Machine Learning Engineer Nanodegree

Capstone Project: MNIST Handwritten Digits Image Classification

Vincent Raerek

April 22, 2017

I. Definition

Project Overview

Humans have been communicating through written word since at least as early as 3100 B.C., and is inarguably one of the most essential contributions to human culture. The ability to record our thoughts by hand is so fast and natural that even today we could scarcely get by without it.

Yet although we often take this ability for granted, it can be a difficult task in computer vision. Vision itself is often taken for granted, as we humans naturally parse and interpret our visual fields so quickly that we have no need to even think about it consciously most of the time. However this apparently simple task is surprisingly so complex that the mountain of existent scientific work has arguably just scratched the surface of vision's mechanics in human biology and its role in the simple behaviors of our daily lives, let alone thought and consciousness. Suffice it to say that computer vision is a hard problem with great room for improvement.

The ability to recognize and interpret written characters involves a number of processes, from breaking down the image into meaningful segments to using context, experience and knowledge to interpret each segment. One area of human life where vision is essential is in commerce, where we frequently exchange currencies and other promissory notes for goods. We need to be able to read these paper documents in order to facilitate the transactions, and in large societies there are at least many millions of transactions daily. It would be helpful if we could enlist the help of computers to make the task of processing currency exchange easier. The first step to achieving this might be to write a program that would enable a computer to recognize numerical handwritten digits, since all currencies could be expected to contain numeric information (like the value of a bank note, or the value promised by a check).

Problem Statement

MNIST ("Modified National Institute of Standards and Technology"), released in 1999, is a commonly used dataset of handwritten images used as a resource for training and benchmarking computer vision classification programs.

In this exercise, my goal will be to correctly classify tens of thousands images of handwritten numeric digits. The data and problem objective has been provided and defined in the Kaggle competition "Digit Recognizer"[1]. The data exists as a series of grayscale images of handwritten digits from zero through nine. The dimensions of the images are 28x28 pixels each, for a total of 784 pixels per image. Each pixel has a value associated with it in the range of 0 to 255 inclusive, with higher values representing darker pixels.

I will attempt to convert each containing 784 pixels to a single value indicating which numeric digit is represented by the image. To do this, I will implement an artificial neural network from scratch in Python 2.7, using no machine learning libraries. In order to learn the information required to achieve this, I completed Andrew Ng's Coursera course on machine learning[2], and read Michael Nielson's "Neural Networks and Deep Learning"[3]. This will require manually translating the calculus of the neural network algorithm, as well as evaluation and tuning algorithms, to code. Although there are plenty of excellent machine learning libraries available today that could more efficiently produce the results I present here, I have chosen to implement the algorithm myself in the interest of my personal education and enjoyment. In a production environment, I would use an appropriate library.

Metrics

Since ultimately my goal is to correctly classify as many examples of handwritten digits as possible, it is natural to use measures of accuracy to quantify how well the model performs.

---

[1] "Digit Recognizer | Kaggle." https://www.kaggle.com/c/digit-recognizer. Accessed 23 Apr. 2017.
[2] "Machine Learning - Coursera." https://www.coursera.org/learn/machine-learning. Accessed 23 Apr. 2017.
[3] "Neural networks and deep learning." http://neuralnetworksanddeeplearning.com/. Accessed 23 Apr. 2017.

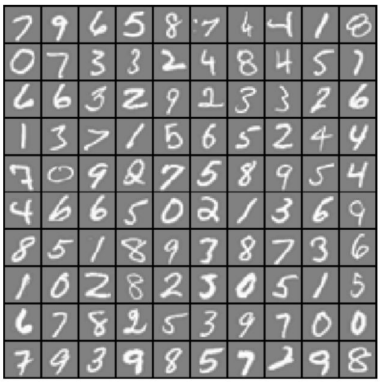
Basic accuracy, defined by the number of correct classifications divided by the total number of classifications, will suffice as a basic metric. The accuracy metric will be provided as the percentage of correct classification out of the total classification completed.

However, because accuracy can sometimes give a misrepresentation of performance simply by random accident, I will supplement this metric with the so-called F1 measure of the balance and valence of precision and recall. Measuring the precision of the model will allow me to gain insight into how often the model falsely positively identifies digits, and measuring the recall will provide insight into how thoroughly the model is able to identify all examples of a given digit in the dataset. Because this is a multiclassifcation problem, i.e. multiple classes exist as options for assignment, the F1 score will be derived as the mean of the F1 score achieved for each individual class. The F1 score will be represented by a fractional value between zero and one, inclusive, with higher values indicating better performance.

II. Analysis

Data Exploration

The MNIST dataset used here contains 42,000 labelled examples of 28x28 pixel images, for a total of 784 pixels per image, with each pixel represented by an integer between 0 and 255 inclusive. Higher pixel values represent darker pixels. The labels (integers 0 through 9) identify the numerical digit represented by the image, and will be used to train the model in a supervised fashion, meaning that the model will learn the characteristics of the digit image representations based on the provided labels. This dataset is provided already processed and cleaned, so it is given that there are no missing or corrupted data, and there is no need for further transformation or segmentation.



There are two statistical concerns I'd like to address regarding the composition of this dataset. First, while some pixels are exhibit significant variance across examples, others exhibit very little variation at all:

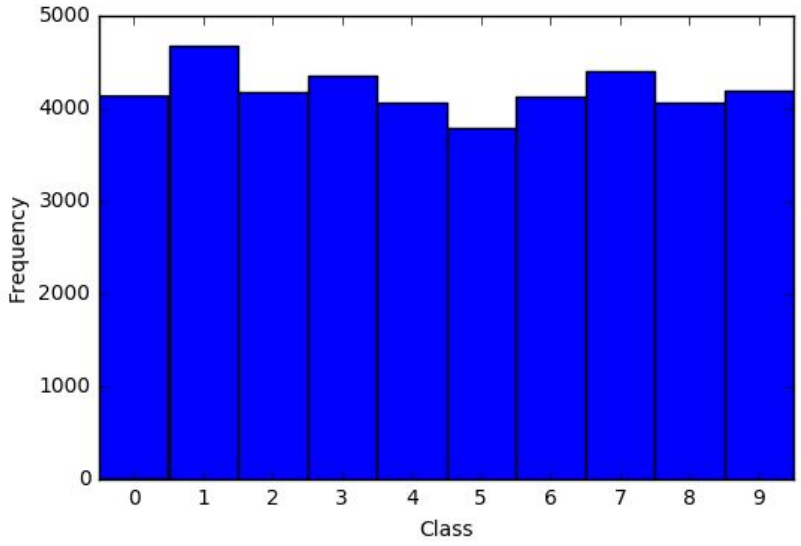
	Pixel 781	Pixel 782	Pixel 783
count	42000	42000	42000
mean	0.0	0.0	0.0
std	0.0	0.0	0.0
min	0.0	0.0	0.0
25%	0.0	0.0	0.0
50%	0.0	0.0	0.0
75%	0.0	0.0	0.0
max	0.0	0.0	0.0

These pixels are just white background in all images. The question might arise, “Can these features be removed from the set?”; but the answer is that they cannot, because although their value may not vary in this training set, it’s possible that this space may be occupied by test examples, and the network may learn a pattern for the each digit class that can be translated up or down or side-to-side in the image space. Therefore, we will let them remain.
The second statistical characteristic of the set is more significant, so we will explore it further with a visualization.

Exploratory Visualization

In classification problems it’s important to determine whether or not the training set is balanced. There should be a relatively even number proportion of examples of each class. Unbalanced datasets can lead to surprisingly high accuracy measurements even when using an obviously flawed model.

For example, suppose there is a training dataset containing labelled examples of medical patient data. The labels on this data indicate whether or not a patient is infected with disease Z. If 80% of the patients in the dataset are labelled as positively having disease Z, we could easily achieve a model with 80% accuracy simply by always predicting positive! We will investigate the current dataset for balance:



The histogram above describes the frequency of examples for each class in the dataset. We can clearly see that the number of examples is relatively even, hovering around 4000 examples per class. Therefore we can disregard concerns of imbalanced representation of accuracy measures. Furthermore, employing the F1 score as a metric also helps to reveal unhelpful patterns due to imbalance, because a model that achieves high accuracy simple by chance will usually also have a poor F1 score, because it will fail to achieve either high recall or precision due to its bias.

Algorithms and Techniques

I implement a basic artificial neural network trained with stochastic gradient descent for this challenge, with several options for tuning and configuration. A neural network is adept at solving non-linear functions for a variety of tasks, and computer vision is a common use case. Neural networks are especially useful when there are a large number of input features, which is the case for this problem.

I provide several options for many default variables and parameters. The regularization parameter, lambda, is tuned via k-fold cross validation, where the best performing regularization value on a validation set is selected and used for testing and prediction. The learning rate parameter, alpha, is set by default to start at 0.05, and decays over time to allow for smaller and smaller steps toward the gradient as the model matures. The number of epochs, or steps of gradient descent, is variable and set to default to 30 steps.

The activation function has two options: logistic and tanh. The logistic function can saturate quickly, and sometimes doesn't perform as well as tanh, which better centers the inputs. I also provide two options for the cost function: mean squared error (mse) and cross-entropy. The mean squared error is simply the mean of the squared errors from the true values, and although

it is an adequate cost function it can cause the model to learn more slowly, whereas cross-entropy helps to reduce learning saturation by ensuring greater cost when the error is large.

I also provide a configurable parameter the batch size of gradient descent. I use a variation of stochastic gradient descent sometimes known as batch gradient descent. I randomly select a batch of example to use for a step of gradient descent, and the size of this batch has a large impact on the speed of convergence.

Finally, I allow an option for random weight initialization before the model is trained. Weights can either be pulled randomly from a standard normal gaussian distribution, or they can randomly initialized from a much narrower distribution, which reduces some of the learning slowdown that can occur with very large or very small initial weights.

Benchmark

The worst model possible would predict classes merely by guessing randomly, and since there are 10 classes, this model would on average achieve 10% accuracy, since for any guess there would be a 1 in 10 chance of guessing correctly. The benchmark could be set as low as this, because anything better suggests an augmentation of our ability to predict the class from noisy data.

However, I've introduced this challenge in the context of banking, where transactions must be handled with extreme reliability. Therefore I will set 90% accuracy as the benchmark for success. In this case, the model can be considered to almost always predict the correct class, which is much suitable for this hypothetical application.

III. Methodology

Data Preprocessing

Because I am using a professionally catalogued dataset, provided preprocessed and pre-cleaned, there is no preprocessing necessary for this project. All images can be assumed to be clean and uncorrupted.

Implementation

I wanted to make an artificial neural network implementation that would be flexible enough to be applied to other datasets apart from just the MNIST image set. I started by building a neural network class that is capable of accepting a number of parameters, including layer quantity and sizes, activation functions, weight initialization, cost function, learning rate, regularization ratio, batch size of gradient descent, and number of epochs. In a separate file, I created a class for

each of the types of functions that could be supplied, one each for activation, cost, evaluation and weight initialization.

I tried to implement the neural network class so that it could be used similarly to the patterns in scikit learn, so the class provides methods for fitting and predicting. It also allows its parameters to be adjusted even after initialization.

I'll begin by describing an overview of the whole process, then I will describe the implementation of the neural network and gradient descent more in depth.

Custom data can be passed into the fit method, but if none is provided it will default to the MNIST training set. The fit method takes a keyword argument called "tuning", and this is a tuple containing the hyperparameter to tune and a function for adjusting the parameter as training progresses. When the fit method is called, the network layers will be initialized according to the layers parameter, and stochastic gradient descent will begin.

When gradient descent begins, a test set representing ten percent of the total data is extracted from the training data. Then k-fold cross validation is performed over the training set; during this time gradient descent is performed and model is trained for each fold. Each fold is a randomly selected subset of the training set, representing ten percent of the total training set size after the test set was extracted.

After all of the training has completed, the fold with the best F1 score is then selected and predictions on the test set are performed and evaluated using that fold's value of the parameter being tuned. The accuracy and F1 score of these predictions are reported in the console and pickled along with the parameters required to reproduce these scores.

Now that an overview of the entire training process has been described, I will explain some of the details of the implementation of the neural network and gradient descent.

For fold in the k-fold cross validation process, the following is performed. First, the weights of the model and the learning rate alpha are re-initialized to clear any prior learning from other folds. Next, some file management is performed to make room for a log of the training results during gradient descent. Next, we enter a routine that is performed for each epoch in each fold. This routing breaks up the training data further into batches. Finally, forward- and back-propagation are performed on each of these batches, and the model's weights are updated with each batch. Additionally, the learning rate alpha is reduced with each step of gradient descent, so that the model's weights are initially updated in large steps, but then in smaller and smaller steps as the cost function generated by the weights approaches its relative minimum value. After this, the models weights are used to make predictions on both the training set and the cross validation set, and these results are recorded in csv format.

Let's dive a little deeper into the feedforward propagation implementation, where a single training example is passed through the network layers and weights. The first layer of the neural network using the MNIST data will always be a 784-length vector representing the darkness of each pixel, using a value from 0 to 255, inclusive. The dot product of these values and the initial weights will result in the next layers inputs, which then pass through in activation function, and which are in turn passed through more weights and layers and activations and so on, until we reach the output layer, which ends in a final series of activation computations. The weighted sums and activation values of each layer are recorded in lists and passed back out of the feedforward function.

The backpropagation function takes the output from the feedforward function, namely the weighted sums and activations of each layer, and implements a derivation of the gradient of the entire network function via the chain rule, which causes the error to pass back out of the output, through activation functions, back across weights, back all the way to the first hidden layer. The error values, or deltas, are recorded in another set of lists, one for each layer (except the input layer).

The model weights are then updated with using the delta values, which causes the model to traverse down the derivative of the cost function toward its minimum. The weights are then adjusted one more time in proportion with the regularization parameter, which prevents these weights from growing unnecessarily large. Weights that grow too quickly tend to be undesirable because they can cause the model to overfit and thus generalize poorly.

Next, I will describe the implementation of the support functions for the neural network provided in the file called "neural_funcs.py". There are several classes of such function in this file, including activation, evaluation, weight initialization and cost. There are two options in each class. In activation, I have implemented the logistic and tanh functions, along with their derivatives. In evaluation, I have implemented functions for computing accuracy and F1 scores. The accuracy is relatively straightforward; it's just the number of correct predictions and out the total number of predictions. The F1 function however, computes the F1 for each class, and then returns the mean.

The weight initialization class allows us to draw random values either from a standard normal distribution, or from a normal distribution shrunk to plus or minus some small number epsilon. Finally, the cost class allows us to use either mean squared error or cross-entropy to calculate the error from true value labels.

The greatest complications I encountered during my implementation of forward- and back-propagation and gradient descent all revolved around my inexperience with numpy and pandas. I had some difficulty with managing the data types and structure of my data while it was passed through the network. Through a good measure of trial and error, along with some long bouts of debugging, I eventually untangled these issues however. I found that the implementations of the math formulas were relatively straightforward and easy to manage.

One other complication I encountered was just the decisions I faced while designing the main neural network class. I wasn't sure how flexible I wanted it to be; should it just be for learning the MNIST set, or would I prefer it to be able to be applied to other data as well. In the end I chose to make it flexible enough to be used more generally, but I think my indecision on this matter led to some messier code than I would usually write.

Refinement

The first time I ran my network, I ran it without regularization. I gave it one hidden layer with 5 units, and I set the learning parameter alpha to 0.5. At the time, I had not yet implemented the F1 metric, but the accuracy result was 11.2%, meaning that the model performed about as well as random chance.

I wasn't sure if my network was incorrectly implemented or if the input parameters weren't suitable, so for my next step I proceeded through trial and error to test various input parameters to see if I could significantly improve this initial result. I decided that if modifying the parameters led to a large improvement, then I would know that the network is correctly implemented and that I could move on past implementation to tuning the model. I soon discovered that my learning rate alpha was too large initially, and that by reducing it tenfold I could achieve vastly improved results: 82.6% accuracy and 0.821 F1.

At this point I felt confident in my network implementation, and proceeded to implement regularization and to perform k-fold cross validation to discover the best input parameters. After implementing regularization, I found to my surprise that the resulting model did not achieve better accuracy; instead it resulted in accuracy of 83% and an F1 of 0.827. However, it converged to this score faster, requiring only 25 epochs, where the previous model required 30 or more.

Finally, I tried adjusting the network layer sizes. Leaving the output layer at the required 10 units, I adjusted the hidden layer sizes, reasoning that a larger network would be able to capture a more complicated non-linear pattern. I used hidden layer sizes of 256 and 64 units. Alpha was set to begin at 0.05, and the best regularization value was nearly zero, at 0.00005. The activation function for these results was logistic and the cost function was mean squared error. This resulted in the highest score I have been able to achieve with this network: 93.3% accuracy and 0.932 F1.

IV. Results

Model Evaluation and Validation

The final model achieved over 93% accuracy, beating the benchmark goal by 3%. This model was derived through a combination of trial-and-error, k-fold cross validation, and theoretical

modeling. Trial and error produced a starting value for the learning rate alpha, which was initially set far too high, causing the gradient descent process to skip around the “bowl” of the cost function. By hand-tuning the parameter to an acceptable starting value, then implementing decay of the rate over time as gradient descent progressed, the model was able to closely approximate the lowest value of error on the test set.

The regularization parameter was tuned with k-fold cross validation; each cross-validation set was used to measure the generalizability of the learned model to unseen data using the current fold’s regularization parameter value. The fold that achieved the highest F1 score on the cross validation set was used as a model to be evaluated against the test set. This allows a reasonable level of confidence in the value used since it has been empirically proven to yield the best results in this context and with this data.

The data is always shuffled prior to splitting into test and cross-validation folds. Therefore we can be confident that variations in the input data to not disrupt the results of the model. Furthermore, because parameters are tuned to unseen data in the cross-validation folds, and the final model is evaluated against the unseen data in the test set, we can be confident that the model can generalize well, since the F1 and accuracy on the test set is still better than the benchmark.

Finally, the number of layers and the sizes thereof were the result of a decision informed by theory, which states that deeper and larger neural networks are capable of describing more and more nonlinear functions, thereby reducing bias that could undermine higher accuracy. I chose to increase the number of layers and the sizes thereof based on this information. I would like to try k-fold cross validation on this parameter as well, to more empirically investigate the best combinations available, but because the results of the current model already exceed the benchmark goal, I decided to leave the final model be.

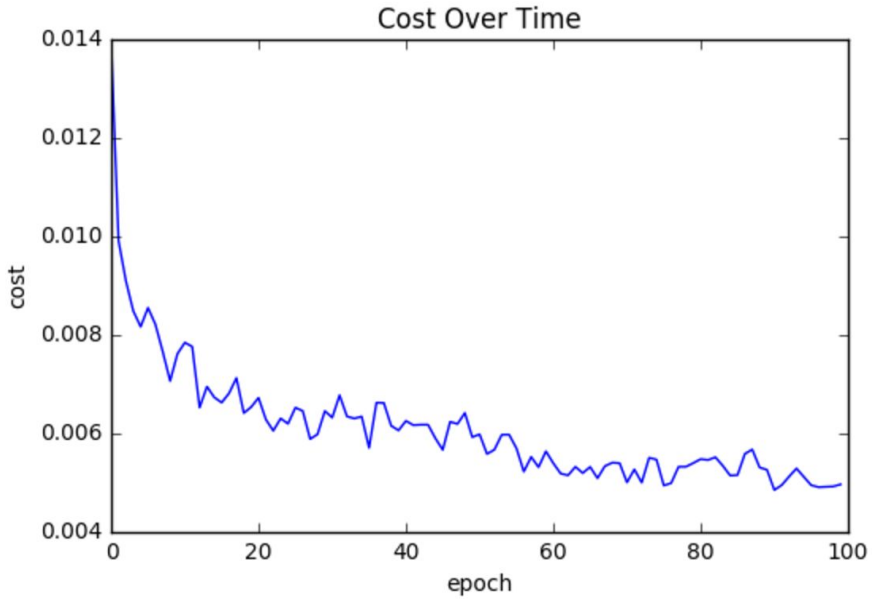
Justification

The benchmark goal for this model was elected to be 90% accuracy. The final model exceeds this goal by 3%. The F1 score accompanying this result is also high, rising up to over 0.932, where the highest possible score is one. This leads me to believe that not only is the model almost always predicting the correct class, but it is doing so with high precision and recall. Furthermore, the model was able to achieve results like these in most folds of cross-validation, in addition of course to the test set, suggesting that the model is highly reproducible. Therefore, the final model exhibits a high degree of reliability and accuracy.

V. Conclusion

Free-Form Visualization

Because the neural network was manually implemented from scratch, I felt it was important to verify that the network solved the gradient descent correctly, so below we plot the cost of the neural network as it learns over time through each epoch.



The results are clear, and the network is indeed learning over time. We can see that after around 60 epochs training the network does not progress much further down the gradient. This proves that the network is modifying the weights of each layer to minimize the cost function, and this is the main objective of gradient descent. It's also interesting to see that the cost falls sharply at the beginning, the oscillates lower and lower until it plateaus around epoch sixty. This observation can be explained by noticing that the learning rate alpha is largest at the beginning of gradient descent, causing the weights to undergo large changes, but alpha decays over time, shrinking smaller and smaller, and thus causing the network to learning more and more slowly.

Reflection

Computer vision is indeed a difficult task, even one as tightly constrained and simplified as this one. Using a neural network to solve the problem proved successful, but much more susceptible than I expected to large changes in the final model as a result of fine changes to the inputs. The network is incredibly adept at describing a nonlinear solution to separating examples in an extremely high dimensional space, a process which is incredibly mathematically complex.

The biggest surprise I encountered was the magnitude of the effect of small changes to initialization parameters. The most difficult challenge I experienced was the initial tuning of the learning rate. I wasn't sure what value would be suitable to start with, and small changes, perhaps a tenfold increase or decrease, had a large impact on the final results.

The final model does indeed fit my expectations to the problem, where I assumed I would be able to beat a 90% benchmark threshold, but I am sure that this network could be tuned further to improve its accuracy. Given the context within which this problem has been introduced, namely banking, I would not use the model I have created here, as banking necessitates a nearly perfect level of reliability, and 93% accuracy, although impressive enough to exceed my learning objectives here, is not high enough to be relied upon for critical financial transactions.

Improvement

There are many improvements that could be made to this implementation. First, cross-validation was only used to tune the regularization parameter, when in fact it could be used to discover the best values for each of the parameters in combination. Furthermore, cross-validation could also be used to investigate the best performing network layer sizes.

I have also read the Rectified Linear Units (ReLU) tend to provide better results for activation over logistic and tanh. ReLU activation are able to avoid much of the unit saturation that can plague networks based on logistic or even tanh. It's possible that the final result presented here suffered from this problem, and ReLU could proved to be a better solution.

Because this problem was pulled from a well-defined and explored Kaggle competition, it is already known that better solutions exist. I think that the current final results would prove to be a very useful benchmark for building a better network.