Title
Classification with Hash Collision Networks

Permalink
https://escholarship.org/uc/item/7421j1pj

Author
Gao, Christian Siyao

Publication Date
2018

Peer reviewed|Thesis/dissertation
Classification with Hash Collision Networks

A thesis submitted in partial satisfaction
of the requirements for the degree
Master of Applied Statistics

by

Christian Siyao Gao

2018
ABSTRACT OF THE THESIS

Classification with Hash Collision Networks

by

Christian Siyao Gao

Master of Applied Statistics

University of California, Los Angeles, 2018

Professor Ying Nian Wu, Chair

Today most of our machine learning methods and classification methods are gradient based methods. Especially in the area of image classification, methods such as deep learning try to tune large amounts of weights associated with training data over many iterations to be able to make predictions. These models, while highly accurate in prediction require many hours to train and are not very easily changed once trained. For instance, it is difficult to combine pre-trained neural networks into one model, difficult to add new observations to a trained model, and difficult for a neural network to forget specific observations it has trained on. Our goal is to explore non-gradient classification methods based on MinHash and locality-sensitive hashing. Instead of having a fixed network structure and fine tuning weights to respond to certain features, we create hash neurons that are only able to hash data and create new links with other neurons. By using this network of hash maps we create an online model that can learn, combine, forget, and change its own structure. We will also show that this technique, while memory intensive, is fast, scalable, performs well, and will work on any classification problem as long as the training data can be hashed into smaller unique parts.
The thesis of Christian Siyao Gao is approved.

Qing Zhou

Mark Stephen Handcock

Ying Nian Wu, Committee Chair

University of California, Los Angeles

2018
# TABLE OF CONTENTS

1 **Introduction** .................................................. 1  
   1.1 Supervised Classification ...................................... 1  

2 **Related Work** .................................................. 3  
   2.1 Hashmap development and large scale data retrieval. ............ 4  

3 **Method Overview** .............................................. 5  
   3.1 Hash Nodes and Hash Functions ............................... 5  
   3.2 Overview of Process ......................................... 7  

4 **Training Hash Collision Network With Numerical Predictors** .... 8  
   4.1 Dataset Description - Iris .................................. 8  
   4.2 Hashing Approaches .......................................... 9  
   4.3 Adding Variation with Bootstrap .............................. 10  
   4.4 Predicting Neurons ........................................... 12  
   4.5 Connecting Neurons ........................................... 13  

5 **Hash Collision Nodes for Image Classification** .................. 14  
   5.1 Data Description ........................................... 14  
   5.2 Image Hashing Process ...................................... 15  
   5.3 Predictions ................................................ 18  

6 **Optimization and Performance Evaluation of Hash Collision Nodes** 20  
   6.1 Boosting hash node Learners ................................. 20  
   6.2 Jackknife Cross Validation and Performance Tracking .......... 20
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Hash Network Framework Architecture and Scalability</td>
<td>23</td>
</tr>
<tr>
<td>7.1</td>
<td>Architecture Overview</td>
<td>23</td>
</tr>
<tr>
<td>8</td>
<td>Experimental Results</td>
<td>26</td>
</tr>
<tr>
<td>8.1</td>
<td>Classification Success Rate</td>
<td>26</td>
</tr>
<tr>
<td>8.2</td>
<td>System Performance</td>
<td>29</td>
</tr>
<tr>
<td>9</td>
<td>Conclusion</td>
<td>30</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

3.1 Hash Collision Definition ........................................... 5
3.2 Hash Node Definition ................................................ 6
4.1 Hash Training- Numeric Predictors ..................................... 9
4.2 Counting Hashes ...................................................... 10
4.3 Filtered Hashes ....................................................... 11
4.4 Prediction by neurons voting. ........................................ 12
4.5 Joint Nodes Hashes .................................................... 13
5.1 Process for Hash Node Training- Image Classification ............... 14
5.2 Overview of Hash Node Training ..................................... 16
5.3 Convolution Methods for Hash Generation ............................. 17
5.4 Categorization Success Rate vs Down Scale Ratio ..................... 17
5.5 Hashing Process ...................................................... 18
6.1 Forgetting Trained Observation for Jackknife Evaluation ............. 22
7.1 Hash Network System Architecture .................................... 23
7.2 Adding New Classes .................................................. 24
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Iris Dataset Features Description</td>
<td>8</td>
</tr>
<tr>
<td>5.1</td>
<td>One Hot Conversion</td>
<td>19</td>
</tr>
<tr>
<td>8.1</td>
<td>Classification Results for Hash-Nodes</td>
<td>26</td>
</tr>
<tr>
<td>8.2</td>
<td>Classification Results for Previous Studies</td>
<td>27</td>
</tr>
<tr>
<td>8.3</td>
<td>Confusion Matrix For Iris Predicted vs. Expected</td>
<td>27</td>
</tr>
<tr>
<td>8.4</td>
<td>Confusion Matrix For MNIST Predicted vs. Expected</td>
<td>28</td>
</tr>
<tr>
<td>8.5</td>
<td>Performance Metrics for Hash Networks</td>
<td>29</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

To my friends and family who gave me this chance;

   to my teachers who gave me an education;

   and to Glenda who shows us the way
CHAPTER 1

Introduction

1.1 Supervised Classification

Recent developments in deep learning has shown that neural networks are well suited for classification tasks with large datasets. However, these models generally take a long time to train and are not very flexible; once neural networks are trained, it is not easy to add new classes to the model’s predictive set. My goal is to create a learning method that utilizes hash maps instead of neurons to replicate the same results of a neural network while maintaining a flexible, combinable model that can be easily trained in a distributed fashion. Our method is based on locality sensitive hashing principles presented in [6, 10, 11], using hashing functions to store data in an easily retrievable manner. We first use randomly initiated filters to hash our data, compressing our features into a binary string, then we store the outputs of these filters into a corresponding hash maps. Finally upon prediction we extract trained values that have hashed similarly and aggregate the votes from each of hash maps.

My main contributions:

- Presenting a framework that utilizes HBase and fast lookup technology to speed up training process.
- Creating a cloud based neural-like model that scales to accept unlimited quantities of classes and features.
- Adding the ability for a hash-network to quickly train and accept new classes as well as “forget” previously trained observations for validation.
Question To Be Answered. Many top image classification algorithms today use neural networks for training filters and use hashing tricks in order to estimate hamming distances when the dataset has a lot of samples and classes. Can we skip neural networks entirely and rather use a network of hash maps to create the same filters and generate the same results?
CHAPTER 2

Related Work

This section will review work that includes advances in image recognition, applications of hash functions, and fast retrieval systems. We will also discuss topics and technologies that allow hash based learning systems to scale beyond single node calculations.

Image Classification and Mnist. In recent years almost all image classification research employs a neural network approach. Explorations in studies such as [3, 5, 17, 20] have pushed the boundaries of digit classification to about .21% error rate. Upon closer examination, within images that these models misclassified, more than 50% of the errors are irreducible and come from erroneous labels in the dataset, where an image is clearly mislabeled. The rest of the misclassifications occur on images that are badly drawn and do not resemble any images. One of our goals is to provide a way to identify badly drawn images in addition to generating the most probable class.

Hashing in Classification. Previous studies have used hashing tricks in conjunction with deep learning not only to help performance but also to augment the number of classes that a model can recognize. Studies such as [21] use hashing tricks in text based problems to extract and compress features from email text in order to make a low dimension feature set for spam detection. Other studies such as [7] developed methods to optimize the performance of generating integer hash keys from raw text while limiting erroneous hash collisions between sentences that are not similar.
Hashing Images and Neural Networks. Hashing techniques are also used in image recognition primarily to expand the number of objects a neural network can identify. Studies such as [11] trained a typical CNN filters on the MNIST dataset and then used filters to create binary hash codes that would be mapped to the original image. Upon prediction, the same filters are applied to the new image and training images with the closest approximate hamming distances are retrieved. A final prediction is then made by comparing new images to previously seen images. The use of hashing techniques to store large image datasets were also explored in [6] in which the authors were able to index large datasets and create an approximate nearest neighbor (ANN) mapping for the images using very little memory.

2.1 Hashmap development and large scale data retrieval.

While recent development in GPUs have accelerated computational power in deep learning, fast and scalable data retrieval have also made great advances. Developments in NoSQL databases such as google BigTable [2] and HBase [18] have allow databases to have near infinite scalability at very little retrieval performance loss. In our experiments we utilize these tools to create a method to map our training procedure to a batch of row inserts and our predictions to a batch of row lookups in HBase. Any additions and subtractions to the model can be done by importing new tables or removing rows in Hbase. This creates a procedure that is just as fast, and scalable as the framework that it is built on.
CHAPTER 3

Method Overview

3.1 Hash Nodes and Hash Functions

Our approach classification without using gradient based methods is to create multiple hash representations of the training features and then store these features mapped to its original class in a hash node. When predicting an object’s class we create the same hash representations of the objects features and count the number of hash collisions from previous examples that these hashes generate, effectively looking up the classes associated with these hashes. A hash collision [Figure 3.1] is defined as an instance in which two or more observations are hashed to the same value. When retrieving the outputs of each hash function based on new observations, we can observe the performance of each individual hash function. The success of these hash functions is then used to train better hash functions. Thus, a hash node has 3 parts: one hash function, one hash map, and one prediction function.

Figure 3.1: Hash Collision Definition

\[
\text{Hash Collision}
\]

\[
\begin{array}{c}
\text{Input 1} \\
\text{Input 2}
\end{array} \quad \overset{\text{Hashes To}}{\rightarrow} \quad 101101
\]
Hashing functions. A typical hash function $h(x)$ or $h(x; S_n, F_n, B_n)$ will have 3 components: $x$ is the training observation, $S_n$ is a separation parameter describing how to break down an observation into smaller pieces, $F_n$ is a filtering parameter describing how to filter emphasize certain features of the data, and $B_n$ is a bootstrap parameter describing how to parametrically create more hash keys if the training set is limited. Given an observation $X_i$ a hash function will generate hashes: \{$h_{1:n}, h_{2:n}, ... h_{i:n}$\} where $i$ corresponds to the index of the observation and $n$ represents the corresponding hash node. In the simple case, the hashes are then stored in a hash map $H_{h_{i:n}}$ (generated from $h(x)$): $C_i$ where $h_{i:n}$ is the row key and $C_i$ is the class of the observation mapped to the hash key $h_{i:n}$. Since a hash map cannot have repeating lookup keys, when adding new data with the same key we can only either increment a counter or overwrite a row, recording the last class that generated this hash. A prediction function $pred_n(x)$ is then used to generate prediction output from a new observation $x$ from the test set. Thus, we can model the probability of each observation’s class $C$ to be of a certain hypothesis class $c$ as:

$$P(C = c|x_i) = \frac{\text{# of Collision Values} = c}{\text{Total # of Collision Values}} \quad (3.1)$$

Which is the total number of collisions mapped to class $c$ divided by the total number of collisions when querying the hash map with the observations generated hash \{$h_1, h_2, h_3...$\}. 
3.2 Overview of Process

The Overall Method is:

1. For dataset $X = \{x_1, x_2, ... x_i\}$ generate hashes $H = \{h_{1:n}, h_{2:n}, ... h_{i:n}\}$ from $h(x)$ in node $N : \{N_1, N_2, ... N_n\}$ with arbitrary or randomly generated parameters $S_n, F_n, B_n$

2. If data is small, generate $H' = \{h'_1:n, h'_2:n, h'_3:n, ...\}$ through parametric bootstrap of the original hash set $H$ for each node and store the in the corresponding node’s hash map.

3. Predict $\{c_1, c_2, ... c_i\}$ classes from each hash node.

4. Do a simple vote between each of the hash nodes for the final prediction $= C_{ipred}$ or use a boosting method to aggregate the predictions.

5. Rank each node in terms of performance and calculate the correlations between the predictION of each node to create a set of N top nodes combining or removing nodes that are too similar or perform badly.

6. Add and train new nodes with randomly set filters and parameters as necessary.
CHAPTER 4

Training Hash Collision Network With Numerical Predictors

Our first application of the hash collision network is to predict categorical values from purely numerical data. Since we are mapping continuous numerical values to a non-numerical outcome variable, a careful approach has to be taken to ensure that within our hash map the correct number of hash collisions occur. For our first iteration we use the Iris dataset [1] to show how we could make predictions by hashing numerical features to categorical values.

4.1 Dataset Description - Iris

The Iris dataset is a popular dataset that refers to three types of plants each with 50 observations with the following features:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sepal length</td>
<td>numerical (cm)</td>
</tr>
<tr>
<td>sepal width</td>
<td>numerical (cm)</td>
</tr>
<tr>
<td>petal length</td>
<td>numerical (cm)</td>
</tr>
<tr>
<td>petal width</td>
<td>numerical (cm)</td>
</tr>
<tr>
<td>class</td>
<td>categorical</td>
</tr>
</tbody>
</table>
4.2 Hashing Approaches

In order to have a good prediction rate we make sure to have a healthy number of hash collisions for any test set. In other words we must have a hashing function, \( h(x; S_n, F_n, B_n) \), that creates hashes which are good representations of the original observation. An easy way to do this is to simply sample from the given features. In [Figure 4.1] we just take “Sepal Length” as our first hash for the first node and “Sepal Height” as our second hash for the second hash nodes. For example, if we train our first node a flower with class “Iris-setosa” in our training set has sepal length of 5.1, the next time we predict an iris with the same sepal length of 5.1 this node would predict the class to be “Iris-setosa”. In this case \( S_n = 1 \) and the resulting hash policy would be to just take the first feature of every observation and to treat that as a hash.

Figure 4.1: Hash Training- Numeric Predictors
If the total number unique hashes generated is large and there are few collisions relative to the total size of all the hashes generated, as long as we randomize the order of the training data, we can generate good results by only keeping the last known class for any particular hash key. One the other hand, if we generate very few total unique hashes for many observations, the probability of collision is very high within the training set, so it is important that for each hash key we also keep a tab of not only the most recently attributed class but also a count of all the previous instances in which this generate key has mapped to that hash. This would make our model unbiased towards the order of the training set. Thus for each node, supposed we define the total number of collisions that the hashes generated by \( h(x) \) for class \( c \) as:

\[
d(x, c) = \sum_{c* \in K(x)} \begin{cases} 
1, & \text{if } c* = c \\
0, & \text{otherwise}
\end{cases}
\]

where \( K(x) \) is a list the classes for each collision. Then for each node we return the \( \max(d(x, c)) \) for all classes of \( c \). For example, if we make a prediction from a trained node with generated hash key 11, then our predicted class for that node is Class B.

Figure 4.2: Counting Hashes

Hash Map Detailed

4.3 Adding Variation with Bootstrap

Next, we realize that since the iris dataset is small, we must augment the dataset to have enough collisions for predicting. Thus we create a Bootstrapping method \( B_n \) to augment the chances of collision. In this case we assume that each flower in the training set represents
many flowers of the approximately same size and class, so we do a simple parametric bootstrap, creating new data points uniformly around our observation. This creates multiple versions of the same hash. For this study we have two bootstrap parameters $\sigma_{\text{boot}}$ which sets our bootstrap interval to $[x - \sigma_{\text{boot}}, x + \sigma_{\text{boot}}]$ and $n_{\text{boot}}$ as total number of samples we want from this interval. Each new datapoint is a small fraction of the overall standard deviation of the feature. For example, suppose we want to bootstrap a spread of .2 standard deviation on each end of our samples. If our original hash for this node is Sepal length 5, then our boot-strapped hash values will also map variations [4.834387, 4.889591, 4.944796, 5.055204, 5.110409, 5.165613] to the same class.

Similar to how a convoluted neural network filters image data in order to bring out certain features of the image, we must also filter our iris data to make sure the hash collisions occur. A simple way to do this is to round the digits of our hash to $n$ - digits so that close numbers would fall into the same bucket, creating a collision. An overall view of the hashing process for numerical features:

Figure 4.3: Filtered Hashes

Hashing Variations and Numerical Bootstrap

<table>
<thead>
<tr>
<th>Simple Hash Node</th>
<th>Aggregated Hash Node</th>
<th>Hash Node Bootstrap: Interval $[5 + 2 \times \text{sd}, 5 - 2 \times \text{sd}]$</th>
<th>Hash Node Bootstrap: Rounded</th>
</tr>
</thead>
<tbody>
<tr>
<td>PK</td>
<td>Class</td>
<td>PK</td>
<td>Class</td>
</tr>
<tr>
<td>5</td>
<td>iris-setosa</td>
<td>4.834387</td>
<td>iris-setosa Count: 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.889591</td>
<td>iris-setosa Count: 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.944796</td>
<td>iris-setosa Count: 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5.055204</td>
<td>iris-setosa Count: 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5.110409</td>
<td>iris-setosa Count: 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5.165611</td>
<td>iris-setosa Count: 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PK</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.8</td>
<td>iris-setosa Count: 1</td>
</tr>
<tr>
<td>4.9</td>
<td>iris-setosa Count: 1</td>
</tr>
<tr>
<td>5</td>
<td>iris-setosa Count: 1</td>
</tr>
<tr>
<td>5.1</td>
<td>iris-setosa Count: 1</td>
</tr>
<tr>
<td>5.2</td>
<td>iris-setosa Count: 1</td>
</tr>
</tbody>
</table>
4.4 Predicting Neurons

In order to do a simple prediction, we allow each node to vote for a class by counting the number of hash collisions occurred corresponding to each of the possible classes. For example, if our iris with petal width 3 and sepal length 4 returned 3 instances of “Iris Setosa” and only one instance of “Iris Virginica” we could classify the flower as “Iris Setosa”. With larger datasets, each node only gets one vote for a class, then the final prediction is the class that has the most votes. Illustrated in [Figure 4.4] With smaller datasets like Iris, each node is able to vote many times for each class based on the number of hash collisions.

Figure 4.4: Prediction by neurons voting.
4.5 Connecting Neurons

It has been hypothesized that part of the human brain’s ability to abstract ideas derives from how it can alter links between neurons creating new structures [16]. In order to generate new hashing functions, we also allow our hash nodes to alter their structure. In simple regression situations such as:

\[ y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 \]  \hspace{1cm} (4.2)

We often join two variables by adding an interaction term \( \beta_3 x_1 \times x_2 \) to the model to see if there is any joint effect to a pair of variables. Similarly, we also combine the hash functions of two or more neurons, creating a new node to represent the joint effect of these neurons. In [Figure 4.5], we illustrate combining two nodes that sample the first and second features respectively. The third node combines the hash values of the first two nodes creating joint distribution of the first two hashes. When we start training a new model, we start with simple hashing functions such as: \( h(x; S = 1) \), getting only the first feature of each observation and treating that as the hash. We then keep a limited but running list of hashes that seem to occur at the same time. For example, if the first node’s hash function always returns 5 when the second node’s hash function returns 4, we know that there is correlation between these two nodes. Then in the next iteration, based on their performance we add new nodes to the previous model, training combinations of these nodes keeping the ones which seem to have a positive predictive significance.

Figure 4.5: Joint Nodes Hashes

<table>
<thead>
<tr>
<th>Hash Node: Feature 1</th>
<th>Hash Node: Feature 2</th>
<th>Hash Node: Feature 1,2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hash</strong></td>
<td><strong>Class</strong></td>
<td><strong>Hash</strong></td>
</tr>
<tr>
<td>11</td>
<td>Class A: 3, Class B: 7, Class C: 3</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>Class A: 6, Class B: 2, Class C: 3</td>
<td>5</td>
</tr>
<tr>
<td>22</td>
<td>Class A: 1, Class B: 4, Class C: 2</td>
<td>6</td>
</tr>
</tbody>
</table>

13
CHAPTER 5

Hash Collision Nodes for Image Classification

Our 2D image classification process is very similar to the previous method used to classify numerical observations. As illustrated in figure 5.1, we first create N nodes where N is set based on the memory and resource limitations, then we iteratively evaluate the performance of our network to add, subtract, and link nodes when necessary.

Figure 5.1: Process for Hash Node Training - Image Classification

5.1 Data Description

For our experiments we used the standard MNIST dataset. MNIST [9] consists of 60000 handwritten digit images with 10 classes (0-9). Each image is grayscale with range with 28x28 pixels. There are 60000 training images and 20000 standard testing images.
### 5.2 Image Hashing Process

Compared to regular numeric data, hashing 2D data provides a challenge because of the number of inherent features in an image. As illustrated in figure 5.2, each image is put through a process of filtering by convolution, downsampling, and image interpolation much like typical classification neural network. The end result is a hash in the form a flattened 1 dimensional array of 0’s and 1’s with a header describing the parameters of the hash function which created the hash. Similar to our numerical example we can estimate the probability of the actual class of an image equal to class $c$:

$$P(C = c|x_i) = \frac{\# \text{ of Collision Values } = c}{\text{Total } \# \text{ of Collision Values}}$$  \hspace{1cm} (5.1)

Since the number of features and nodes required to categorize images is relatively large, we also need a baseline approach to eliminate nodes without extensive testing. Assuming that the distribution of classes is uniform, as it is in the MNIST dataset then we remove a node from our network if $d(x, c = C)$(i.e. number of collisions with the correct class) $< aR$, where $R$ is expected number of collisions if for a randomly selected image and $a$ is the minimum acceptance threshold [Figure 5.2].
Filter by convolution. Our approach to create filters for convolution is to select a combination of well known group of filters (Gaussian, Edge Detectors, Sharpening Filters) along with a group of randomly initialized filters of three different sizes (5x5, 7x7, 10x10) to be the seed filters of our hash network. The process of convolving the filters is similar to that of a traditional CNN, except, in addition to taking the sum of the results of the filter, we also use the convolved output of each section of the image to produce our hash keys [Figure 5.3]. This allows us to not only make the conclusion that a particular feature was apparent in a certain area on the image but also let us record the interactions of each filter on each region on the image.
Figure 5.3: Convolution Methods for Hash Generation

Figure 5.3: Convolution Methods for Hash Generation

**Down sampling.** In order to have a healthy number of collisions we must down sample our hashes to an acceptable degree. After experimenting with different parameters we notice that on average node’s predictive success rate is highest when down sampling at a ratio of .5 [Figure 5.4]. However the single nodes with the highest predictive abilities often have higher or lower down sampling ratio. The performance range of the hash nodes does not stop drop off considerably until the down sampling ratio is less than .3. Thus, we decide to initialize new nodes with down sampling ratio ranging [.3,.9] .

Figure 5.4: Categorization Success Rate vs Down Scale Ratio
Image interpolation. We also experimented with image interpolation thresholds. As previously mentioned, the MNIST images are grayscale with pixels darkness ranging \([0, 280]\). By setting and interpolation thresholds we can convert the image to monochrome or we can convert the image to grayscale with a specified number of buckets. Holding all else constant the average success rate of the hash function is fairly reasonable for monochrome conversion with interpolation thresholds between 80 and 250. Thus our new nodes are initialized with thresholds within \([80, 250]\).

Figure 5.5: Hashing Process

5.3 Predictions

Predictions for 2D image data is also similar to the 1D case [Figure 4.1] with a few exceptions. Since we have to train a large number of nodes to generate our classification votes, we use boosting methods to increase our classification success rate. We also translate our predictions from each node into ten one-hot vectors to create ten binary classifiers from one classifier [Figure 5.1]. This allows us to use packages more optimized for binary classification boosting.
# Table 5.1: One Hot Conversion

<table>
<thead>
<tr>
<th>Original Prediction</th>
<th>Prediction = 1</th>
<th>Prediction = 2</th>
<th>Prediction = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>
CHAPTER 6

Optimization and Performance Evaluation of Hash Collision Nodes

We use boosting methods to combine multiple learners to make the final prediction. We also demonstrate the capabilities of the hash nodes to “unlearn” specific observations, providing a flexible way for us to jackknife validate our training set. We also show how we can use these metrics to form links between nodes, create new nodes, and train our image filters.

6.1 Boosting hash node Learners

In order to combine our hash node predictions we use XGBoost [4] to boost our hash nodes; experimentation was also initially done with random forest and AdaBoost. While these boosting methods also produced similar results, they were many times slower than xgboost and would not meet the scalability needs of creating a hash node network. In order to get the best performance from xgboost, we encoded all of our predictions into one hot vectors [Figure 5.1], and used this data to train XGBoost. From our test set, we were able to generate a vector of probabilities, one for each class in the MNIST set. Since we already know that the dataset is balanced in terms of classes we choose to normalize all of our predictors to a common $\mu$, and the variation of the predicted probabilities to a common $\sigma$.

6.2 Jackknife Cross Validation and Performance Tracking

Since the widespread use of neural networks of deep learning, the jackknife technique has been largely unused because it is a computationally expensive way of validating a model.
Previous attempts have either used simple networks [12, 14] or more complex networks with a larger portion of the sample size left out for cross validation [13]. This mainly because neural networks aim to optimize weights between its layers and it is difficult, after many iterations, to remove the effects of one single observation. Jackknife resampling however, is very useful in training a model. It estimates a parameter of the population \( \hat{\theta} \) by removing only the \( i \)-th observation each time the model is trained to create an bias-corrected estimator \( \hat{\theta}_{\text{Jack}} \):

\[
\hat{\theta}_{(\cdot)} = \frac{1}{n} \sum_{i=1}^{n} \hat{\theta}_{(i)} \tag{6.1}
\]

\[
\hat{\theta}_{\text{Jack}} = n\hat{\theta} - (n - 1)\hat{\theta}_{(\cdot)}
\]

With standard error:

\[
\text{SE}(\hat{\theta}_{\text{Jack}}) = \left\{ \frac{n-1}{n} \sum_{i=1}^{n} (\hat{\theta}_{(i)} - \hat{\theta}_{(\cdot)})^2 \right\}^{1/2} \tag{6.2}
\]

This is particularly useful in cases where the sample size is low and data collection is very expensive, for example, medical related images where edge cases are particularly hard to attain. Normally the Jackknife estimation requires one to retrain the model \( n \)-times, once for each observation left out. Since many image recognition models require many hours to train, this is practically hard to do.

If we use the hash collision model however, jackknifing is very simple. Instead of training our model multiple times with different data, we train our model only once with all the data, and iteratively “forget” each observation during the jackknife. In order to un-train the model for a specific observation we reuse all the hashes for the training observation and decrement the count of the class associated for each of the hashes. For example suppose our observation hashed to value: 11 and 13, then the effects of “forgetting” in the hash map this observation is:

Since both the training and forgetting process only need to happen once for each observation, we are skipping the re-training of the model. We show that the training complexity for the
simple hash node model without boosting happens only once and can perform the jackknife cross validation in linearly increasing $O(n)$ time compared to the usual quadratic $O(n^2)$ time to jackknife a neural network.
CHAPTER 7

Hash Network Framework Architecture and Scalability

In this section we will review the general architecture and implementation of a hash network to show how our hash network can be implemented to scale and work in parallel. We will also go over alternative implementation options given each user’s needs.

7.1 Architecture Overview

In [Figure 7.1] we present an overview of a hash network architecture with three parts: a data access layer, a hash node abstraction layer, and a boosting layer. Each layer consists of many servers and can only communicate with the layer directly below or above it.

Figure 7.1: Hash Network System Architecture
**Data Storage Layer.** The data access layer is the bottom layer of our architecture that communicates directly to our distributed big data store. It is responsible for batching queries including batch inserts, batch increments, and batch retrievals from our hash nodes and storing the hashed data in the data store. For our implementation we decided to use HBase because of its fast performance in row key lookup and its scalability in large clusters. In fact, any wide column NoSQL database would be suitable to serve as the datastore. Since HBase is a NoSQL database, it allows our model to be flexible; we can add new nodes to augment the number of classes our hash network can identify by simply adding more columns to existing rows. In [Figure 7.2] we demonstrate adding classes to a pre-trained hash network. If the model trained with a large dataset or large set of hash nodes it would requires large amounts of hashes. Therefore, the user can simple add more servers to the database layer to augment its capabilities.

![Figure 7.2: Adding New Classes](image)

<table>
<thead>
<tr>
<th>Before: HBase 3 Classes</th>
<th>After: HBase 4 Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hash</strong></td>
<td><strong>Class</strong></td>
</tr>
<tr>
<td>11</td>
<td>Class A: 3, Class B: 7, Class C: 3</td>
</tr>
<tr>
<td>13</td>
<td>Class A: 0, Class B: 2, Class C: 3</td>
</tr>
<tr>
<td>22</td>
<td>Class A: 1, Class B: 4, Class C: 2</td>
</tr>
</tbody>
</table>

**Hash Node Abstraction Layer.** For our hash node layer we chose to implement our hash nodes in python creating different hash node classes each with its own hashing function and prediction function. The hash nodes are initialized with parameters to define its filters and each hash node connects to the data storage layer to store the generated hash keys in HBase. As we show in figure [Figure 7.1], since the database is distributed, we can train hash nodes in parallel on multiple servers. Similar to the data layer, if the model contains complex convolution and many hash node, we can simply scale by adding more servers that connect to the same database layer since a database such as HBase can handle multiple requests from many servers simultaneously. We can make predictions in parallel as well.
because HBase’s bloom filter technology [18] allows it to rapidly process batch row lookups. Since a hash network is easily scalable and each step for training is just an insert into a database, the model can also simultaneously train and predict at the same time.

**Boosting hash nodes.** Since the xgboost model [4] extremely fast and can be incrementally updated, it fits well into our model. The boosting model is first trained from the training results of the hash network, then as the hash nodes generate more data we update the model in batches. Since this just assigns weights to each of the hash node predictions it does not need to be retrained often. Even though xgboost is designed to be widely scalable [4], it is usually fast enough to be trained on one server.
CHAPTER 8

Experimental Results

In this chapter we will discuss our classification success rates and performance for multiple versions of our model. For both datasets we aim to limit the classification error rate. Table [Figure 8.1] shows the error rates attained by our model on each dataset. Tables [Figure 8.4] and [Figure 8.3] show a detailed confusion matrix and illustrates areas of misclassification. Table [Figure 8.5] details the performance on these models as well as performances from previous studies. We also included results for MNIST classification in [Figure 8.2] from state of the art neural network approaches and other alternative methods [8, 15, 19].

8.1 Classification Success Rate

Using the confusion matrix [Figure 8.4] we can not only see the overall performance but we can also see areas that need the most improvement. For example we can see that the model often confuses 9 as 7. Knowing this we can adjust our aggregation step to search for nodes that can distinguish between 9 and 7 very well.

<table>
<thead>
<tr>
<th>Model</th>
<th>Dataset</th>
<th>Error</th>
<th>Error Range For Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Hash Node</td>
<td>Iris</td>
<td>8%</td>
<td>-</td>
</tr>
<tr>
<td>Multi Hash Node</td>
<td>Iris</td>
<td>4%</td>
<td>8% - 20%</td>
</tr>
<tr>
<td>Single Hash Node</td>
<td>MNIST</td>
<td>4.9%</td>
<td>-</td>
</tr>
<tr>
<td>Multi Hash Node</td>
<td>MNIST</td>
<td>4.8%</td>
<td>4.9%-31.1%</td>
</tr>
<tr>
<td>Boosted Hash Network</td>
<td>MNIST</td>
<td>3.6%</td>
<td>4.9%-31.1%</td>
</tr>
</tbody>
</table>
Table 8.2: Classification Results for Previous Studies

<table>
<thead>
<tr>
<th>Model</th>
<th>Dataset</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logistic Regression (Qin et al. 2009)</td>
<td>MNIST</td>
<td>14.5%</td>
</tr>
<tr>
<td>Nearest Neighbor (LeCun et al. 2006)</td>
<td>MNIST</td>
<td>4.4%</td>
</tr>
<tr>
<td>SVM (LeCun et al. 2006)</td>
<td>MNIST</td>
<td>1.1%</td>
</tr>
<tr>
<td>Convoluted Neural Network (LeCun et al. 2006)</td>
<td>MNIST</td>
<td>.95%</td>
</tr>
<tr>
<td>DropConnect Neural Network (Wan et al. 2013)</td>
<td>MNIST</td>
<td>.21%</td>
</tr>
</tbody>
</table>

Table 8.3: Confusion Matrix For Iris Predicted vs. Expected

<table>
<thead>
<tr>
<th></th>
<th>Iris-setosa</th>
<th>Iris-versicolor</th>
<th>Iris-virginica</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iris-setosa</td>
<td>50</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>0</td>
<td>47</td>
<td>3</td>
</tr>
<tr>
<td>Iris-virginica</td>
<td>0</td>
<td>3</td>
<td>47</td>
</tr>
</tbody>
</table>
Table 8.4: Confusion Matrix For MNIST Predicted vs. Expected

<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>479</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>6</td>
<td>2</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>525</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
<td>481</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>491</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>484</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>436</td>
<td>5</td>
<td>0</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>470</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>548</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>465</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>441</td>
</tr>
</tbody>
</table>

28
8.2 System Performance

System performance tests [Figure 8.5] were done on a single machine with an i7-7700k processor and 32GB of RAM. Most of the time was taken in the inserting and incremental part of the model.

Table 8.5: Performance Metrics for Hash Networks

<table>
<thead>
<tr>
<th>Model</th>
<th>Data</th>
<th>Total Nodes</th>
<th>Training</th>
<th>Prediction</th>
<th>Boosting</th>
<th>Epochs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Hash Node</td>
<td>Iris</td>
<td>1</td>
<td>&lt;1 secs</td>
<td>&lt;1 secs</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Multi Hash Node</td>
<td>Iris</td>
<td>20</td>
<td>2-3 secs</td>
<td>2-3 secs</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Single Hash Node</td>
<td>MNIST</td>
<td>1</td>
<td>38-45 mins</td>
<td>2-3 mins</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Multi Hash Node</td>
<td>MNIST</td>
<td>10</td>
<td>4-5 hours</td>
<td>20-22 mins</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>Boosted Hash Network</td>
<td>MNIST</td>
<td>10</td>
<td>4-5 hours</td>
<td>20-22 mins</td>
<td>&lt;1 min</td>
<td>2</td>
</tr>
</tbody>
</table>
CHAPTER 9

Conclusion

In this project we developed a learning model with two important characteristics. Inspired by the associative learning properties of the brain, we created a classification system based on memory rather than tuning weights, storing memories in pieces that can only be recalled if given the correct stimulus or trigger. Rather than manually creating a rigid architecture of fixed neuron relationships, we use neurons with different properties to let them create their own connections. These characteristics result in a few important properties of the model. Our network built on hash nodes are able to warp and forget previous events, learn in a dynamic and online fashion, and combine previous knowledge with new unrelated experiences. Future avenues of exploration may include using this structure to explore the creative nature of the human brain, to create a system that can generate meaningful conversations, emulate emotion, and make creative decisions in new environments.
Bibliography


