Devloping an handwitten Digit Recognizer app for Android
Project Report
Kabya Basu

I. Definition

Project Overview

In this project, we will be training a deep learning model to automatically recognize handwritten digits within an image. The science here can be applied to a large scale problem, such as reading postal pin code in address view images. To help understand this, we can build a classifier that understands how to classify digits and use a convolution neural networks, returning the prediction of the digit. Our primary dataset for this study is MNIST dataset.

Problem Statement

In this project, we are going to train a machine learning model to predict a handwritten digits within the canvas of the android app. To do so, we'll first use the MINST dataset and use that to train our model. We'll then validate the model with the test data. Once we are satisfied with the results, we'll validate that the model performs well using the app. If it performs well, we would have done a great job. If it doesn't, we'll review our model and learn what was not working.

STEP 0: Exploring using the 28x28 pixels images

The first step followed was to explore the dataset with simple Logistic model. We would create a test by randomly selecting a few entries. We then use these values to train the network.

Step 1: Modelling

Here, we'd create a convolution neural networks using a tensorflow and Keras. We would then train our network using the training and testing dataset.

Step 2: Evaluation

We'd then evaluate the model using the test dataset to see how well it performs. Also we will check how CNN better performs then the Logistic Model

Step 3: Deployment

We'd deploy the model to an android app.

Step 4: Testing with Live data

We will run the app and will check how good it is predicting live.

Metrics

Since this problem is quite delicate, we don't want to mislead people with wrong house numbers, we would only give credit when all digits in the sequence are predicted. We would use the percentage of correctly predicted sequences to measure the performance of our model.

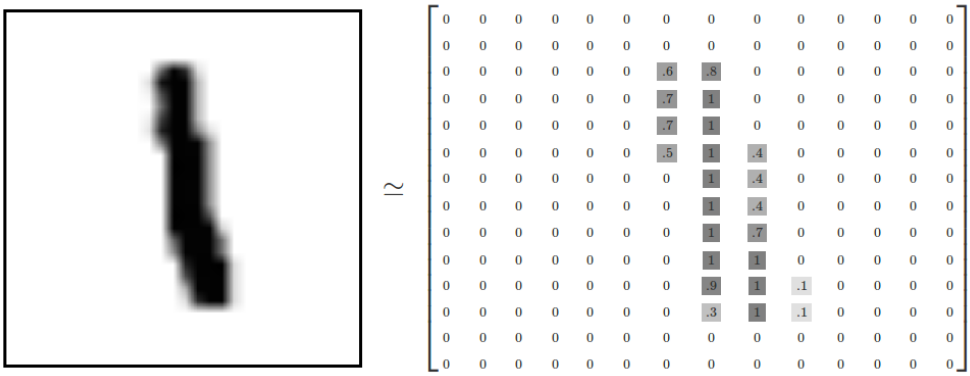
II. Analysis

Data Exploration

The dataset has been separated into two. One for train and test data. The data has 28x28 pixel images. With this train a logistic regression model and see how it performs.The MNIST data contains images like below.

As mentioned earlier, every MNIST data point has two parts: an image of a handwritten digit and a corresponding label. We'll call the images "x" and the labels "y". Both the training set and test set contain images and their corresponding labels; for example the training imag are mnist.train.images and the training labels are mnist.train.labels.

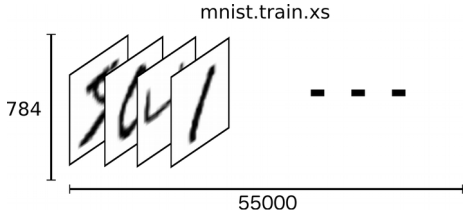
Each image is 28 pixels by 28 pixels. We can interpret this as a big array of numbers:



We can flatten this array into a vector of 28x28 = 784 numbers. It doesn't matter how we flatten the array, as long as we're consistent between images. From this perspective, the MNIST images are just a bunch of points in a 784-dimensional vector space, with a very rich structure (warning: computationally intensive visualizations).

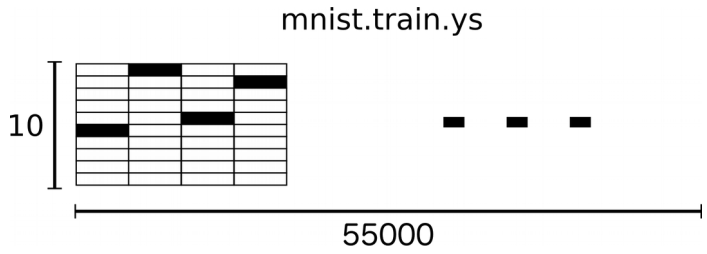
Flattening the data throws away information about the 2D structure of the image. Isn't that bad? Well, the best computer vision methods do exploit this structure, and we will in later tutorials. But the simple method we will be using here, a softmax regression (defined below), won't.

The result is that mnist.train.images is a tensor (an n-dimensional array) with a shape of [55000, 784]. The first dimension is an index into the list of images and the second dimension is the index for each pixel in each image. Each entry in the tensor is a pixel intensity between 0 and 1, for a particular pixel in a particular image.



Each image in MNIST has a corresponding label, a number between 0 and 9 representing the digit drawn in the image.

For the purposes of this tutorial, we're going to want our labels as "one-hot vectors". A one-hot vector is a vector which is 0 in most dimensions, and 1 in a single dimension. In this case, the digit will be represented as a vector which is 1 in the dimension. For example, 3 would be. Consequently, mnist.train.labels is a[55000, 10] array of floats.

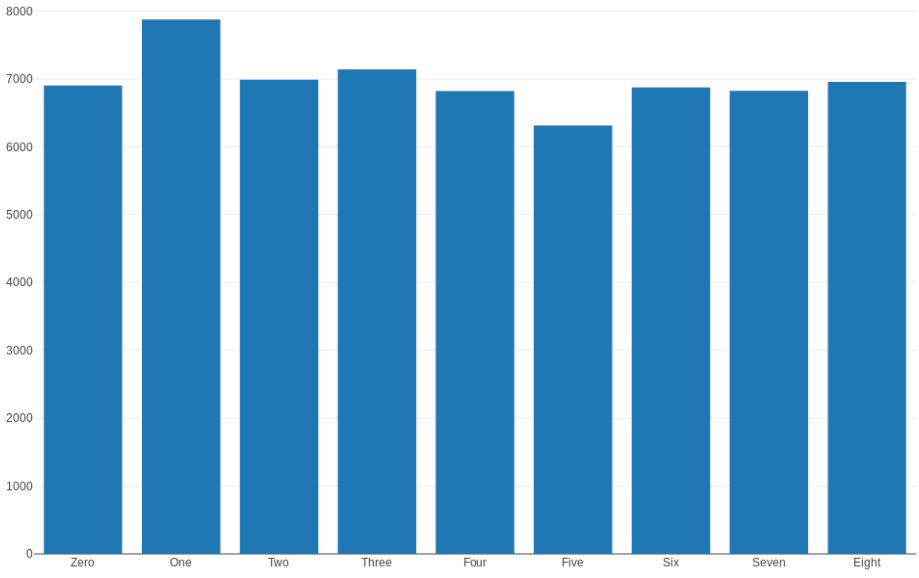


Exploratory Visualization

In the dataset, each digit image is 28 pixels in height and 28 pixels in width, for a total of 784 pixels. Each pixel has a single pixel-value, denoted as xi, associated with it, indicating the lightness or darkness of that pixel with smaller meaning darker. These pixel-values, xi, range from 0 for a completely black pixel to 255 for a completely white pixels. It is convenient to flatten the image and represent them as a vector x = (x1, x2, ..., xd) T (in this case d = 784). The Arabic numerals represented by the images are the labels that we want to predict. Basically, we intend to assign each image to ten classes.

Before introducing any model,we also investigated the distribution of data in train dataset and test dataset. From the Figure 1, it indicates that the number of images in each class are uniform. If we randomly guess a number, which is the most trivial model, we will get about 10% accuracy.

Figure 3: Data Distribution for Each Class



Algorithms and Techniques

Initially we built a Softmax Regressions function, We know that every image in MNIST is of a handwritten digit between zero and nine. So there are only ten possible things that a given image can be. We want to be able to look at an image and give the probabilities for it being each digit. For example, our model might look at a picture of a nine and be 80% sure it's a nine, but give a 5% chance to it being an eight (because of the top loop) and a bit of probability to all the others because it isn't 100% sure.

This is a classic case where a softmax regression is a natural, simple model. If we want to assign probabilities to an object being one of several different things, softmax is the thing to do, because softmax gives us a list of values between 0 and 1 that add up to 1. Even later on, when we train more sophisticated models, the final step will be a layer of softmax.

A softmax regression has two steps: first we add up the evidence of our input being in certain classes, and then we convert that evidence into probabilities.

To tally up the evidence that a given image is in a particular class, we do a weighted sum of the pixel intensities. The weight is negative if that pixel having a high intensity is evidence against the image being in that class, and positive if it is evidence in favor.

We also add some extra evidence called a bias. Basically, we want to be able to say that some things are more likely independent of the input. The result is that the evidence for a class i given an input x is:

$$\text{evidence}_i = \sum_j W_{i,\,j} x_j + b_i$$

where Wi is the weights and bi is the bias for class i, and j is an index for summing over the pixels in our input image x. We then convert the evidence tallies into our predicted probabilities y using the "softmax" function:

$$y = \text{softmax}(\text{evidence})$$

Here softmax is serving as an "activation" or "link" function, shaping the output of our linear function into the form we want -- in this case, a probability distribution over 10 cases. We can think of it as converting tallies of evidence into probabilities of our input being in each class. It's defined as:

$$\text{softmax}(x) = \text{normalize}(\exp(x))$$

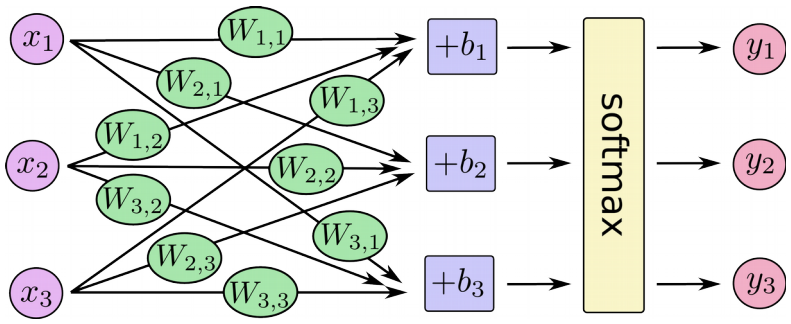
If we expand that equation out, we get:

$$\text{softmax}(x)_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

But it's often more helpful to think of softmax the first way: exponentiating its inputs and then normalizing them. The exponentiation means that one more unit of evidence increases the weight

given to any hypothesis multiplicatively. And conversely, having one less unit of evidence means that a hypothesis gets a fraction of its earlier weight. No hypothesis ever has zero or negative weight. Softmax then normalizes these weights, so that they add up to one, forming a valid probability distribution.

We can picture our softmax regression as looking something like the following, although with a lot more xs. For each output, we compute a weighted sum of the xs, add a bias, and then apply softmax.



If we write that out as equations, we get:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax}\left(\begin{bmatrix} W_{1,1}x_1 + W_{1,2}x_2 + W_{1,3}x_3 + b_1 \\ W_{2,1}x_1 + W_{2,2}x_2 + W_{2,3}x_3 + b_2 \\ W_{3,1}x_1 + W_{3,2}x_2 + W_{3,3}x_3 + b_3 \end{bmatrix} \right)$$

We can "vectorize" this procedure, turning it into a matrix multiplication and vector addition. This is helpful for computational efficiency.

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax}\left(\begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right)$$

More compactly, we can just write:

$$y = \text{softmax}(Wx + b)$$

We can use this softmax function with a regression model and use for prediction.

Convolution Neural Network

As we know softmax regression is a very basic model, so we will be using more advanced Convolution Neural Network (convnet) to predict recognize digits. This is largely due to to the network being able to identify images at different scales. Another advantage here is that a convnet is able to directly work on raw pixels. We'll be using an architecture similar to the one made popular by the tensor flow.

Figure 4: Convolution Neural Network

Images / Input Layer :
This layer holds the the input which is equivalent to a store of the the pixels of all images.

Convnet / Convolution Layer:
Convnets are neural networks that share their parameters across space. It works by sliding over the vector with depth, height and width and producing another matrix(called a convolution) that has a different weight, depth height. Essentially, rather than having multiple matrix multipliers, we have a set of convolutions. For example, if we have an image of size 1024x1024px in 3 channels, we could progressively create convolutions till our final convolution has a size of 28x28px and a much larger depth.

Pooling (Max Pooling)
Pooling is a technique that can be used to reduce the spatial extent of a convnet. Pooling finds a way to combine all information from a previous convolution into new convolution. For our example, we'll be using maxpooling, takes the maximum of all the responses in a given neighborhood. We choose it because it does not add to our feature space and yet gives accurate responses.

Fully Connected Layer
This layer connects every neuron from the previous convolutions to every neuron that it has. It converts a spatiallike network to a 1d network, so we can then use this network to produce our outputs.

The Output layer
The output from this layer are logits which represents matrix showing the probabilities that of having a character in a particular position

Benchmark

The primary benchmark here would be the accuracy of prediction. Accuracy score is the percentage of correctly predicted data.To create a model that can help, it needs to be able to perform either close to or better than humans, I'd say between 98 and 100%.

III. Methodology

Data Preprocessing

We performed the following steps to preprocess data:

1. Extract information from the digit Struct and save it in a python friendly format.
2. Resize our new images to 28x28 pixels.
3.Now we generate final training, testing dataset.

Implementation

My implementation is broken down into following steps, and covered in Instructions_model_Training&Saving.ipynb jupyter notebook file.

01-Download and Extraction:
Here, we pull download and extract the MNIST dataset.

02-Exploration
Here we explore the dataset and generate visualizations such as a histogram of Data Distribution for each class.

03-Model Training and Prediction:

First, we retrieve the already saved dataset. We also define our model here as well as the accuracy function.As discussed eariler we have created 3 model.We then train our training dataset, but in batches. Once that is done, we save the trained Tensorflow and Keras model to be used in the app.

We also print the prediction accurecy of the each model.

Model Training with tensorflow :

1. We load up the saved dataset
2. We define our convnet (Figure: 4)
3. We define the weights and biases for our 10 logits. The weights are initialized using a tensor flow function that ensures that weights are balanced randomly based on the number of neurons.
4. We define our Loss and accuracy function
5. We train our model and log important information like accuracy of the test set and the loss.
6. Save and exported the trained model to disk.

Our Convolution model
- C1: convolutional layer, batch_size x 28 x 28 x 16, convolution size: 5 x 5 x 1 x 16
- S2: sub-sampling layer, batch_size x 14 x 14 x 16
- C3: convolutional layer, batch_size x 10 x 10 x 32, convolution size: 5 x 5 x 16 x 32
- S4: sub-sampling layer, batch_size x 5 x 5 x 32
- C5: convolutional layer, batch_size x 1 x 1 x 64, convolution size: 5 x 5 x 32 x 64

● Dropout F6: fully-connected layer, weight size: 64 x 16
● Output layer, weight size: 16 x 10
That brings it up to 7 layers.

To train, we read the already preprocessed data into the model and train it in batches.During the training, we try to minimize loss and log the accuracy we are achieving so we can keep track of how well our model is improving. Once the model is trained, we evaluate it using our test step. One step further would be to save the model and load he model so it can be used in other applications such as an android app.

Similar steps were also taken for Keras trnsorflow backend model.

Refinement

I performed the following improvements to improve the accuracy score.

1. I added a dropout layer to the network just before fully connected layer. This was to prevent over fitting. It works by randomly dropping weights from the model.

2. I also changed the learning rate 1E-4 for tensorflow model, so that it learns accurately even if it takes time.

This refinement improved the prediction accurecy.

IV. Results

At the end of my training, I was able to achieve the following:

Tensorflow model :

Test accuracy: 98.25%

Keras model :

Test accuracy: 92.42%

According to our benchmark, Tensorflow model produced much more accurate predictions when compared with the softmax regrssion classifier which has 92% prediction accurecy in testing dataset, although not as good as humans do.

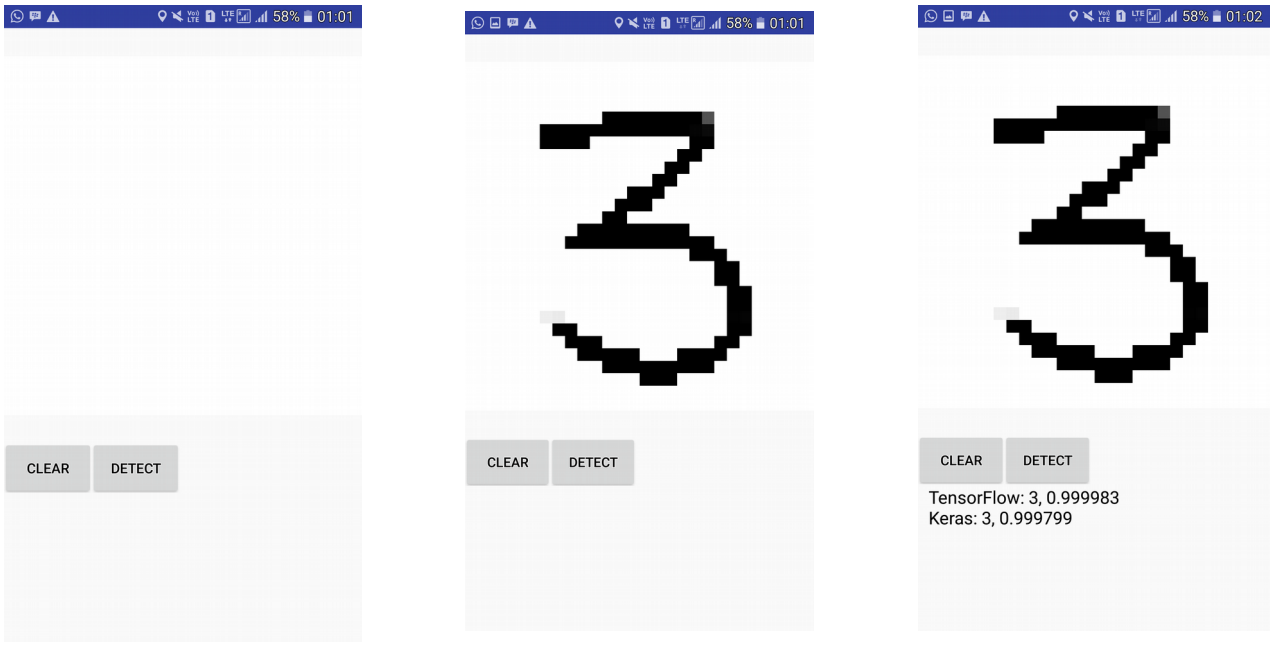
Justification

Our benchmark of human recognition is 100% while that of a softmax regression classifier was 92%. Our Tensorflow model was able to achieve 98.25% accuracy while training on 60000. This is much better than the softmax regression. If we have more images to train on, we expect the accuracy to continue to improve moving closer to human recognition. According to the paper written by on this model, one can achieve over 100% accuracy with this model.

V. Conclusion

FreeForm Visualization

To test how well our model is working, we exported our model into an android application, and then drwan a hand wriiten digit on the canvas and tried to predtict it with the detect button.



We can see that our both model did a pretty good job at predicting data and it did so with a training set of just over 60000.

Reflection

Deep learning is an interesting field of machine learning that can be applied to many exciting problems. The problem we have applied it to is handwritten digit recognition.

While it is capable of solving many problems, the part that I find most challenging is the specialized compute resource needed to get good results. GPUs are becoming cheaper and I believe that every device would soon be able to train deep learning models, or at least retrain their models based on new information.

To close this project I also took up the Udacity android devloper cource the instructer are great their.

The Udacity forum has been really helpful as many of the ideas from this project comes from the course and the forum.

Improvement

There are several ways this implementation can be improved.

One of which is to run this model on a GPU. Scholars have written on how this can provide up to 10x improvement in the training time for deep neural networks.

Also instead of canvas we may connect the app to camera and then can predict the hand written digit.