms: Percent accuracy is introduced and the MSE is updated to be faster
- Version 5 - Learning Rate Decay: This is the latest update and has not been tested yet. The main addition is the incorporation of a system that stores the parameters of the ANN of the previous epoch, so that if accuracy decreases between two epochs, the code reverts back to the older parametric settings, redefines the learning rate as 2/3 of itself, and continues training. This allows for the existence of learning rate decay without another hyperparameter.

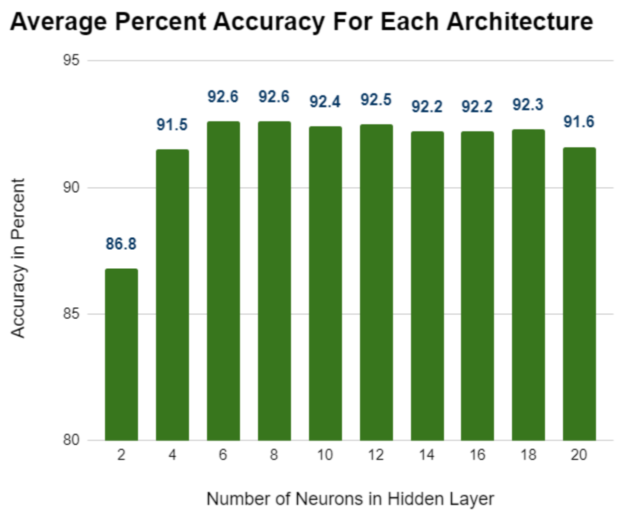
A much more inclusive documentation can be found in the project written logbook.

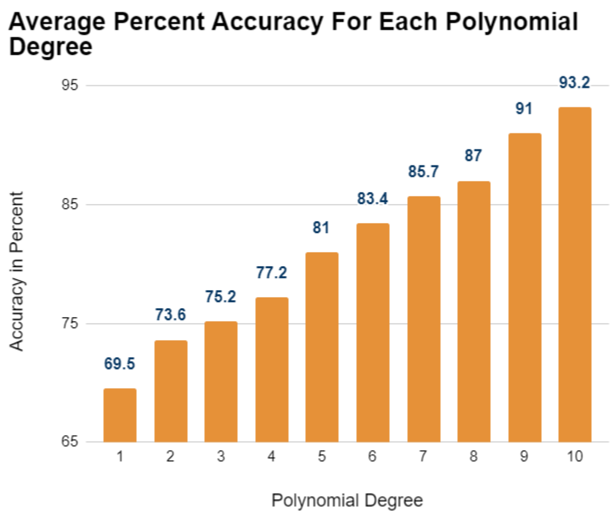
4 Data

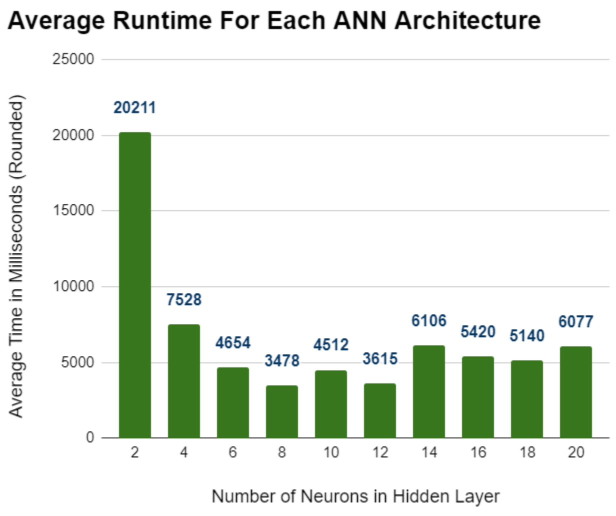
The cumulative data acquired can be summarized with the following graphs: A comprehensive collection of data can be found in Appendix B

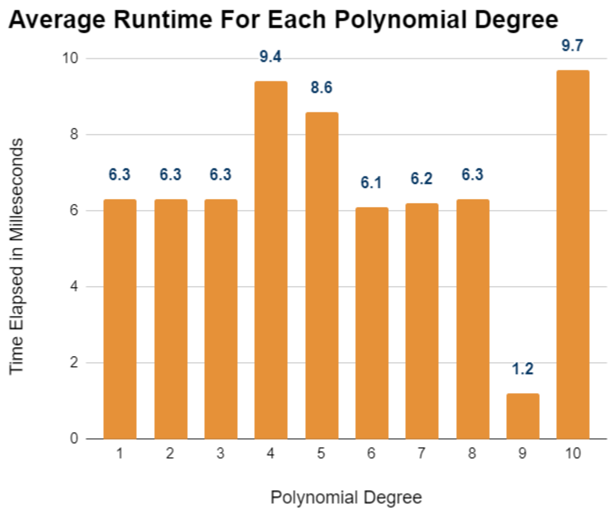


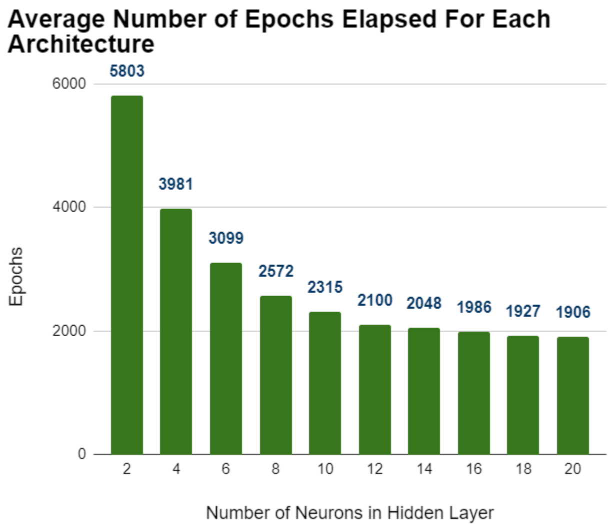












5 Conclusion

5.1 General Project Conclusions

In this project the plausibility of using Artificial Neural Networks (ANNs) as a regression model was tested. It was found that three-layer ANNs usually took longer than polynomial regression and had roughly similar accuracy, however, their rate of overfitting was incredibly lower and their usability was remarkably higher than polynomials. It was also observed that they were much less reliant on initial conditions, unlike polynomials which were very sensitive to their degree, and for this reason, ANNs are so much more resilient than their ANN counterparts. It was observed that in a three layer network, the optimum number of neurons for speed was around 12, and although there was an improvement of accuracy as the number of neurons increased, there was no big improvement after 14 neurons It is theorized that using deeper neural networks will greatly improve performance due to more layers of complexity, allowing the network to model more complicated datasets. This might introduce new problems, however, such as overfitting, but these can be solved with regularization, altering the activation function, and other techniques. It is intended to continue development of the ANN model and perform testing on multi-dimensional datasets to investigate its scalability. The full potential of ANNs as a curve fitting model was not revealed, but rather previewed in this project. It is conjectured that it will however be revealed upon further development and testing upon multivariate datasets with deeper networks, as this is where we can no longer apply standard methods with confidence.

5.2 Implications Of Data

The major implication of the collected data can be summarized as

- The accuracies of the ANN and polynomial models approached an asymptotic curve, meaning that another dimension of freedom must be introduced to break through that barrier. For ANNs, this means adding another layer. For polynomials, this does not exist.
- The ANN model never overfit to the training dataset, but rather it actually slightly underfit which is conjectured to be because the SGD training algorithm was too slow or 3 layers of neurons holds insufficient complexity to overfit.
- There was a general favorable response of runtime of the ANN model to its complexity. Adding more layers is expected to decrease runtime a little more.

Unlike the polynomial graphs, there exists variation in the ANN graphs as parameters are initialized randomly (local minima)

5.3 Real-World Applications

Curve fitting has a huge plethora of applications in the real-world, some of which include:

- Predicting concrete failure time through manufacturing data (compressive strength)
- Mass spectrometry (Applications in radar, sonar, MRI/XRAY, signal processing).
- Behavioral analysis of populations
- Migration patterns of refugees
- Comparing reaction rates (analyzing effects if various catalysts)
- Predicting motion of one mechanism given another that is contiguous to it, or contiguous to other mechanisms that are contiguous to the mechanism of interest.

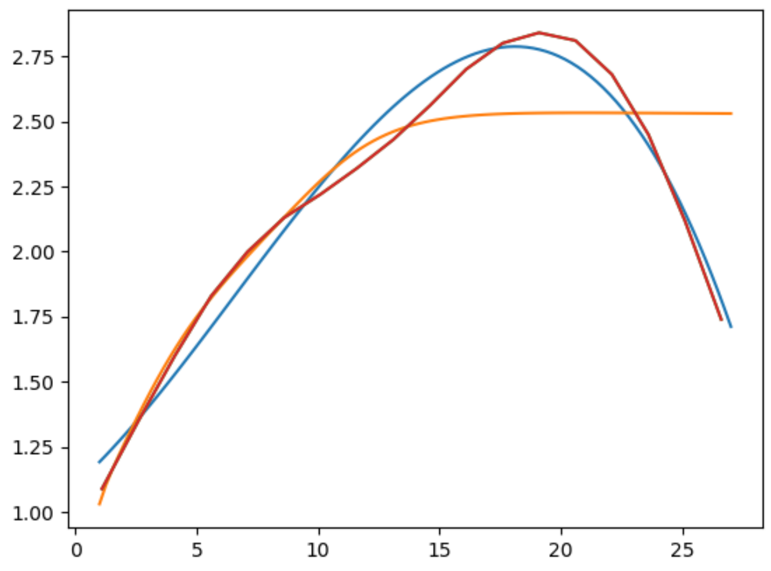
The last point on this list was lightly tested with ANNs and polynomial and although it was found that the polynomial model attained an accuracy higher by 1.3% (given that the correct order was chosen), it was demonstrated that ANNs are indeed well working predictors of natural phenomena.

Background: The Deep drawing press is a metal forming operation which essentially compresses metals with a hydraulic press to alter the shape of the metal.The mechanism used for this consists of three 4 bar chains which link the crank to the sliding press. The process of curve fitting through an ANN has been applied to the link mechanism of this machine.

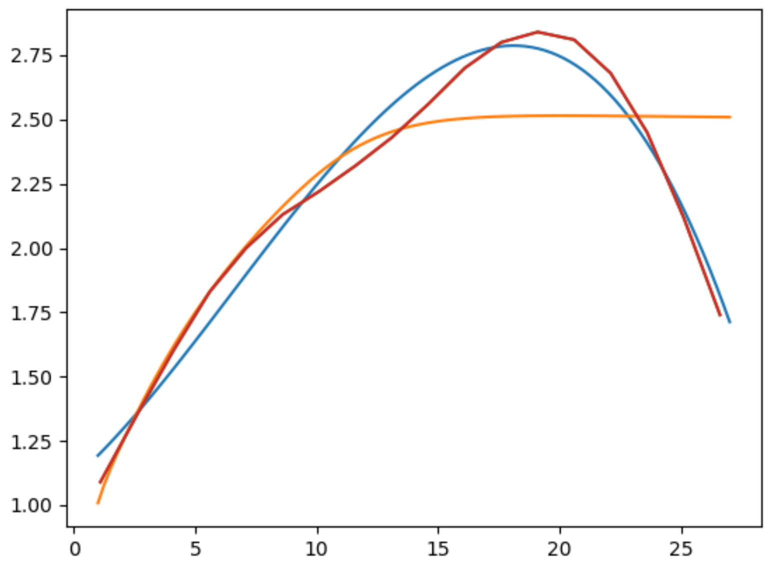
Training Data: [(0, 46.9), (60, 53.8), (90.0, 58.527), (120, 61.453), (150, 63.302), (180, 65.824), (220, 68.436), (270, 64.256), (313, 53.727), (320, 51.918)]

Test Data: [(91, 50.9), (106, 53.5), (121, 56), (136, 58.3), (151, 60), (166, 61.3), (181, 62.2), (196, 63.2), (211, 64.3), (226, 65.6), (241, 67), (256, 68), (271, 68.4), (286, 68.1), (301, 66.8), (316, 64.5), (331, 61.2), (346, 57.4)]

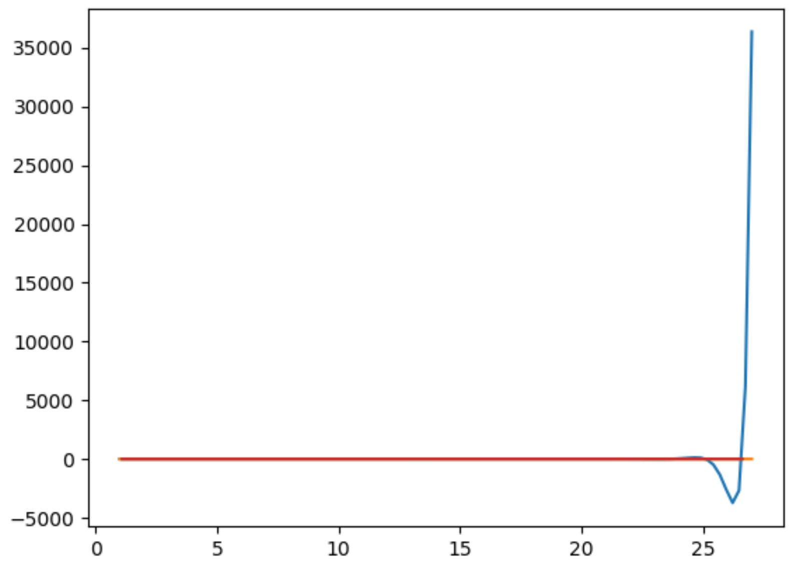
Again, the orange model is an ANN ([1-15-1]), the blue model is polynomial (degree 3) and the red line is the pure function.



What is interesting is the effect of drastically changing the number of degrees of freedom for each model. For example, changing the ANN architecture to [1-50-1] while maintaining the polynomial model at degree 3 yields the following graph.



However, if we consider the effect of changing the polynomial degree to 50 while maintaining the ANN model at [1-15-1] we get the following graph.



The polynomial has overfit to such an extent that when there is a lack of training data near the right edge of the graph, it explodes. In comparison to the ANN model, the polynomial model has a serious overfitting problem. What's more is that the number of degrees of freedom in the ANN model is actually double the amount of the polynomial one due to 50 weights from layer 1 to layer 2 and likewise from layers 2 to 3. Now add on to this the number of individual bias values and we see that the ANN model is remarkably more useful for cases where we don't know the data complexity and hence cannot determine the optimal degree of the polynomial that may curve fit the data.

5.4 Future Research

Future research is intended to be on curve fitting with more complex ANNs, particularly deeper ones. Three layer ANNs has shown that ANNs in general have a huge amount of potential in numerical curve fitting and increasing the number of dimensions of freedom will unleash it. The following is a rough list of future work to be done.

- Using a different gradient descent algorithm. SGD is actually the second slowest. Adagrad or some other momentum based one seems desirable as it avoid local minima.
- Testing different network architectures. Testing deeper networks is hypothesized to vastly decrease runtime and increase usability, and networks mimicking multivariate functions are hypothesized to be more practical
- Using the ReLU or Hyperbolic Tangent activation functions. Using the hyperbolic tangent functions as the activation functions for the network could expand the range of the network to negative values, and using ReLU activation functions such as Leaky ReLU could get rid of the need to scale and rescale data entirely. Both of these increase network usability and to a certain extent, accuracy and speed.
- This is the most important of all: Testing in a more practical way. Current testing was limited to simulating datasets, but curve fitting real-world datasets provides to the testing process a more accurate representation of what the network will be used for and thus is more rigorous.

5.5 Error Analysis

The following are some choices that may have affected the results:

- There were five functions used for testing the models. Using more functions to test the two models would have caused more accurate results since they would model a much wider variety of graph shapes.
- The choice of functions. Sufficiently high degree polynomials fit every function with a high degree of accuracy ($>$ 90%) except for the asymptotic Function 2 due to the fact that polynomials have poor asymptotic properties. The representation of asymptotic function used while testing may not accurately reflect the distribution of data in the real world which models an asymptotic pattern, and for that reason the data could have been skewed. This problem can be solved in the future by increasing the diversity of the functions used.

- The constant choice of η, the learning rate. For different architectures, the optimum learning rates can be different and therefore the data might have been skewed towards the architectures for which $\eta = 6.5$ is the optimum learning rate. This issue can be solved in the future by incorporating a random hyperparameter search algorithm for optimizing the learning rate based on the architecture.