# Mathematical Paper Author Identification with Neural Network

By

TZU FENG LEUNG

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

March 2019

ABSTRACT.

This senior thesis is about using Neural Networks(NN) to identify mathematical paper authors. The data I use in my experiments consists of 180 papers from 19 different Fields Medalists.

I preform three types of experiment with NN. In the first experiment, I randomly select $n$ papers as testing data, train the NN with papers that are not chosen, and use the NN to predict the authors of the testing papers. In the second experiment, I choose one author at random and use half of his/her papers as training data and the other half as testing data. Next, from the papers that do not belong to the chosen author, I choose some amount of papers randomly and divide them into training set and testing set. I then combine the two testing data sets as one big testing set and combine the two training data sets as one big training set. Finally, I use the big training data set to train the NN and use the NN to make predictions on the big testing set. In the third experiment, I select two authors at random to preform classification. From each of the two chosen authors, I randomly choose half of his/her papers as training data and the other half as testing data. Finally, I train the NN with the testing set and use the NN to distinguish testing papers from the two authors.

I reach 68% accuracy in the first experiment, 98% accuracy in the second experiment, and 92% in the third experiment.

ACKNOWLEDGMENTS.

First, I would like to thank Professor Jesús De Loera for giving me the opportunity to write this senior thesis with him. In the process of writing this thesis, I learn to typeset in LaTeX, learn the basics of feed-forward neural networks, and learn to use Python to train and test FNN.

Moreover, I would like to offer my special thanks to my experiment partners, Xiaotie(Jessie) Chen and Michi Kirihara, for providing insightful ideas to me when I run into problems.

Lastly, I would like to thank the National Science Foundation for support provided via the NSF grant 1818969 for Professor De Loera.

# Contents

CHAPTER 1

# Introduction

## 1.1. Text Mining and Author Identification

The main topic of this thesis is to use text mining techniques and neural networks to identify authors of mathematical papers. We are interested in the following problem: given a fixed number of papers written by distinct authors, how can we use our computer to predict the author of a paper?

Author identification has many real-life applications such as identifying anonymous papers and finding plagiarism. People have conducted many author identifications on different types of writing. One example is text mining on newspaper headlines using logistic regression and convex optimization to discover how different Dutch newspapers portray the news [**9**]. Unlike other projects, this thesis focuses on mathematical papers.

In math papers, theorems and math equations convey important mathematical ideas to readers, yet a computer does not understand them when they are written in LaTeX format. Therefore, we delete all LaTeX formats from our data and only keep English words. Thus, we lose information in the deletion process and have less data to perform author identification with NN models.

## 1.2. Data

The data we have consists of 180 math papers written by Fields Medal prize winners. The papers cover different topics, such as probability theory, partial differential equations, combinatorics, and so on. Note that the Fields Medal is considered by many as the highest honor that a mathematician can get; it is sometimes described as the "Nobel Prize" for mathematicians. All the papers are all written in LaTeX syntax. To preprocess the data, we first apply a shell script to the data to remove all symbols, numbers and LaTeX formatting. For example, "`\vec{a}`" will be removed because it is written in LaTeX format. Also, "#" and "123" will be removed because the first is a symbol and the second is a number.

To transform our math papers into a format that is readable by computers, we greatly reduce the word count of the papers by deleting all **stop words**, or words that do not have significant meaning in a sentence. For example, the sentence, "The cat is brown", has two stop words: "the" and "is." They are stop words because they have little meaning and the

main idea of the sentence is the brown cat. We use the programming language, Python, to eliminate all the stop words from our 180 papers. My research partner, Michi Kirihara, and I first used the stop word list available in the NLTK python module and removed all natural language stop words. Moreover, we also created our own customized list of mathematical stop words. Some common math stop words are "definition", "example", and "assumption." After removing the unnecessary stop words, the total word count of our papers was significantly reduced and the preprocessed data consisted of mostly meaningful words.

The next task for us is to find a way to combine words in different parts of speech that have the same root. We call this part of preprocessing "intelligent stemming" the remaining words. I will illustrate the idea of "intelligent stemming" with an example. Assume that we have five "big", two "bigger", and one "biggest" in a paper; each word has its own word count. We do not want "big", "bigger", and "biggest" to be counted separately because they all mean large; Counting them separately will affect the term frequency of the word "big," which decreases its importance to the paper. To fix the problem, we write a Python script that "intelligently" stems the words. After running the script, we would combine the count of the less frequent word and add it to the word with the maximum count; in our example, since "big" has more counts compared to "bigger" and "biggest," we would have 8 "big" after running the script. After intelligent stemming, the quality of our vocabulary is further refined.

After that, we proceed to count the number of n-grams in the papers. We define **n-grams** as meaningful $n$ words that appear consecutively frequently in our paper. Examples of mathematical n-grams are: "Convex Hull," "Partial Derivative," and "Singular Value Decomposition."Here is an example of how we count bi-grams. Suppose that we have five "partial" and 8 "derivative" in our paper and the two words appear right next to each other exactly 4 times. Given the info, we would count the bi-gram, "partial derivative" as a word on its own. After counting all the n-grams in our papers, we are done with the preprocessing and ready to transform the preprocessed data into matrix form.

## 1.3. Data Representation in Matrix Form

**1.3.1. Data Matrix.** We will use an *m x n term-by-document matrix* to represent the preprocessed data. In the data matrix $A$, each row represents one math paper and each column represents a word in the papers [**4**]. Note that the matrix initially contains every word in all of the papers. In our experiment, we set matrix $A$ to have 180 rows and 3000 columns. Note that the total vocabulary in the papers is more than 3000; we pick the number 3000 because we believe 3000 is an appropriate size for computers to perform

matrix operations on $A$.

Here is what matrix A look like:

$$\text{Data Matrix A} = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} & \dots & x_{1,3000} \\ x_{2,1} & x_{2,2} & x_{2,3} & \dots & x_{2,3000} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{180,1} & x_{180,2} & x_{180,3} & \dots & x_{180,3000} \end{bmatrix} .?? \tag{1.1}$$

I will give an example of how we represents two documents in a matrix. Suppose we have two documents:

(1) "Alex has a turtle."
(2) "Natalie loves to dance."

After we remove that stop words from the two documents, we would get the following:

(1) "Alex has a turtle." $\longrightarrow$ "Alex" , "turtle"
(2) "Natalie loves to dance." $\longrightarrow$ "Natalie", "love", "dance"

Then we can transform the documents into matrix form. In this example, I use the frequency of each word in the entry of the matrix.

$$\begin{array}{c} \begin{array}{ccccc} Alex & turtle & Natalie & love & dance \end{array} \\ \begin{pmatrix} \frac{1}{5} & \frac{1}{5} & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} \end{pmatrix} \begin{array}{l} Document1 \\ Document2 \end{array} \end{array}$$

We construct data matrix $A$ with the same approach, but instead of using word frequency as entry in the matrix, I will use the numerical statistic, *tf-idf*.

## 1.4. Tf-idf

Next we will use the numerical statistic "*term frequency-inverse document frequency*" (*tf-idf*) to assign weight to each word in the data matrix. The *tf-idf* metric was introduced by the British computer scientist, Karen Spärck Jones, in 1972 [**11**], and is commonly used in text-mining.

Let $x_{ij} \in A$. We know that $x_{ij}$ means the $j^{th}$ word in document $i$. Define $W(x_{ij})$ as weight of $x_{ij}$, then according to [**8**],

$$W(x_{ij}) = tf(x_{ij}) * idf(x_{ij}),$$

where

$$tf(x_{ij}) = \frac{\text{Number of times the word appears in document i}}{\text{Total number of words in document i}},$$

and

$$idf(x_{ij}) = log\Big(\frac{N}{D_{x_{ij}}}\Big),$$

with

$$N = \text{Total number of documents in the matrix.}$$
$$D_{x_{ij}} = \text{The \# of documents in corpus that contain the word.}$$

Initially, we try using only term frequency as a measure of word importance, but we did not get good results due to the high frequency nature of stop words. Consider the stop word, "definition", in our papers. The word has a high term frequency weight, yet it means little and does not help us distinguish work from different authors. Therefore, only using the $tf(.)$ metric is not good enough; we correct for the problem by doing $tf(.) * idf(.)$.

The significance of multiplying by $idf(.)$ is as follows: we want words with high weight to have a high $tf$ and also be important and meaningful for certain papers. That is, we want to assign more weight to word that have high $tf$ in one particular paper but low $tf$ in all other papers. The intuition is that words with such characteristics are especially important to the particular papers that contains them in high frequency.Therefore, they deserve more weight. On the other hand, words that have high $tf$ in every papers are usually stop words because we know the papers cover different topics and each topic is associated with topic-specific math vocabulary.

I will now illustrate how *tf-idf* assigns low weight to stop words through an example. Let's take a look at the stop word "definition" again; it is common to all math papers, yet it provides virtually no useful information for author identification. If we assume that all the papers contains the word "definition", then:

$$idf(\text{"definition"}) = log\Big(\frac{N}{D_{\text{"definition"}}}\Big) = log(1) = 0 \implies W(\text{"definition"}) = 0.$$

So, we see that the *ti-idf* metric successfully assigned low weight to "definition", and in general, most stop words will have low weight under *tf-idf* [**2**]. In contrast, a key word for a specific paper will have high $tf$ in that specific paper only. This implies that both the $tf$ and *idf* of the word will be high, which results in higher weight for the key word. Altogether, under the *tf-idf* metric, high frequency key words that only appear in one or few papers have high weight and common words that appears in many papers have low weight [**8**].

CHAPTER 2

# A Quick Introduction to Neural Networks

In this section, I will introduce neural networks (NN) and explain how to train feed-forward neural networks (FNN).

## 2.1. Artificial Intelligence, Machine Learning, and Deep Learning

First off, I will briefly talk about artificial intelligence, machine learning, deep learning, and their relations: **Artificial intelligence (AI)** is a field of computer science that aims to use machines to automate tasks that usually require human intelligence [**1**]; **Machine learning (ML)** is a subset of AI that uses algorithms to analyze data, learn from the data, and come up with decision rules based on what it learned from the data; finally, **deep learning (DL)** is a subfield of ML that uses neural network models to help machines learn from successive layered data representation [**7**]. Note that DL is a subset of ML, and ML is a subset of AI.

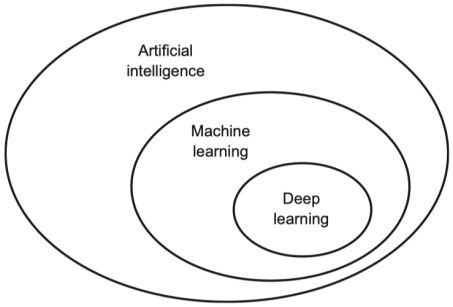


FIGURE 1. Relation of AI, ML, and DL from [**5**]

Another key point is that ML is fundamentally different from classical programming: in classical programming, rules and data are explicitly given to the machine to compute the desired answer, whereas in ML, data and answers are given to a machine and the machine would try to come up with the rules to solve for the answer. Hence, "all ML systems are trained rather than explicitly programmed" [**5**].

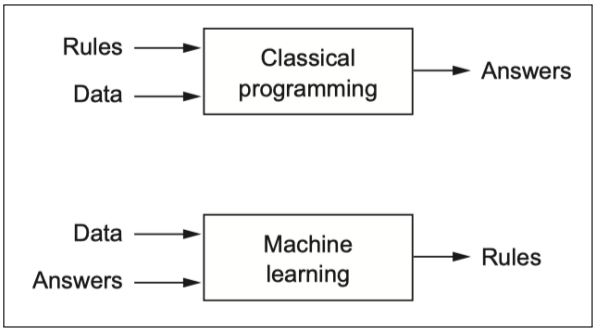


FIGURE 2. Difference between classical programming and machine learning from [**5**]

## 2.2. Structure of Neural Network Models

There are many classes of neural network models in DL. For example, feed-forward neural networks, recurrent neural networks, and convolutional neural networks are commonly used in industry. For this project, I choose to use feed-forward neural network because it is the simplest NN model.

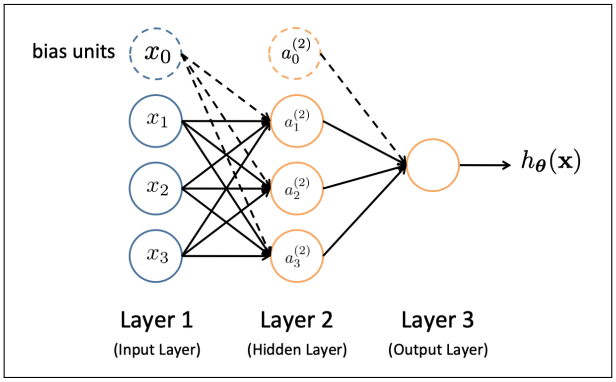


FIGURE 3. Structure of NN from [**6**]

Figure 3 is an example of a three-layer FNN. According to the book [**10**] written by Michael Nielsen, Neural network models are composed of layers. A **layer** is a set of nodes. Three types of layers exist: the input layer, the hidden layer, and the output layer; all of the layers are made up of nodes. The **input layer** is where we input the data to the NN. The **hidden layer** is all the layers that are in between the NN's first and last layer; they usually contain fewer nodes than the input layer. Finally, the **output layer** is the place where the NN make the prediction. In the input layer and each of the hidden layers, there exists a bias node. The bias node represents constant value.

Each node in the NN has an **input value**, an **activation function**, and an **output value**. Nodes are connected by **links**; each link has a **weight** attached to it.

For a NN model with $L$ layers, all nodes in the $i^{th}$ layer are connected to all non-bias nodes in the $i+1^{th}$ layer. In Figure 3, we see that all nodes in layer one is connected to each node in layer two except for the bias node. In other words, nodes $x_0, x_1, x_2, x_3$ are connected to $a_1^{(2)}, a_2^{(2)}, a_3^{(2)}$. Bias nodes are always *not* connected to the nodes in their respective previous layer. Also, all nodes in the same layer are not connected.

## 2.3. Feed-forward Process

Next, I will talk about how feed-forward neural networks work (FNN). FNN always start at the input layer and work forward; FNN never works backward.

I will first illustrate the feed-forward process with an example and then talk about the general case. Recall that each node in the FNN has an input value, an activation function, and an output value.

**2.3.1. Notation for Feed-Forward Process.** Assume that our NN model has $L$ layers.

Definition 1. Let $n_x^{(y)}$ be $x^{th}$ node in the $yth$ layer.

Definition 2. Let $W^{(y)}$ denote the weight matrix that contains the weight of each link that connects layer $y-1$ and layer $y$.

Note that:

(1) The rows would represent the nodes in layer $y$ and the columns are nodes in layer $y-1$.
(2) $W^{(y)}$ have dimension (# of nodes in layer $y$) $\times$ (# of nodes in layer $y-1$).
(3) $W_{jk}^{(y)}$ is the weight of the link that connects (node $k$ in layer $y-1$) and (node $j$ in layer $y$).

Definition 3. Define $B^{(y)}$ to be the bias nodes in layer y. Next, define $b_x^{(y)}$ to be the associated bias node for the $x^{th}$ node in layer $y$:

$$b_x^{(y)} = W_x^{(y)} * B^{(y-1)}.$$

where

- Vector $\vec{W_x^{(y)}}$ = row $x$ of matrix $W^{(y)}$. i.e., weights of links of nodes that are connected to $n_x^{(y)}$, in the same order as $\vec{a^{(y-1)}}$.

Note that This definition says that the specific bias node, $b_x^{(y)}$, is the product of the bias node in layer $y$ and the weight of the links between $n_x^{(y)}$ and $B^{(y-1)}$. Notice that each node in layer two or in any layer after has its own unique bias node.

DEFINITION 4. The **input value** of node $n_x^{(y)}$, $Z_x^{(y)}$, is defined as:

$$Z_x^{(y)} = \vec{W_x^{(y)}} \cdot \vec{a^{(y-1)}} + b_x^{(y)}$$

, where:

(1) The operation "$\cdot$" means dot product between two vectors of the same size.
(2) Vector $\vec{a^{(y-1)}}$ = all activation of nodes in the layer $y-1$ that are connected to $n_x^{(y)}$.
(3) Vector $\vec{W_x^{(y)}}$ = row $x$ of matrix $W^{(y)}$. i.e., weights of links of nodes that are connected to $n_x^{(y)}$, in the same order as $\vec{a^{(y-1)}}$.

DEFINITION 5. Let $a_x^{(y)}$ be the activation value of node $n_x^{(y)}$, then $a_x^{(y)} = g_x^{(y)}(Z_x^{(y)})$

Where $g_x^{(y)}(.)$ is the **activation function** of node $n_x^{(y)}$ and $Z_x^{(y)}$ is the **input value** of node $n_x^{(y)}$.

Once again, we look at Figure 3. $x_0, x_1, x_2, x_3$ are in the input layer: we can also call them $b^{(1)}, a_1^{(1)}, a_2^{(1)}, a_3^{(1)}$. In order to compute the nodes values in layer two, we will do the following:

(1) Compute the input value of each node in the second layer
(2) Plug the input value into each node's respective activation function and compute value of each node.

Here I will demonstrate how we compute $a_1^{(3)}$, the value of the first node in layer three. First, we use the input layer's nodes to compute the nodes values in layer two. By our definitions, we get that:

$$W^{(2)} = \begin{bmatrix} W_{00}^{(2)} & W_{01}^{(2)} & W_{02}^{(2)} & W_{03}^{(2)} \\ W_{10}^{(2)} & W_{11}^{(2)} & W_{12}^{(2)} & W_{13}^{(2)} \\ W_{20}^{(2)} & W_{21}^{(2)} & W_{22}^{(2)} & W_{23}^{(2)} \\ W_{30}^{(2)} & W_{31}^{(2)} & W_{32}^{(2)} & W_{33}^{(2)} \end{bmatrix}, \tag{2.1}$$

and

$$\begin{aligned} a_1^{(2)} &= g(\big[W_{11}^{(2)} * a_1^{(1)} + W_{12}^{(2)} * a_2^{(1)} + W_{13}^{(2)} * a_3^{(1)} + b_1^{(2)}\big]), \\ a_2^{(2)} &= g(\big[W_{21}^{(2)} * a_1^{(1)} + W_{22}^{(2)} * a_2^{(1)} + W_{23}^{(2)} * a_3^{(1)} + b_2^{(2)}\big]), \\ a_3^{(2)} &= g(\big[W_{31}^{(2)} * a_1^{(1)} + W_{32}^{(2)} * a_2^{(1)} + W_{33}^{(2)} * a_3^{(1)} + b_3^{(2)}\big]). \end{aligned}$$

Altogether, we get that,

$$a_1^{(3)} = g(\big[W_{11}^{(3)} * a_1^{(2)} + W_{12}^{(3)} * a_2^{(2)} + W_{13}^{(3)} * a_3^{(2)} + b_1^{(3)}\big]).$$

In general, the feed-forward process starts at the input layer and then computes the value of nodes in layer two, layer three, etc. , all the way until we reach the output layer! The activation of nodes in the output layer would be the prediction of the NN.

## 2.4. Training Neural Networks

From the previous section, we learn how NN preform the feed-forward process and make predictions given any input data vector. In our data matrix, each row represents an input vector; each input vector has a corresponding correct "answer." We define the **label vector** as the vector that contains all the corresponding answers from the input data. To train the NN, we will divide the data into two parts: the **training data** and the **testing data**, and divide the data label into: **training label** and **testing label**. It is of the utmost importance that each data vector is paired with the correct label vector. The splitting of the data is up to the user to configure. Some possible divisions are "50-50" and "70-30" splits, where the first number is the percent of training data and the second is the percent of testing data. We first train the NN using the training data. After that, we find the accuracy of the NN by making a prediction of the training data and comparing them to the testing label.

We often train our neural network with the same training data multiple times. To keep track of the number of times we train the NN, we define an **epoch**.

Definition 6. One **epoch** means that the NN has completed one feed-forward process and one back propagate process with all the training inputs.

For example, if we have 100 training inputs and mini-batch size of 20, then NN would have to perform 5 feed-forward processes and 5 back propagate processes to complete one epoch (since $20 * 5 = 100$).

## 2.5. Cost Functions

To measure how close the NN's prediction is to the actual answer, we use a **cost function** . In essence, the smaller the cost, the better the prediction.

Definition 7. 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 [**10**].

To make things more concrete I will define the **quadratic cost function**:

Definition 8. We define the **quadratic cost function** as:

$$C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a(w,b)\|^2,$$

where:

- $w$ is the vector that contains all the weight used in NN.
- $b$ is the vector that contains all biases used in NN.
- $n$ is the total amount of training data.
- $x$ is a training vector in the training data.
- $y(x)$ is the corresponding label vector to input vector $x$.
- $a$ is the vector that contains all the activation of nodes in the output layer (the prediction). $a$ is a function of of $x, w$, and $b$. (i.e. $a = a(x, w, b)$).
- the notation $\|\vec{v}\|$ means the length of a vector.

From the quadratic cost function, we observed that the better the prediction of the NN, the less the cost will be. We will use the optimization algorithm **gradient descent** to tune weights $w$ and the biases $b$ of the NN in order to minimize the cost function [**10**].

Moreover, I use the cross-entropy cost function in my NN models. I will define it formally below.

Definition 9. The **cross-entropy function** is defined as follows [**10**]:

$$CE(a) \equiv -\frac{1}{n}\sum_{x}\big[y * ln(a) + (1-y)ln(1-a)\big],$$

where

- $a = \sigma(z)$, where $\sigma$ is the activation function and $z = \sum_j w_j x_j + b$.
- $x$ is all the training inputs $\implies \sum_x$ is summed over all training inputs.
- $y$ is all the training labels.

## 2.6. Gradient Descent

Recall from the previous section that our current goal is to find weights $w$ and biases $b$ that minimize cost function $C(w, t)$. To illustrate the intuition of gradient descent, I will talk about a special case where the cost function only takes in two inputs values, $v_1$ and $v_2$. Consider $C(v_1, v_2)$. We can plot the cost function in a three-dimensional Cartesian plane. For simplicity, we can think that $C(v_1, v_2)$ has the following shape:

Note that $C(v_1, v_2)$ can take any shape in general; Figure 4 is merely an example. From here, our goal is to find $v_1$ and $v_2$ such that we locate the global minimum of cost function. Here is an analogy of Gradient descent: imagine that the $C(v_1, v_2)$ plot is a valley, and the two points $(v_1, v_2)$ represent a ball's position in the plot. In order to reach the minimum, we just need to find the direction down the valley and move the ball towards that direction. Naturally, one direction will not necessarily be enough to reach the bottom. Hence, we must constantly make adjustments to the direction of the ball to make sure the ball arrives at the minimum. I will present this idea with math notations.

Definition 10. Let the $\Delta v_1$ denote a small change in the variable $v_1$ and $\Delta v_2$ denote a small change in $v_2$.

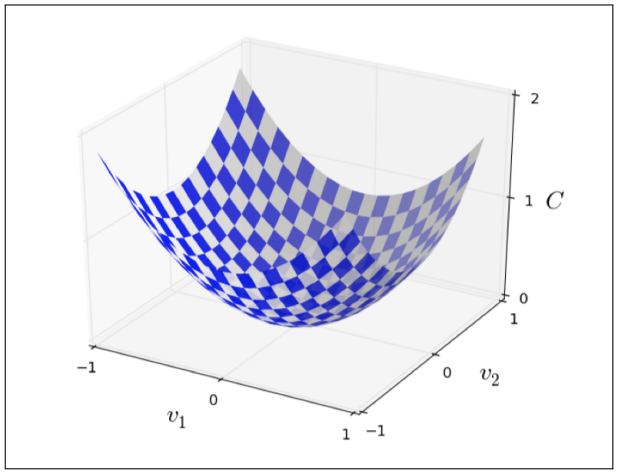


FIGURE 4. Example plot of cost function from [**10**]

By calculus, we know that:

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2, \tag{2.2}$$

where C is the cost function.

This equation says that a change in $C(v_1, v_2)$ can be approximated by computing the partial derivative with respect to each variable of $C$ and then move $v_1$ and $v_2$ accordingly. Next I will define what a gradient vector is.

DEFINITION 11. Let $C(v_1, v_2, ..., v_n)$ be any function, the **gradient of C** is defined as $\nabla C$, where:

$$\nabla C = \left[\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}, \ldots, \frac{\partial C}{\partial v_n}\right]^T.$$

Note that "T" means the transpose of a vector.

DEFINITION 12. Let $v_1, v_2, ..., v_n$ be inputs of cost function C, then define the vector $\Delta v$ as:

$$\Delta v = \left[\Delta v_1, \Delta v_2, ..., \Delta v_n\right]^T.$$

Using the notations described above, we can rewrite Equation 2.2 as:

$$\Delta C \approx \nabla C \cdot \Delta v. \tag{2.3}$$

Recall from calculus that the gradient vector $\nabla C$ points to the direction that increases C the most. Therefore, we will use this property of $\nabla C$ and move in the **opposite** direction to decrease $C(v_1, v_2)$. We will carefully choose $\Delta v$ to ensure that $\Delta C$ is negative (which means that the $C$ is slowly decreasing).

For instance, we can choose

$$\Delta v = -\eta \nabla C. \tag{2.4}$$

DEFINITION 13. Let $\eta$ be a positive parameter, we call $\eta$ the **learning rate** of the NN.

By plugging Equation 2.4 into Equation 2.2, we get

$$\begin{aligned} \Delta C &\approx \Delta v \cdot \nabla C \\ &= (-\eta \nabla C) \cdot \nabla C \\ &= -\eta \|\nabla C\|^2. \end{aligned}$$

Since $\|\nabla C\|^2 \geq 0$, $\Delta C$ is guaranteed to be $\leq 0$. We will make frequent adjustments to $\nabla C$ and $\Delta v$. In each update, we compute the amount to move $v$ by Equation 2.4, which yields:

$$v \longrightarrow v' = v + \Delta v = v - \eta \nabla C. \tag{2.5}$$

Each time we update $\Delta v$, we would decrease $C$ and get closer and closer to a global or local minimum of $C$.

Finally, we are ready to define gradient descent.

DEFINITION 14. **Gradient descent** (GD) is an algorithm that aims to minimize a function $C$ by repeatedly computing the gradient $\nabla C$ and then moving in the opposite direction of the gradient.

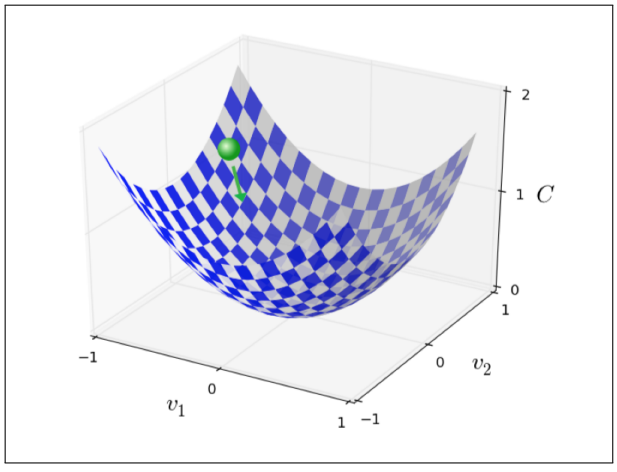


FIGURE 5. Figure to illustrate intuition of GD from [**10**]

From Figure 5, we see that the ball has a direction attach to it. That direction is obtained by computing gradient $-\nabla C$. To apply GD to the NN, we need to find $\nabla C$ of $C(w, b)$, where

$$\nabla C = [\frac{\partial C}{\partial w_1}, \ldots, \frac{\partial C}{\partial w_n}, \frac{\partial C}{\partial b_1}, \ldots, \frac{\partial C}{\partial b_m}]^T.$$

After we have computed $-\Delta C$, we adjust each weight $w_k$ and $b_l$ by:

$$w_k \longrightarrow w_k' = w_k - \eta \frac{\partial C}{\partial w_k}. \tag{2.6}$$

$$b_l \longrightarrow b_l' = b_l - \eta \frac{\partial C}{\partial b_l}. \tag{2.7}$$

## 2.7. Stochastic Gradient Descent

Notice that the quadratic cost function is actually the average of sum of cost for each individual training vector. That is:

$$\begin{aligned} C(w, b) &\equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2 \\ &= \frac{1}{n} \sum_x C_x, \end{aligned}$$

where:

- $n$ is the number of training vectors.
- $C_x \equiv \frac{\|y(x)-a\|^2}{2}$, which is the cost of **one** training vector $x$.

It follows that we must average each $\nabla C_x$ to get the gradient $\nabla C$:

$$\nabla C = \frac{1}{n} \sum_x \nabla C_x.$$

It is common to use at least thousands of training data to train NN. Therefore, it would a long time to use all of the training vectors to compute $\nabla C$. To speed up the process, we will use an algorithm call **stochastic gradient descent** [**10**].

Definition 15. Given N training inputs, $I = \{X_1, ..., X_N\}$. We say a subset of $I$ is a **mini-batch** of size m if each element in the subset is selected from $I$ randomly and $m < N$.

For example, let $I = \{X_1, ..., X_{100}\}$. One possible mini-batch of size $m = 5$ is $\{x_1, x_{50}, x_{60}, x_{70}, x_{99}\}$.

Definition 16. **Stochastic gradient descent** is an algorithm that works by picking a mini-batch from the training data and then uses the gradient of each member in the

mini-batch to estimate the true $\triangledown C$. That is:

$$\frac{1}{m}\sum_{j=1}^{m}\triangledown C_j \approx \frac{1}{n}\sum_{x}\triangledown C_x = \triangledown C,$$

where $m$ is the size of the mini-batch and $n$ is total number of input vectors.

With the stochastic gradient descent algorithm, we are able to approximate the true gradient $\triangledown C$ at a fast rate and optimize the NN quicker.

To adjust the weights and biases, we can first choose a mini-batch and then compute the gradient of the mini-batch and approximate $\triangledown C$:

$$\triangledown C \approx \frac{1}{m}\sum_{j=1}^{m}\triangledown C_j. \tag{2.8}$$

It follows that:

$$w_k \longrightarrow w_k' = w_k - \eta\Big[\frac{1}{m}\sum_{j=1}^{m}\frac{\partial C_{x_j}}{\partial w_k}\Big]. \tag{2.9}$$

$$b_l \longrightarrow b_l' = b_l - \eta\Big[\frac{1}{m}\sum_{j=1}^{m}\frac{\partial C_{x_j}}{\partial b_l}\Big]. \tag{2.10}$$

## 2.8. Backward Propagation

Backward Propagation is the way we compute the gradient of the cost function $\triangledown C$. Previously, we know that:

$$C(w,b) \equiv \frac{1}{2n}\sum_{x}\|y(x)-a\|^2 \Longleftrightarrow C_x(w,b) = \|y(x)-a\|^2,$$

$$Z_j^{(L)} = \Big[\vec{W_j^{(L)}}\cdot\vec{a^{(L-1)}} + b_j^{(L)}\Big],$$

$$a_j^{(L)} = \sigma(Z_j^{(L)}).$$

Note:

(1) $x$ denotes an input vector.
(2) There are a total of $N$ input vectors.
(3) $L$ denotes the layer of the node.
(4) $j$ denotes the $j^{th}$ node in the layer.

The goal is to calculate the gradient of the cost function $\triangledown C$:

$$\triangledown C = \big[\frac{\partial C}{\partial w_1},\ldots,\frac{\partial C}{\partial w_n},\frac{\partial C}{\partial b_1},\ldots,\frac{\partial C}{\partial b_m}\big]^T,$$

where:

$$\frac{\partial C}{\partial w_{jk}^{(L)}} = \frac{1}{N}\sum_{x}\frac{\partial C_x}{\partial w_{jk}^{(L)}}.$$

Using the chain rule from calculus, we get that:

$$\frac{\partial C_x}{\partial w_{jk}^{(L)}} = \frac{\partial C_x}{\partial a_j^{(L)}} * \frac{\partial a_j^{(L)}}{\partial Z_j^{(L)}} * \frac{\partial Z_j^{(L)}}{\partial w_{jk}^{(L)}}. \tag{2.11}$$

$$\frac{\partial C_x}{\partial b_j^{(L)}} = \frac{\partial C_x}{\partial a_j^{(L)}} * \frac{\partial a_j^{(L)}}{\partial Z_j^{(L)}} * \frac{\partial Z_j^{(L)}}{\partial b_j^{(L)}}. \tag{2.12}$$

By calculus, we get that:

$$\begin{aligned}
\frac{\partial C_x}{\partial a_j^{(L)}} &= \big(y_j - a_j^L\big), \\
\frac{\partial a_j^{(L)}}{\partial Z_j^{(L)}} &= \sigma'(Z_x^{(y)}), \\
\frac{\partial Z_j^{(L)}}{\partial w_{jk}^{(L)}} &= a_x^{L-1}, \\
\frac{\partial Z_j^{(L)}}{\partial b_j^{(L)}} &= 1.
\end{aligned}$$

Therefore:

$$\frac{\partial C_x}{\partial w_{jk}^{(L)}} = \big(y_j - a_j^L\big) * \sigma'(Z_x^{(y)}) * a_x^{L-1}. \tag{2.13}$$

$$\frac{\partial C_x}{\partial b_j^{(L)}} = \big(y_j - a_j^L\big) * \sigma'(Z_x^{(y)}). \tag{2.14}$$

Let variable $M$ denote that the layer of the nodes. We need to calculate $\frac{\partial C_x}{\partial w_{jk}^{(M)}}$ for $M \in [0, 1, \ldots, L-1]$. Recall that:

$$\frac{\partial C_x}{\partial w_{jk}^{(M)}} = \left[\frac{\partial C_x}{\partial a_j^{(M)}}\right] * \frac{\partial a_j^{(M)}}{\partial Z_j^{(M)}} * \frac{\partial Z_j^{(M)}}{\partial w_{jk}^{(M)}}.$$

One problem arises when we calculate the first term in the product $\frac{\partial C_x}{\partial a_j^{(M)}}$. Since $a_j^{(M)}$ is *not* in the last layer $L$, we take the partial derivative the following way:

$$\frac{\partial C_x}{\partial a_j^{(M)}} = \sum_{i=1}^{n_{M+1}} \left[\frac{\partial C_x}{\partial a_i^{(M+1)}} * \frac{\partial a_i^{(M+1)}}{\partial Z_i^{(M+1)}} * \left[\frac{\partial Z_i^{(M+1)}}{\partial a_j^{(M)}}\right]\right], \tag{2.15}$$

where:

- $i$ denotes the $i^{th}$nodes in layer $M+1$.
- There is a total of $n_{M+1}$ nodes in layer $M+1$.

In particular, the third term in Equation 2.15 is:

$$\begin{aligned}\left[\frac{\partial Z_i^{(M+1)}}{\partial a_j^{(M)}}\right] &= \frac{\partial}{\partial a_j^{(M)}}\left[W_x^{(\vec{M}+1)} \cdot a^{(\vec{M})} + b_x^{(M+1)}\right] \\ &= \frac{\partial}{\partial a_j^{(M)}}\left[W_{i1}^{(M+1)} * a_1^{(M)} + \cdots + W_{ij}^{(M+1)} * a_j^{(M)} + \cdots + W_{in}^{(M+1)} * a_n^{(M)}\right] + 0 \\ &= W_{ij}^{(M+1)}.\end{aligned}$$

From the Equation 2.15, we see that we always use derivative in the $M+1$ layer to compute derivative $\frac{\partial C_x}{\partial b_j^{(M)}}$ and $\frac{\partial C_x}{\partial b_j^{(M)}}$ where $M \in [0, 1, \ldots, L-1]$. The backward propagation algorithm starts by computing derivative in the last layer $L$ and gradually works backward to layer $L-1$, layer $L-2$, ... , all the way back to layer 1. Hence, the algorithm is called: "backward" propagation.

Each time we update the NN's weights and bias using gradient descant, we say the NN have completed one **backward propagation process**. Note that we can complete a backward propagation process with any number of training data inputs.

**2.8.1. Example of Backward Propagation.** I will show how we apply the backward propagation algorithm with an example. Suppose that the we have the NN model in Figure 6.

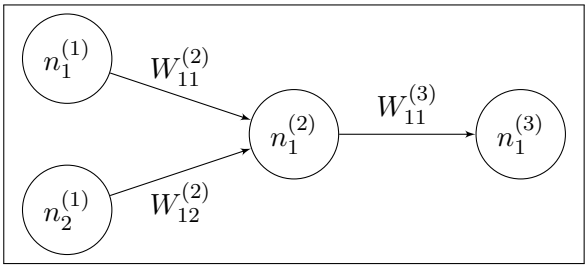


FIGURE 6. Example of Backward Propagation

For simplicity , assume that we are working with only one input vector $x$ (i.e., N=1). To compute the gradient vector $\triangledown C$, we start with the third (output) layer and compute

$\frac{\partial C_x}{\partial w_{11}^{(3)}}$:

$$\frac{\partial C_x}{\partial w_{11}^{(3)}} = \frac{\partial C_x}{\partial a_1^{(3)}} * \frac{\partial a_1^{(3)}}{\partial Z_1^{(3)}} * \frac{\partial Z_1^{(3)}}{\partial w_{11}^{(3)}}. \tag{2.16}$$

Next, we will compute $\frac{\partial C_x}{\partial w_{11}^{(2)}}$ and $\frac{\partial C_x}{\partial w_{12}^{(2)}}$:

$$\frac{\partial C_x}{\partial w_{11}^{(2)}} = \left[\frac{\partial C_x}{\partial a_1^{(2)}}\right] * \frac{\partial a_1^{(2)}}{\partial Z_1^{(2)}} * \frac{\partial Z_1^{(2)}}{\partial w_{11}^{(2)}}, \tag{2.17}$$

$$\frac{\partial C_x}{\partial w_{12}^{(2)}} = \left[\frac{\partial C_x}{\partial a_1^{(2)}}\right] * \frac{\partial a_1^{(2)}}{\partial Z_1^{(2)}} * \frac{\partial Z_1^{(2)}}{\partial w_{12}^{(2)}}, \tag{2.18}$$

where inside the brackets we have:

$$\frac{\partial C_x}{\partial a_1^{(2)}} = \sum_{j=1}^{n_3=1} \left[\frac{\partial C_x}{\partial a_1^{(3)}} * \frac{\partial a_1^{(3)}}{\partial Z_1^{(3)}} * \left[\frac{\partial Z_1^{(3)}}{\partial a_1^{(2)}}\right]\right]. \tag{2.19}$$

with

$$\left[\frac{\partial Z_1^{(3)}}{\partial a_1^{(2)}}\right] = w_{11}^{(3)}$$

By plugging Equation 2.19 into $\frac{\partial C_x}{\partial w_{11}^{(2)}}$ and $\frac{\partial C_x}{\partial w_{12}^{(2)}}$, we are done with computing the partial derivatives with respect to all the weights. From the example, we see that we must use partial derivatives from the *next* layer $L+1$ to compute partial derivative in the current layer $L$. Hence, we call this algorithm of computing gradient "Backward Propagation"; we always start from the last layer and work backwards until we have computed all partial derivatives.

Similarly, we compute all partial derivatives $\frac{\partial C_x}{\partial b_j^{(L)}}$ as follows:

$$\frac{\partial C_x}{\partial b_1^{(3)}} = \frac{\partial C_x}{\partial a_1^{(3)}} * \frac{\partial a_1^{(3)}}{\partial Z_1^{(3)}} * \frac{\partial Z_1^{(3)}}{\partial b_1^{(3)}}. \tag{2.20}$$

$$\frac{\partial C_x}{\partial b_1^{(2)}} = \left[\frac{\partial C_x}{\partial a_1^{(2)}}\right] * \frac{\partial a_1^{(2)}}{\partial Z_1^{(2)}} * \frac{\partial Z_1^{(2)}}{\partial b_1^{(2)}}. \tag{2.21}$$

$$\frac{\partial C_x}{\partial b_2^{(2)}} = \left[\frac{\partial C_x}{\partial a_1^{(2)}}\right] * \frac{\partial a_1^{(2)}}{\partial Z_1^{(2)}} * \frac{\partial Z_1^{(2)}}{\partial b_2^{(2)}}. \tag{2.22}$$

Just as before, we plug Equation 2.19 into $\frac{\partial C_x}{\partial b_1^{(2)}}$ and $\frac{\partial C_x}{\partial b_2^{(2)}}$. Therefore gradient $\nabla C$ is:

$$\nabla C = \left[\frac{\partial C}{\partial w_{11}^{(2)}}, \frac{\partial C}{\partial w_{12}^{(2)}}, \frac{\partial C}{\partial w_{11}^{(3)}}, \frac{\partial C}{\partial b_1^{(2)}}, \frac{\partial C}{\partial b_2^{(2)}}, \frac{\partial C}{\partial b_1^{(3)}}\right]^T \tag{2.23}$$

This concludes the example for $N = 1$. For any $N \geq 2$, we will modify the equations of $\frac{\partial C}{\partial w_{jk}^{(L)}}$ and $\frac{\partial C}{\partial b_j^{(L)}}$ to:

$$\frac{\partial C}{\partial w_{jk}^{(L)}} = \frac{1}{N}\sum_x \frac{\partial C_x}{\partial w_{jk}^{(L)}}, \tag{2.24}$$

$$\frac{\partial C}{\partial b_j^{(L)}} = \frac{1}{N}\sum_x \frac{\partial C_x}{\partial b_j^{(L)}}. \tag{2.25}$$

After all the computations, we obtain the gradient of the cost function $\nabla C$.

## 2.9. Summary of Neural Networks

In the previous sections, we have gone through how we train the NN with input vectors. To make things clear, here is the procedure to train a NN to make classifications based on given data:

(1) Divide the data and the data label into training set and testing set.
(2) Input the training data into the NN and use the feed-forward process to come up with predictions of the training data.
(3) Use the gradient descent algorithm to find weights and bias that decrease the loss and increase accuracy
    - Note that we calculate the gradient of the cost function $\nabla C$ by backward propagation.
(4) Use NN to make prediction on the testing data with updated weights $\vec{w}$ and biases $\vec{b}$.
(5) Repeat steps two, three, and four until we reach desired accuracy (e.g., 80%).

Below is a diagram of the procedure. In essence, we are trying to find the optimal weights and biases such that the NN can make accurate predictions on unseen data. There is no set formula on how to tune the parameters of the NN. The best approach now is to refer to other successful NN models and use the trail and error method to find good weights and biases. In the next chapter, I will talk about the experiment results I obtained with my NN models.

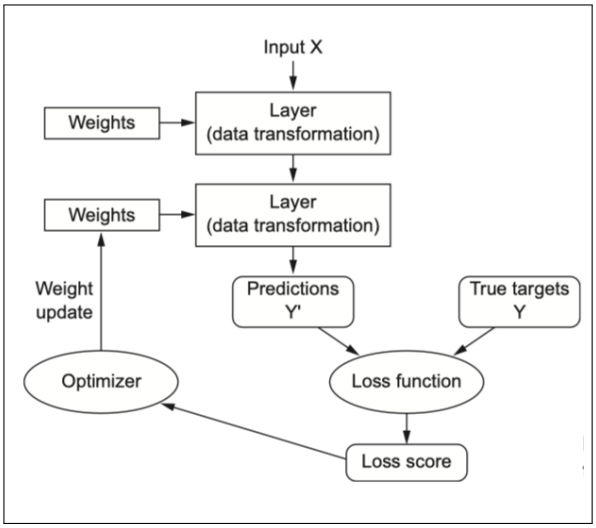


FIGURE 7. NN Procedure Diagram from [**5**]

CHAPTER 3

# Experiments

I conduct three different types of author identification experiments using our data. The three experiments are:

(1) 19-class identification
(2) Binary identification
(3) Pairwise identification

I will discuss each experiment in detail in the following sections. For my experiments, I use the Python library, TensorFlow, to construct my own NN models. Tensorflow is an open source software library that allows users to easily construct NN models [**3**]. Recall that our data consists of 180 math papers from 19 different Fields Medalists. In Table 1, I report the number of papers each author has. Also, the data matrix has size $180 \times 3000$, where the 180 rows are the papers and the 3000 columns are vocabulary from the 180 papers.

## 3.1. Activation Functions Used

In my neural network models, I used the ReLU, Sigmoid, and the Softmax activation function in different layers. I will define them formally below.

Definition 17. The **ReLU function** is defined as follows [**5**]:

$$ReLU(x) \equiv max(0, x).$$

Definition 18. The **Sigmoid function** is defined as follows [**10**]:

$$Sigmoid(z) \equiv \frac{1}{1 + exp(-z)} = \frac{1}{1 + exp(-\sum_j w_j * x_j - b)}.$$

Definition 19. The **Softmax function** is defined as follows [**10**]:

$$Softmax(z_j^{(L)}) \equiv \frac{exp(z_j^{(L)})}{\sum_k exp(z_k^{(L)})},$$

where in the denominator, the sum $\sum_k$ denotes the sum over all outputs activation in the $L^{th}$ layer.

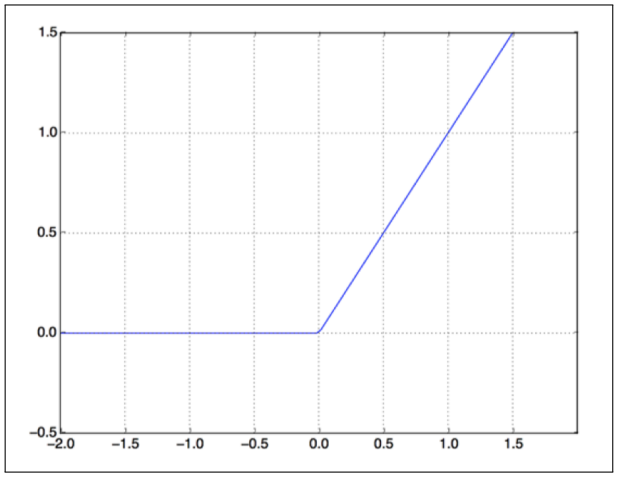


FIGURE 1. Graph of ReLU function from [**5**]

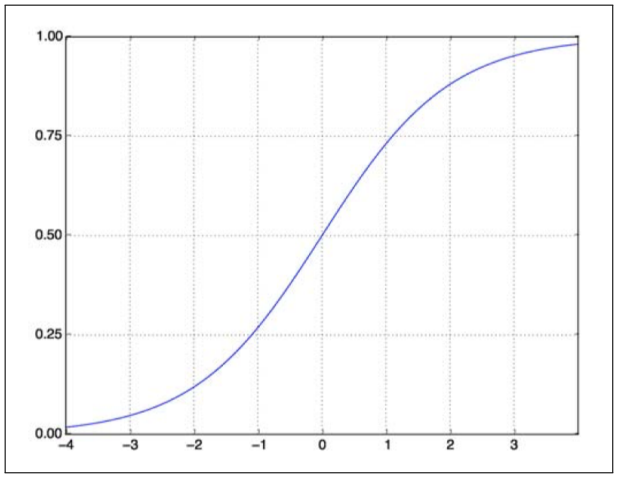


FIGURE 2. Graph of Sigmoid function from [**5**]

### 3.2. 19-class Identification

First, I conduct a 19-class identification with the NN. Given a paper from the 19 authors, I want to use the NN model to predict the author of the paper. In this experiment, I randomly choose $N$ papers from the data as testing data, and use the rest $180 - N$ papers to train the NN. For example, if I choose to use 20 papers as testing data, then I would have $180 - 20 = 160$ papers for training.

I perform the experiment using different training data sizes. In particular, I test the accuracy of the NN with training size $\in \{100, 110, 130, 150, 160\}$. To make sure my results

| Author | Num of papers |
|---|---|
| Bhargava | 10 |
| Borcherds | 10 |
| Bourgain | 12 |
| Chau | 3 |
| Gowers | 11 |
| Hairer | 10 |
| Kontsevich | 11 |
| Lindenstrauss | 11 |
| Louis | 12 |
| McMullen | 3 |
| Mirzakhani | 8 |
| Okounkov | 10 |
| Smirnov | 12 |
| Tao | 12 |
| Villani | 10 |
| Voevodsky | 11 |
| Werner | 10 |
| Yoccoz | 11 |
| Zelmanov | 3 |

TABLE 1. Table of Number of papers by each author

are accurate, I repeat the experiment 10 times for each training size, each time with random training data and testing data. The accuracy presented in the table is the average accuracy of the 10 experiments for each training size. That is:

$$\text{Average Accuracy} = \frac{1}{10}\sum_{i=1}^{10} \text{Accuracy of the } i^{th} \text{ experiment .}$$

The NN model has three layers:

(1) First layer has 100 nodes and uses ReLU as activation function.
(2) Second layer has 60 nodes and uses ReLU as activation function.
(3) Third layer has 19 nodes and uses Softmax as activation function.

The NN optimizer is RMSProp and the cost function is categorical cross-entropy, which are both available in TensorFlow.

I only include three layers in my NN model because I want to avoid over fitting. Over fitting means that the NN is trained too well for the given train data; which means that the NN model has high accuracy on the testing papers but poor accuracy for any data that deviate from the testing data. Since we only have 180 papers, I try to keep the NN model simple.

Next, I pick the input nodes size to be 100 because it yields that highest accuracy. My last layer has 19 nodes because each node corresponds to one author. The Softmax activation function spits out the predicted probability of how likely the input paper belongs to the corresponding author of the output node.

| Epoch | Batch | # input nodes | Training Size | Average Accuracy |
|---|---|---|---|---|
| 30 | 10 | 100 | 160 | 0.68 |
| 30 | 10 | 100 | 150 | 0.69 |
| 30 | 10 | 100 | 130 | 0.61 |
| 30 | 10 | 100 | 110 | 0.58 |
| 30 | 10 | 100 | 100 | 0.58 |

TABLE 2. Table of 19-author identification

From the table, we see that the larger the training set is, the higher the NN accuracy will be.

### 3.3. Binary Identification

In the second experiment, I choose one author at random and use half of his/her papers as training data and the other half as testing data. More specifically, if the author has $k$ papers and $k$ is an odd number, then I will use $\lfloor \frac{k}{2} \rfloor$ as testing data and $\lceil \frac{k}{2} \rceil$ as training data. I will refer to the papers from the 18 unchosen authors as the "leftover" data set. I will use the variable $n$ to refer to the number of leftover training papers and use $m$ to refer to the number of leftover testing papers. I randomly choose $n$ papers from the leftover data set and add them into the training set. Similarly, from the leftover data set, I exclude the $n$ papers that I have chosen and pick $m$ papers and add them to the testing set. Finally, I train the NN using the training set and use the NN to make predictions on the testing set. Note that $m$ and $n$ can be any positive natural number as long as $m + n \leq$ number of leftover papers (which depends on the chosen author).

Here is one example of the experiment: assume that my randomly chose author is Maxim Kontsevich, who has 11 papers in our data set. I will randomly pick $\lfloor \frac{11}{2} \rfloor = 5$ papers as testing data and pick $\lceil \frac{11}{2} \rceil = 6$ as training data. My leftover data set has size $180 - 11 = 169$. Next, I can randomly select $n = 30$ papers and $m = 10$ papers from the leftover data set and add them into the training set and testing set, respectively. In this case, $n$ and $m$ can be any natural number as long as $n + m \leq 169$.

The NN model has three layers:

(1) First layer has 100 nodes and uses ReLU as activation function.

(2) Second layer has 45 nodes and uses ReLU as activation function.
(3) Third layer has 1 nodes and uses Sigmoid as activation function.

The NN optimizer is RMSProp and the cost function is binary cross-entropy, which are both available in TensorFlow.

To see how accurate the NN is, I have done experiments with $n \in \{10, 20, ..., 160\}$ and $m \in \{10, 30, 50, 70, 90, 110, 130, 150\}$. I report my results in Table 4 and in Table 5 below. Similar to the Table 2, the accuracy reported is the average accuracy of the five experiments for each training size. That is:

$$\text{Average Accuracy} = \frac{1}{5}\sum_{i=1}^{5} \text{Accuracy of the } i^{th} \text{ experiment .}$$

Note that we are including vast amounts of leftover testing data relative to the number of testing papers from the chosen author. I suspect that the NN will blindly make predictions and guess that all papers are from the leftover data set. To make sure that the NN model did learn from our data, I included a column that reports the difference between actual accuracy of the NN and the expected accuracy of the NN. I define expected accuracy of the NN as the accuracy that the NN would have if it predicts that all testing samples are from the leftover data set. In the previous example, the expected accuracy would be $\frac{10}{10+5} = \frac{2}{3} \approx 66.66\%$. Fortunately, this column is positive most of the time, which means that the NN is able to distinguish between papers that belong to the chosen author and papers that do not.

### 3.4. Pairwise Identification

In the third experiment, I select two authors at random to perform classification. From each of the two chosen authors, I randomly choose half of his/her papers as training data and the other half as testing data. Just like the second experiment, if the author has $k$ papers and $k$ is an odd number, then I will use $\lfloor \frac{k}{2} \rfloor$ as testing data and $\lceil \frac{k}{2} \rceil$ as training data. Finally, I train the NN with the testing set and use the NN to distinguish testing papers from the two authors.

The NN model has three layers:

(1) First layer has 100 nodes and uses ReLU as activation function.
(2) Second layer has 45 nodes and uses ReLU as activation function.
(3) Third layer has two nodes and uses Softmax as activation function.

The NN optimizer is RMSProp and the cost function is categorical cross-entropy, which are both available in TensorFlow.

The results are reported in Table 3. The accuracy reported is the average accuracy of the 20 experiments for each training size. That is:

$$\text{Average Accuracy} = \frac{1}{20}\sum_{i=1}^{20} \text{Accuracy of the } i^{th} \text{ experiment .}$$

In conclusion, the three NN models are quite accurate at predicting authors of unknown papers. The first NN model reach 68% accuracy in the first experiment. The second reach

| Epoch | Batch | Accuracy |
|---|---|---|
| 10 | 10 | 0.83 |
| 10 | 20 | 0.92 |
| 20 | 10 | 0.90 |
| 20 | 20 | 0.87 |
| 30 | 10 | 0.91 |
| 30 | 20 | 0.88 |

TABLE 3. Table of Pairwise author identification

98% accuracy in the second experiment. The third NN model reach 92% accuracy in the third experiment. Here is a list of interesting experiments that one can work on:

- Run the three experiments using Convolutional neural networks (CCN).
- Tune weights and biases of the three NN models above and try reach a higher accuracy.
- Gather more input data for the three NN models and test how they preform on unseen mathematical papers.

| leftover test size | leftover train size | Accuracy | Percent > Expected Acc |
|---|---|---|---|
| 10 | 10 | 0.69 | 2.67 |
| 10 | 20 | 0.77 | 1.25 |
| 10 | 30 | 0.70 | 0 |
| 10 | 40 | 0.73 | 2.67 |
| 10 | 50 | 0.72 | 2.58 |
| 10 | 60 | 0.73 | 5.17 |
| 10 | 70 | 0.79 | 2.67 |
| 10 | 80 | 0.79 | 8 |
| 10 | 90 | 0.77 | 1.33 |
| 10 | 100 | 0.70 | 5.08 |
| 10 | 110 | 0.68 | 1.33 |
| 10 | 120 | 0.77 | 1.25 |
| 10 | 130 | 0.72 | 1.33 |
| 10 | 140 | 0.73 | 6.67 |
| 10 | 150 | 0.76 | 10.42 |
| 10 | 160 | 0.91 | 0 |
| 30 | 10 | 0.85 | -0 |
| 30 | 20 | 0.88 | 1.13 |
| 30 | 30 | 0.86 | 0.56 |
| 30 | 40 | 0.91 | 2.30 |
| 30 | 50 | 0.87 | 1.14 |
| 30 | 60 | 0.89 | 1.14 |
| 30 | 70 | 0.89 | 1.71 |
| 30 | 80 | 0.87 | 1.71 |
| 30 | 90 | 0.87 | 1.70 |
| 30 | 100 | 0.92 | 4 |
| 30 | 110 | 0.86 | 1.11 |
| 30 | 120 | 0.91 | 2.86 |
| 30 | 130 | 0.89 | 1.13 |
| 30 | 140 | nan | nan |
| 50 | 10 | 0.94 | 1.82 |
| 50 | 20 | 0.91 | 0.36 |
| 50 | 30 | 0.92 | 0 |
| 50 | 40 | 0.92 | 0.36 |
| 50 | 50 | 0.92 | 1.09 |
| 50 | 60 | 0.95 | 2.53 |
| 50 | 70 | 0.92 | 1.08 |
| 50 | 80 | 0.91 | 0.71 |
| 50 | 90 | 0.92 | 0.72 |
| 50 | 100 | 0.94 | 0.36 |
| 50 | 110 | 0.94 | 0.36 |
| 50 | 120 | nan | nan |

TABLE 4. Table of Binary Author Identification Part 1

| leftover test size | leftover train size | Accuracy | Percent > Expected Acc |
|---|---|---|---|
| 70 | 10 | 0.95 | 0.79 |
| 70 | 20 | 0.95 | 0 |
| 70 | 30 | 0.94 | 0.80 |
| 70 | 40 | 0.95 | 0 |
| 70 | 50 | 0.95 | 0.53 |
| 70 | 60 | 0.94 | -0.53 |
| 70 | 70 | 0.94 | 0.80 |
| 70 | 80 | 0.95 | 0.53 |
| 70 | 90 | 0.94 | -0.27 |
| 70 | 100 | 0.99 | 0 |
| 90 | 10 | 0.96 | -0 |
| 90 | 20 | 0.96 | 1.26 |
| 90 | 30 | 0.95 | 0.42 |
| 90 | 40 | 0.95 | 0.21 |
| 90 | 50 | 0.95 | 0.21 |
| 90 | 60 | 0.96 | 0.42 |
| 90 | 70 | 0.95 | 0.21 |
| 90 | 80 | nan | nan |
| 110 | 10 | 0.96 | 0.17 |
| 110 | 20 | 0.95 | 0.17 |
| 110 | 30 | 0.96 | 0.17 |
| 110 | 40 | 0.96 | 0.52 |
| 110 | 50 | 0.97 | 1.39 |
| 110 | 60 | nan | nan |
| 130 | 10 | 0.98 | 0.44 |
| 130 | 20 | 0.99 | 0 |
| 130 | 30 | 0.98 | 0.15 |
| 130 | 40 | nan | nan |
| 150 | 10 | 0.97 | 0 |
| 150 | 20 | nan | nan |

TABLE 5. Table of Binary Author Identification Part 2

# Bibliography


1. *Artificial intelligence (ai) vs. machine learning vs. deep learning*, https://skymind.ai/wiki/ai-vs-machine-learning-vs-deep-learning. 5
2. *Tf-idf :: A single-page tutorial*, http://www.tfidf.com. 4
3. Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, 2015, Software available from tensorflow.org. 21
4. Michael W. Berry and Murray Browne, *Understanding search engines: Mathematical modeling and text retrieval (software, environments, tools), second edition*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2005. 2
5. Francois Chollet, *Deep learning with python*, 1st ed., Manning Publications Co., Greenwich, CT, USA, 2017. 5, 6, 19, 21, 22
6. Eric Eaton, *Neural networks*, University Lecture, 2015. 6
7. Brett Grossfeld, *A simple way to understand machine learning vs deep learning*, 2017, https://www.zendesk.com/blog/machine-learning-and-deep-learning/. 5
8. Bartosz Gralewicz, *The tf*idf algorithm explained*, 2018, https://www.elephate.com/blog/what-is-tf-idf/. 3, 4
9. Anneke Jacobs, *Statistical analysis of newspaper headlines with optimization*, (2011). 1
10. Michael A. Nielsen, *Neural networks and deep learning*, 2018. 6, 9, 10, 11, 12, 13, 21
11. Stephen Robertson, *Understanding inverse document frequency: On theoretical arguments for idf*, Journal of Documentation **60** (2004). 3