Capstone Report

An Introduction to Handwritten Digit Algorithms

Marco Sousa

University of Massachusetts Dartmouth
Dartmouth, Spring 2021

Abstract

The following is an introduction to classification of handwritten digits, which is a standard problem in pattern recognition. Subjects in this paper provide a brief introduction in carrying out several machine learning algorithms for classification, in addition to reporting results for test cases. The MNIST and US Postal Service databases were used to measure algorithm performance. Methods of classification for K-Means, SVD, Tangent Distance, K-Nearest Neighbors, Neural Networks, and Convolutional Neural Networks are introduced, then results of test cases are reported after each algorithm. Basic meta modelling was carried out using a weighted average and stacked ensemble. C-NNs seemed the most promising for the task, and produced the greatest accuracy among the models.

Contents

1 Project Goals

This paper serves to achieve three project objectives. Most directly, the goal is to classify a dataset of handwritten digits accurately and appropriately. Additionally, this project served as a means to acquire a better understanding of image processing basics, classification techniques, and machine learning more broadly. As such, subjects in this paper provide a brief introduction in carrying out several machine learning algorithms for classification, in addition to report results. Several classification algorithms are described, then results of test cases are reported and discussed (simultaneously) after each algorithm. Additionally, reference code for each algorithm can be found via my digit classification github repository.

2 Introduction

2.1 Classification

The essential idea of classification is to use a set of classified/identified objects (a training set) to construct a model that will further classify unknown objects (a test set). The 'dependant variables' to be classified are discrete values or categories, as opposed to continuous values, which would entail techniques such as regression. We are interested in the success or error rate, and also speed of the classification. The success rate or accuracy are simply and intuitively represented as (correct classifications)/(total classifications).

2.2 Supervised vs Unsupervised

There are four primary types of learning that are typically discussed when referring to types of machine learning algorithms: supervised, unsupervised, semisupervised, and reinforcement. The goal of machine learning broadly is to determine a model that takes in attributes X as inputs, and outputs values Y in relation to those attributes. Constructing a function f that maps X_{test} attributes to to Y_{test} predictions, in the form $f(X_{test}) = Y_{test}$, in some manner, would be a sufficient model, most generally. Supervised learning uses Y_{train} labels of the X_{train} attributes to 'supervise' the training of the model. A model is then used to attempt to predict the correct Y_{test}, sometimes denoted as $\hat{Y}$. Unsupervised learning, in contrast, does not have access to these training Y labels, and must only rely on techniques using X training labels. Semi-supervised learning combines, typically, a small amount of labelled data with a larger amount of unlabelled data. Reinforcement learning is outside the scope of this paper.

All of the following algorithms discussed in this paper that were carried out were supervised techniques, with the exception of K-Means.

3 Digit Data

There are two popular databases when classifying digits: US Post Office Zip Code Data and the MNIST handwritten digit database. The latter has been classified using convolutional neural networks with an error rate as low as 0.25 percent. The training and test data for both databases are explicitly split to facilitate algorithm performance comparisons.

Both handwritten digit databases considered in this paper provide training X_n and Y_n in addition to testing X_m and Y_m. The objective, naturally, is to train a model using X_n and (if supervised) Y_n, then test that model against X_m, and finally observe how well the predicted $\hat{Y}_m$ matches the actual Y_m.

3.1 US Post Office Zip Code Data

The US Postal Service data can be acquired as provided by Stanford. As is stated in the preliminary information on the Stanford site, the original scanned digits are binary and of different sizes and orientations; the images have been deslanted and size normalized, resulting in 16 x 16 grayscale images (Le Cun et al., 1990).

The dataset contains 7291 training observations and 2007 test observations. Column observations represent images, which can be represented in 3 formats in particular. Each row contains 256 cells that acts collectively as an observation, preceded by a digit label. Such rows are vectors in R^{256}, and can be transposed to be column vectors. Each observation can be reshaped into 16 by 16 pixel grayscale images that represent a handwritten digit. One can furthermore consider the 16 by 16 pixel image as a function of two variables, f(x,y).

The training or test data may be visualized as mxn matrices as follows:

$$\left[\begin{array}{c|cccc} y_1 & x_1 & x_2 & \cdots & x_{256} \\ y_2 & x_1 & x_2 & \cdots & x_{256} \\ \vdots & x_1 & x_2 & \cdots & x_{256} \\ y_m & x_1 & x_2 & \cdots & x_{256} \end{array}\right]$$

Programming Observations

If the reshape command in MATLAB is simply used, you may need to additionally perform a reflection and rotation. It is worth considering converting the values from [-1,1] to a grayscale [0,1], depending on the algorithm that is being used. To better display the image, one may consider calling imshow(1-image), so that black inverts with white. Alternatively, one can conserve their original values by reversing the color mapping in MATLAB settings.

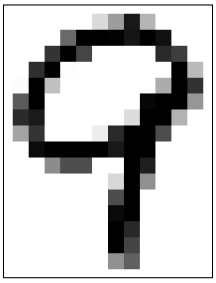


Figure 1: A representation of a formatted digit (US Postal).

3.2 MNIST Data

The MNIST (Modified National Institute of Standards and Technology) dataset can be acquired as provided by Yann LeCun. Depending on your browser, accessing the data directly through Yann's site may be difficult, but there are several other locations where the renowned MNIST dataset is available.

The MNIST dataset consists of 60,000 training images and 10,000 test images with grayscale pixel values ranging from 0 to 255. Each set also has a corresponding set of labels. Each image observation fits a 28x28 matrix, and separately has a digit associated with it. For both the training and test set we have the following arrays:

$$\text{X Data: } [[x]_{28\text{x}28}, ..., [x]_{28\text{x}28}]_{1\text{x}m}$$

$$\text{Y Labels: } [y_1, ..., y_m]_{1\text{x}m}$$

The X training data appears as follows:

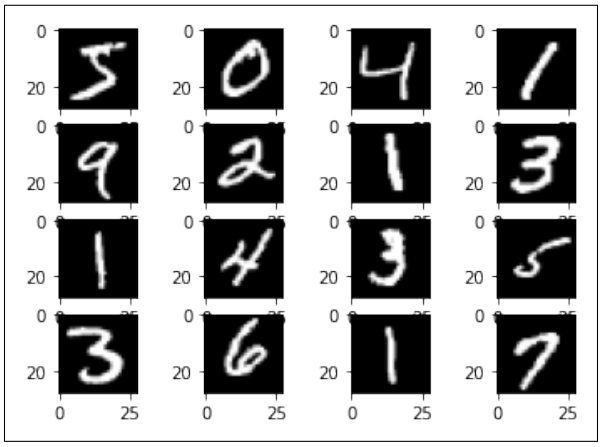
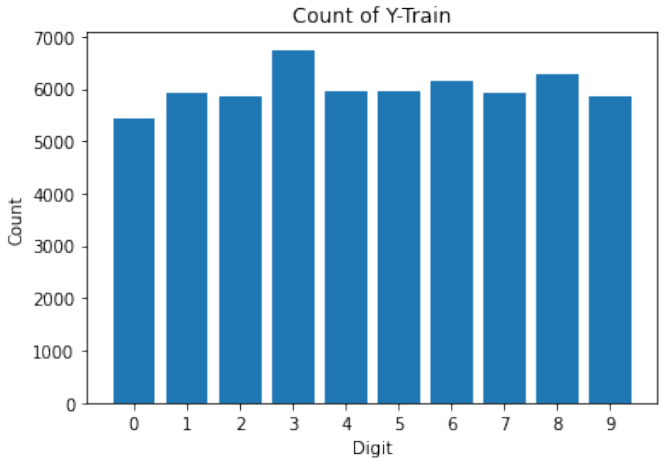
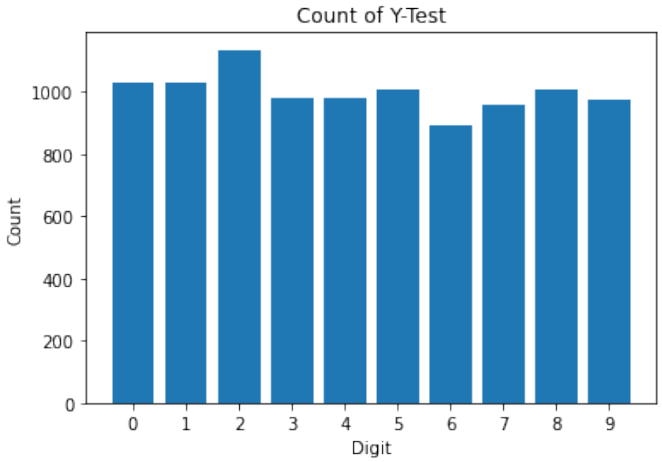


Figure 2: The first 16 MNIST training digits

Histograms of the Y labels for each the training and test appear mostly uniform, and are shown below:





Programming Observations

The data as acquired from keras/tensorflow takes the form of uint8, a common graphical data type consisting of whole numbers ranging from 0 to 255. While indeed images exist within this boundary, such a datatype restricts the allowance for negative values which may occur when element-wise subtraction takes place, such as when calculating distances. As such, the data type of the data was converted to avoid such complications.

4 K-Means Classification

The classification of an image often revolves around the conception of minimizing differences. As such, conceptions of difference can be measured with forms of distances. The essential idea of a K-means clustering technique is to compare the euclidean distance of one image to each centroid representing a digit class. Note that euclidean distance can be calculated on an image regardless of whether it is a matrix of $R^{16\times16}$ or a vector in R^{256} without loss of generality because of the robustness of the norm operation. Such a difference between two images p and q can be represented as follows:

$$\text{Euclidean Distance} := \|p - q\|_2$$

$$= \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2}$$

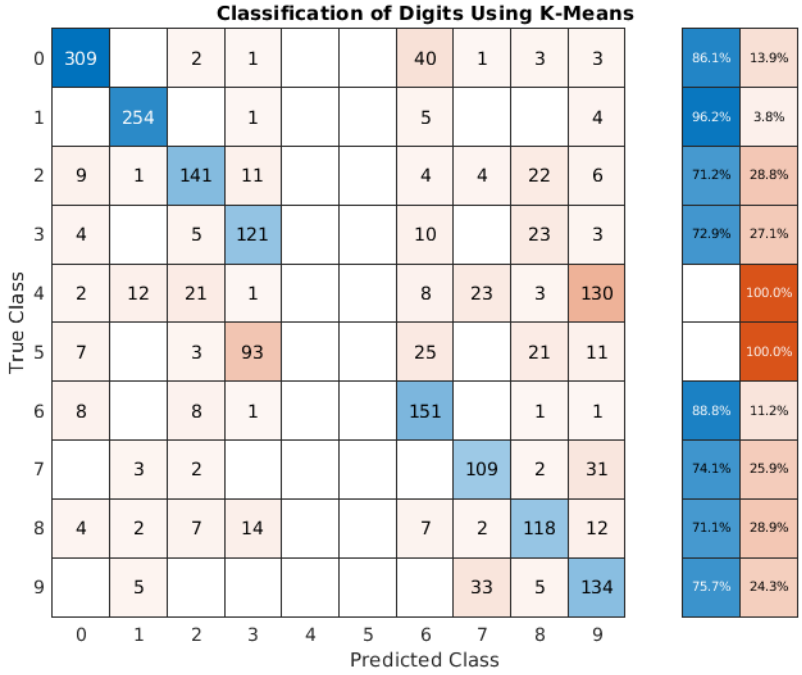
4.1 The Algorithm

The data is trained by performing a K-Means clustering on the training set. From such, 10 centroids represent each digit. This process is partially supervised at the start, as each cluster must be manually classified to represent a digit. In classification, for every digit p and each cluster c_k, the euclidean distance $\|p - c_k\|_2$ is calculated. The mean centroid with the minimum distance is chosen as the classifier for that test digit.

One common question for K-Means clustering is to consider how many k clusters are appropriate. However, with this data set, assuming the digits form distinct clusters, one must consider at least 10 clusters. However, since I found clusters were not sufficiently distinct from one another, there may be a repeat in considering a digit, and thus, by pigeon hole principle, all classifications of the given digit result in error. This was seen in the confusion matrix found on the next page, where the 4 and 5 were often classified as 9 and 3 wrongfully.

4.2 Results

Because of the error with 4 and 5, this particular K-Means algorithm produced a success rate of 66.62 %. However, in other cases, this can by chance be as high as approximately 75 %. As such, I found a good deal of variability in my results for this algorithm. This variability in the algorithm occurs when sampling the training set. Random sampling can improve the accuracy in this method case because it by chance can remove poorly written digits.



5 SVD Classification

Another classification algorithm involves the use of SVD. The essential idea of the SVD algorithm is to suggest all the digits of a single kind span a subspace. That subspace, represented in an orthogonal manner because of the structure of SVD, can be identified, and compared with each test digit, most simply. Rather than using euclidean distance, the difference is measured from the residual between the image and the computed subspace of the classes of a particular digit. The residual can be represented as follows:

$$\left\| z - U_k U_k^T z \right\|_2$$

The relative residual can be represented as follows, similarly:

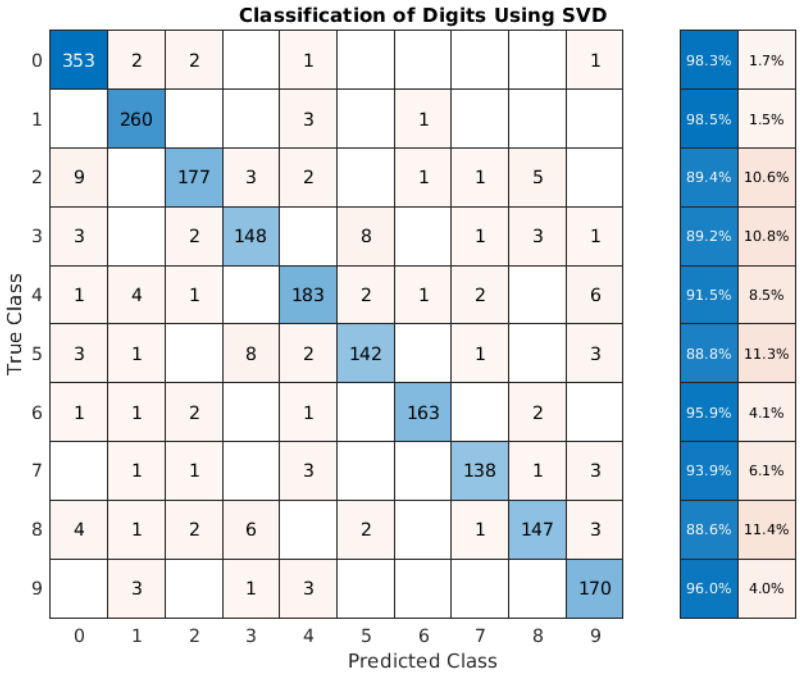
$$\left\|(I - U_k U_k^T)z\right\|_2 / \|z\|_2$$

5.1 The Algorithm

Separate the digits of the same kind in the training set and compute the SVD of each to form a subspace of k singular values. For each test digit, compute the residual between the test digit and the 10 generated subspaces. Choose the minimum and classify the test in accordance to its respective subspace. One may also consider relative residual.

5.2 Results

Such an algorithm holds some advantages. Since it is not necessary to perform a full SVD, one may instead choose less k singular values, which saves more computation time. Furthermore, it's fairly accurate, with accuracy around 93 to 94 %, and low variability. One result is represented in the confusion matrix that follows below.



6 Tangent Distance Classification

6.1 Motivations for Tangent Distance

While the SVD method was a significant improvement from the K-means method, an upper bound of 94 % is not entirely accurate. One consideration we have yet to take into account is that the digits may be altered in some way. A digit can be rotated, translated, thickened, scaled, and so forth. A human eye can typically recognize these adjustments in writing and yet still classify an image appropriately. As such, the tangent distance procedure seeks to remedy this discrepancy in writing by conceptualizing a distance between two curves. The essential idea for the tangent distance method is to form a new conception of differences which approximates the distance between two curves by the distance produced between their tangent lines. These adjustments in writing are called 'invariances' because the transformation on the digit is invariant to the calculated tangent distance. In this manner, we may consider small transformations on digits, yet still conserve the same tangent distance, and thus, under small invariances, still retain an appropriate classification.

6.2 Tangent Distance

Considering a digit p, we can perform a transformation, s, on the image, with parameter a, in order to adjust the image. In such a manner, we can represent a transformation on an image as s(p,a), with s(p,0) = p. To approximate this nonlinear curve, or manifold, under a transformation s, we can construct a linear approximation. A Taylor expansion can be performed to approximate the curve as follows:

$$s(p_1, a) = s(p_1, 0) + \frac{ds}{da}(p_1, 0)a \approx p_1 + t_{p1}a$$

If we have a second point p_2, then we can define the transformation similarly:

$$s(p_2, a) = s(p_2, 0) + \frac{ds}{da}(p_2, 0)a \approx p_2 + t_{p2}a$$

Noticeably, the calculation of the tangent vector, $\frac{ds}{da}$(p,0), is required, and such tangent vector will be calculated in a later section. This approximation also satisfies our aforementioned essential idea. To approximate the curve s(p,a), we use a linear approximation represented by the tangent vector. To find the difference between curve one and curve two, we subtract the approximation of each curve. Then the difference can be taken as follows:

$$\begin{aligned}
&\min \|\text{(curve one) - (curve two)}\|_2 \\
&= \min \|s(p_1, a_{p1}) - s(p_2, a_{p2})\|_2 \\
&\approx \min \|p_1 + t_{p1}a_{p1} - (p_2 + t_{p2}a_{p2})\|_2 \\
&= \min \|(p_1 - p_2) + t_{p1}a - t_{p2}a\|_2
\end{aligned}$$

$$\min_{a_{p1},a_{p2}} \left\| (p_1 - p_2) - \begin{bmatrix} -t_{p1} & t_{p2} \end{bmatrix} \begin{bmatrix} a_{p1} \\ a_{p2} \end{bmatrix} \right\|_2$$

The prior difference considers two images p_1 and p_2, and a single transformation parameter, a. However, we can imagine moving each digit along several curves transformed by many a. Thusly, we may consider a set of transformations parameterized by $\{ a_1 ... a_l \}$ with respective $T_p = (\frac{ds}{da_1} \frac{ds}{da_2} ... \frac{ds}{da_l})$ and continue as follows:

$$\min \left\| (p_1 - p_2) - \begin{bmatrix} -T_{p1} & T_{p2} \end{bmatrix} \begin{bmatrix} a_{p1} \\ a_{p2} \end{bmatrix} \right\|$$

Such a formulation is of course in the form of solving a classic least squares question:

$$\min \|b - Ax\|_2$$

Note that the solution for a minimum alpha is not what we are directly interested in, but rather, the residual of the solution, described by the norm. Recall that alpha is decided apriori as a paramaterization of the transformation of image p under s, in the form of s(p,a). An important consideration when solving such a system is whether matrix A is singular or not. Simard et. al.[1] remarks that matrix A composed of tangent vectors may become singular when some of the tangent vectors for p_1 and p_2 become parallel. As such, to the best of my understanding, when constructing tangent vectors for p_1 and p_2, there may exist tangent vectors that are parallel, and furthermore linearly dependent, and thus singular. One solution is to use a QR method. Solving with a QR method must split into two cases of singular and nonsingular. In implementation, I chose to follow the method of controlled deformation, proposed Simard [1] and again used by Jose Israel Pacheco [2].

6.3 Transformations

Recall that we are considering slight perturbations of image p under s paramaterized by a. As such, different transformations, such as scaling, rotating, translating in x or y directions and so forth, act on image p, and produce respective tangent vectors t. It would be expected that a transformation by a nonzero alpha would change the curvature of image p, along with its tangent vectors. Furthermore, the change on the curve and tangent vector would naturally be different for different transformations. As such, it is necessary to describe and derive each transformation accordingly.

Transformation as Linear Combination

Most simply, in translation, we aim to move the image pixels horizontally or vertically. The direct mapping of the transformation is as follows for a horizontal translation:

$$t_a = \begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} x + a \\ y \end{pmatrix}$$

If one considers p =p(x,y) as a function in two variables, then s becomes s(p,a_x)(x,y) = p(x+a_x,y). Finding the derivative will create our respective tangent vectors through our paramaterized operator L. $p_x = \frac{dp}{dx}$ can be derived with the chain rule:

$$L_x = \frac{d}{da_x}(s(p, a_x)(x, y))|_{a_x=0} = \frac{d}{da_x}p(x + a_x, y)|_{a_x=0} = p_x(x, y)$$

Such a mechanism constructs the derivative of the respective transformation. This produces a specific tangent vector for transformation s, of which we can paramateraize as operator L with respect to a in the form of $p + La$, analogous to the prior $p + t_p a$. The prior operator, L_x, is the derivative for a transformation s by translating in the x direction. Notice for any operator L, when considering a=0, the original p is retained: $p + 0L$ = p.

Table of Operators

The derivation of other transformations can be found in Analysis and Tests of Handwritten Digit Recognition Algorithms(Savas)[3] or Matrix Methods in Data Mining and Pattern Recognition (Eldén)[4]. It can be seen that the tangent vectors in the form of transformation operators can all be represented as linear combination of the x and y derivatives p_x and p_y. A table of the operators are as follows.

Table 1: Transformation Operators and their Derivative Representations

Transformation	Mapping from p(x, y)	Derivative
x-Translation	p$(x + a, y)$	p_x
y-Translation	p$(x, y + a)$	p_y
Rotation	p$(x\cos(a_r)+y\sin a_r, -x\sin a_r+y\cos a_r)$	$yp_x - xp_y$
Parallel Hyperbolic	p$(x + ax, y - ay)$	$xp_x - yp_y$
Diagonal Hyperbolic	p$(x + ay, y + ax)$	$yp_x + xp_y$
Scaling	p$(x + xa, y + ya)$	$xp_x + yp_y$
Thickening	N/A	$(p_x)^2+(p_y)^2$

Programming Observations

Regardless of the method in which you calculate the tangent vector, you will need to access each pixel at some point. As such, you are bounded by a minimum of kmn FLOPs, where k are the necessary flops to carry out your operator procedure for image of size m×n pixels. One should take notice, therefore, that the size of an image matters in respect to the FLOP count when considering many images. Thus, downscaling, as was done with the MNIST database, is worth considering for very large datasets.

If one chooses to implement tangent distance by means of mapping pixels directly, then some translations do not correspond in their pixel domains. When considering image $R^{16\times16}$ and P[x][y], x and y are elements of integers ranging from 1 to m and 1 to n respectively. However, for mappings such as rotation, x and y can map to noninteger values when

considering the output of trig functions. To adjudicate for this, an interpolation must be done for each rotation to map in the integer domain of x,y = 1...16. For many images, for the set of a_l parameters, considering each image of size m × n, an interpolation would be required, and can become quite expensive. As such, the next section details more appropriate methods of attaining the tangent vector, and operator similarly, by constructing a linear combination of p_x and p_y.

6.4 Tangent Vectors

In the calculation of the tangent distance, a necessary calculation is the construction of a tangent vector t_p, or $\frac{ds}{da}$. The appropriate derivation of the tangent vector requires differential geometry concepts I do not completely understand. However, I will briefly discuss the essential idea, and a practical approximation in the following section.

6.4.1 Derivation of Tangent Vector by Gaussian Convolution

In order to construct the tangent vector of the nonlinear manifold, we must convert an image from a discrete matrix P[i][j], for integer indices i and j, to a tensor function in two variables, p(x,y). An image P[i][j] can be written as a single discontinuous function using the 'delta' function (Savas,42)[3]:

$$P'(x,y) = \sum_{i,j} P[i,j]\delta(x-i)\delta(y-j)$$

The discontinuity of the function is obvious when expanded into summed piecewise functions:

$$P'(x,y) = P[i,j]\delta(x-1)\delta(y-1)+P[i,j]\delta(x-1)\delta(y-2)+P[i,j]\delta(x-2)\delta(y-1)+...+P[i,j]\delta(x-16)\delta(y-16)$$

Convolution with a Gaussian produces a continuous function f in the form of:

$$f(x,y) = \sum_{i,j} P[i,j]g_\sigma(x-i,y-j)$$

While I am familiar with the convolution of an image, the application of the convolution of the image with a Gaussian function is not distinctly clear to me. Furthermore, it's not exactly clear how the delta function constructs the derivative or why. As such, in implementation, I choose to approximate the tangent vector t_p for image p under transformation s by a method of finite differences.

6.4.2 Approximation of Tangent Vector by Finite Differences

The transformation of s on p necessitates the construction of tangent vector t_p. This tangent matrix, derived from the tensor p = p(x,y) under operation s, typically requires

the derivative in terms of x or y, dependant on the specific operation. These derivatives $\frac{\partial p(x,y)}{\partial x}$ and $\frac{\partial p(x,y)}{\partial y}$, can be calculated with finite differences, such as central distance, as defined below:

$$\frac{\partial f(x,y)}{\partial x} \approx \frac{f(x+h,y) - f(x-h,y)}{2h}$$
$$\frac{\partial f(x,y)}{\partial y} \approx \frac{f(x,y+h) - f(x,y-h)}{2h}$$

Applying the central difference directly on the matrix, with h=1, is defined as below:

$$P_y(i,j) \approx \frac{P[i+1,j] - P[i-1,j]}{2h}$$
$$P_x(i,j) \approx \frac{P[i,j+1] - P[i,j-1]}{2h}$$

You would expect $P_x(i,j)$ and $P_y(i,j)$ to be swapped. However, note that in practice, the i in image P[i][j] corresponds to a row, and thus operates in the y dimension, such as in MATLAB. That is to say adding to your i will move you down rows, and so you are working along y; not the x. Although this runs opposite of the notation described by Savas [3], I conjecture in implementation the derivatives are calculated in correct correspondence to ∂x and ∂y because of the direction of i and j when coding.

When considering the transformation operators in the prior section, x and y become somewhat less clear to me when we considering the image in respect to finite differences instead of a tensor function. x and y can no longer represent pixel domains, since a diagonal hyperbolic transformation would seem to imply a tangent vector in the form of 16px+16py for P[16][16], as an example. Yet this clearly cannot be the case, since the grayscale images range from [0...1], and would completely blacken an image. As such, my understanding is that x and y represent gradient matrices, in which the column or row values range from $[\frac{1}{n}, \frac{2}{n}, ...\frac{n}{n}]$ or $[\frac{1}{m}, \frac{2}{m}, ...\frac{m}{m}]$ respectively. In such a manner, we multiply p_x or p_y by a row or column gradient, which represents x or y. This intuition may be incorrect, however.

6.5 Pre-processing

When considering the classification of a digit, the representation of the digit is of significant importance. One method of improving classification performance is to employ a preprocessing technique, where images are adjusted in some way prior to training and testing. While there are several ways to preprocess grayscale images, I chose to blur (or smooth) the images with a gaussian kernal ($\sigma = 0.9$), so that the edges are more smoothed and less sharp. Berkant Savas [3] suggests a method of approximating the blurring process by cutting the tail of the kernal, which saves preproccessing time for large datasets. While this is useful, in my implementation, I simply performed a full gaussian blur. This may become more important when considering larger datasets such as the MNIST dataset.

If you construct the tangent vector by the product of gaussian functions, then the blurring occurs locally and simultaneously with the construction of the tangent vectors. However, if constructed by finite differences, then blurring must be done ahead of time.

6.6 The Algorithm

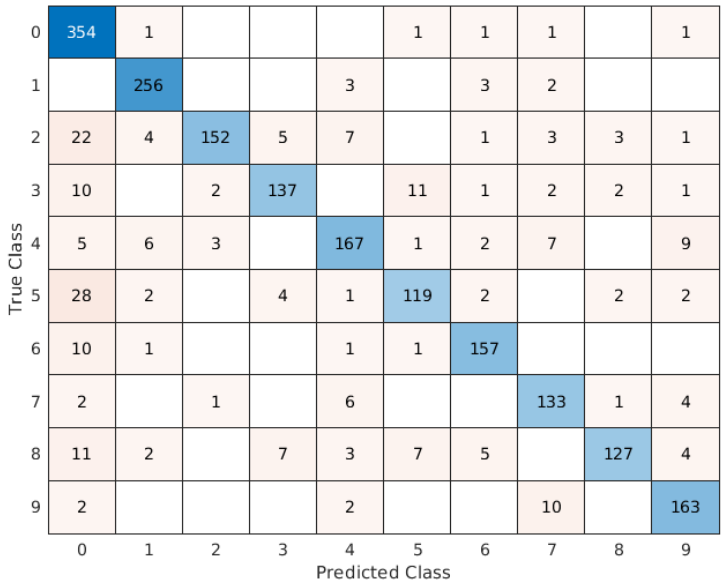
Compute the Tangent Matrix T_p for each digit in the training set. For each test digit, compute its Tangent Matrix. Then compute the tangent distance to that of each training digit.

The clear disadvantage is of course the fact we must compare to every training digit. This can be very expensive, but will hopefully improve accuracy. It is also possible to instead consider a set of images that most distinctly represent the image, and use that training set.

6.7 Results

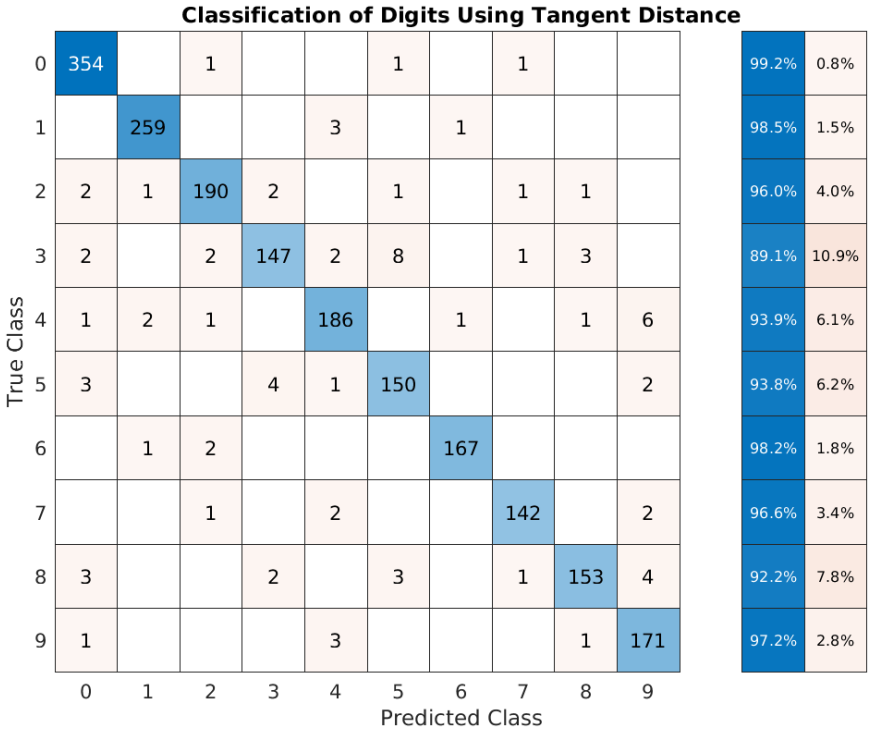
The following confusion matrix demonstrates my tangent distance algorithm with horizontal and vertical translation, rotation, and thickening, using the direct mapping and interpolation method on constructing tangent vectors and transformation operators. In this procedure, the image set was not blurred. This produced a matrix with accuracy of approximately 88.25 %.

Tangent Distance Confusion Matrix - Pixel Mapping



The following is a confusion matrix with an success rate of 95.9%, using finite derivatives to approximate the and construct tangent vectors considering the same transformations. I implement a full gaussian blur before classification.

Tangent Distance Confusion Matrix - Finite Differences



7 K-Nearest Neighbors

K-NN reuses the concept of euclidean distance between two images. A smaller distance implies there is less distance between the two images. In prior algorithms, the minimum distance determined which class displayed the least difference, and that minimum class was chosen. However, it's also possible to consider the bottom k minimum distances, and choose the most popular class among these. Doing such a technique is called majority voting. A probability distribution can be constructed by counting the occurrences of each class, and dividing those respective classes by the total k minimum values considered.

7.1 The Algorithm

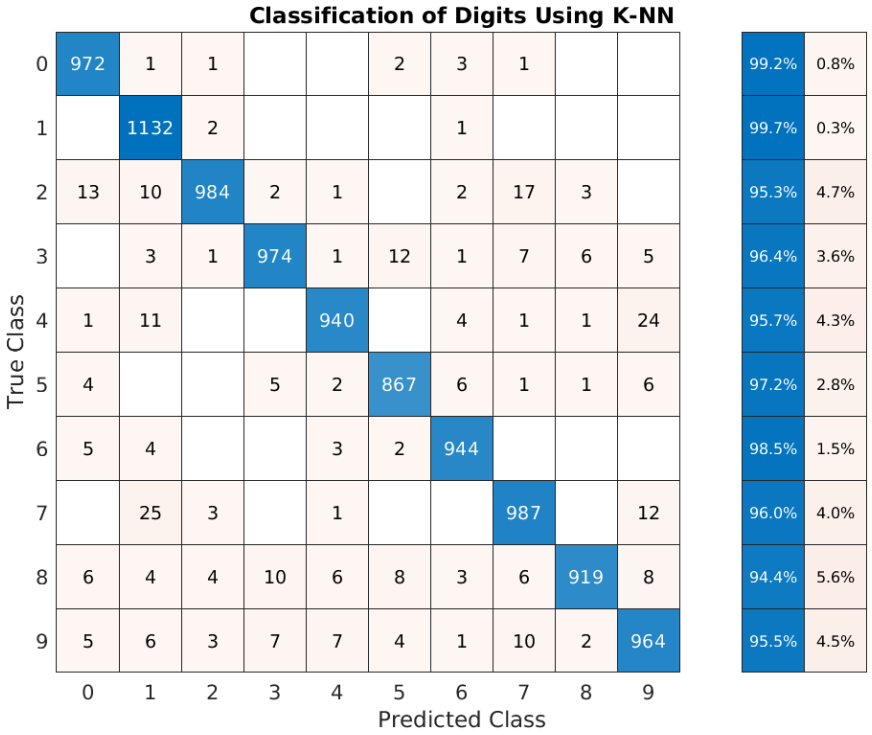
1. For each test digit, compute the euclidean distance against each of the training digits.
2. Determine the minimum k distances

3. Count how many of each class occur respective to the minimum k distances
4. Choose the most frequent class

The determination of k minimum distances is an important component in regards to the computational complexity of the algorithm. If the entire training and test sets are used, then the algorithm will need to determine the minimum k distances 60,000*10,000=6×10^8. If sorting is used to determine the minimum k, then the algorithm will need to undergo 6×10^8 sorts likewise. Furthermore, the bottom k algorithm cannot simply choose the bottom k distances for x, but must also be able to in some way maintain its relationship to its respective training label y.

7.2 Results

Despite the simplicity of the K-NN algorithm, the algorithm can acquire fairly high accuracy. Running K-NN on the entire MNIST test set against all 60,000 training digits produced an accuracy of 96.83%. This should always be the case given the same paramaters because there is at base no stochastic element inherent to the algorithm. The following is a confusion matrix for such a procedure:



8 Neural Networks

8.1 Introduction to NN

One thing that tends to be fairly good at determining the difference between simple digits is our brain, or perhaps our eyes as well by extension. Brains seem to be composed of neurons connected in a complex network, working in fairly complicated and perhaps mysterious ways. Similarly, in computer science, a node can be connected to other nodes in a network, simulating this phenomenon in the brain and constructing an architecture than can pass data along nonlinearly and learn.

8.2 Neurons

Neurons are the simple building block of a neural network. They accept input, perform some math on them, and produce a single output. A neuron with x_n input follows the following function:

$$f((w_1x_1 + w_2x_2 + ... + w_nx_n) + b) = y$$

x_n are parameter inputs x_1 through x_n
w_n are weights associated with each input x_1 through x_n
b is a bias that is added
f is an activation function

An activation function decides whether a particular neuron should fire or not. Given the combination of weights, parameters, and bias, the output could range from $(-\infty, \infty)$, and does not give a clear determination on whether a neuron should be fired or not. The activation function receives such a calculation, and maps it to another distribution that can be used to determine whether the neuron will fire or not. There are several activation functions, but one very common function is the sigmoid function, shown as follows:

$$A = \frac{1}{1 + e^{-x}}$$

Which follows the following distribution:

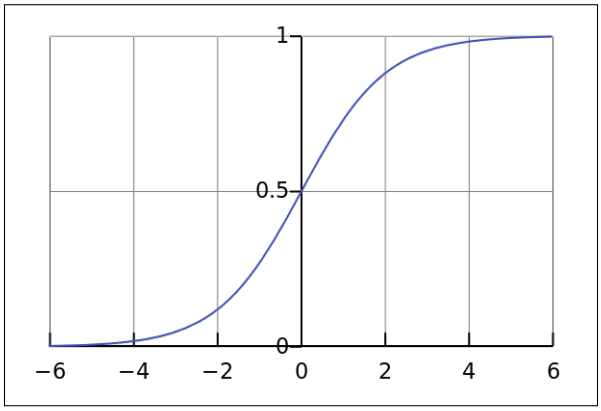


Figure 3: Sigmoid Function

It can immediately be seen that the distribution is nonlinear. This is a useful condition when training a network, which relies on partial derivatives, discussed later. Furthermore, we can see the function maps values from $(-\infty, \infty)$ to [0,1], which gives a distinct range and cutoff.

8.3 Networks

A neural network is the combination of such neurons. The following is one simple neural network:

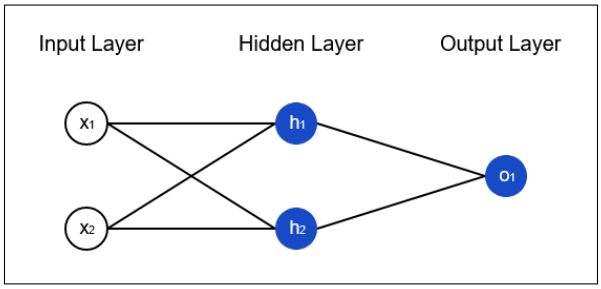


Figure 4: Simple Neural Network (by Victor Zhou)

This network has an input layer, a single hidden layer, and an output layer. The input layer contains inputs x_1 and x_2, the hidden layer contains neurons h_1 and h_2, and the output layer contains a single neuron o_1. Generally, a neural network can have any number of layers with any number of neurons. The process of performing calculations and passing information forward along the network is called feedforward.

8.4 Training a Neural Network

8.4.1 Loss

However, simply passing information along is not sufficient for learning. Rather, we need to optimize our weights and biases in relation to our activation function in some manner that will improve our predictions. As such, we need a method of training the algorithm to improve, and in order to improve, we need a mechanism that describes what a good result is, and how it can be better. This is the purpose of a loss function, which quantifies how bad a calculation is in relation to a particular event. This is to suggest we ask, given a set of weights and biases, how will our error change as we change these weights and biases? As such, it is common practice to create a loss function and somehow minimize our loss, which is something like our error given some conditions. There are many loss functions, one common such function is the Mean Squared Error, as can be seen below:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

y_i resembles our y_{actual} values, and $\hat{y}$ represents our y_{pred} values. Since it doesn't make sense to take the difference of digit outcomes, or even a binary outcome, another method of loss called cross-entropy loss will be used, and is discussed later.

8.4.2 Partial Derivatives for Loss

The goal is naturally to find optimal weights and bias to minimize the loss, and likewise error. Training the algorithm will intuitively be a manner of adjusting the biases and weights to minimize loss, and thusly reduce error. The primary question is how do we appropriately adjust values in our algorithm to reduce such loss. Consider a general loss function that depends on weights and biases:

$$L(w_1, w_2, ...w_n, b_1, ...b_k)$$

Such a notation assumes weights are respective to their node, that is, weights 1 through n include all weights among all nodes, but a particular node does not contain all weights 1 through n. Each node will contain a single bias for k nodes. Changing the weights and biases would hopefully change the loss in some way. Calculating the partial derivatives $\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, ..., \frac{\partial L}{\partial w_n}, \frac{\partial L}{\partial b_1}, ..., \frac{\partial L}{\partial b_k}$ shows how the loss may change under an adjustment with a respective weight or bias, in regards to a particular chosen loss function. However, simply calculating $\frac{\partial L}{\partial w_1}$ requires more information, and should somehow relate to the data that is being passed through our variables. The chain rule can be applied to a partial derivative to calculate the example derivative for weight 1 for figure 4 as follows:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}$$

It should be observed that adjusting the weights and bias changes loss in a compositional manner. Loss is a metric of the change in the y_{pred}, which is a consequence of the activation function, which takes in weights and biases. As such, derivatives of each function should be considered when chaining, and one should be careful to understand what each partial derivative is in relation to. $\frac{\partial L}{\partial y_{pred}}$ relates to the derivative of the loss function in regards to y_{pred}, and the derivative of the given loss function will be taken into account here. $\frac{\partial y_{pred}}{\partial w_1}$ relates to the predicted values in regards to a particular weight, this time being w_1, yet we do not immediately have a function that corresponds as such. Instead, ∂y_{pred} is calculated as a predicted output passed along from neuron output, which is a function of the activation functions and its respective weights. As such, it can be split in relation to the neuron h which contains w. When considering the specific example of figure 4 above, the splits of each are as follows:

$$\frac{\partial y_{pred}}{\partial h_1} = w_j * f'(w_j h_1 + w_i h_2 + b)$$

The predictions are a consequence of the output layer, which uses neurons from the prior layer, in this case h_1 and h_2. Since we are interested with a weight in the first neuron, it will further split on regards to that weight.

$$\frac{\partial h_1}{\partial w_1} = x_1 * f'(w_1 x_1 + w_2 x_2 + b)$$

This is fairly straightforward. h is a neuron which has activation function f and passes w and b into it. The partial derivative for w_1 is the chain rule.

8.4.3 Backpropagation and Gradient Descent

Putting this together results in a chain of partial derivatives which determines how the change in that particular weight will affect the loss function. Such a system of calculating partial derivatives backwards is called backpropagation.

In general, we need an optimization algorithm to carry out such a process in an iterative manner. Gradient descent is one such algorithm that updates the parameters by calculating the slope (partial derivatives), and updating the value by subtracting a constant times that slope. In such a manner, the slope is being updated so that the algorithm converges to a minimum through iterations. The update for a particular weight can be seen as a simple update function:

$$w_1 \leftarrow\!\shortmid w_1 - \eta \frac{\partial L}{\partial w_1}$$

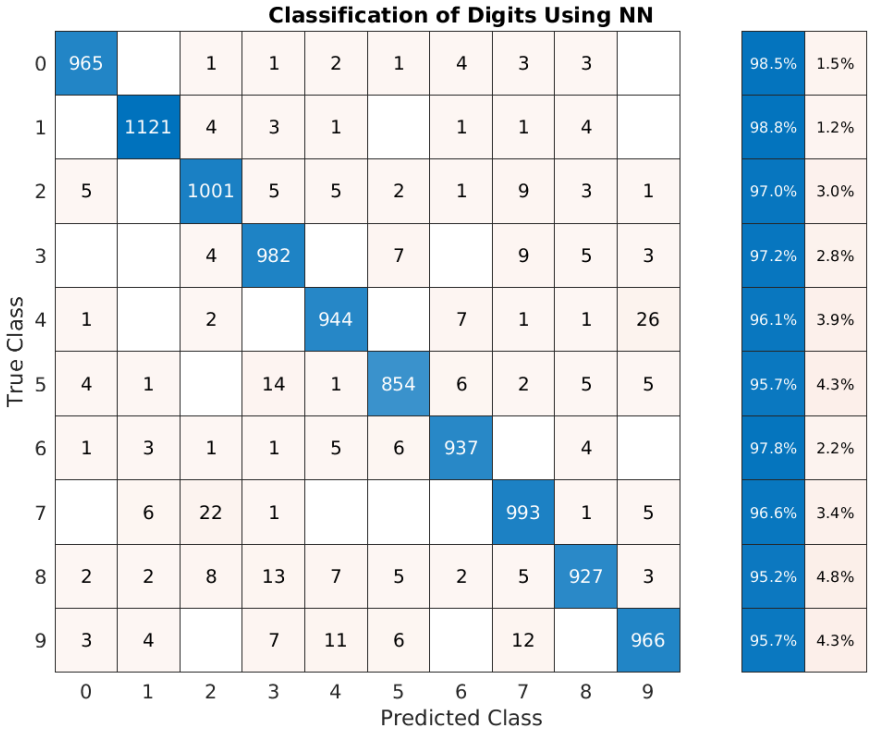
η is a stepsize, or learning rate.

To carry out gradient descent passing a single training observation at a time is called stochastic gradient descent. Alternatively, one can pass groups of observations at a time,

called batch gradient descent. Training the entire dataset a single time is called an epoch. Carrying out too few or too many epochs is a problem of underfitting or overfitting.

8.5 Results

A neural network was carried out using tensorflow/keras. A sequential neural network was carried out on the entire training set with two layers of size 64 with ReLu activation, and a third layer of ten nodes with a softmax activation. The model was compiled with an adam optimizer and categorical cross entropy loss. Gradient descent was carried out with a batch size of 32. Due to the random nature each model at initialization, results may vary. Below is the confusion matrix with an accuracy of 96.9% for one such model:



Programming Observations

The keras model expects vectors that can be readily multiplied with weights, as such, flattening your 28x28 matrix to an array of 1x784 vectors is ideal. Furthermore, normalizing the vectors to exist within [-.5,0.5] can improve performance. This is due to the nature of backpropagation. Limiting the range of values to exist within a similar and smaller range of values prevents the partial derivative gradients from bounding out of control. Keras expects categorical training targets to relate to the number of values in our softmax layer

(10). If a single digit is outputted, then keras will not accept the classified digit. As such, a digit needs to be converted from a single to a vector of size 10 (e.g. 3 = [0,0,0,1,0,0,0,0,0,0]). The to_categorical function is one utility method that was used to do such a conversion.

9 Convolutional Neural Networks

9.1 Motivations for CNN

Convolutional neural networks are a class of neural networks that introduce additional methods for processing the data, and is particularly effective for image processing. The essential idea is to construct filters that perform calculations on areas sweeping through an image. One such method is convolution, naturally.

There are quite a few motivations for using a C-NN over a general neural network, and here are a few:

Large Images: Consider a larger image. 224x224 in size, with 3 RGB. Thus, we would have 224x224x3 features, or roughly 150,000 features. If we consider a weight for each feature, this is extraordinarily expensive. However, the process of convolution allows a network to consider many less features in regards to a filter, which hopefully detects something like lines, which can be considered a pixel feature in relation to neighboring pixel features.

Invariances: Convolution better handles edge shifting, that is, if we slightly transform an image, it still retains its essential identification to a degree.

The following introductory processes are articulated with greater detail in Victor Zhou's Introduction to C-NNs.

9.2 Feedforward

Feedforward works similarly to prior, in which information from prior layers is fed forward to new layers. However, one should take special consideration as to the dimensions that are expected as information is passed through each layer. With C-NNs, the size of the input may change throughout, and whether the information is flattened or not can hinder results.

9.2.1 Convolution

Convolution takes an element wise multiplication within a nxn domain, then sums those values. Padding the edges will retain the size of the input to be the same after convolution. Non-padding is called valid convolution.

Convolution for a discrete nxn image with a mxm convolution filter may be carried out as follows:

1. Superimpose the mxm filter on top of the nxn image.

2. Perform element-wise multiplication
3. Sum the products and store in an output matrix
4. Repeat for other available locations in your nxn region

An example of such a convolution can be seen as follows. The blue region is being multiplied by a 3x3 convolution filter (not shown) to output at a corresponding location in an output matrix.

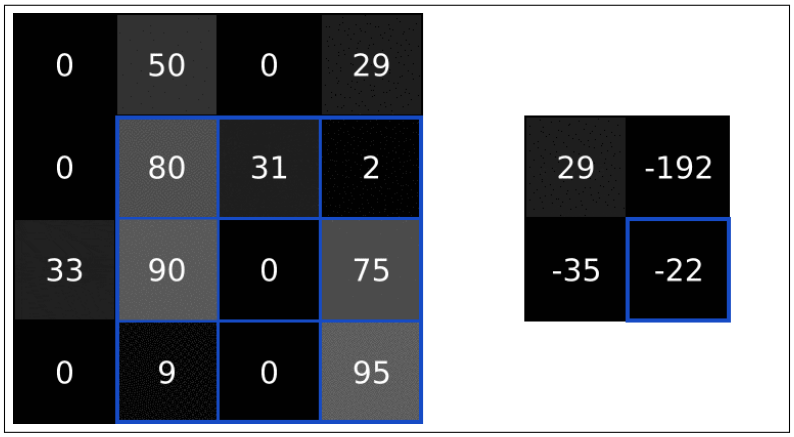


Figure 5: Example Convolution (Victor Zhou)

Convolution, and filtering in this manner, can be used for edge detection. Below, for example, a Sobel filter is used to determine edges.

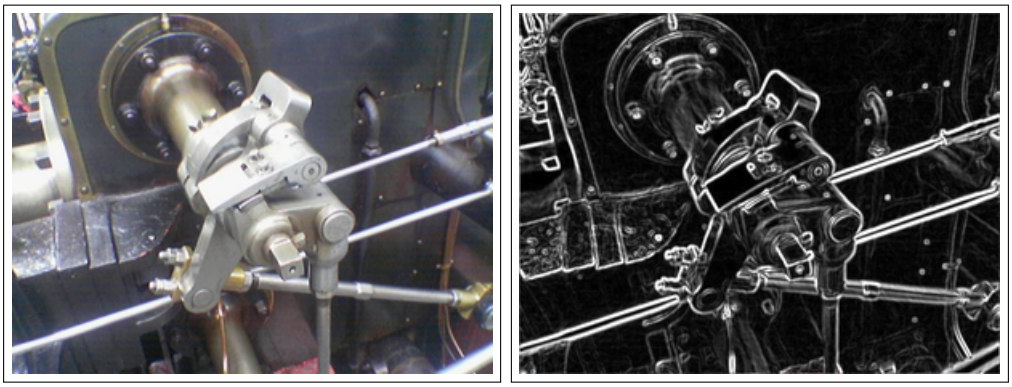


Figure 6: Example Sobel Filter (Wikipedia)

9.2.2 Maxpool

Because much of the information contained in a convolutional layer's output is redundant, a maxpool provides a way of pooling information together. Consider if an image contains an edge of sorts. If we consider many pixels along that edge which a convolutional layer detected, it could be suggested adjacent pixels aren't detecting anything especially new, but rather the same edge many times. As such, pooling combines this information using some operation like a max, min, or average. A max pool naturally uses the max.

Maxpooling a nxn image may be carried out as follows:

1. Choose a mxm maxpool grid on the edge of the nxn image matrix
2. Find the max value in the mxm region and store in output matrix
3. Repeat for other available locations (without overlap) in your nxn region

9.2.3 Softmax

The output of the feedforward thus far may output values that are somewhat arbitrary. Even with a final 10 neuron layer, one for each digit, the output will be varying values. We could perhaps pick the greatest value, but such a number does not give an associated probability, simply a max among the other numbers. It would be preferable to have a probability distribution for the 10 digits, and select the greatest probability instead. Softmax converts a list of values into a probability distribution that, of course, sums to one.

Softmax Formula:

$$\frac{e^{x_i}}{\sum_{i=1}^{i=n} e^{x_i}} \text{ for } x_1 ... x_n$$

9.2.4 Cross-Entropy Loss

Cross-entropy loss is well designed to measure the loss in relation to a probability distribution, and makes it valuable for the classification problem.

Cross-Entropy Loss formula:

$$L = -ln(p_c)$$

An example with a probability p for a digit class c:

$$p_c = 0.7, L = -\ln 0.7 = 0.356$$

Such a loss is not the sum of the loss along the probability distribution. Recall that loss is a measure of how good or bad a prediction is. Say you have a training digit and label. Then you would hope the probability for it being the right label is close to 1. Knowing the supervising label, you may simply calculate the loss for the probability at its respective digit class p_c. For example, if you know the correct digit label, 7, then you would want your probability distribution to have a high probability for 7, and low for other values. So you may simply calculate the loss for $p_c = p_7$.

9.3 Dropout

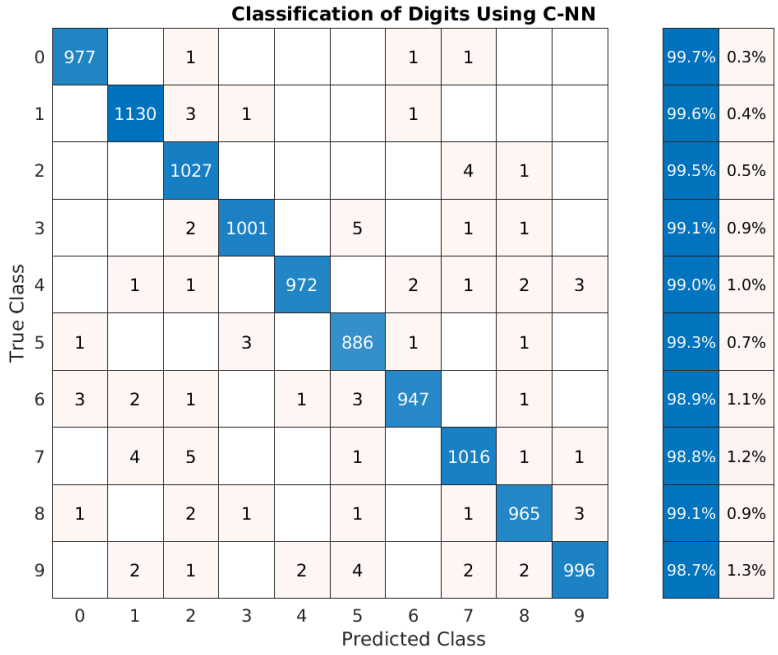
Dropout is randomly randomly ignores k neurons and their connections within a layer. This is useful in training to prevent overfitting. Furthermore, it prevents the co-adaptation of neurons along with one another.

9.4 Results

Three convolutional neural networks were produced. The first coded each class and its backpropagation in relation to Victor Zhou's Introduction to C-NNs. The C-NN involved a 3x3 convolution layer with 8 filters (randomized at first initialization), a 2x2 maxpool layer, and a softmax layer with 10 neurons (for each digit class). This produced an accuracy of 83.40% with three epochs. Such a technique only considered the first 1,000 training and test observations.

The same model layers using an adam optimizer were carried out to 3 epochs with tensorflow/keras with the entire training and data set to an accuracy of 96.82%.

Lastly, following the keras documentation, an improved model with many additional layers ran to 15 epochs produced an accuracy of 99.22%. The model contained a 3x3 Convolution followed by a 2x2 maxpool layer, twice, with a 0.5 (50%) dropout, and finally a softmax activation layer. The resulting confusion matrix can be seen below:



10 Ensemble Techniques

Metamodels make 'meta' considerations about the nature, structure, rules, constraints, applications, and so on, about the models in question, and provide a syntax for the usage of such models. Much like meta data refers to the description of data in question, meta models describe the model or models in question. Metamodelling can most simply be understood as the modelling of models.

One technique in meta modelling is the usage of an ensemble. A musical ensemble is a collection of musicians that produce music through the collection of the independent members. There are several types of ensembles, and in a particular ensemble, particular members may contribute to the piece as a whole in similar or very different ways. Similarly, an ensemble in machine learning is a particular collection of models with members in it in a structure that contribute to the task as a whole.

There are several considerations to be made from a meta-modelling approach to learning. The following section only covers three ensemble ideas that were attempted on the models as a brief introduction to meta-modelling. The first ensemble considers several models with equal weights working at equal levels. The second ensemble considers several models with unequal weights working at equal levels. The last model ensemble is a stacked ensemble, in which a Logistic regression model is stacked on top of a lower layer of models.

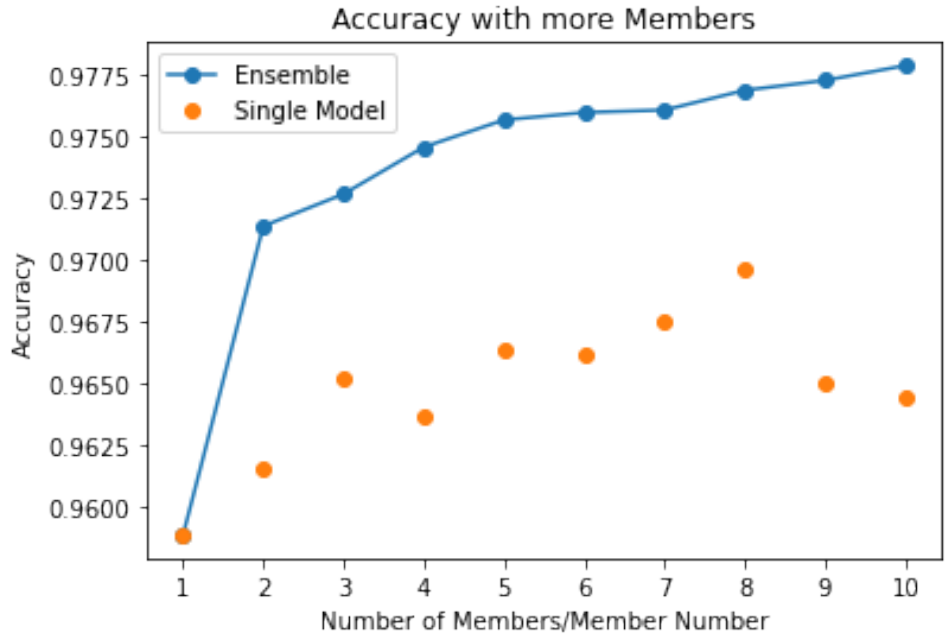
10.1 Uniform Average Ensemble

The term "Uniform Average Ensemble" is not a known term, but simply what I chose to call the procedure for the following metamodel. The idea is very simple. Consider m models. For each model, it produces a probability distribution in regards to a single test digit classification. If the probability distributions for m models were summed, then divided by the total number of models (m), this would produce a new probability distribution based on the 'average' of the models. Dividing is not actually necessary for the collection of the models to produce a collective distribution, but simply to normalize the new ensemble probability distribution to sum to 1 rather than to m. You may assign a weight ω_1 to ω_m to each of the models, which multiplied by their respective probability distribution, contribute more or less to the ensemble prediction. However, if the weights are exactly equal for all of the models ('uniform' distribution of weights), then this is equivalent to having no weights at all.

10.1.1 Results

Each model was a different instance or realization of the same neural network infrastructure. Since the weights and bias are randomly initialized, neural networks may not converge to identical models. Each model resembled the neural network produced in the neural network section (8.5 results). 10 models were used to produce an equal weight ensemble.

The following line chart is a visualization demonstrating the effectiveness of a single model versus an ensemble of models. The orange dots represent the accuracy for models m_1,...m_k,...,m_{10} working independently. The blue line represents the accuracy of the models ensembled together considering the first k models (arbitrarily ordered). It can be observed that an ensemble of models is always better than an independent model within the context of the model and data sets. Furthermore, more models seem better than less models generally speaking.



10.2 Weighted Average Ensemble

The idea for the weighted average ensemble is identical to the 'uniform' average ensemble, except the weights that are multiplied to each model are no longer equal. Yet what models should get more weight, and what models should get less weight, and by how much? Such a question can be formulated into an optimization question. Consider models m_1 through m_k and weights ω_1 through ω_k. For each test digit, the sum of the probability distributions for each model and weight are as follow:

$$\omega_1 m_1 + \omega_2 m_2 + ... + \omega_k m_k = m_E$$

It could be said ensemble m_E is a product of the models with associated weights, or the probability distrubtion similarly is similarly the sum of the probability distributions of

each model multiplied by weights. We would hope we get a better accuracy with a new distribution of weights. A simple loss function can directly relate to accuracy as follows:

$$\text{Loss} := 1 - \text{accuracy}(m_e)$$

Loss, in such a manner, is acting directly as error. With the prior formulation of the ensemble, with the given loss, one can treat the minimization of such loss with the linear combination of the weight parameters (to be optimized) and respective models.

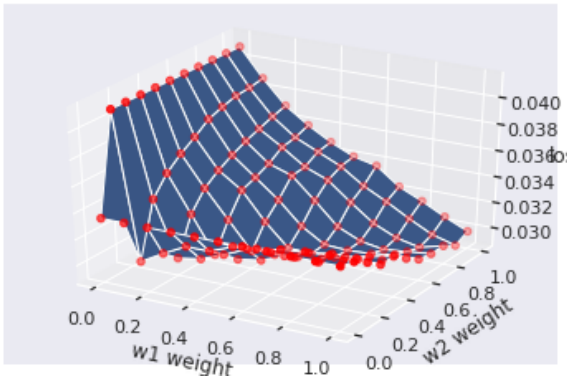
An advantage of such is that any change in the loss directly relates to an improvement in accuracy. Furthermore, it seems to be a very direct manner of defining the problem statement. However, there seem to be some disadvantages to the construction of such an optimization problem. In the case where models perform similarly well (such as is the case for different stochastic instances of the same model), the loss manifold seems to degenerate to a nearly flat plane. As such, finding improvement along that plane can be difficult at a global level. Determining a local optimum is entirely possible given an adequate starting point.

Alternatively, instead of constructing weights in relation to models, weights may be associated to a digit for a model. This would be useful if one algorithm were better at detecting certain classes (in this case digits) than others. However, this was not carried out within this paper.

10.2.1 Results

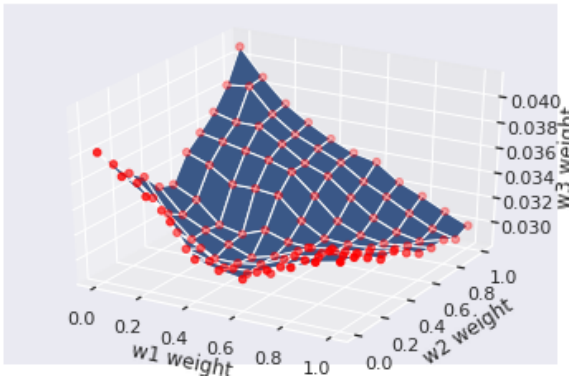
Given a ensemble of 3 nerual network models (constructed identical to that described in the neural network results section), a global optimum was attempted with scikit's differential_evolution function. No global optimum was able to be produced, and ran until runtime was terminated. SciPy's minimize function was ran with Nelder-Mead optimization, and did not always iterate to a local minimum. The first attempt assumed equal weights as a starting weight distribution, and did not converge.

The following visualization is a loss manifold with discretized weights ω_1 and ω_2 for their respective models m_1 and m_2. In this instance, the weight for the third model in the ensemble w_3, was fixed, in this case, to zero. The loss range of the weight tensor exists approximately within 0.03 to 0.04. What this indicates is that even the greatest improvement in the distribution of the weights will only increase the accuracy by about 0.01, or 1% from the worst to the best.

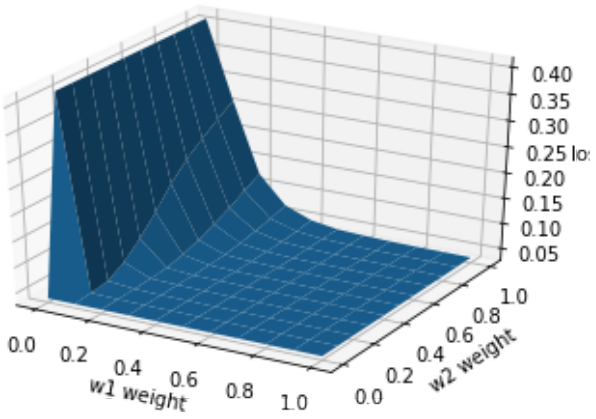


Finding the minimum point along the manifold, and using a local minimum optimizer using a point near that minimum (an exact region of stability was not determined) will achieve convergence. However, in our model this only increased accuracy as small as .01% from the equal weights. It should also be noted that the manifold includes weights outside our viable domain, since weights must follow a unit vector.

A similar visualization was produced for weights ω_1 and ω_2. In this case, however, the weight for model 3 was not fixed. Rather, was equal to whatever was not caught by the other weights: $\omega_3 = 1 - (\omega_1 + \omega_2)$. This, however, was an error in calculation. The sum of the weights shouldn't directly equal one, but rather, the magnitude of the weights (norm) should sum to one. Nonetheless, it can be observed that the manifold seems to still exist mostly within the same range for loss including additional similar models (here m_3).



Lastly, two models were compared in which one model was the standard neural network used prior, versus another of the same neural network architecture with only 200 training observations fed into it. This is intended to observe the manifold produced by a good model versus a poor model. The poor model produced a validation accuracy of about 58%, versus the good model's 96%. The following is the loss manifold for such models.



It can immediately be observed that the change in loss is most affected by the heaviness of the weight for the better model, that being w_2. Considering the weight along the w_1 direction, for a particular w_2 weight, having greater w_1 weight leads to an increase in loss until the manifold flattens out when a sufficiently large weight for the better model can overtake prediction results. This is exactly as expected, since this shows (1) that more weight given to the better model decreases loss, and (2) having more weight given to the poor model increases loss when w_2 is not great enough. Lastly, this shows that the plane is still mostly flat even when models are drastically different.

10.3 Stacked Ensemble

Rather than considering every model at an equal level, we could construct many models, then pass those models through another model. In this manner, the models are stacked on top of one another. At one level, the models take in input data, and learn to make predictions given the data. Stacked on top of that, another layer, takes in output from the first layer and learns to make predictions given that output. This stacked layer is called a 'meta-learner'. There are some concerns regarding the proper use of a stacked ensemble. Some of these concerns, such as overfitting, are discussed in the following article by Jason Browlee named Stacking Ensemble for Deep Learning Neural Networks in Python.

10.4 Results

The prior ensemble techniques were done again so that each model are the same, and each ensemble keeps the same members. Five members were chosen for the ensemble. The accuracy for each independent model were 96.17%, 96.81%, 96.64%, 96.79%, and 96.49%. The uniform weighted average model produced an accuracy of 97.62 , and the optimized with by the local minimum produced an accuracy of 97.63%. The ensemble for both methods, thus, were greater than their independent members. A ensemble model using logistic regression as the stacked model produced an accuracy of 98.0%. This showed another improvement in accuracy. Due to the random nature of models, values may vary when tests are reproduced. However, using an independent model should generally be less accurate than an ensemble method, and a stacked ensemble seemed to produce the greatest of results among the tested ensemble methods.

11 Results Summary

All algorithms were tested on the US Postal database. K-Means, SVD, and Tangent Distance were not tested on the MNIST database. For more information on the results of a particular algorithm, such as confusion matrix results, see its independent results section. Note that the confusion matrices for K-NN, NN, and C-NN correspond with MNIST tests, however.

The following table shows algorithm performance as measured by accuracy for some test cases.

Table 1 - Algorithm Accuracy (%)

	US Postal Accuracy	MNIST Accuracy
K-Means	66.62	
SVD	93.72	
Tangent Dist (old)	88.25	
Tangent Dist (new)	95.95	
K-NN	93.97	96.83
NN	93.57	96.9
CNN-1		96.82
CNN-2	96.26	99.22

The US Postal Data is known to be a more difficult dataset, according to the Stanford site which hosts the data. K-Means is an unsupervised algorithm. Naturally it is expected to produce a worse error rate than the supervised algorithms, even when performing optimally. Note that these are a particular result from a sample of possible model results. Neural Networks initialize their weights and biases randomly, and then train through back-propagation and gradient descent, and thus results may vary.

The K-Means algorithm experienced a large error because of the pigeon hole principle. When choosing k to be 10, we expect 10 different classes which resemble each digit 0 through 9. However, because the data is unsupervised, the centroids may converge to produce copies of the same digit. As such, all classifications for the digits not included are immediately incorrect. A solution for this is to produce a single centroid for each grouped training class, but then the algorithm is no longer unsupervised. This is also done better through the SVD method anyhow. It is also an interesting idea to group by each digit class and perform a k means on each group, with k greater than one (but equal for each class), constructing several different versions of the same class, then run K-NN against these clusters to pick the majority winner. In this way, we hope that we produce several versions of the same digit to test against, similar to an idea used in tangent distance.

The SVD algorithm was fairly accurate considering how fast and simple it was. Since it is not necessary to perform a full SVD; one may instead choose k singular values, which saves more computation time.

The Tangent Distance algorithm was first completed by manually computing each invarience pixel by pixel. While in theory this should produce approximately the same results, I did not use the same amount of invariences because of computation time. This method is incredibly slow, as for each training digit, several invariances must be made, each by accessing every pixel several times. Furthermore, at this point, the images were not blurred, which is important for the Tangent Distance algorithm. Another improved Tangent Distance model with Gaussian blurring and finite derivative approximations was produced and tested with an accuracy of 95.95%. Both methods used 2000 test digits instead of the full 2007.

Running K-NN on the entire MNIST test set against all 60,000 training digits produced an accuracy of 96.83%. This should always be the case given the same parameters because there is at base no stochastic element inherent to the algorithm. This algorithm was extraordinarily slow. Since the entire training and testing set was used, it was necessary to determine 60,000*10,000=6×10^8 minimum k distances. If sorting is used to determine the minimum k, then the algorithm will need to undergo 6×10^8 sorts likewise.

A neural network was carried out using tensorflow/keras for the MNIST dataset. A sequential neural network was carried out on the entire training set with two layers of size 64 with ReLu activation, and a third layer of ten nodes with a softmax activation. The model was compiled with an adam optimizer and categorical cross entropy loss. Gradient descent was carried out with a batch size of 32. Due to the random nature each model at initialization, results may vary. One model produced an accuracy of 96.9%.

Another neural network was constructed for the US Postal dataset. It has the same architecture, except it ran with a smaller batch size (8). This produced an accuracy of 93.57%, but again results may vary.

Three C-NNs were produced. The first was coded entirely without an API following along Victor Zhou's methodology. This produced an accuracy of 83.40, but was only trained with 1000 training and test observations. The same C-NN architecture, that being a 3x3 convolution layer with 8 filters (randomized at first Xavier initialization), a 2x2 maxpool layer, and a softmax layer with 10 neurons (for each digit class), was recreated with keras using the entire training and data set to an accuracy of 96.82%. Lastly, following the keras documentation, an improved model with many additional layers ran to 15 epochs produced an accuracy of 99.22%. The model contained a 3x3 Convolution followed by a 2x2 maxpool layer, twice, after with a 0.5 (50%) dropout, and finally a softmax activation layer. This proved to be the best architecture and the best model among all the models. This architecture was used again for the US Postal data, except with stochastic gradient descent (batch size of 1) and another dropout layer.

When observing the results for the US Postal data, there is less of a clear 'winner'. CNNs indeed performed the best again, but not by as great a margin. Furthermore, Tangent Distance performed second best, even above Neural Networks and K-NN. SVD, K-NN, and the attempted Neural Network performed similarly well in regards to accuracy.

Table 2 - Ensemble Accuracy (%)

	Uniform Average	Weighted Average	Stacked
Accuracy	97.62	97.63	98.0

Five models were considered for ensembling, all different instances of the same trained neural network. The architecture for these networks were identical to what was carried out for the neural network discussed above. The accuracy for each independent model were 96.17%, 96.81%, 96.64%, 96.79%, and 96.49%. The uniform weighted average model produced an accuracy of 97.62 , and the optimized with by the local minimum produced an accuracy of 97.63%. The ensemble for both methods, thus, were greater than their independent members. A stacked model using logistic regression as the stacked model produced an accuracy of 98.0%. Using an independent model should generally be less accurate than an ensemble method, and a stacked ensemble seemed to produce the greatest of results among the tested ensemble methods. The optimal weights did not seem to improve accuracy significantly. This is suspected to occur because of the similarity in the models produces a loss manifold that degenerates to a nearly flat plane, and is discussed more in the ensemble section.

Loss was chosen simply as the error, and 1-accuracy, likewise. Weights for each model were multiplied by the entire probability distribution for a prediction by a model. This way, the ensemble acts simply as a linear combination of the weights and their models. When comparing accuracy per digit class, most models seemed to simply improve each digit, generally speaking. This can be seen by observing the marginal accuracy for each digit class in the confusion matrices for each model. This was not necessarily the case when comparing the neural network to the K-NN, however. Combining these models would provide a justification to weight based on the digit class rather than weight on the linear combination of the entire probability distribution for a respective model, itself.

12 Conclusions

Precise algorithm comparisons were not made. This was in part because of the stochastic random nature of some algorithms. However, it was still observable how each model approximately performed. Neural networks were fast and accurate for digit classification. SVD were also very fast and performed decently well. Tangent Distance and K-NN were exceptionally slow, but under right circumstances could perform nearly as well or just as well. CNNs outperformed all other models, and are very promising for image processing techniques. This paper barely scratched the surface of meta-modelling and ensemble techniques. These techniques improved the accuracy of the combined neural networks. 'Community engagement' requirements were met by commenting on Victor Zhou's blog.

13 Moving Forward

Moving forward I would start by filling in some gaps in my report. The report was intended to be more instructive (especially to myself), so proper and precise algorithm comparisons were not made. I would like to make more proper comparisons and tend to the random nature of some algorithms. This can easily be done by constructing a confidence interval to more clearly demonstrate performance. Some sort of multi-class ROC would additionally help in the comparison task. I would also like to add a Bayesian approach. Computational complexity was not calculated for each algorithm, but I'm sufficiently satisfied with a general idea their respective efficiency for now.

While the tasks in a neural network infrastructure seem clear, it's not distinctly clear how to develop the most appropriate network structure for the general case. For example, how many dense layers are appropriate, how much dropout, what size filters should be used, and so on, are all useful questions that remain unanswered within the scope of what has been produced here. I would like to investigate the effects of adjusting these networks, to demonstrate why a particular network is favourable. I suspect this can easily be done simply by trying many similar models adjusting a single component, and graphing their accuracy results superimposed on top of one another.

Most importantly, to increase best performance, C-NNs seemed the most promising. I would focus more attention to peer review of the current best C-NNs, and see what higher level techniques can be reproduced to attain the current best algorithm performance.

14 Citations

[1] Patrice Y. Simard, Yann A. Le Cun, John S. Denker, and Bernard Victorri. Transformation invariance in pattern recognition – tangent distance and tangent propagation. 2000.

[2] José Israel Pacheco.A comparative Study for the Handrwritten Digit Recognition Problem - Tangent Distance. California State University, May 2011, pp 17-20.

[3] Berkant Savas.Analysis and Tests of Handwritten Digit Recognition Algorithms. Linköping, 2003.

[4] Lars Eldén, Matrix Methods in Data Mining and Pattern Recognition - Classification of Handwritten Digits. SIAM, Society for Industrial and Applied Mathematics, 2019, pp. 113–128.

[5] Ian Goodfellow, Yoshua Bengio, Aaron Courville. Deep Learning (2016). MIT Press.

[6] Marco R. Sousa, Digit Classification (2021), Github, https://github.com/MarcoRSousa/DigitClassification

[7] Standford University, US Post Office Zip Code Data, https://web.stanford.edu/~hastie/StatLearnSparsity_files/DATA/zipcode.html

[8] Yann A. Le Cun, Corinna Cortes, Christopher J.C. Burges. The MNIST Database of handwritten digits, http://yann.lecun.com/exdb/mnist/

[9] Victor Zhou. July 24, 2019, An Introduction to Neural Networks, https://victorzhou.com/blog/intro-to-neural-networks/

[10] Victor Zhou. November 10, 2019, An Introduction to Convolutional Neural Networks, https://victorzhou.com/blog/intro-to-cnns-part-1/

[11] Jason Brownlee. August 28,2020, Stacking Ensemble for Deep Learning Neural Networks in Python, https://machinelearningmastery.com/stacking-ensemble-for-deep-learning-neural-networks/

[12] François Chollet. April 21, 2020, Keras Documentation, Code Examples, Simple MNIST covnet, https://keras.io/examples/vision/mnist_convnet/