# Symbol Recognition: Featurization and Neural Network Optimization

Colter Radden-LeSage

May 2, 2016

# 1 Introduction

The problem of symbol recognition is already a well-studied field in artificial intelligence. This document is an exploration of the methods traditionally used in symbol recognition in order to solve a small subset of the greater symbol-recognition problem space.

# 2 Image Processing

## 2.1 Pre-Processing

In order to simplify the task, symbol images were simplified from their native format (.jpg files which contain red, green, and blue values for each pixel) into a bitmap format using the Python Imaging Library (PIL). These bitmap images are a translation of the symbols' pixels from the provided three-dimensional space (red, green, blue) into a one-dimensional space where each pixel is either "on" or "off." This greatly simplifies the algorithms needed for image processing, simplifies the input vectors used in the symbol-recognizing neural network, and has the added bonus of allowing the network to recognize symbols drawn in any color of ink.

To further facilitate image processing, Python's NumPy package was used to unpack the bitmap, which contains file header info as well as other unused information about the bitmap, into a two-dimensional array of pixels which can be searched and iterated over using familiar algorithms.

## 2.2 Image Hashing

A number of increasingly complex hashing functions were written to provide features for a neural network. Each generation of hashing functions represents an input vector passed to a neural network for testing.

1. **First Generation (naive)**
   (a) Count the number of black pixels in the array
   (b) Locate the boundaries of the symbol (one for each edge)
   (c) Calculate the height and width of the symbol from the boundaries
2. **Second Generation (more informed, inflexible)**
   (a) Using the boundaries found in the first generation, crop the array to the boundaries of the symbol
   (b) Divide the cropped region into 16 equally-sized rectangles
   (c) Count the black pixels in each region
3. **Third Generation (informed and flexible)**
   (a) Using the cropped region from the second generation, divide the array into an N-by-N grid of equally sized rectangles
   (b) Count the black pixels in each of the $N^2$ regions
   (c) Divide each of the pixel counts by the total number of black pixels in the image to normalize the input vector.

## 3 Network Accuracy

A neural network containing a single layer of five hidden neurons between the input and output layers was constructed using the PyBrain library to test the first generation of hashing functions. An input vector was constructed which contained the total number of black pixels in the bitmap, the indices of the first and last rows to contain black pixels, the indices of the first and last columns to contain black pixels, and the calculated height and width of the given symbol. After training, this neural network did not perform better than guessing randomly (78% error.)

The second generation hash function performed significantly better than the first generation when built and trained on an identical network (18% error.) This revealed that the generalized locations of pixel groupings is a far more predictive of the symbol than the dimensions of the symbol or location of the symbol within the image. This discovery was further refined into the third generation of hashing function.

The final hash function written has two major advantages over the second generation hash function. First, it takes a parameter, $N$, which is used to scale an $N$-by-$N$ sized grid of pixel buckets. This grid captures the black pixels in the bitmap based on their locations. In this way, the resolution of the hash function can be scaled to find a value for $N$ which optimizes the efficiency of the neural network by minimizing the size of the input layer, which is of size $N^2$. Additionally, the final implementation normalizes the data by dividing the number of black pixels found in each region by the total number of black pixels found in the bitmap. The vector therefore represents the *percentage* of all black pixels in each region of the symbol. The observed increase in accuracy (9% error when $N$=4) is likely due to subtle differences between otherwise identical symbols, such as line thickness, being better captured by a percentage than an absolute value.

## 4 Optimization

A grid search was performed to find optimal values for $N$ (between 2 and 7, inclusive) and the number of neurons in the hidden layer (between 1 and 8, inclusive.) Each neural network trained for 150 epochs and the validation set errors were recorded. The optimal size of the hidden layer was five nodes and the optimal value for $N$ was four. Interestingly, when $N = 2$, (very low hash resolution) the error was still less than 25% when the number of hidden layer nodes was high ($> 5$.)

## Footnote

The source of error for this project appears to come exclusively from differentiating hash symbols from dollar symbols. This is likely due to the similar pixel distributions in these shapes.