

Machine Learning and Real Roots of Polynomials

By

ZEKAI ZHAO

SENIOR THESIS

Submitted in partial satisfaction of the requirements for Highest Honors for the degree of

BACHELOR OF SCIENCE

in

MATHEMATICS

in the

COLLEGE OF LETTERS AND SCIENCE

of the

UNIVERSITY OF CALIFORNIA,

DAVIS

Approved:

Jesús A. De Loera

June 2019

ABSTRACT.

This senior thesis is about an experiment that uses artificial neural network technology to predict the number of real roots of a polynomial.

In this thesis, I introduce some machine learning and deep learning technologies that I am using in this experiment, and I highlight their popularity. In addition, I review some background about roots of univariate polynomials, and I compare the performance of traditional method of computing the number of real roots of a polynomial with the one of the machine learning method. Then I illustrate my experiment, which includes explanations of why I choose recurrent neural network for this experiment, how do I obtain the data, how I train the model and what are the results.

Contents

Chapter 1. Introduction	1
1.1. Machine Learning	1
1.2. Neural Network	2
1.3. Recurrent Neural network	9
Chapter 2. The Roots of Univariate Polynomials	12
2.1. Number of Roots	12
2.2. Algorithms	12
Chapter 3. Experiments	16
3.1. Why RNN	16
3.2. Datasets	16
3.3. Model Description	16
3.4. Results	17
Chapter 4. Future directions	19
Acknowledgements	20
Bibliography	21
Appendix A. MATLAB Code	22
Appendix B. Python Code	25

CHAPTER 1

Introduction

In this thesis, we are going to build a machine learning model that predicts the number of real roots of a given polynomial. Machine learning, especially deep learning, is extremely popular now. It is interesting to see how machine learning can perform on algebra, so we decided to test it on determining the number of real roots of a polynomial. For humans, we cannot get the answer with a glance, and the traditional algorithm (e.g., Sturm's) is also time-consuming on solving this problem. The difficulty makes this experiment meaningful.

To better introduce the model, we will first introduce some machine learning algorithms and structures that we are going to use. Then we will review the concepts of real roots and algorithms of computing the number of real roots of a polynomial. Finally, we will explain the experiment: how we obtained our training data, what programming languages and packages we were using, how we improved the performance of the model and the experiment results.

1.1. Machine Learning

Machine learning is an algorithm that computers use to learn from the data. Here is a general definition:

[Machine learning is the] field of study that gives computers the ability to learn without being explicitly programmed.

– Arthur Samuel, 1959

Unlike the traditional way of programming to solve a problem, which gives the specific rules and makes the computer solve the problem following the rules (see Figure 1), machine learning method feeds the computer with data, making it learn the rules by itself with algorithms (see Figure 2). This method will be efficient especially in those tasks with complicated rules (e.g., spam mail filter).

Machine learning is now everywhere in our life. For example, Amazon uses machine learning to train model to provide customers with online shopping recommendations [2]; Google uses machine learning to build models for auto-drive cars [3]; and DeepMind uses it to create the most powerful Go “player” [4]. The growth of the computational power makes the machine learning model building faster than ever.

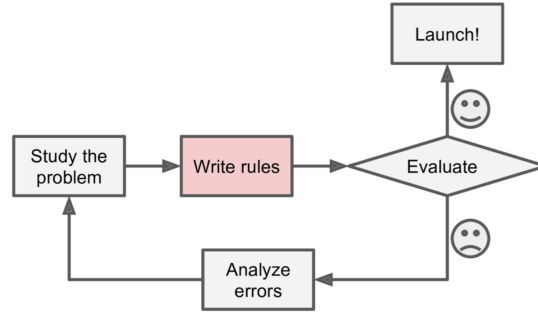


FIGURE 1. The traditional approach.
Picture taken from [1].

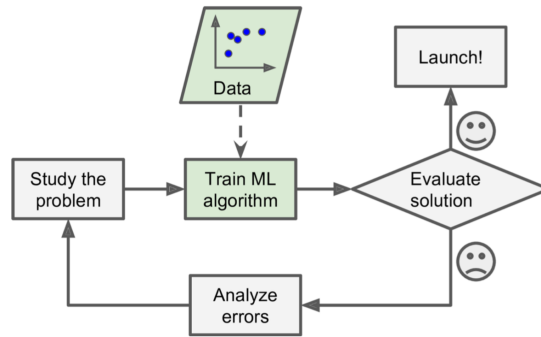


FIGURE 2. The machine learning approach.
Picture taken from [1].

1.2. Neural Network

Neural network (or artificial neural network) is

”...a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs.”

– Dr. Robert Hecht-Nielsen

It is one of the machine learning models. The basic idea behind it is inspired by the biological neural network (see Figure 3). The basic computational unit in the neural network is neuron (node). From Figure 4, we can see the basic structure of the neural network. Let us take a close look at its structure.

1.2.1. Layers. *Layers* are the main components of an artificial neural network. There are three types of layers.

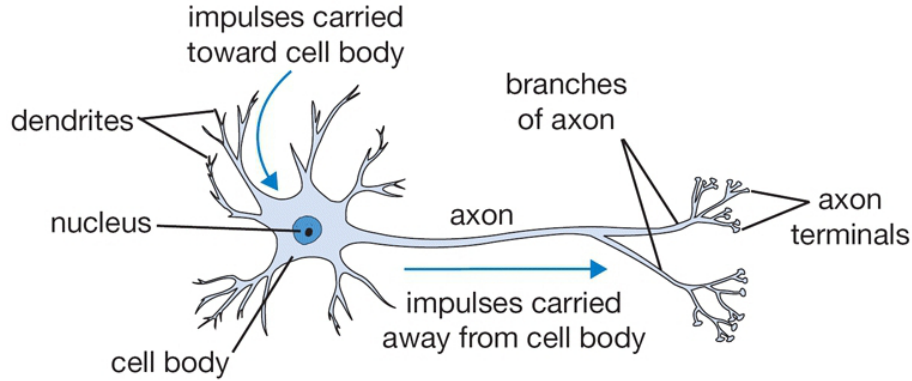


FIGURE 3. Biological neuron.
Picture taken from [5]

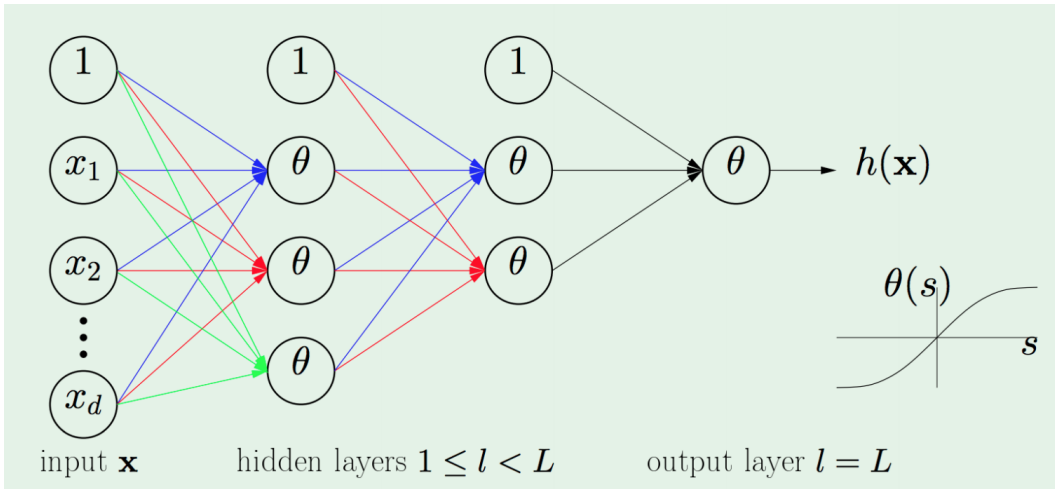


FIGURE 4. Computational neuron
Picture taken from ECS 171 lecture notes by Cho-Jui Hsieh

1.2.1.1. *Input Layer.* Inputs are placed in this layer. In most cases, inputs are in vector form, where every block of the vector is fed into every node of the input layer. No computation is done in this layer.

1.2.1.2. *Hidden Layers.* Most computations are done in these layers. By passing data from the input layer to the output layer (or the next hidden layer), the hidden layer computes and transfers the information (weights) to the next layer. J-th neuron in the l-the layer can be represented as

$$(1.1) \quad x_j^{(l)} = \theta(s_j^{(l)}) = \theta\left(\sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)} x_i^{(l-1)}\right),$$

where θ is the *activation function* and \mathbf{w} is the *weight*.

1.2.1.3. *Output Layer.* The output is generated by using activation functions in this layer. In most cases, the output nodes have two status: 1 (On) or 0 (Off). The output is represented as

$$(1.2) \quad h(x) = x^{(L)}.$$

1.2.2. Connections and Weights. Connections are between neurons (nodes) and each layer. Every node in the previous layer has connections to all the nodes in the next layer. There are weights on all connections, determining if the previous information (node) is important or not (valued from 1 down to 0).

We represent weight \mathbf{w} as

$$(1.3) \quad W_{ij}^l = \begin{cases} 1 \leq l \leq L & \text{:layers} \\ 0 \leq i \leq d^{(l-1)} & \text{:inputs} \\ 1 \leq j \leq d^{(l)} & \text{:outputs} \end{cases}.$$

The whole purpose of neural network training is to find good weights that lead to making correct predictions for training sets.

1.2.3. Activation Functions. Since the status of output nodes of the neural network can only be on or off. It needs a transfer function to map the original results into the final output. The activation functions map the results of neural network computation into a range such as 0 to 1.

1.2.3.1. *Sigmoid.* The sigmoid function is one of the popular activation functions. It is defined by

$$(1.4) \quad S(x) = \delta(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1}.$$

It is in the “S” shape (Figure 5), and it monotonically increases in the \mathbb{R} domain, with returning results from 0 to 1. Meanwhile, since it has a well-defined nonzero derivative everywhere, at every step, *gradient descent* method can make process.

1.2.3.2. *tanh.* tanh (Figure 6) is another activation function defined by

$$(1.5) \quad \tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1},$$

and we can find a relation between the sigmoid function and the tanh function:

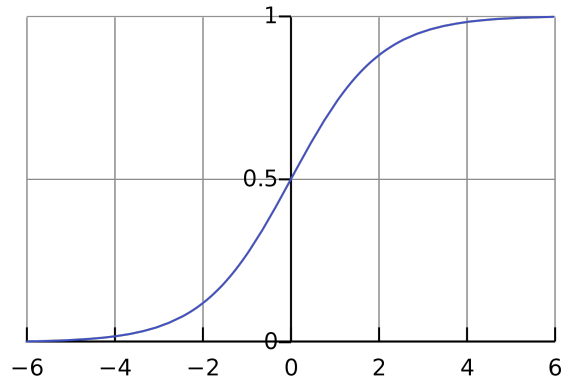


FIGURE 5. The sigmoid function

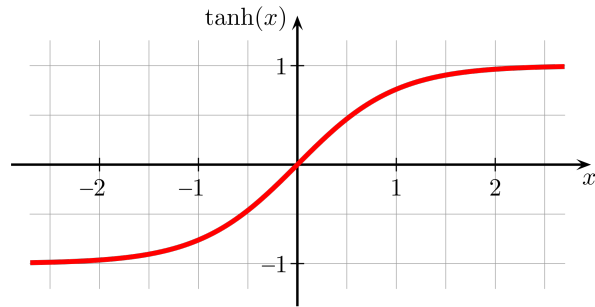


FIGURE 6. The tanh function

$$(1.6) \quad \tanh(x) = 2S(2x) - 1.$$

The domain of \tanh function is \mathbb{R} and the function range is $(-1, 1)$. Tanh function shows up frequently in the neural network.

1.2.3.3. *ReLU*. The ReLU function is defined by

$$(1.7) \quad \text{ReLU}(x) = \max(0, x).$$

It is continuous but not differentiable at $x = 0$. However, it performs good in practice.

After introducing the structure of the neural network, let us take a look at an example to see how neural network works in a general way.

EXAMPLE 1. A very simple example of the neural network application can be recognizing the handwritten digits, where each input image here has 28×28 pixels. The input layer of the model will have 784 nodes, and each node is fed by one pixel of an image. In our example model, there are two hidden layers (see Figure 7). The output layer contains number 0 – 9. As we can see in Figure 7, in the hidden layer one, the model recognizes the five components (colored in the figure) from 784 pixels. Then with the information of components in hidden layer one, the model recognizes the two larger components: circle in the above and a vertical bar in the bottom (number “8” has the same circle component, and number “1” has the same vertical bar component). With these two components, the ANN model can clearly classify our input image as the number “9”.

1.2.4. Training Process. Then let us dive in and see how the learning process is operated. Here are some definitions before we continue.

DEFINITION 1. An *epoch* is a measure of the number of times all of the training vectors are used once to update the weights.

DEFINITION 2. A *cost function* is a function that returns a numerical value as a measure of how close the prediction is to the actual answer (if given any). We can also refer to the cost function as a *loss function* or an *objective function* [7].

In different cases, we use different cost functions. For example, in the logistic regression model, we make predictions according to the estimated probability, and the cost function here is defined as

$$(1.8) \quad C(\theta) = \begin{cases} -\log(\hat{p}) & \text{if } y = 1 \\ -\log(1 - \hat{p}) & \text{if } y = 0 \end{cases},$$

where \hat{p} is the probability and $-\log(\hat{p})$ will be large when \hat{p} is close to 0. While, in some other models we use quadratic cost function to measure the average distance between the prediction and the correct answer, which is defined as

$$(1.9) \quad C(\mathbf{w}, \mathbf{b}) = \frac{1}{2n} \sum ||y(\mathbf{x}) - a(\mathbf{w}, \mathbf{b})||^2,$$

where:

- \mathbf{w} is the weight vector;
- \mathbf{b} is the biases vector;
- n is the total number of training data;
- \mathbf{x} is the training data vector;
- $y(\mathbf{x})$ is the labeled value vector, which is the correct answer;
- $a(\mathbf{w}, \mathbf{b})$ is the predict value vector.

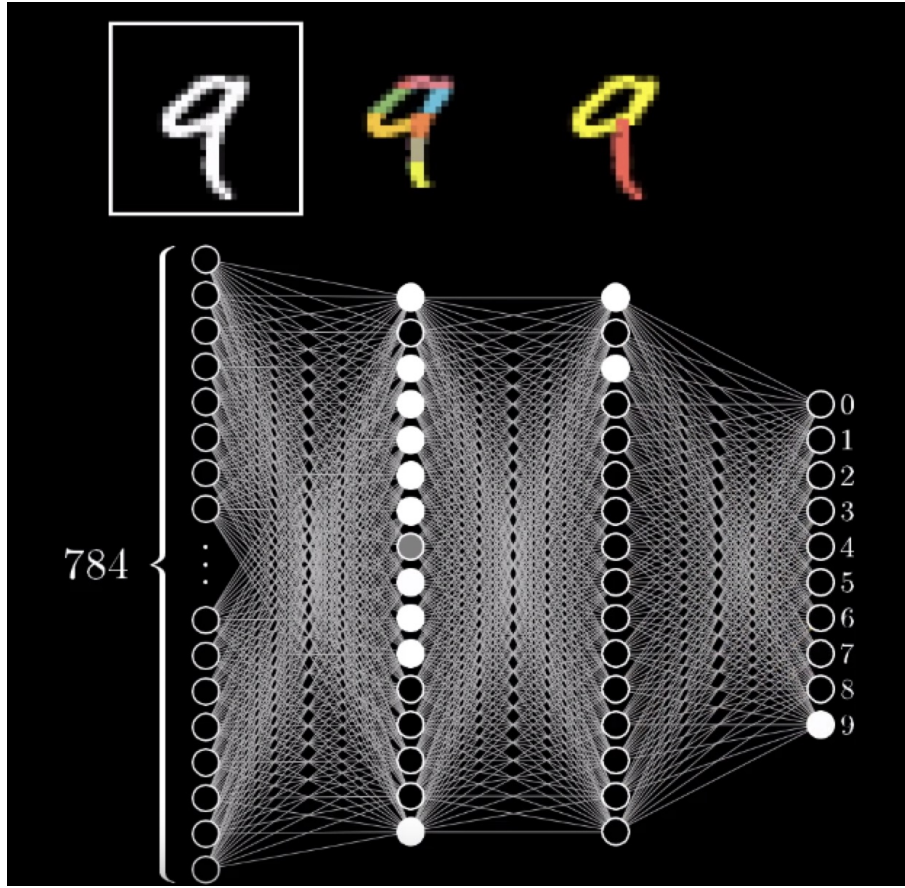


FIGURE 7. Handwriting writing digit model layers visualization, where white nodes are activated nodes (value close to 1); Black nodes are not activated nodes (value close to 0). Picture taken from [6]

The whole process of the neural network training is to look for a set of optimized weights between different layers in order to minimize the loss (i.e., the value of the cost function). The smaller value the cost function has, the closer distance our predictions are from the correct solutions.

In detail, in every epoch, the algorithm feeds every training instance to the neural network and calculates the output. Then it compares the output with the correct answer and measures the error (i.e., the value of a cost function). After that, it computes how each node in the previous layer contributes to the error. Then it keeps computing the contributes of nodes in previous layers until the input layer. This *backtrack process* measures error gradient across all the connection weights. The last step is applying the gradient descent algorithm on connection weights to minimize the error.

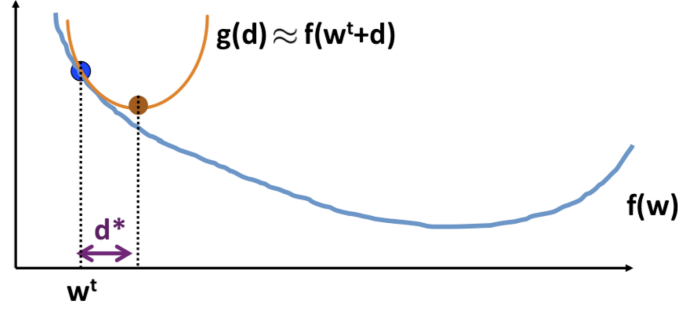


FIGURE 8. Illustration of gradient descent algorithm
image from ECS 171 lecture notes by Cho-Jui Hsieh

DEFINITION 3. *Gradient descent* is an optimization algorithm that utilizes the gradient of a function to find its minimum.

Gradient descent is one of the most important components of the training process. We will see how gradient descent algorithm looks for the minimum of the cost function.

For a convex cost function $f(x)$, one can easily find the global minimum by $\nabla f(x) = 0$. For a non convex function, we use gradient descent: repeatedly do

$$(1.10) \quad \mathbf{w}^{t+1} \leftarrow \mathbf{w}^t - \alpha \nabla f(\mathbf{w}^t),$$

where α is the *step size* and $\alpha > 0$. The choice of step size is crucial. If the step size is too big, the gradient descent will diverge, while if the step size is too small, the convergence speed will be too slow.

Let us do a step of gradient descent by hand. As shown in Figure 8, the cost function is $f(\mathbf{w})$. We start at $\mathbf{w} = \mathbf{w}^t$, and we assume the first step is \mathbf{d} . Then we can form a quadratic approximation:

$$(1.11) \quad f(\mathbf{w}^t + \mathbf{d}) \approx g(\mathbf{d}) = f(\mathbf{w}^t) + \nabla f(\mathbf{w}^t)^T \mathbf{d} + \frac{1}{2\alpha} \|\mathbf{d}\|^2.$$

After that we minimize $g(\mathbf{d})$:

$$(1.12) \quad \nabla g(\mathbf{d}^*) = 0 \Rightarrow \nabla f(\mathbf{w}^t) + \frac{1}{\alpha} \mathbf{d}^* = 0 \Rightarrow \mathbf{d}^* = -\alpha \nabla f(\mathbf{w}^t).$$

After getting the \mathbf{d}^* , we are able to update the weight:

$$(1.13) \quad \mathbf{w}^{t+1} = \mathbf{w}^t + \mathbf{d}^* = \mathbf{w}^t - \alpha \nabla f(\mathbf{w}^t).$$

We keep doing previous steps until we reach or close to the minimum, where $\nabla f(\mathbf{w}) \approx 0$.

1.3. Recurrent Neural network

Human never builds thoughts from scratch. For example, people will analyze the meaning of a word according to the previous word or context. That is the motivation of creating a *recurrent neural network* (RNN). Unlike traditional artificial neural network (NN) model (e.g., feed-forward neural network), RNN has memories of the previous input. For example, if your input is “She likes shopping,” in the view of NN, sentences “She likes shopping” and ”shopping likes she” have the same meaning since NN does not care too much about the order. All the input nodes have no relations with each other. However, in RNN, only ”She likes shopping” is correct, which is what we expect. With memory, RNN has the ability to search correlations between events in different periods (long-term dependencies).

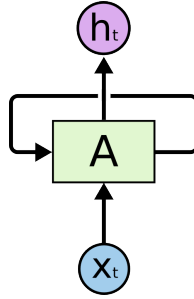


FIGURE 9. Loop of recurrent neural network
Picture taken from [8]

The structure of RNN is similar to the structure of NN, except there is a loop in the hidden layer (Figure 9). A formula of computing the hidden layer of RNN can be

$$(1.14) \quad h_t = \phi(W\mathbf{x}_t + Uh_{t-1}),$$

where:

- h_t is the hidden state at time t ;
- \mathbf{x}_t is the input vector;
- W is the weight;
- h_t is the hidden state at previous time t ;
- U is the memory rate;
- ϕ is an activation function, which can be a sigmoid function or tanh.

1.3.1. LSTM.

DEFINITION 4. *Long Short Term Memory* (LSTM) is the structure inside RNN that controls memorizing or forgetting the context.

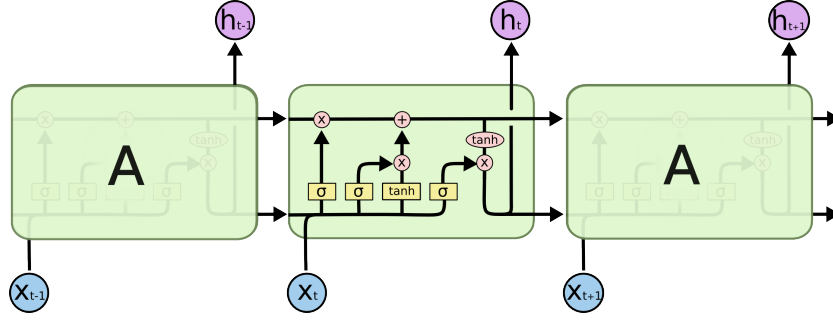


FIGURE 10. LSTM Model
Picture taken from [8]

There are three parts of the LSTM mode (Figure 10). The first part is to decide what information to forget in the current state:

$$(1.15) \quad f_t = \sigma(W_f \cdot [h_{t-1}, \mathbf{x}_t] + \mathbf{b}_f),$$

where f_t is a number from 0 to 1, which determines how completely we should keep the memory of previous state C_{t-1} in current state C_t .

The second part is to decide how many new information we are going to remember in the current cell state:

$$(1.16) \quad i_t = \sigma(W_i \cdot [h_{t-1}, \mathbf{x}_t] + \mathbf{b}_i),$$

and

$$(1.17) \quad \bar{C}_t = \tanh(W_c \cdot [h_{t-1}, \mathbf{x}_t] + \mathbf{b}_C),$$

where \bar{C}_t is the information candidate and i_t determines how much we should remember the candidate information.

In the third part, we first update the current cell state:

$$(1.18) \quad C_t = f_t * C_{t-1} + i_t * \bar{C}_t.$$

Then we calculate the output h_t :

$$(1.19) \quad o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o),$$

and

$$(1.20) \quad h_t = o_t * \tanh(C_t).$$

The sigmoid layer decides the output cell states, and the tanh is an activation function that compresses the value into -1 to 1. C_t and h_t will then enter the next cell state loop.

CHAPTER 2

The Roots of Univariate Polynomials

I found definitions and theorems in this chapter from Professor De Loera's notes.

We have started to learn to solve the equations or find the roots of a polynomial since elementary school. Polynomials and roots are basic mathematics components. In this chapter, we will review some definitions, theorems, and algorithms about them.

2.1. Number of Roots

DEFINITION 5. A root or a zero of a polynomial is a complex vector \bar{c} such that $f(\bar{c}) = 0$

THEOREM 1 (Fundamental theorem of algebra (Gauss)). Let $P(x) = a_d x^d + a_{d-1} x^{d-1} + \dots + a_2 x^2 + a_1 x + a_0$ be a polynomial with coefficients on \mathbb{C} (or \mathbb{Q} or \mathbb{R} or any subfield of \mathbb{C}) then the polynomial P has d roots, counting multiplicities, in the field of complex numbers.

According to the fundamental theorem of algebra, a polynomial $P(x)$ of degree n has n roots.

THEOREM 2 (complex conjugate root theorem). if P is a polynomial in one variable with real coefficients, and $a + bi$ is a root of P with a and b real numbers, then its complex conjugate $a - bi$ is also a root of P [9].

There are two types of roots: real roots and complex roots, where, according to the complex conjugate root theorem, a complex root always appears with its complex conjugate. Therefore, complex roots always appear in pairs. Thus there is a finite number of combinations of the number of roots of a polynomial. For example, for a polynomial of degree six, the combinations of its roots can only be

- no real root and six complex roots;
- two real roots and four complex roots;
- four real roots and two complex roots;
- six real roots but no complex root.

2.2. Algorithms

There are several existing algorithm that helps us find the number of real roots of a polynomial. One is *Descartes' rule of signs*.

2.2.1. Descartes' rule of signs.

DEFINITION 6. For a polynomial $p(x)$, two consecutive terms present a *sign variation* if their coefficients have different signs.

THEOREM 3 (Descartes' rule of signs). *The number of positive real roots of a real equation either is equal to the number v of sign variations or less than v by an even integer. Here roots of multiplicity m are counted m times.*

Knowing the sign variation, we can use Descartes' rule of signs to get a rough guess of the number of roots of a polynomial in the positive domain. Let's take a look at some examples.

EXAMPLE 2. $x^6 - 3x^2 + x + 1 = 0$ has either two positive real roots or none.

However, since Descartes' rule of signs only gives hints for the number of positive, we need to use some tricks to get a better guess.

EXAMPLE 3. $f(x) = x^4 + 3x^3 + x - 1 = 0$. What are its roots? By Descartes' rule of signs it has one root in the positive part of the line, which is not a multiple root. $f(-x) = x^4 - 3x^3 - x - 1 = 0$ has one sign variation, which means $f(x) = 0$ has one root in $x < 0$.

COROLLARY 1. The number of negative real roots of $f(x) = 0$ is equal to the number of positive roots of $f(-x)$.

With this corollary, we are able to guess the number of real roots.

LEMMA 1. *If $c > 0$ is a positive real number, and if $(x - c)f(x)$ is equal to $F(x)$, the number of sign variations of $F(x)$ is equal to that of $f(x)$ increased by a positive odd integer.*

Although one can run this algorithm in a very short time to make a guess of the number of real roots of a polynomial, the probability of making a correct selection from the guess results is really low. For a polynomial of degree six, unless it has a same number of positive signs and negative signs, the probability of a correct selection is below 50%, where our neural network model has 97% accuracy (it will be illustrated in the next chapter). The low accuracy makes it not practical.

To make an accurate computation, we need to use *Sturm's Sequence*.

2.2.2. Sturm's Sequences: Consider now the problem of determining the number of real roots of $f(x)$ inside $[a, b]$.

DEFINITION 7. The (canonical) *Sturm sequence* of f is given by $P_0 = f(x)$, $P_1 = f'(x)$, .., $P(i) = -\text{rem}(P_{i-2}(x), P_{i-1}(x)) = P_{i-1}(x)Q_{i-2}(x) - P_{i-2}(x)$, where $\text{rem}(P_{i-2}(x), P_{i-1}(x))$ is the remainder when polynomial long division is used to divide $P_{i-2}(x)$ by $P_{i-1}(x)$, and

$Q_{i-2}(x)$ is the quotient of said long division. Let P_m be the last non-zero polynomial in the sequence.

Note: $P_i(x)$ is the negative of the remainder (a modified GCD sequence).

THEOREM 4 (Sturm 1892). *Let f be a polynomial without multiple roots. If $a < b$ in \mathbb{R} and neither is a root of $f(x)$, then the number of DISTINCT real roots of $f(x)$ inside the interval $[a, b]$ is the number of sign changes in the sequence $[P_0(a), P_1(a), \dots, P_m(a)]$ minus number of sign changes in $[P_0(b), P_1(b), \dots, P_m(b)]$.*

Compared to Descartes' rule of signs, Sturm's Sequence can give the specific number of real roots in a range.

EXAMPLE 4. Find the distinct real roots of $x^3 + x + 1$.

First, we compute Sturm's sequence: $x^3 + x + 1$, $3x^2 + 1$, $-\frac{2}{3}x - 1$, $-\frac{31}{4}$. Then we check how many real roots in $[-1, 1]$:

$$\begin{cases} [-1, 4 - \frac{1}{3}, -\frac{31}{4}] \rightarrow 2 \text{ sign changes} \\ [3, 4, -\frac{5}{3}, -\frac{31}{4}] \rightarrow 1 \text{ sign change} \end{cases} \Rightarrow 1 \text{ real root in } [-1, 1].$$

If we let $a = -\infty$ and $b = \infty$, then we can use Sturm's Sequence to find the number of real roots in \mathbb{R} . However, computing all the derivatives and the remainder is time-consuming.

EXAMPLE 5. We try to compute the number of real roots of $P := 33x^6 + 43x^5 - 38x^4 - 60x^3 + 51x^2 - 11x + 10$, with both Sturm's theorem and recurrent neural network model (it will be mentioned in the next chapter). For Sturm's theorem, We ran it on *Maple* from the workstation at Math department, UC Davis. For the recurrent neural network model, We implemented and ran it on a laptop (CPU: 2.2GHz Core i7-8750H 6 CPU Cores). Results are shown as below.

```
[> P:= 33*x^6+43*x^5-38*x^4-60*x^3+51*x^2-11*x+10;
      6      5      4      3      2
      P := 33 x  + 43 x  - 38 x  - 60 x  + 51 x  - 11 x + 10

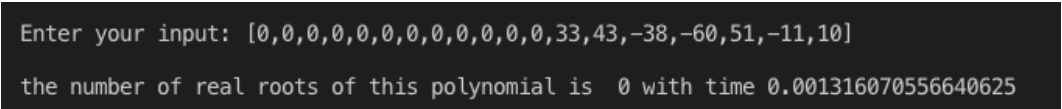
[> st:=time[real]():sturm(sturmseq(P,x),x,-infinity,infinity):time[real]()-st;
      0.106

[> st:=time():sturm(sturmseq(P,x),x,-infinity,infinity):time()-st;
memory used=4.9MB, alloc=8.3MB, time=0.20
      0.011

[> sturm(sturmseq(P,x),x,-infinity,infinity);
      0
```

FIGURE 1. Running time with *Maple*

As we can see from Figure 1 and Figure 2, Both methods give us the same and correct solution. The processing of finding the number of real roots for polynomial P with Sturm's



```
Enter your input: [0,0,0,0,0,0,0,0,0,0,0,0,33,43,-38,-60,51,-11,10]
the number of real roots of this polynomial is 0 with time 0.001316070556640625
```

FIGURE 2. Running time with RNN model

theorem took 0.011 seconds, and the Machine Learning method took 0.0013 seconds, which runs about 7.4 times faster than the normal algorithm. It again proves the feasibility of applying machine learning on this problem.

CHAPTER 3

Experiments

The entire experiment includes generating the dataset, building the model and testing the model.

3.1. Why RNN

Sometimes it is hard to determine which type of neural network to use to solve a problem. There are thousands of types of neural network structure, including the modified version [10]. At first, we were using normal feedforward neural network to solve this problem. However, the results become terrible as the degree of polynomial increases. Our model does not learn anything from our training data in that situation. Then we decided to change the neural network structure, and we found the recurrent neural network (RNN). Since RNN has LSTM layers, it has the ability to remember the previous inputs, in other words, the order of inputs nodes matters. In our experiment, the input vector is the coefficients of a polynomial, and the order of coefficients matters (the coefficient of degree six term is different from the coefficient of degree five term of a polynomial). Thus, we decided to use RNN to do this experiment, and it turned out to be a success.

3.2. Datasets

Table 1 lists the training examples used to learn the number of real roots of a polynomial. We generate polynomials of degree d by randomly choosing (using uniformly distributed random numbers) the $d + 1$ coefficients. For each polynomial, we then count the number of real roots n using MatLab. This is done for each degree d from three to six until we have enough training and test instances. The feature vector we use for training is just the vector of coefficients of length seven (since the degree is bounded by six). We believe the model trained based on natural distribution is the most useful model (people look for the number of real roots of those polynomials most often), though we also did experiments on even distribution model.

3.3. Model Description

We chose five layers in our model. Beside the input and output layer, we constructed two LSTM layers both with 200 nodes. After that, we built a dense layer with 100 nodes. Drop out rates (percentage of inputs to remember) for these three layers were 0.2, 0.1, and

deg.	# real roots	# complex roots	# data(ND)	# data(ED)
3	1	2	7783	1000
3	3	0	2217	1000
4	0	4	2537	1000
4	2	2	7127	1000
4	4	0	336	1000
5	1	4	6630	1000
5	3	2	3341	1000
5	5	0	29	2000
6	0	6	1949	1000
6	2	4	7211	1000
6	4	2	837	1000
6	6	0	3	2000

TABLE 1. Training data used for learning the number of real roots of polynomials on a natural distribution (ND) with 10,000 examples in each degree, and on an even distribution(ED) where the samples are chosen such that there are 2000 examples for each number of real roots from 0 to 6.

0.2. Input layer was fed by a vector of coefficients of a polynomial. Output layer gave the predicted number of real roots of a given input polynomial.

For the whole process, we randomly chose 80% of the dataset to be training data, 10% as validation data, and the result 10% as test data.

Before the training process, the weights between layers were initialized to a small number. With learning rate = 0.001, after each epoch of training, the weights changed to achieve a lower error (loss) by the back-propagation algorithm

The loss vector was computed according to the cross entropy algorithm in our model:

$$(3.1) \quad Loss(t) = desired(t) - predicted(t),$$

where $desired(t)$ was the correct number of real roots, and $predicted(t)$ was the predicted result.

After about 40 epochs, the loss converged to a small number.

3.4. Results

For the natural distribution data set, the total accuracy is about 97%, with a mean squared error below 0.017. For the even distribution data set the total accuracy is about 95% (See Figures 1 and 2). The results show that our model successfully learned to identify the number of real roots of a polynomial up to degree 6. This experiment serves as a proof

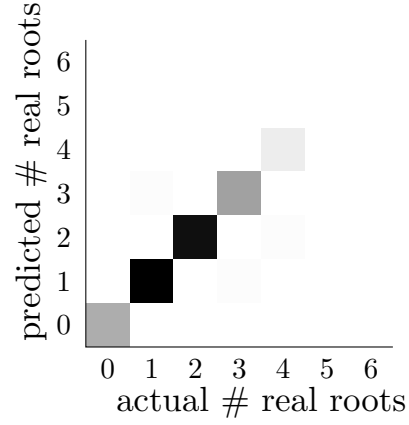


FIGURE 1. Confusion matrix for polynomials of degrees 3-6 on natural distribution.

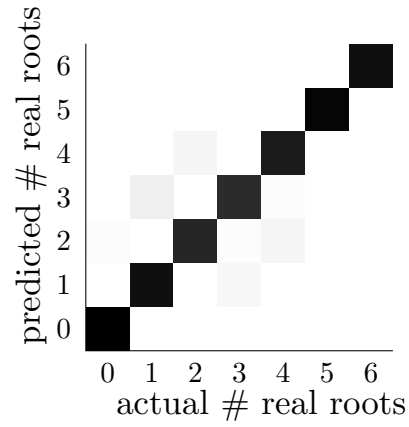


FIGURE 2. Confusion matrix for polynomials of degrees 3-6 on even distribution.

of concept that machine learning (neural networks) can be applied to polynomial algebra problems.

CHAPTER 4

Future directions

In this chapter, we will discuss different ways this study can be extended.

The first direction can be testing the algorithm on polynomials with higher degrees. In the previous experiments, we used a normal feedforward neural network to test on polynomials with up to degree 18. The results were bad. However, we haven't tried using the recurrent neural network on this problem. In the future, we can apply the recurrent neural network on predicting the number of real roots of high degree polynomials and to see if it works as good as on polynomials with degrees up to six.

Another direction can be testing on the model with multivariable polynomials (e.g., $x^2y^3 + xz^2 + y^2 + z = 0$). It will be more challenging since the coefficient-only input data may not work as well as before. We need to come up with another good way to describe and represent the polynomial. One possible solution is use features such as the number of terms in the polynomial, the number of terms contains variable x , and etc.

The experiment in Chapter three proves that machine learning or deep learning can work well in some areas of algebra, but where does it work well and where does it fail is still unknown. According to the fact that the neural network solves the problem in its own way, it is hard to analyze how it specifically works.

Acknowledgements

We thank the National Science Foundation for support provided via the NSF grant 1818969 for Prof. De Loera.

Bibliography

- [1] Aurlien Gron. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems*. OReilly, 2018.
- [2] Amazon. Real-time personalization and recommendation. <https://aws.amazon.com/personalize/>, 2019.
- [3] Andrew J. Hawkins. Inside the lab where waymo is building the brains for its driverless cars. <https://www.theverge.com/2018/5/9/17307156/google-waymo-driverless-cars-deep-learning-neural-net-interview>, May 2018.
- [4] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dhharshan Kumaran, Thore Graepel, and et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):11401144, 2018.
- [5] David Fumo. A gentle introduction to neural networks series - part 1. <https://towardsdatascience.com/a-gentle-introduction-to-neural-networks-series-part-1-2b90b87795bc>, Aug 2017.
- [6] 3Blue1Brown. But what *is* a neural network? — deep learning, chapter 1. <http://www.3blue1brown.com/>, Oct 2017.
- [7] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.
- [8] Christopher Olah. Understanding lstm networks, Aug 2015.
- [9] Gary McGuire and A. G. OFarrell. *Maynooth Mathematical Olympiad manual*. Logic Press, 2002.
- [10] Jason Brownlee. When to use mlp, cnn, and rnn neural networks. <https://machinelearningmastery.com/when-to-use-mlp-cnn-and-rnn-neural-networks/>, Jul 2018.

APPENDIX A

MATLAB Code

In this section, we are going to present the MATLAB code to generate the two datasets used for this experiment.

The first part is the code that generates the even distribution dataset.

```
0 %% create the files
  file_coef = fopen('coef.txt','w');
2 file_nroots = fopen('n_roots.txt','w');
  file_record = fopen('record.txt','w');
4
  %% for degree 3 to 6
6 for expon = 3:6
    n_roots = zeros(1,expon+1);
8    ceiling = 40000; %% normally we do 40000 times iterations
    if expon == 5 || expon == 6
10        ceiling = 300000 %% if degree is 5 or 6, we do more iterations
    end
12
    for j = 1:ceiling
14        f = [];
        for i = 1:18-expon
16            f(end+1) = 0; %% set 0 on all other places
        end
18        rand_number = round(rand*200-100); %% randomly generate a number
        from -100 to 100
        while(rand_number == 0)
20            rand_number = round(rand*200-100);
        end
22        f(end+1) = rand_number; %% to avoid 0 at first place
        for i = 1:expon
24            f(end+1) = round(rand*200-100);
        end
26        root = roots(f); %% get the root of polynomial
        root = root(imag(root) == 0); %% get all the polynomial with
no image root
28        for k = 0: 0
            if((length(root) == k && n_roots(k+1) < 1000) || (length(
root) == k && (expon == 5 || expon == 6) && k == expon && n_roots(k
+1) < 2000))
30                fprintf(file_coef, '[ ');
                fprintf(file_coef, '%d, ', f(1:end-1));
```

```

32         fprintf(file_coef, '%d', f(end));
33         fprintf(file_coef, ']\n');
34         n_roots(k+1) = n_roots(k+1)+1;
35         fprintf(file_nroots, '%d\n', k);
36     end
37 end
38
39 end
40
41 fprintf(file_record, 'expon = %d \n', expon);
42
43 for k = 0: 0
44     fprintf(file_record, '# of %d roots = %d \n', k, n_roots(k+1));
45 end
46
47 end
48 fclose(file_record);
49 fclose(file_nroots);
50 fclose(file_coef);

```

./code/randomPoly_direct_even.m

The second part is the code that generates the uneven distribution dataset.

```

0 file_coef = fopen('coef.txt', 'w');
1 file_nroots = fopen('n_roots.txt', 'w');
2 file_record = fopen('record.txt', 'w');
3 top_degree = 18;
4
5 for expon = 3:top_degree
6     n_roots = zeros(1, expon+1);
7     for j = 1:10000 %% iterate 10000 times for each degree
8         f = [];
9
10        rand_number = round(rand*200-100);
11        while(rand_number == 0)
12            rand_number = round(rand*200-100);
13        end
14        f(end+1) = rand_number; %% to avoid 0 at first place
15        for i = 1:expon
16            f(end+1) = round(rand*200-100);
17        end
18        derivate_f = polyder(f); %% get the derivate of the polynomial
19        root = roots(f); %% compute the roots
20        root = root(imag(root) == 0); %% select unimage roots
21        k = length(root);
22
23        newf = [zeros(1, max(0, top_degree+3-numel(f))), f] ;
24        new_derivate_f = [zeros(1, max(0, top_degree+2-numel(f))),
25        derivate_f];
26
27        %%result includes coefficient of polynomials and it's derivatives

```

```

28         result = [newf new_derivate_f];
30
31         %% printing the result
32         fprintf(file_coef, '[' );
33         fprintf(file_coef, '%d, ', result(1:end-1));
34         fprintf(file_coef, '%d', result(end));
35         fprintf(file_coef, ']\n');
36         n_roots(k+1) = n_roots(k+1)+1;
37         fprintf(file_nroots, '%d\n', k);
38
39     end
40
41     fprintf(file_record, 'expon = %d \n', expon);
42     for k = 0: expon
43         fprintf(file_record, '# of %d roots = %d \n', k, n_roots(k+1));
44     end
45
46 end
47 fclose(file_record);
48 fclose(file_nroots);
49 fclose(file_coef);

```

./code/randomPoly_direct_uneven.m

APPENDIX B

Python Code

This part presents the python code used for

- data preprocessing;
- data training;
- data testing;
- results analysis.

```
0 import tensorflow as tf
1 from tensorflow import keras
2 import time
3 import ast
4 import numpy as np
5 import matplotlib.pyplot as plt
6 from collections import Counter
7 from sklearn import svm
8 from sklearn import linear_model
9 import itertools
10 from sklearn.preprocessing import RobustScaler
11 import random
12 import glob
13 import os
14
15
16
17 input_file = '3-hypergraphs-on-15-vertices'
18 folder_path = 'PolyLearn/data-direct-even/'
19
20 n_test = 100 # of data points to reserve for testing
21 n_val = 100 # of data points to validate during training
22 draw = 0 # 1 if draw the validation acc vs training acc graph; 0
23     otherwise
24
25 #CNN
26 n_nodes_hl = [80] # number of nodes in each hidden layer
27
28 #RNN
29 n_nodes_dense = [100] # number of nodes in each dense hidden layer
30 n_nodes_LSTM = [200,200] # number of nodes of LSTM layers
31 n_drop_out = [0.2,0.1,0.2]
32
33 hm_epochs = 40 # number of training times
```

```

34
36 # main function
37 def main():
38
39     input_data = user_input()
40
41     data = np.array([[input_data['coef'][i], input_data['root'][i]] for i
42 in range(len(input_data['coef']))])
43     n_classes = np.amax(input_data['root']) + 1 # number of classes for
44     classification
45
46     print("Number of Classes = ", n_classes)
47     print("n_nodes_hl = ", n_nodes_hl)
48     print("hm_epochs = ", hm_epochs)
49     print("Total Number of Data = ", len(data))
50     n_train = 1 # number of experiment time
51     for i in range(n_train):
52         # if doing experiment on uneven distribution data set call this
53         function
54         train_and_test(data, n_classes)
55
56         # if doing experiment on even distribution data set call this
57         function
58         #train_and_test_fix_test_set(data, n_classes)
59         print("\n")
60
61 # training adn test on uneven distribution dataset
62 def train_and_test(data, n_classes):
63
64     np.random.shuffle(data) # randomly shuffle the data
65
66     n_test = int (0.1*len(data)) # take 10% of data as test data
67     n_train = len(data)-n_test
68     assert n_train > 0 # make sure number of training data is greater
69     than 0
70     train_data = data[0:n_train] # separate data into training data
71     test_data = data[n_train:] # and testing data
72
73     tfmodel = RNNModel(n_classes, n_nodes_hl)
74     tfmodel.train(train_data, epochs=hm_epochs)
75     # build a tensorflow RNN model
76
77     preds = tfmodel.test(test_data, n_classes)
78     # test on model
79
80     diff_from_correct_answer(preds, test_data, n_classes)
81     #analyze the results
82
83 # raining and testing on even distribution data set
84 def train_and_test_fix_test_set(data, n_classes):

```

```

82 # testing up to degree 6
83 # each number of root we pick 200 as test and rest as train
84
85 n_test = 1400
86 n_train = len(data)-n_test
87 assert n_train > 0
88 train_data = data[0:800]
89 test_data = data[800:1000]
90 for i in range(1,14): # since it's even distribution, we can obtain
data evenly
    train_data = np.concatenate((train_data, data[i*1000:i
*1000+800]), axis=0)
92     # separate data into training data
    test_data = np.concatenate((test_data, data[i*1000 + 800: i*1000
+1000]),axis = 0)
94
95
96 tfmodel = RNNModel(n_classes, n_nodes_hl)
97 tfmodel.train(train_data, epochs=hm_epochs)
98 # build a tensorflow RNN model
99
100 preds = tfmodel.test(test_data, n_classes)
101 # test the model
102
103 diff_from_correct_answer(preds, test_data, n_classes)
104 # analyze the results
105
106 # parent class for learning models
class Model:
107     def preprocess(self, data): # a function for converting input file
data to model input
        return np.array([np.array(datum) for datum in data])
110     def train(self, data): pass # train the model on a list of input/
output pairs
    def predict(self, datum): pass # predict the class of one input
112     def test(self, data, n_classes): # test the model on a list of input/
output pairs
113
114         n_data = len(data)
115         preds = [self.predict(datum) for datum in data[:, 0]]
116
117         num_acc = len([i for i in range(n_data) if preds[i] == data[i
,1]])
118
119         # computing mean square error
120         mean_squared_error = 0.0
121         for i in range(n_data):
122             mean_squared_error = mean_squared_error + ((preds[i]-data[i
,1])**2)**0.5
123         mean_squared_error = mean_squared_error / float(n_data)
124         print("mean_squared_error = ", mean_squared_error)
125
126         #compute the accuracy

```

```

128         accuracy = num_acc/float(n_data)
129         print("accuracy = ", num_acc/float(n_data))
130         return preds
131
132     def predict_stdio(self): # predict the class of input from stdio
133         while(1):
134             test_feat = input('\nEnter your input: ')
135
136             if (test_feat == '0'):
137                 break
138             try:
139                 datum = ast.literal_eval(test_feat)
140             except SyntaxError:
141                 continue
142
143             pred, start_time, end_time = self.predictWithTime(datum)
144
145             print('\nthe number of real roots of this polynomial is ',
146 pred, "with time", end_time - start_time )
147
148     def save_model(self): # save the model
149         model_json = self.model.to_json()
150         with open("./models/model.json", "w") as json_file:
151             json_file.write(model_json)
152         self.model.save("./models/PolyLearnModel.h5")
153
154     def load_model(self, file_path): # load the existed model
155         # load json and create model
156         json_file = open('./models/model.json', 'r')
157         loaded_model_json = json_file.read()
158         json_file.close()
159         self.model = tf.keras.models.model_from_json(loaded_model_json)
160         self.model.load_weights(file_path)
161
162     # RNN model
163     class RNNModel(Model):
164         # initialize and build the structure of the RNN model
165         def __init__(self, n_classes, n_nodes_hl, preprocess=None):
166             if preprocess != None: self.preprocess = preprocess
167             model = keras.Sequential()
168             print("Model initializing")
169
170             # added layers for test
171             for i in range(len(n_nodes_LSTM)):
172                 if i == 0:
173                     model.add(keras.layers.LSTM(n_nodes_LSTM[i], activation='
174 relu', return_sequences=True))
175                 else:
176                     model.add(keras.layers.LSTM(n_nodes_LSTM[i], activation='
177 relu'))
178                 model.add(keras.layers.Dropout(n_drop_out[i]))

```



```

178         for j in range(len(n_nodes_dense)):
179             model.add(keras.layers.Dense(n_nodes_dense[j], activation='
relu'))
180             model.add(keras.layers.Dropout(n_drop_out[i+j]))
181
182         model.add(keras.layers.Dense(n_classes, activation='softmax'))
183
184         opt = tf.keras.optimizers.Adam(lr=0.001, decay=1e-6)
185         model.compile(optimizer=opt,
186                       loss='categorical_crossentropy',
187                       metrics=['accuracy', 'mean_squared_error', '
mean_absolute_error'])
188         self.model = model
189         self.accuracy = 0
190         print(self.model.variables)
191
192     #function to train the model with training data
193     def train(self, data, epochs=hm_epochs):
194         input_data = self.preprocess(data[:, 0]) # preprocessing the data
195         input_data = input_data[:, input_data.shape[1]-7:input_data.shape
[1]] # abstract the non-zero vector
196         input_data = np.reshape(input_data, (input_data.shape[0], 1,
input_data.shape[1]))
197         tbCallBack = keras.callbacks.TensorBoard(log_dir='./Graph',
198                                                  histogram_freq=0, write_graph=True, write_images=True)
199
200         self.history = self.model.fit(input_data, keras.utils.
to_categorical(data[:, 1]), verbose = 1, validation_split=0.1, epochs
=hm_epochs, callbacks=[tbCallBack])
201         if (draw == 1):
202             self.plot() # plot if it is asked
203
204     # function to predict the result
205     def predict(self, datum):
206         input_data = self.preprocess([datum])
207         input_data = input_data[:, input_data.shape[1]-7:input_data.shape
[1]] # abstract the non-zero vector
208         input_data = np.reshape(input_data, (input_data.shape[0], 1,
input_data.shape[1]))
209         return np.argmax(self.model.predict(input_data)[0])
210
211     # function to predict the result with timing
212     def predictWithTime(self, datum):
213         input_data = self.preprocess([datum])
214         input_data = input_data[:, input_data.shape[1]-7:input_data.shape
[1]] # abstract the non-zero vector
215         input_data = np.reshape(input_data, (input_data.shape[0], 1,
input_data.shape[1]))
216         start_time = time.time()
217         result = np.argmax(self.model.predict(input_data)[0])
218         end_time = time.time()
219         return result, start_time, end_time

```

```

220
222 def plot(self):
224     history = self.history
226     history_dict = history.history
228
230     # graph of training and validation loss
232     acc = history.history['acc']
234     val_acc = history.history['val_acc']
236     loss = history.history['loss']
238     val_loss = history.history['val_loss']
240
242     epochs = range(1, len(acc) + 1)
244
246     # "bo" is for "blue dot"
248     plt.plot(epochs, loss, 'bo', label='Training loss')
250     # b is for "solid blue line"
252     plt.plot(epochs, val_loss, 'b', label='Validation loss')
254     plt.title('Training and validation loss')
256     plt.xlabel('Epochs')
258     plt.ylabel('Loss')
260     plt.legend()
262
264     plt.show()
266
268     # graph of training and validation accuracy
270     plt.clf() # clear figure
272     acc_values = history_dict['acc']
274     val_acc_values = history_dict['val_acc']
276
278     plt.plot(epochs, acc, 'bo', label='Training acc')
280     plt.plot(epochs, val_acc, 'b', label='Validation acc')
282     plt.title('Training and validation accuracy')
284     plt.xlabel('Epochs')
286     plt.ylabel('Loss')
288     plt.legend()
290
292     plt.show()
294
296 # reading the data from files
298 def user_input():
300     input_data = {}
302
304     # reading the coefficients
306     lines = tuple(open(folder_path+'coef.txt', 'r'))
308     print("file name = ", folder_path+'coef.txt')
310     print("total line = ", len(lines))
312     input_data['coef'] = np.array([ast.literal_eval(lines[i]) for i in
314 range(0, len(lines))])
316     #input_data['coef'] = np.array([ast.literal_eval(lines[i]) for i in
318 range(0, 10000)])
320
322 # reading the number of roots

```

```

270     lines = tuple(open(folder_path+'n_roots.txt', 'r'))
271     print("file name = ", folder_path+'n_roots.txt')
272     print("total line = ", len(lines))
273     input_data['root'] = np.array([ast.literal_eval(lines[i]) for i in
274     range(0, len(lines))])
275     #input_data['root'] = np.array([ast.literal_eval(lines[i]) for i in
276     range(0, 10000)])
277
278     return input_data
279
280 # function to analyze the results
281 def diff_from_correct_answer(preds, data, n_classes):
282     distance = np.zeros(n_classes)
283     root_correct = np.zeros(shape=(11))
284     root_wrong = np.zeros(shape=(11))
285     root_ans_and_pred = np.zeros(shape=(11,11))
286     n_data = len(data)
287     for i in range(n_data):
288         distance[abs(preds[i]-data[i,1])] += 1
289         if abs(preds[i]-data[i,1]) == 0: # if the prediction is correct
290             root_correct[data[i,1]] += 1
291         else: # if the prediction is not correct
292             root_wrong[data[i,1]] += 1
293             root_ans_and_pred[data[i,1],preds[i]] += 1
294
295     for i in range(n_classes):
296         print("number of predict data that has distance ", i, " from the
297         correct answer is ", int(distance[i]))
298
299     for i in range(0,n_classes+1):
300         if root_correct[i]!= 0 or root_wrong[i] != 0:
301             print("For class ",i, " there are ",int(root_correct[i]), "
302             data predict correct and ", int(root_wrong[i]), " data predict wrong"
303             )
304
305         for j in range(1,n_classes+1):
306             if root_ans_and_pred[i,j] != 0:
307                 print("The number of number of real roots should be
308                 predict as", i , "but be predicted as ", j," is ", int(
309                 root_ans_and_pred[i,j]))
310
311 if __name__ == "__main__":
312     main()

```

./code/polyLearn.py