Digit Classification

Multi-Class Perceptron Classifier

David Young

November 2015

Contents

1 Introduction **2**
- 1.1 Submission Overview . . . 2
- 1.2 Problem Definition. . . . 2

2 Background **3**
- 2.1 Neural Tissue: . . . 3
- 2.2 Perceptrons . . . 3
- 2.3 Modeling a Perceptron . . . 4
- 2.4 Supervised Learning . . . 4

3 Overview of Source **5**

4 Implementation **6**
- 4.1 Representing Digits . . . 6
- 4.2 Reading the Data From Files . . . 6
- 4.3 Representing a Perceptron . . . 6
- 4.4 Running the Perceptron . . . 8
- 4.5 Training the Perceptron . . . 9
- 4.6 Supervised Learning . . . 10
- 4.7 Classifying Test Data . . . 12
- 4.8 Acquiring the Confusion Matrix . . . 13
- 4.9 Tying it All Together . . . 14

5 Results **15**
- 5.1 Baseline Test . . . 15
- 5.2 Ordering of Training Examples (fixed vs. random) . . . 16
- 5.3 Varying Epoch, Training Curves & Overfitting . . . 17
- 5.4 Initializing Weights (Random vs. Zero) . . . 19
- 5.5 Inclusion of Bias . . . 20

6 Analysis & Discussion **21**

7 Potential Improvements **22**

1 Introduction

1.1 Submission Overview

This writeup summarizes the procedure and results of test digit classification using a multi-class perceptron. It also contains discussion of said results and attempts to provide some insight and reflection on the behavior of implemented algorithms. This report was produced for course "CS-440: Artificial Intelligence" at University of Illinois Urbana Champaign.

1.2 Problem Definition.

Apply the multi-class (non-differentiable) perceptron learning rule to classify digits.

- **Features**: The basic feature set consists of a single binary indicator feature for each pixel. Specifically, the feature $F_{i,j}$ indicates the status of the (i,j)th pixel. Its value is 1 if the pixel is foreground (no need to distinguish between the two different foreground values), and 0 if it is background. The images in the training set are of size 28*28, so there are 784 features in total.
- **Training**: The multi-class perceptron will need to learn a weight vector for each digit class, based on training digits from the training data set.
- **Testing**: The classification performance of the trained perceptron will be tested using a separate reserved testing data set that has not yet been seen by the perceptron.
- **Evaluation**: The true class labels of the test images from a testlabels file are used to check the correctness of the estimated label for each test digit. Performance is reported in terms of the classification rate for each digit (percentage of all test images of a given digit correctly classified). Also reported is a confusion matrix. This is a 10x10 matrix whose entry in row r and column c is the percentage of test images from class r that are classified as class c. Additionally, a training curve is generated to demonstrate the accuracy on the training set as a function of the epoch (i.e., complete pass through the training data).

2 Background

2.1 Neural Tissue:

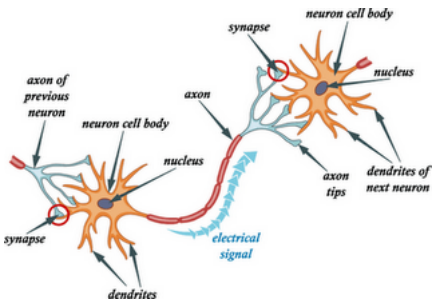

Figure 1: Simplified Neuron

Human neural tissue can manage extremely complex computation and function. The physiology gets rather complicated but at the most basic level, neural tissue is a biological circuit which manifests as a network of interconnected neurons. To oversimplify things, neurons can be thought of as binary junctions working with electrochemical signals. If signal are received in such a way to trigger the neuron to fire, the neuron will propagate the signal to any other neurons it innervates. A human brain has tens if not hundreds of billions of neurons that are interconnected in multi input and multi output networks. Because all neurons operate simultaneously, processing with this network is carried out in a massively parallel way as opposed to computer processors which for the most part process in a procedural and serial fashion. Basic neural communication (signaling) involves complex chemical gradients, diffusive and electrical forces, and chemical messengers which are all affected by a seemingly infinite number of factors, proteins, second messengers etc. The end result is an extremely large state space of possible conditions. The control logic that would be required to oversee such function is too complicated to generate and too large to store. There is very little programming preloaded in the human brain at birth. Instead, most of our neurological function is adaptive and learned over time in place (ie: there is no control center, the neurons themselves adapt/learn their function amidst everything else). Every time neurons fire or do not fire, they are reinforcing certain responses to given stimuli. Over time, complicated pathways in the brain form like circuits that accomplish different computations. The interconnected nature of all this reinforcement is so complicated that we have yet to understand how the brain accomplishes a lot of cognitive function. However, we do have a decent grasp of how the base unit (single neuron) operates.

2.2 Perceptrons

We are long away from developing an artificial human brain, but we have managed to accomplish function from artificial models of neural networks inspired from their much more complex human counterparts. Artificial networks of interconnected neurons are the topic of bleeding edge computer science research today. But a computational model of a single neuron, arguably the simplest neural network, is very easy to model and train in code. Such an implementation is called a "perceptron". Typical applications for neural networks involve tasks that are difficult for computers but easy for humans. These tasks are often based around pattern recognition of various sorts. While the variety of applications is limited, perceptrons can yield real-world usable results on appropriate tasks such as simple feature based classification.

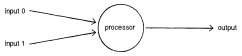

Figure 2: Simplest Perceptron

The adaptive nature of artificial neural networks must be accomplished via some change in the internal structure of the neurons. Most commonly this means adjusting some kind of weight given to connections between neurons (input and output paths). Thinking of a perceptron a simple neural network composed of only a single neuron, the only paths to consider are the inputs and outputs.

2.3 Modeling a Perceptron

A perceptron model requires a way to input information, process that information and yield an output. The simplest model can be broken into a 4 step process:

1. receive inputs
2. weigh those inputs given an adaptive set of weights
3. sum the weighted inputs
4. pass the sum through an activation function to produce an output

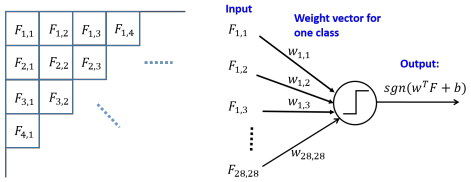

Figure 3: Features as input to perceptron

2.4 Supervised Learning

Several strategies for learning exist (supervised, unsupervised, reinforcement learning etc), but this report will deal with supervised learning. Supervised learning is a training strategy involving a supervisor that is "more intelligent" than the entity being trained. In supervised learning, the supervisor provides example problems to which it knows the answers. During training the learning entity will generate responses to these example problems and the supervisor will indicate whether or not the response was correct. The learning entity may then adjust its internal state as a reflection of its confirmed success or failure. Such a process aims to adapt the learning entity such that it produces successful responses more frequently.

3 Overview of Source

Obtaining the source code
The entirety of the code written for this project can be found at the following repository under the branch "PerceptronClassifier". A direct link to the branch is also provided.

https://github.com/dcyoung/DigitClassification

https://github.com/dcyoung/DigitClassification/tree/PerceptronClassifier

Summary of source code
The following source files were written from scratch. All code is well commented with Javadocs; it should be no burden to browse for specific details.

Filename	**Description**
AccuracyStates.java	Builds confusion matrix and other statistics.
Digit.java	Holds the pixel and numeral data for a single digit.
FileReader.java	Reads digits from properly formatted .txt files.
MultiClassPerceptron.java	A peceptron that can operate on inputs to produce a classification estimate considering multiple classes.
OrganizedDataSet.java	Organizes the training data into groups by class.
PerceptronTrainer.java	Individual trainer used to conduct a single data training on a perceptron.
TestRunner.java	Provides a demo-able program.
TrainingManager.java	Creates and manages many PerceptronTrainers to effectively train a perceptron on a training dataset.

4 Implementation

4.1 Representing Digits

A Digit class represents each digit from the files, and consists of that digit's true value and it's image data (a 2D array of 1's and 0's).

```
public class Digit {
    //actual true integer value of the digit between 0 and 9
    private int trueValue;
    //28x28 array of values for the pixels of this digit
    private int[][] pixelData;

    /**
     * Constructor
     * @param pixelData - array of pixel values
     * (pix = 0 if background, and 1 if part of the digit)
     * @param trueValue - true integer digit value between 0->9
     */
    public Digit(int[][] pixelData, int trueValue){
        this.pixelData = deepCopy2dArray(pixelData);
        this.trueValue = trueValue;
    }

    ...

}
```

4.2 Reading the Data From Files

This is done through a function that takes as input a label and image data filename and returns a list of Digit objects. Using Java's file scanner, for every digit, It retrieves the true digit value from the label file, and the data from the image data file. It uses that data to build an object of Digit class, and add that to the list which it returns.

4.3 Representing a Perceptron

The actual model of a multi-class perceptron is extremely simple and requires very little state information. Later methods will have to be written to operate the perceptron, but the state of the perceptron at any time can be represented by:

- A list of weight vectors, one for each class (in this case 10 different digit classes). Each weight vector represents the weights for all inputs. Here there are 28x28 pixel value features that will be used as input.

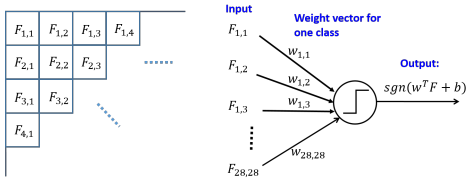

Figure 4: Weight vector for an array of pixel features as inputs. 1 vector will be needed for each digit class.

- A learning rate.

The constructor is written to support any number of classes with any number of inputs, but here it will only be tested on digit classification (10 classes each with 28x28 inputs). The constructor also initializes the weight vectors to either all 0's or random values. The comparison between these settings will be shown later.

```
/**
 * MultiClassPerceptron:
 * A peceptron that can operate on inputs to produce a classification
 * estimate considering multiple classes.
 * @author dcyoung
 */
public class MultiClassPerceptron {

    //weight vector for each class
    private ArrayList<double[]> weights;
    //learningRate
    private double alpha;
    private Random randomGenerator = new Random();

    /**
     * Constructor
     * @param numClasses - # of classes considered for classification,ie:10 digits
     * @param numInputs - number of inputs to this peceptron
     * @param bInitRandWeights - true if the perceptron should be initialized
     * with random weights, false if the weights should be initialized to 0
     * @param learningEpoch
     */
    public MultiClassPerceptron(int numClasses, int numInputs, boolean bInitRandWeights,
        int epoch){

        //create weight vectors for each class (initialize all weights to 0)
        this.weights = new ArrayList<double[]>();
        for(int i = 0; i < numClasses; i++){
            double[] weightVec = new double[numInputs];
            this.weights.add(weightVec);
        }

        if(bInitRandWeights){
            //initialize all the weights to random values between -1:1
            for(int c = 0; c < numClasses; c++){
                for(int i = 0; i < numInputs; i++){
                    weights.get(c)[i] = randomGenerator.nextDouble()-1;
                }
            }
        }

        //calculate the learning rate (alpha)
        this.alpha = 1000.0/(1000+epoch);
    }

    ...

}
```

4.4 Running the Perceptron

For each class considered in classification (digits 0:9 for example)...

1. Multiply each input by its corresponding weight from the current class' weight vector.
2. Sum all of the weighted inputs.
3. Generate an output (best class guess) based on an activation function applied to the weighted inputs. In this case the activation function is merely selecting the class with the max sum of weighted inputs.

```
/**
 * feeds the input through the perceptron
 * @param inputs: any features representable by a double array
    (digit pixel values for example)
 * @return the perceptron's best guess at a classification (digit class)
 */
public int feedForward(double[] inputs){

    //create a container to store the calculated sum for each class
    ArrayList<Double> sumsByClass = new ArrayList<Double>();

    for(int c = 0; c < this.weights.size(); c++ ){
        double sum = 0;
        for(int i = 0; i < this.weights.get(c).length; i++){
            sum += inputs[i]*weights.get(c)[i];
        }
        sumsByClass.add(sum);
    }

    int bestClassGuess = sumsByClass.indexOf(Collections.max(sumsByClass));
    return bestClassGuess;
}
```

4.5 Training the Perceptron

Remember, the function of real neural tissue is adapted in place through reinforcement of responses to various stimuli. Similarly, the adaptive nature of artificial neural networks must be accomplished via some change in the internal structure of the neurons. Most commonly this means adjusting some kind of weight given to connections between neurons (input and output paths). For a perceptron, we need only be concerned with the weight vectors associated with the feature inputs.

Using this information to setup a training scheme for a perceptron is quite simple. The perceptron is provided inputs. Using its current set of weight vectors it evaluates the inputs in the "feedForward()" method to generate a classification guess. The perceptron is then informed whether or not the classification was accurate (this is what makes the learning supervised). If the guess was accurate, the perceptron knows the current weights worked well at classifying the input and leaves the weights alone. If the guess was inaccurate, the perceptron knows the current weights did not yield a good classification. In this case the the weight vectors must be updated. To increase learning speed beyond what would evolve from random perturbations of weight vectors, the weights are adjust intelligently.

In the event of an incorrect classification, only two weight vectors are updated: the vectors associated with the expected and incorrectly chosen classes. The adjustment to the two weight vectors is accomplished in the same way, but in opposite directions. Weights from the expected class vector are bolstered up (increased in magnitude) according to the inputs, while weights from the incorrect class vector are stripped down (decreased in magnitude) according to the inputs. This ensures that input values commonly present in a class are weighted heavily, while inputs not commonly present in a class are not weighted as heavily.

```
/**
 * Trains the perceptron by running the input through the perceptron, comparing
 * the perceptron's best classification to the true value and updating the weights
 * if the classification was inaccurate.
 * @param inputs: any features representable by a double array
 * @param expectedClass: the correct classification
 */
void train(double[] inputs, int expectedClass){
    //let the perceptron classify the given inputs
    int bestClassGuess = feedForward(inputs);
    //compare the classification to the expected value
    if(bestClassGuess == expectedClass){
        //do nothing
    }
    else{
        //if the classification was incorrect
        //update the weight vectors of the expected and incorrectly chosen classes
        updateWeightVec(expectedClass, inputs, this.alpha);
        updateWeightVec(bestClassGuess, inputs, -this.alpha);
    }
}
/**
 * updateWeightVec: updates a specified weight vector as part of training
 * @param targetClass: the class who's weight vec must be updated
 * @param inputs:
 * @param updateFactor: either positive or negative learning factor
 */
private void updateWeightVec(int targetClass, double[] inputs, double updateFactor) {
    for(int i = 0; i < this.weights.get(targetClass).length; i++){
        this.weights.get(targetClass)[i] += updateFactor*inputs[i];
    }
}
```

4.6 Supervised Learning

The perceptron "train" method allows the perceptron to adapt its weights according to its confirmed performance on a single piece of data (in this case a single training image). To conduct the entire process of training however, this adaptation process must be run many times given a lot more and varied data. To handle the entire training process from start to finish, two additional classes were created: "TrainingManager.java" and "PerceptronTrainer.java". The PerceptronTrainer class represents an individual trainer that can be used to conduct a single data training on a perceptron. A TrainingManager is used to create and manage many PerceptronTrainers.

The PerceptronTrainer class is shown below. Representing a single unit of training data, it creates a set of features to use as inputs to the MultiClassPerceptron.train() method.

```
/**
 * PerceptronTrainer:
 *  An individual trainer that can be used to conduct a single data
 *  training on a perceptron. A TrainingManager is used to create
 *  and manage many PerceptronTrainers.
 *
 *  (for this project, this could be replaced functinally by just using
 *  a Digit class, but this class was created to dedicate the functionality
 *  of a single train on a perceptron to a class)
 * @author dcyoung
 *
 */
public class PerceptronTrainer {
    private double[] inputs;
    private int trueValue;

    /**
     * Constructor:
     * @param features - 2D int array of features (pixel values)
     * @param trueValue - true value or answer for the features
     */
    public PerceptronTrainer(int[][] features, int trueValue){
        //populate the inputs array given the features
        int numFeatureRows = features.length;
        int numFeatureCols = features[0].length;

        //number of inputs will be 1 for each feature + 1 for the bias
        int numInputs = numFeatureRows*numFeatureCols +1;

        this.inputs = new double[numInputs];
        for(int row = 0; row < numFeatureRows; row++ ){
            for(int col = 0; col < numFeatureCols; col++ ){
                this.inputs[row*numFeatureCols + col] = features[row][col];
            }
        }
        this.inputs[numInputs-1] = 1;

        //note the trueValue which will be used during training
        this.trueValue = trueValue;
    }
    ...
}
```

The majority of the TrainingManager class is shown below. It takes a dataset and provides methods to train a given perceptron using the entire training dataset in different ways. These methods create PerceptronTrainers for every piece of data in the dataset and use these generated trainers to train the Perceptron itself.

```
/**
 * TrainingManager:
 *  Creates + manages many PerceptronTrainers to effectively train a perceptron
    on a dataset. Provides method to test performance of a trained peceptron.
 * @author dcyoung
 */
public class TrainingManager {
    //the organized training data
    private OrganizedDataSet trainingData;

    /**
     * Constructor
     * @param dataset
     */
    public TrainingManager(OrganizedDataSet dataset){
        this.trainingData = dataset;
    }

    /**
     * Trains a perceptron on all the data in the organized dataset, in the order
     * data was originally read in... before any organization/separation by group
     * @param perceptron
     */
    public void trainAllDataRandomly(MultiClassPerceptron perceptron){
        trainPerceptron(perceptron, this.trainingData.getAllDigits());
    }

    /**
     * Trains a perceptron on all the data in the organized dataset considering
     each class sequentially. Trains perceptron on every example from that class.
     * @param perceptron
     */
    public void trainAllClassesSequentially(MultiClassPerceptron perceptron){
        for(int digClass = 0; digClass < this.trainingData.getGroupedDigits().size();
            digClass++){
            trainPerceptron(perceptron, this.trainingData.getGroupedDigits().get(digClass)
                );
        }
    }

    /**
     * Train a perceptron on a specifiable list of training data.
     * @param perceptron
     * @param trainingData
     */
    public void trainPerceptron(MultiClassPerceptron perceptron, ArrayList<Digit>
        trainingData){
        PerceptronTrainer[] trainer = new PerceptronTrainer[trainingData.size()];
        for(int i = 0; i < trainer.length; i++){
            Digit trainingDigit = trainingData.get(i);
            int actualDigClass = trainingDigit.getTrueValue();
            trainer[i] = new PerceptronTrainer(trainingDigit.getPixelData(),
                actualDigClass);
        }

        for(int i = 0; i < trainer.length; i++){
            perceptron.train(trainer[i].getInputs(), trainer[i].getTrueValue());
        }
    }
}
```

4.7 Classifying Test Data

The following method was added to TrainingManager to generate statistics about the classification performance of a perceptron on a given test dataset.

```
/**
 * Test a perceptron on a set of test data.
 * @param perceptron
 * @param testData
 * @return accuracy statistics about the classification performance of the
   perceptron on the testing data.
 */
public AccuracyStats testTrainedPerceptron(MultiClassPerceptron perceptron,
    OrganizedDataSet testData){
    AccuracyStats stats = new AccuracyStats();

    for(int digClass = 0; digClass < testData.getGroupedDigits().size(); digClass++){
        for(int i = 0; i < testData.getGroupedDigits().get(digClass).size(); i++){
            Digit digit = testData.getGroupedDigits().get(digClass).get(i);

            int digClassGuess = perceptron.feedForward(digit.generateInputsWithBias());
            stats.addDatapoint(digClass, digClassGuess);
        }
    }
    return stats;
}
```

Assuming the training and testing dataset have already been read into organized datasets, the testing of the perceptron can then be accomplished in relatively few steps.

```
TrainingManager trainingManager = new TrainingManager(trainingDataset);
MultiClassPerceptron perceptron = new MultiClassPerceptron(10, (28*28+1), true, 1);
trainingManager.trainAllDataRandomly(perceptron);
AccuracyStats stats = trainingManager.testTrainedPerceptron(perceptron, testingDataset);
```

4.8 Acquiring the Confusion Matrix

An AccuracyStats class exists to build the confusion matrix. This class takes datapoints one at a time; where a datapoint consists of a Digit's actual class and the class predicted by the perceptron. A non-normalized confusion matrix, where each entry is the number of datapoints of a particular value, is updated for every new datapoint. For example, if a datapoint of actual = 9, classifiedAs = 10 comes through, row 9, column 10 in the non-normalized confusion matrix is incremented by 1.

To build the actual confusion matrix after all data has been processed involves normalizing the non-normalized confusion matrix by the sum of the value in each row.

addDatapoint can be seen here:

```
public void addDatapoint(int actual, int classifiedAs) {
    this.confusionMatrixNonNormalized[actual][classifiedAs]++;
}
```

Normalizing can be seen here:

```
public double[][] getConfusionMatrix() {
    // Normalize confusion matrix
    int[] rowTotals = new int[10];
    for (int i = 0; i < rowTotals.length; i++) {
        for (int j = 0; j < 10; j++) {
            rowTotals[i]+=confusionMatrixNonNormalized[i][j];
        }
    }

    double[][] confusionMatrix = new double[10][10];
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 10; j++) {
            confusionMatrix[i][j] = 1.0 * confusionMatrixNonNormalized[i][j]/rowTotals[i];
        }
    }

    return confusionMatrix;
}
```

It can be noted that the diagonal of this confusion matrix is the correct classification rates for each class.

4.9 Tying it All Together

All of these components are tied together in the TestRunner class. This class uses all of the components to create and train a perceptron on the training dataset, and then test the classification accuracy on the test dataset. The classification rates and confusion matrix are printed to the console.

```
public static void main(String[] args) {
    DecimalFormat df = new DecimalFormat("0.00");
    df.setMaximumFractionDigits(2);

    FileReader fr = new FileReader();
    String imgDataFilename = "digitdata/trainingimages";
    String labelFilename = "digitdata/traininglabels";
    ArrayList<Digit> allTrainingDigits = fr.readDigitData(imgDataFilename, labelFilename);

    imgDataFilename = "digitdata/testimages";
    labelFilename = "digitdata/testlabels";
    ArrayList<Digit> allTestingDigits = fr.readDigitData(imgDataFilename, labelFilename);

    OrganizedDataSet trainingDataset = new OrganizedDataSet(allTrainingDigits);
    OrganizedDataSet testingDataset = new OrganizedDataSet(allTestingDigits);

    TrainingManager trainingManager = new TrainingManager(trainingDataset);
    MultiClassPerceptron perceptron = new MultiClassPerceptron(10, (28*28+1), true, 1);
    trainingManager.trainAllDataRandomly(perceptron);
    AccuracyStats stats = trainingManager.testTrainedPerceptron(perceptron, testingDataset
        );

    System.out.println("Average Classification Rate:");
    System.out.println(stats.getAverageClassificationRate());
    System.out.println();
    System.out.println("Confusion Matrix:");
    stats.printConfusionMatrix();
}
```

5 Results

5.1 Baseline Test

Testing a perceptron can be done under various conditions and combinations of parameters. In this report, the following are considered for variation.

- Learning rate decay function
- Bias vs. no bias
- Initialization of weights (zeros vs. random)
- Number of epochs (i.e., complete pass through the training data)

Before examining the impact of varying these factors, its nice to have a baseline for performance. The following results were generated as baseline using the following conditions:

Learning Rate Decay Fxn	**Bias?**	**Weights Initialization**	**# of epochs**
$\frac{1000}{1000+epoch}$	true	random	1

Table 1: Baseline Performance Test Conditions

An epoch count of 1 means the perceptron will only see the training data once.

Classification Rates By Digit:
The average classification rate for the baseline perceptron setup was 0.83 (83%). The breakdown by digit is as follows.

Digit Class	0	1	2	3	4	5	6	7	8	9	Average
Accuracy	0.94	0.98	0.83	0.83	0.81	0.76	0.90	0.83	0.73	0.70	**0.83**

Table 2: Classification Accuracy by Digit Class for Baseline

Confusion Matrix
The baseline setup produces the following confusion matrix.

0	1	2	3	4	5	6	7	8	9
0.94	0.00	0.01	0.00	0.00	0.00	0.02	0.01	0.01	0.00
0.00	0.98	0.00	0.00	0.00	0.00	0.01	0.00	0.01	0.00
0.00	0.01	0.83	0.03	0.01	0.00	0.07	0.02	0.03	0.01
0.00	0.00	0.05	0.83	0.00	0.02	0.01	0.06	0.02	0.01
0.00	0.01	0.08	0.00	0.81	0.00	0.04	0.03	0.02	0.01
0.02	0.00	0.00	0.05	0.00	0.76	0.05	0.01	0.10	0.00
0.02	0.01	0.03	0.00	0.01	0.00	0.90	0.02	0.00	0.00
0.00	0.02	0.08	0.01	0.01	0.00	0.00	0.83	0.01	0.05
0.02	0.01	0.06	0.08	0.02	0.06	0.01	0.02	0.73	0.00
0.00	0.00	0.04	0.03	0.03	0.01	0.00	0.18	0.01	0.70

5.2 Ordering of Training Examples (fixed vs. random)

The previous results demonstrated the performance after a random ordering of training examples. The average classification accuracy when ordering training examples sequentially (ie: by running through each class of grouped digits and training first all 0's and then all the 1's etc..) is terribly low. The classifier ends up being coached to detect only the last digit class it was trained on, effectively overwriting the effects/reinforcement of previously trained digits. The following code was used to test the performance of a baseline peceptron by applying the training in a sequential order.

```
/**
 * Trains a perceptron on all the data in the organized dataset by going through
 * each class sequentially and training the perceptron on every example from that class.
 * @param perceptron
 */
public void trainAllClassesSequentially(MultiClassPerceptron perceptron){
    for(int digClass = 0; digClass < this.trainingData.getGroupedDigits().size(); digClass
        ++){
        trainPerceptron(perceptron, this.trainingData.getGroupedDigits().get(digClass));
    }
}
```

Such a training scheme produces the following unsatisfactory results with an average of 10%.

Digit Class	0	1	2	3	4	5	6	7	8	9	Average
Accuracy	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	**0.10**

Table 3: Classification Accuracy by Digit Class for Sequential Training

5.3 Varying Epoch, Training Curves & Overfitting

Training Curves

Here the effects of epoch on classification accuracy are observed for the **Training Dataset**. It should be noted that a classifier should never be tested on its training data. Data should always be separated into training and unseen test data from the very beginning. These training curves serve to illustrate how the perceptron is learning to fit the training data exactly over time. As epoch increases the classification accuracy on training data increases. Eventually the perceptron can classify the training data with 100% accuracy. This is expected. The training is being repeatedly reinforced on the exact same data, causing the perceptron to learn the specific training data incredibly well. The following training curves demonstrate that regardless of starting conditions, the perceptron will learn to classify the training data perfectly given enough passes through that same data.

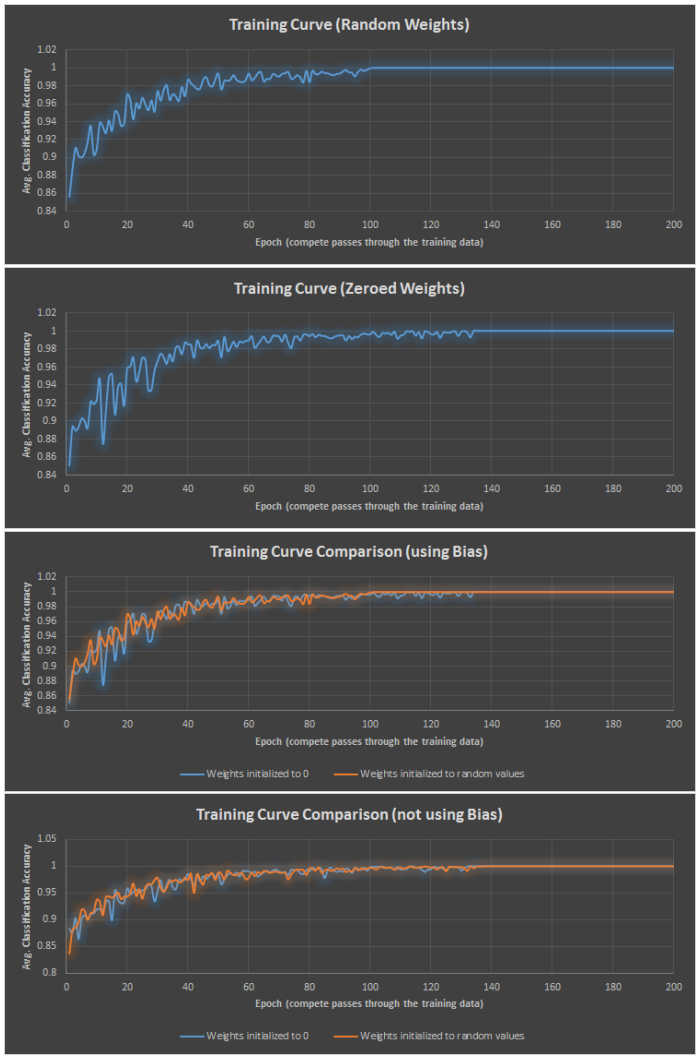

Figure 5: Classification Accuracy vs Epoch, comparing the initialization of weights.

Test Data & Overfitting

Here the effects of epoch on classification accuracy are observed for the **Test Dataset**. Varying the epoch (number of complete passes through the training data) has an impact on the average classification accuracy on test data as well. Since weights are adjusted each time, the average performance can increase or decrease between each epoch but the overall trend is upwards. Eventually the performance seems to converge. While performance is expected to increase in the lower epoch values, as evidenced by the upward trend towards convergence, the performance is not expected to stay at the converged level indefinitely. In reality training the perceptron on the same data too many times will result in overfitting. Overfitting occurs when a learning entity begins to reinforce patterns in the training data that may not be indicative of the general case (which might be reflected in unseen test cases). So while the entity may increase its performance on the training data, too much training on the same data will risk overfitting and decrease performance on unseen test data.

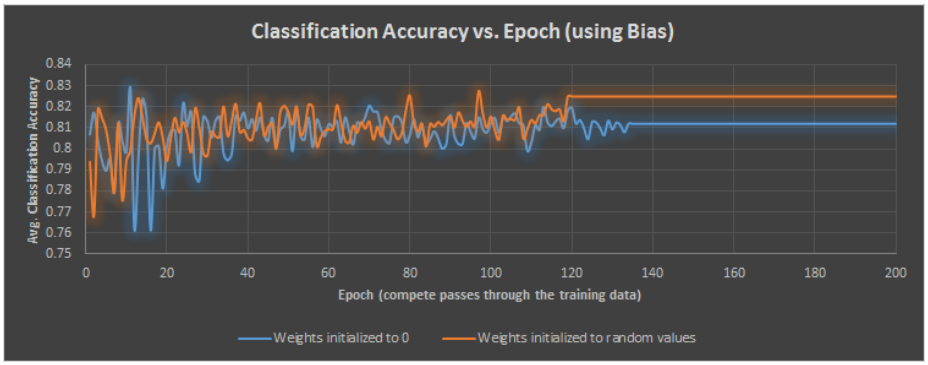

Figure 6: Classification Accuracy vs Epoch.

5.4 Initializing Weights (Random vs. Zero)

The way weights are initialized impacts the final classification accuracy. Initializing the weights at zero will always produce the exact same classification accuracy. Consider the case where all baseline conditions are left the same, except weights are initialized to zero.

Learning Rate Decay Fxn	Bias?	Weights Initialization	# of epochs
$\frac{1000}{1000+epoch}$	true	zero	1

Table 4: Zeroed Weights Perceptron Conditions

Such a setup will always produce the following classification results, because the weights are starting from the same place.

Digit Class	0	1	2	3	4	5	6	7	8	9	Average
Accuracy	0.96	0.94	0.83	0.58	0.54	0.75	0.86	0.86	0.78	0.83	**0.83**

Table 5: Classification Accuracy by Digit Class for Zeroed Weights

Comparing two strategies for initializing weights over many epochs (iterations through the training data), it can be seen that random weights tend to converge to higher average accuracies.

Figure 7: Classification Accuracy vs Epoch, comparing the initialization of weights.

5.5 Inclusion of Bias

To avoid rare issues originating from odd combinations of inputs, the perceptron can be modified with a bias input to avoid such situations all together. A bias input will always have the value of 1 and is passed into the perceptron just like any other input. It receives a weight and contributes to the summation passed into the activation function. The more inputs a perceptron has, and the more complex the input values can be, the lower the likelihood of receiving benefit from this bias input. The inclusion of the bias in this perceptron did not make a very noticeable impact. Compare the blue lines on each graph (representing perceptrons whose weight vectors were initialized to zero). With and without the bias the performance values are very similar.

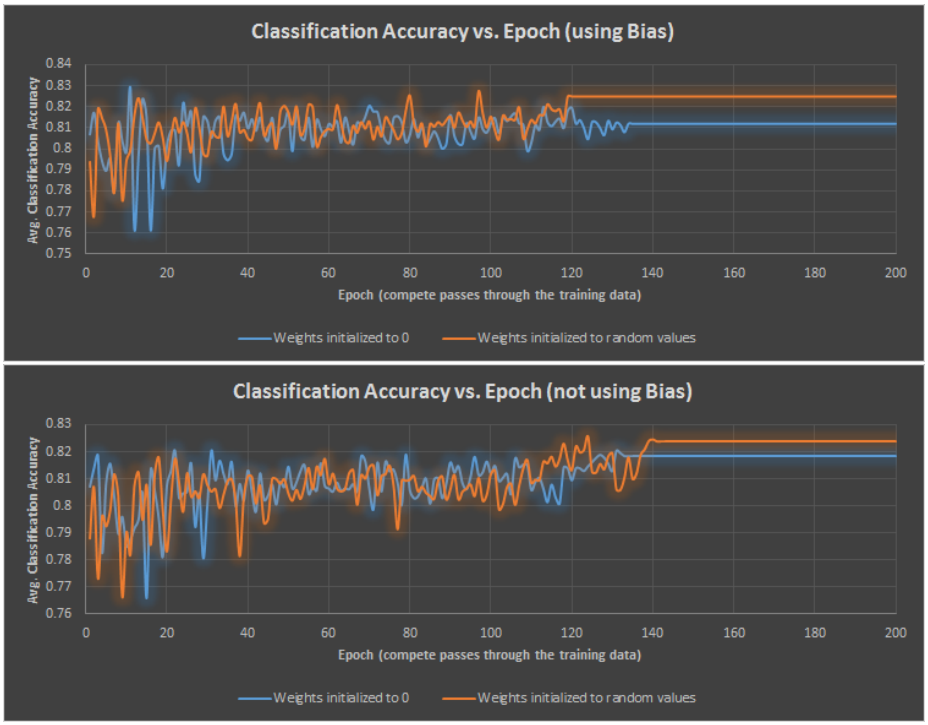

Figure 8: Classification Accuracy vs Epoch, comparing the inlcusion of bias.

6 Analysis & Discussion

The accuracy attained from a naive bayes classifier working on single pixel features was 0.77 (77%). This is lower than the minimum observed accuracy of a perceptron under any setup/conditions. Therefore the perceptron is yielding higher performance. Comparing the consistency of both classifiers by digit class, the standard deviation of the naive bayes classifier (0.09643) was slightly higher than that of the perceptron (0.08925). This would indicate that the perceptron had a more consistent classification accuracy across all classes. However, the difference is so small it is negligible for these results.

Looking at factors that impact performance of the perceptron classifier, there are a few key takeaways.

- Zeroing weights does not yield the best performance. Each weight will have an ideal best initial value, and initializing the weights to random values averaged out to better performance over initializing all to zero. This is most likely because the random values came closer (on average) to the ideal initial weights than did the value zero.

- The order that training data is presented to the perceptron during training is arguably the most important factor. The perceptron reinforces weights that provided accurate classifications. If the data is sorted by class and presented in that sorted order, then the perceptron will reinforce similar weights for the duration of training that is focused on a given class. Each time it moves on to a new class it will begin to reinforce only the weights relevant to that class and the other impact of the earlier training risks being forgotten. The end result is a classifier that is only good at classifying its most recently trained class. The workaround for this is to train the perceptron with training data presented in a random order. Ideally this would change every epoch as well, but here the data is simply presented in the same randomized order (the order it was written in the file) at every epoch.

- Lastly, it is important to separate training data and testing data early on. Then, it must be enforced that all training is conducted on training data and any performance measures are conducted on unseen test data. Ideally the perceptron would be trained on an enormous set of training data only once, ensuring that each instance of training is fresh. Because training data can be particularly difficult to gather and organize, often it is only feasible to train on a small data set. With a small data set it is possible to increase performance by training on the same data repeatedly. But such repetitions must be cut short to avoid overfitting the classifier to a particular data set. If the classifier trains for too long it will learn patterns in the data set that are not indicative of univeral patterns of classes. Overfitting risks decreased performance on unseen test data.

7 Potential Improvements

The performance of the single perceptron classifier could be further improved in various ways.

- Acquire more training data.
- Tune and optimize the learning rate function.
- Ensure that a new randomized ordering for training data is created every epoch (currently the random ordering is the same each epoch).
- Determine the most effective initialization value for each weight individually so that the weights need not be randomized.

Better yet, the perceptron could consider different features. There are likely better ways to define features than a single pixel. Is it better to use a group of pixels perhaps? Are there certain pixels that are more important (IE edges of the number), and in certain areas? Modern digit classifying algorithms likely consider more than the naive model used here.

Lastly, the single perceptron model implemented here is the most basic neural network: a single neuron. Grouping multiple perceptrons together to form a more complicated network would undoubtably be more powerful. With the human brain as the most complicated example and a single perceptron as the most simplistic, there is a lot of middle ground for artificial neural networks to cover.

References