Touch Screen Gesture Recognition

Elijah Allmon, Jack Erwin, and Matthew Madden

University of North Carolina at Wilmington

Abstract

In the modern world, touch screen devices are a massive part of everyday life. For ease of use and accessibility purposes gesture recognition features are implemented into many touch screen devices. In this paper we will explore three classification algorithms that recognize and classify gestures resembling the Arabic numerals: Chain Code, Hidden Markov Model, and Neural Networks. In the interest of emulating the input to a touch screen device all gestures are inputted into the algorithms as sets of Cartesian coordinates.

Among the algorithms explored are a Hidden Markov Model implementation using the Forward-Backward algorithm, an Artificial Neural Network, and a Chain Code algorithm. Algorithms will be evaluated by comparing the misclassification rate, time complexity, and the amount of space required to store an algorithm.

1. Introduction

Beginning with the introduction of the iPhone in mid-2007, the implementation of touch screen interfaces has become the standard for portable technology like cell phones, tablets, and laptops. This streamlined user interface allows for an unprecedented ease of use. The primary form of data input on these devices is a virtual keyboard with each virtual key corresponding to a letter. What if there were more ways to input data simply by writing them on the touch screen? A user could simply use writing gestures to write out a text, search the internet, make a post, etc. without the use of any other input.

The goal of our project is to find a way to efficiently and accurately evaluate touch screen gestures and translate those to strings and other numeric data types for that device to use. To do this we will use three different algorithms: Hidden Markov Model, Chain Code, and Neural Network. Each algorithm will train and test using the MNIST database of handwritten digits. The database contains 60,000 training character entries and 10,000 testing character entries for the Arabic numerals 0 through 9. Each character entry is a 28x28 pixel grayscale image.[1] Joseph Redmon has created a CSV format of this dataset that will use for the implementation of all three algorithms[2].

2. Context

In order to institute a consistent basis of comparison for the classifier algorithms confusion matrices were used. A confusion matrix is used to store the frequencies of what a classifier algorithm identifies test data which has an expected outcome as. Figure 1 pictured below is an example of a confusion matrix.
The frequencies of correct classifications are stored in the cells where the index of the row and column is the same. All other cells contain the frequencies of misclassification. In order to calculate the successful classification rate the frequencies of the cells where the index of the row and column are the same are summed and divided by the cumulative frequency. In order to find the misclassification rate we can simply subtract the successful classification rate from 1.

3. Informal Statement of Problem

Recognize a character inputted as a touch screen gesture as one of Arabic numerals (0-9) with maximum accuracy. The misclassification and classification of each character will be analyzed using a confusion matrix.

4. Formal Statement of Problem

The formal statement of our problem is as follows:

Let $C$ represent a set of $n$ inputted characters.

Let $T$ represent a set of $a$ pre-defined characters.

Identify an algorithm which classifies each element of $C$ as an element of $T$ which minimizes the misclassification rate given by:

$$misclassification\ rate = 1 - \frac{1}{n} \sum_{i=1}^{a} m_{i, i}$$

Where $m_{i, i}$ is the element in the $i$th column of the $i$th row of a confusion matrix, $m$, of size $a \times a$.

5. Hidden Markov Model

In his paper serving as an introduction into Hidden Markov Models, Ghahramani describes a Hidden Markov Model as “a tool for representing probability distributions over sequences of observations” [3]. We implemented our Hidden Markov Model using methodology proposed in an article by Fatema Mahmoud published in Medium [4]. This article proposes the idea of using
data normalization in the training phase in place of an Expectation Maximization algorithm in order to optimize the initialization of the model. In the interest of implementing the most efficient model possible we used this method.

In order to initialize the classifier we initialized a Hidden Markov Model for each character that exists in our data set. We used the following process to initialize the Hidden Markov Model for each character:

1. Normalize the training data representing the desired character.
2. Divide the normalized training data into $x$ parts (where $x$ is the given number of states), calculate the mean and covariance for each part. Use the mean and covariance to initialize each corresponding state as a Gaussian Distribution.
3. Initialize a transition matrix with $x$ rows and $x$ columns (where $x$ is the given number of states). For each row assign a high probability of staying in the same state (in our implementation the probability initialized is .75) and a lower probability of transitioning to other reachable states.

The number of states is subjective to each character and varies based on the number of states that the points in a given character are normally distributed in.

A character from the test dataset is then classified by iterating though the Hidden Markov Model for each character, then classifying the test character as the character whose model returns the highest probability of representing the test character. Evaluating the probability of a given model representing a given character is done using the Forward-Backward algorithm. The process for evaluating the probability of a character belonging to a given model is as follows:

1. Normalize the inputted character.
2. Create two matrixes each representing the emission probabilities for each point in each state. One transition matrix will contain the emission probabilities for the forward pass of the algorithm, the other will contain the emission probabilities for the backward pass of the algorithm.
3. Calculate the probability of each point fitting the model in the forward direction by:
   i. For each state, take the sum of multiplying the observation value by the probability of transitioning for every possible state into the current state.
   ii. Multiply the sum calculated by the probability of the next point belonging in the current state (calculated using the Gaussian Probability Distribution Function).
   iii. Store the product in the matrix responsible for the forward emission probabilities.
4. Calculate the probability of each point fitting the model in the backward direction using a similar methodology, however, we traverse the points in the reverse order this time:
   i. For each state, take the sum of multiplying the observation value by the probability of transitioning for every possible state into the current state.
   ii. Multiply the sum calculated by the probability of the previous point belonging in the current state (calculated using the Gaussian Probability Distribution Function).
   iii. Store the product in the matrix responsible for the backward emission probabilities.
5. Find the probability of the inputted character fitting the model by summing the product of the probability of each point fitting the model forward and backward for each state.

The probability from running the Forward-Backward algorithm is then returned to be used for classification of the given character.

6. Chain Code

The concept of the chain code was first defined by Herbert Freeman in 1961. Freeman detailed a method that, “permits the encoding of arbitrary geometric configurations so as to facilitate their analysis and manipulation by means of a digital computer” [5] The chain code uses an eight-directional compass to identify paths along a two-dimensional shape. For this implementation, we will be using a compass labeled with numbers 0-7 starting at the middle-right direction and moving counter-clockwise. A unique histogram with the frequencies of each direction is recorded for a two-dimensional character. [6] The histogram will be represented by a list that is of a size 8 with each index (starting from 0) representing each direction.

Each entry in the MNIST database is represented by a set of cartesian coordinates. Each training and testing entry will have a histogram created for it and added to a list of histograms for that character. To create a histogram for each entry:

1. Normalize the data.
2. Create a base histogram: [0,0,0,0,0,0,0,0].
3. For each point in the entry that has a succeeding point:
   a. Identify the x and y values of both points and calculate the slope using 
      \[
      \frac{(y_2 - y_1)}{(x_2 - x_1)} \]
      where \(x_1\) and \(y_1\) represent the x and y values of the current point and \(x_2\) and \(y_2\) represent the x and y values of the succeeding point.
   b. The slope will evaluate to one of eight statements:
      i. If the slope is equal to 0 and \(x_1\) is less than \(x_2\), the frequency of the 0 direction will increment by 1.
      ii. If the slope is equal to 0 and \(x_1\) is greater than \(x_2\), the frequency of the 4 direction will increment by 1.
      iii. If the slope is greater than 0 and \(y_1\) is less than \(y_2\), the frequency of the 1 direction will increment by 1.
      iv. If the slope is greater than 0 and \(y_1\) is greater than \(y_2\), the frequency of the 5 direction will increment by 1.
      v. If \(x_2-x_1\) evaluates to zero (indicating the slope does not exist) and \(y_1\) is less than \(y_2\), the frequency of the 2 direction will increment by 1.
      vi. If \(x_2-x_1\) evaluates to zero (indicating the slope does not exist) and \(y_1\) is greater than \(y_2\), the frequency of the 6 direction will increment by 1.
      vii. If slope is less than 0 and \(y_1\) is less than \(y_2\), the frequency of the 3 direction will increment by 1.
      viii. If slope is less than 0 and \(y_1\) is greater than \(y_2\), the frequency of the 7 direction will increment by 1.
4. Add the histogram to the histogram list.
In order to train the chain code for each specific character we create a list of average histograms that contains the average of all training data for each character.

1. For each histogram list created for each training character
   a. Find the average between the first two histograms in the list to create the base average histogram.
   b. For each other histogram in the list, find the average of it and the base average histogram.
   c. Add the final average to the list of average histograms.

To evaluate a character’s testing data set:

1. Create a list of histograms for each entry in the testing dataset.
2. For each histogram in the histogram list for each testing character
   a. Find the variance between the histogram and every histogram in the training character’s average histogram list. Insert those variance values into a separate list with each index (starting from 0) representing each different character.
   b. Find the smallest variance in the list and keep the index.

The index is then used to identify which column in the row of the current character of the confusion matrix to increment by 1.

7. ANN

Artificial neural networks are designed and behave much in the same way as neurons in our own brains. In the conventional approach to programming, we tell the computer what to do and how to solve the problem. A neural network, however, learns from data it receives and creates its own solution to the problem[7]. The first artificial neurons were developed in the 1950s and 1960s and were named perceptrons. They receive signals (inputs) from other neurons and when a threshold is met the neuron fires a signal of its own. More advanced artificial neurons have been developed since the 60s.

Our neural network will use a sigmoid neuron. Rather than using a step function like previous perceptrons where the output can be only a zero or one, we will use the sigmoid function. This allows our output to be anything between zero and one and will enable us to better monitor the changes in the output of our neurons. Our network will contain three layers. An input layer made up of 784 neurons, one for each of the pixels in the image data. A hidden layer with a varying number of neurons and an output layer with ten neurons for each of the ten digits. An image is classified as a digit if that digits output neuron “fired” the most.

For our network to learn we must assign weights and biases to our neurons. Weights are applied to the outputs of neurons and biases are applied to the neurons themselves in order to change their “importance.”

1. Our network will determine the correct digit by assigning lighter or heavier weights/biases to neurons depending on its relationship with the desired output. The
challenge is determining how we change the weights and biases so that our network
becomes more accurate and learns from the data.
2. In order to do this our network uses an algorithm called back-propagation.
3. The back-propagation algorithm computes a gradient that shows how our classification
error will change with respect to changes in weights and biases.
4. Next, we used a gradient descent optimization algorithm to minimize our error function
iteratively along the gradient that the back-propagation algorithm created.
   i. Our network will take a sample of the test data, calculate a gradient of its error
      and how it will change with respect to the weights and biases of our neurons, and
      change those weights and biases according that gradient.
   ii. Using a sample, or batch, of the training data allows us to update the weights and
       biases more accurately and more often during for iteration of the gradient descent.
   iii. The gradient descent updates our network after every batch until it has iterated
       through every image in the training data one time.
5. Once our network has been trained, and our weights and biases have been tuned, we run
our test data through the network and compare the output of our network to the label of
the test image in order to receive our results.

8. Results
   Our three algorithms will be compared based on misclassification rate, time complexity, lines
of code required to implement, and storage space required to store on a device.

As seen in the chart above complaining the misclassification rates and successful
classification rates between algorithms, the Neural Network Algorithm produced the smallest
misclassification rate, followed by the Hidden Markov Model, and then the Chain Code.

The Hidden Markov Model produced a misclassification rate of 45% (thereby meaning the
successful classification rate was 55%). As seen in the below confusion matrix resulting from
running the Hidden Markov Model on the testing data set (Figure 2), when a given row has a
high misclassification rate, it is misclassified as a number of different characters rather than
being misclassified heavily towards one character. The characters with the highest misclassification rates (the characters three and four) tended to have features similar to the characters zero, eight, and nine, causing confusion between the characters.

The Chain Code Algorithm produced a misclassification rate of 56% (thereby meaning the successful classification rate was 44%) The frequency of successes and failures for each character with this approach can be seen in the confusion matrix below (Figure 3). The characters with the lowest misclassification rate at 0% and 5% were the numerals one and three. These results are logical considering the numeral one is universally written in a straight vertical line and no other numerals were misclassified as this. The numeral three has a horizontally symmetrical design and is rarely written in different ways. The characters with the highest misclassification rate were the numerals four, five, six, and eight with rates of 92%, 78%, 82%, and 83% respectively. The numerals five, six, eight are all most commonly misclassified as the numeral three. The similarity amongst these numerals’ shape and therefore the frequency of directions that each character’s chain code would travel along would inform these misclassifications.
The Neural Network produced a misclassification rate of 13% (with a success rate of 87%). The successes and failures are represented in the confusion matrix below. The most misclassified characters were predictable in that they have similar curves. Fives were often mistaken for threes, and fours were often mistaken for nines. Eight was also often misclassified presumably because it has similar curves to multiple other digits. The highest classified digit was one.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>929</td>
<td>0</td>
<td>9</td>
<td>3</td>
<td>0</td>
<td>6</td>
<td>25</td>
<td>2</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1109</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>5</td>
<td>876</td>
<td>19</td>
<td>16</td>
<td>9</td>
<td>15</td>
<td>20</td>
<td>52</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>25</td>
<td>867</td>
<td>2</td>
<td>57</td>
<td>7</td>
<td>18</td>
<td>26</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>1</td>
<td>882</td>
<td>3</td>
<td>10</td>
<td>4</td>
<td>14</td>
<td>52</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>4</td>
<td>15</td>
<td>71</td>
<td>23</td>
<td>666</td>
<td>25</td>
<td>9</td>
<td>41</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>28</td>
<td>5</td>
<td>14</td>
<td>0</td>
<td>27</td>
<td>19</td>
<td>860</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>10</td>
<td>28</td>
<td>1</td>
<td>13</td>
<td>4</td>
<td>0</td>
<td>905</td>
<td>18</td>
<td>47</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>4</td>
<td>18</td>
<td>31</td>
<td>19</td>
<td>41</td>
<td>7</td>
<td>8</td>
<td>820</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>5</td>
<td>6</td>
<td>13</td>
<td>60</td>
<td>22</td>
<td>1</td>
<td>34</td>
<td>19</td>
<td>838</td>
</tr>
</tbody>
</table>

(Figure 4)

The time complexity for the training process of the Hidden Markov Model is $O(c^5p^3s^3)$ where $c$ represents the number of characters in the training data set, $p$ represents the number of points in each character, and $s$ represents the number of states in a given model. The time complexity for evaluation of the probability of a given character belonging to each model is $O(p^7s^8)$ where $p$ represents the number of points in the given character, and $s$ represents the number of states in the model the character is being evaluated by.

The time complexity for finding the average histogram for each character in the Chain Code algorithm is $O(tw^2u)$ where $t$ represents the number of characters in the training data set, $w$ represents the number of histograms in the histogram list for each character and $u$ represents the number of points in each character. The time complexity for the evaluation of an entry in the training data set is $O(av^2u)$ where $a$ represents the length of the average list, $v$ represents the length of the variance list between each average histogram and the current character’s histogram and $u$ represents the number of points.

The time complexity for training the neural network is $O(nt(ij+jk))$ where $n$ represents the number of epochs, $t$ represents the number of training images, $i$ represents the number of neurons in the input layer, $j$ represents the number of neurons in the hidden layer, and $k$ represents the number of neurons in the output layer. The time complexity for the evaluation of the test data is $O(tk)$ where $t$ represents the number of test images and $k$ represents the number of possible outcomes.

The time complexities for the Chain Code algorithm are by far the most simple, which was expected as the Chain Code implements the most simple approach. All of the time complexities for our algorithms were polynomial in nature, while constant or logarithmic would have been
preferable, for a problem as complicated as gesture recognition polynomial time complexities is reasonable.

The Hidden Markov Model and the Neural Network approach have around the same amount of code overall with Hidden Markov Model containing 151 lines of code and Neural Network containing 140 lines. However, the amount of code required to train and evaluate is different between the two approaches because of their respective focus towards one aspect. The Neural Network approach is involved with training a system of neurons to identify characters, making the evaluation much less involved. The Hidden Markov Model approach trains each character in the set by creating a transition matrix and then uses that matrix to evaluate the testing data which takes substantial effort. The Chain Code Algorithm approach has the smallest amount of code (98 lines of code) due to its simplicity in relation to the other two approaches.

The Hidden Markov Model is concise and clean when written in code. The Neural Network Approach had the marginally largest file size requirement with 20 kilobytes, more than the other two approaches’
requirements combined. This is most likely to due to the complexity and the amount of code required to successfully create a neural network.

8. Conclusion

Overall, the factor for comparison that yielded the largest difference between the algorithms was misclassification rate. The Artificial Neural Network had the lowest misclassification rate at 13% followed by the Hidden Markov Model with 45% then followed by the Chain Code algorithm with 56%. The Artificial Neural Network had by far the largest storage space (over twice as much as the next largest algorithm), however, since the storage for all of the algorithms is in relatively small units (KB) the difference in storage space is not significant enough to suggest that the Artificial Neural Network would be a poor choice for this situation. Since the time complexities of the other algorithms are relatively similar (all polynomial in nature) and the algorithm with the smallest amount of code (Chain Code) having the highest misclassification rate it would be fair to say that the Artificial Neural Network is the best choice out of our algorithms for situations where we have very point-dense data with a large training dataset.

9. Future Works

Potential future work would include implementing our algorithms in an application and collecting data on functionality by performing some sort of survey sample. Implementing the Chain Code and Hidden Markov Model applications using a dataset with less point-dense characters could also produce interesting, potentially higher accuracy results, as those models are better suited to such a dataset. Implementing the Hidden Markov Model in other related fields (such as hand gesture recognition or part of speech tagging) could also provide for very interesting future work.
Works Cited:


