Machine Learning Engineer Nanodegree

Capstone Project

Tushar Bansal
August 20th, 2017

I. Definition

Project Overview

Handwriting recognition is the ability of a computer to receive and interpret intelligible handwritten input from sources such as paper documents, photographs, touch-screens and other devices. The image of the written text may be sensed "off line" from a piece of paper by optical scanning (optical character recognition) or intelligent word recognition. Alternatively, the movements of the pen tip may be sensed "on line", for example by a pen-based computer screen surface, a generally easier task as there are more clues available. This is a classification problem for which support vector machines are very successfully used. Determining optimal support vector machine is another hard optimization problem that involves tuning of the soft margin and kernel function parameters. Off-line handwriting recognition involves the automatic conversion of text in an image into letter codes which are usable within computer and text-proThe goal in this is to take an image of a handwritten single digit and determine what that digit is.

For every ImageId in the test set, you should predict the correct label.

This is evaluated on the categorization accuracy of your predictions (the percentage of images you get correct).cessing applications. The data obtained by this form is regarded as a static representation of handwriting. Off-line handwriting recognition is comparatively difficult, as different people have different handwriting styles. And, as of today, OCR engines are primarily focused on machine printed text and ICR for hand "printed" (written in capital letters) text.

Problem Statement

Narrowing the problem domain often helps increase the accuracy of handwriting recognition systems. A form field for a U.S. ZIP code, for example, would contain only the characters 0-9. This fact would reduce the number of possible identifications.

Primary techniques:

Specifying specific character ranges

Utilization of specialized forms

Our Final goal is to predict the digit written in an image.

Metrics

The goal in this is to take an image of a handwritten single digit and determine what that digit is.

For every ImageId in the test set, you should predict the correct label.

This is evaluated on the categorization accuracy of your predictions (the percentage of images you get correct).

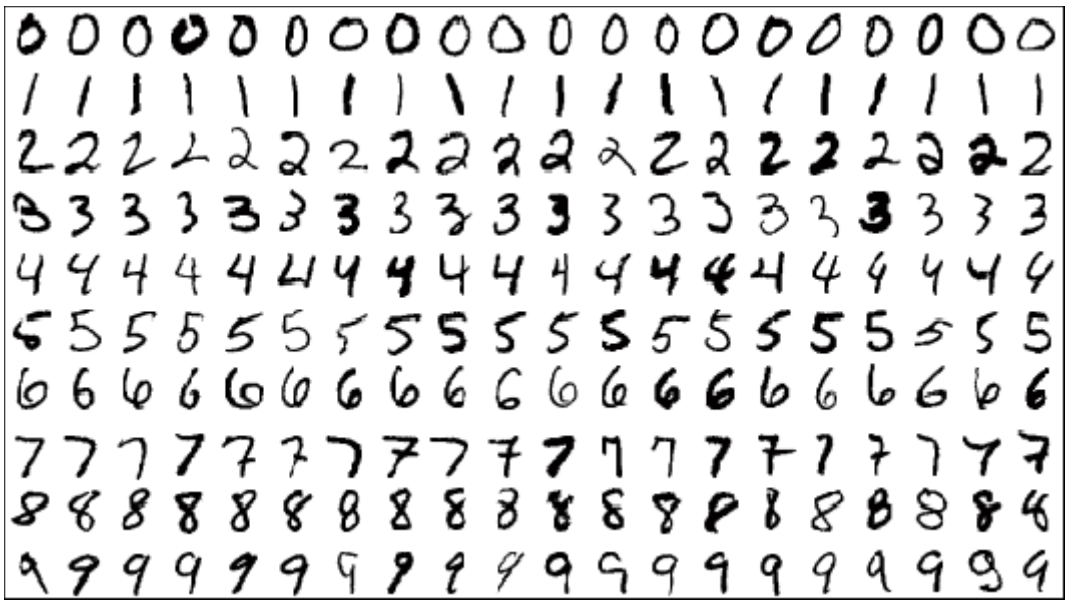
II. Analysis

Data Exploration

The original black and white (bilevel) images from NIST were size normalized to fit in a 20x20 pixel box while preserving their aspect ratio. The resulting images contain grey levels as a result of the anti-aliasing technique used by the normalization algorithm. the images were centered in a 28x28 image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the 28x28 field.

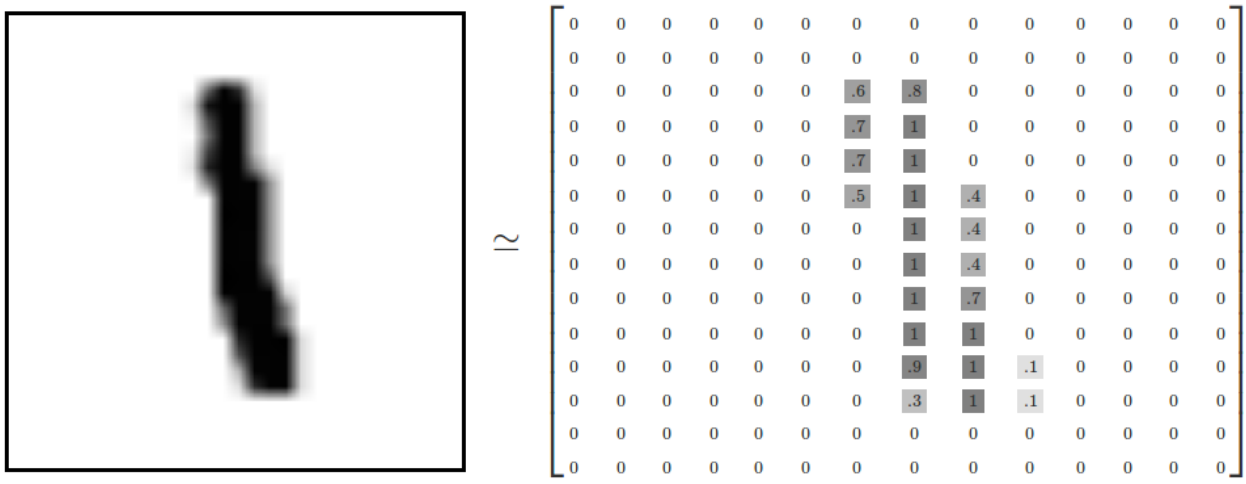
The MNIST data is split into three parts: 55,000 data points of training data (`mnist.train`), 10,000 points of test data (`mnist.test`), and 5,000 points of validation data (`mnist.validation`). This split is very important: it's essential in machine learning that we have separate data which we don't learn from so that we can make sure that what we've learned actually generalizes!

Exploratory Visualization



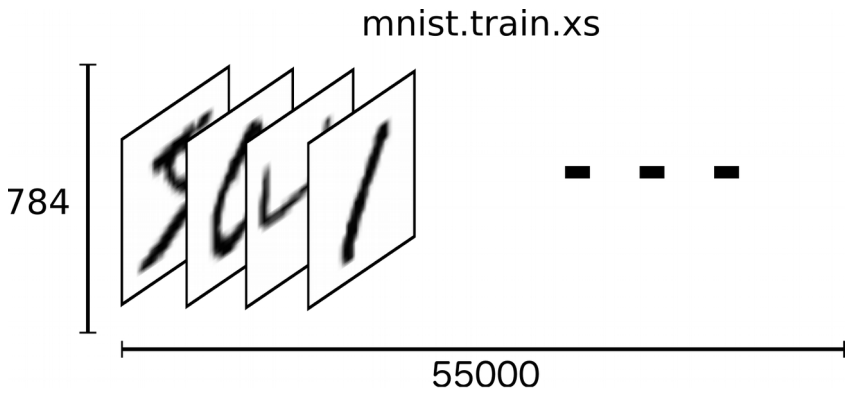
Every MNIST data point has two parts: an image of a handwritten digit and a corresponding label. We'll call the images "x" and the labels "y". Both the training set and test set contain images and their corresponding labels; for example the training images are `mnist.train.images` and the training labels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

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

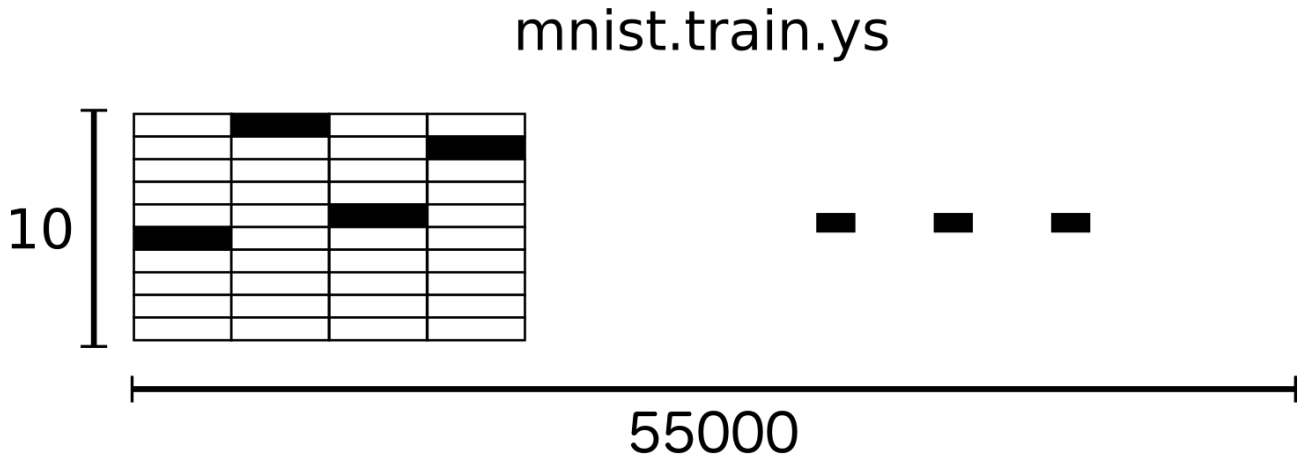


We can flatten this array into a vector of 28x28 = 784 numbers. It doesn't matter how we flatten the array, as long as we're consistent between images. From this perspective, the MNIST images are just a bunch of points in a 784-dimensional vector space, with a very rich structure.

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.

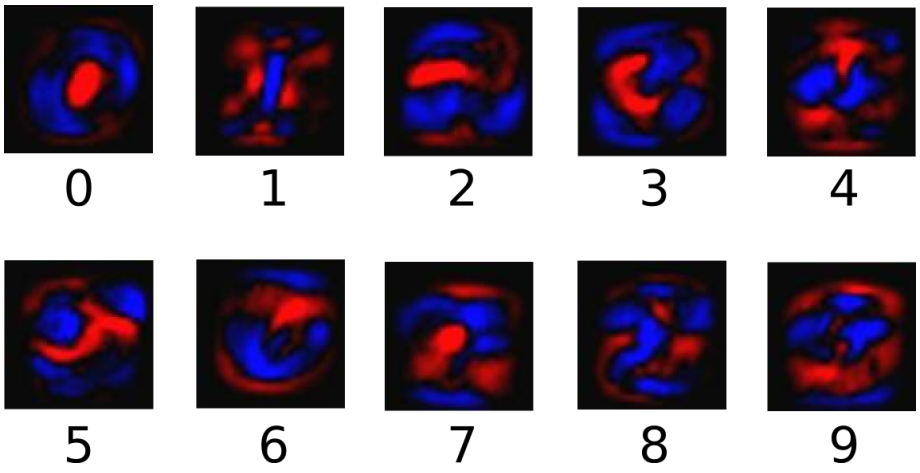


Each image in MNIST has a corresponding label, a number between 0 and 9 representing the digit drawn in the image.



Algorithms and Techniques

Every digit will lie between zero and nine so we know we have 10 possibilities for each case. For example the digit is 0 so it will also have some probability that it is 8 or 9. but probability of besing 0 will be greatest. We will use softmax to train out model because it gives of values between 0 and 1 that all adds upto 1.



Benchmark

I used the SVM with RBF kernel as my benchmark and several kernels on Kaggle competition SVM with RBF takes much more time to train than neural nets we can use several parameters with SVM_RBF but it takes much more time to train as compared to neural networks. My accuracy score was ~92% with neural nets.

III. Methodology

Data Preprocessing

After Downloading the data we processed them into 4D uint8 numpy array and then converted scalars to one-hot vectors.

Extracted labels into 1D uint8 array.

We split the data into testing and training set

TRAIN_IMAGES = 'train-images-idx3-ubyte.gz'

TRAIN_LABELS = 'train-labels-idx1-ubyte.gz'

TEST_IMAGES = 't10k-images-idx3-ubyte.gz'

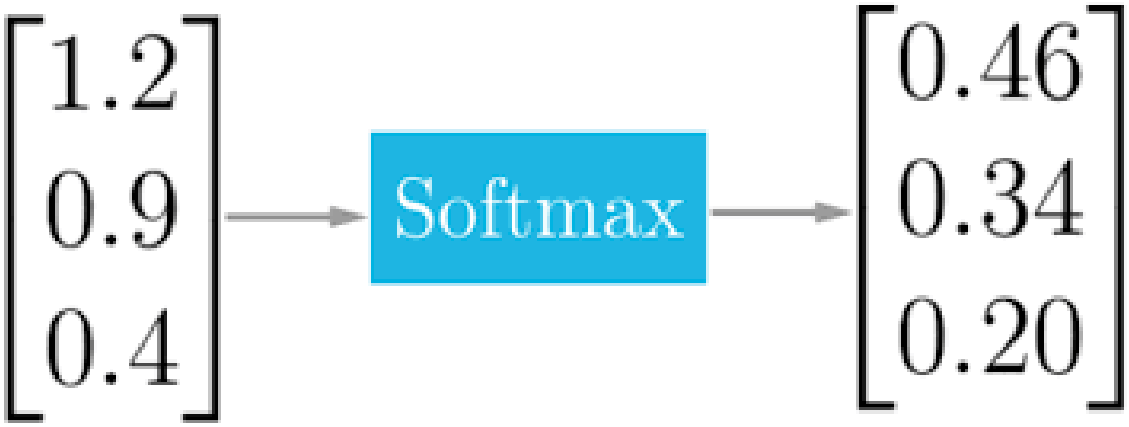
TEST_LABELS = 't10k-labels-idx1-ubyte.gz'

Implementation

We first Downloaded the input dataset, and split it into testing and training data then preprocess it using the data_dl.py script.

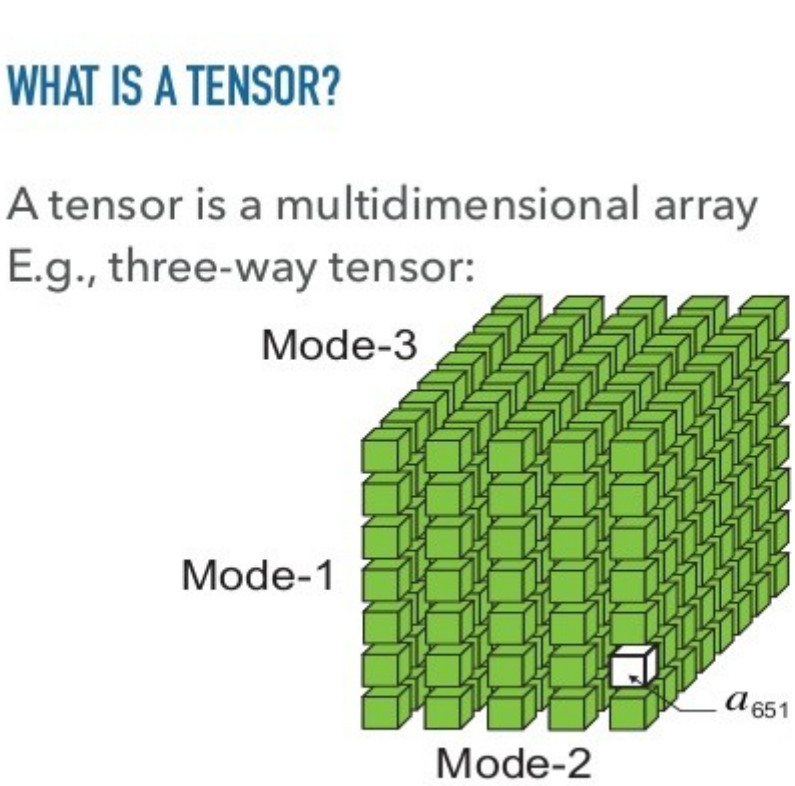
We need to set the Hyper parameters first i.e Learning Rate, training_iteration, batch_size and display_step.

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$$



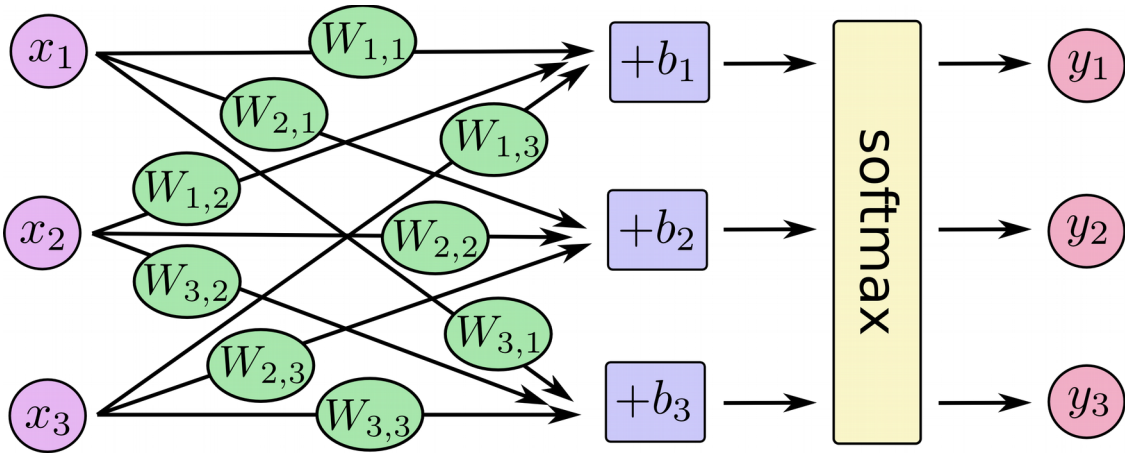
Tensorflow Represents data in a Tensor. It takes input as a tensor and output as a tensor as well.

Tensor is a multidimensional array of numbers and and they flow between operations

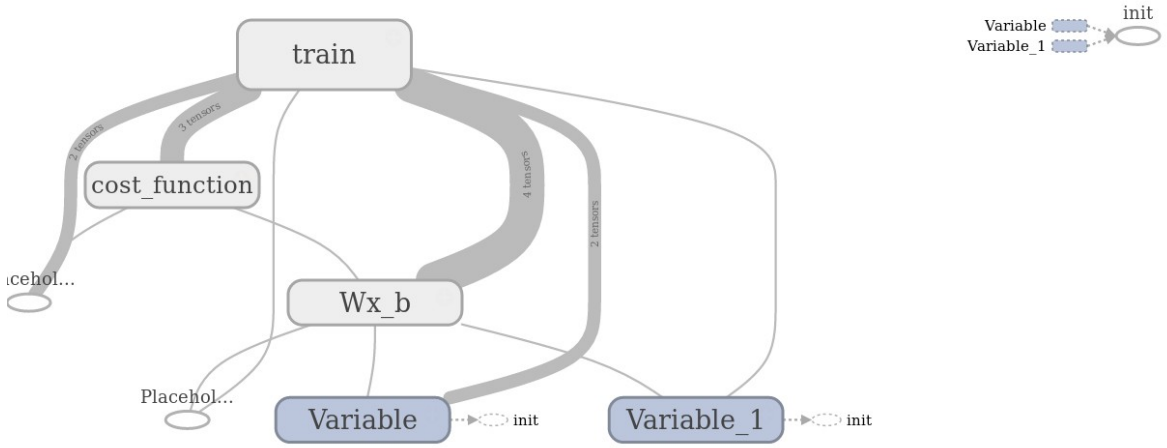


We will create two placeholder operations first. And assign 784 dimension to x and 10 to y as x is an image with 28x28 (which are converted into 1D array) and y are labels.

Now we will set our Weights and biases and set value of them to zero.



Now we will implement our model logistic regression by matrix multiplication. We will also visualize weights and biases by creating summary operation. We wil also create cost funtion which will minimize errors during training by sung cross entropy option and also create scalar summary to visualize it.



And in last scope we will improve our model during training by using optimisation function. We will be using Gradient Descent algorithm which will take our leraning rate into consideration and cost_function will minimize the errors.

We will initialize all of our variables we will launch our graph in an Tensorflow session. And set the location to the summary writer which we will later visualize using tensorboard.

Now we will set for loop for number of iterations and we will also print our average cost on every iteration so that we can check that the model is actually learning or not.

We will compute our batch size and start training our model over each data. After training we will fit the batch data in our Gradient Descent Algorithm for the back propogarion and computing loss and writing logsfor each iterations by using summary writer.

We will display the error logs in terminal for each display step.

Now at the end we will test the model by comparing the model values to the output values. And Calculate the accuracy.

Refinement

We will tune our hyper parameters to tune out model and gain more accuracy. In final i set them to

learning_rate=0.01

training_iteration = 50

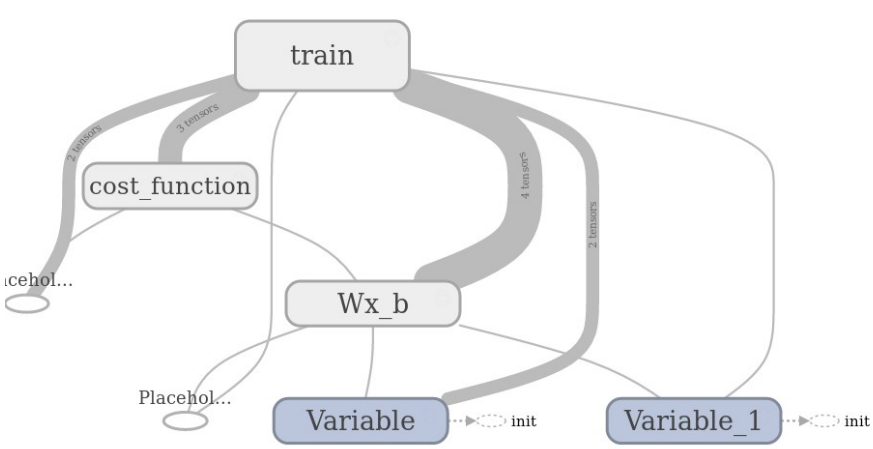
batch_size = 64

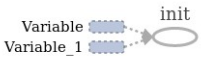
IV. Results

Model Evaluation and Validation

Cost Function

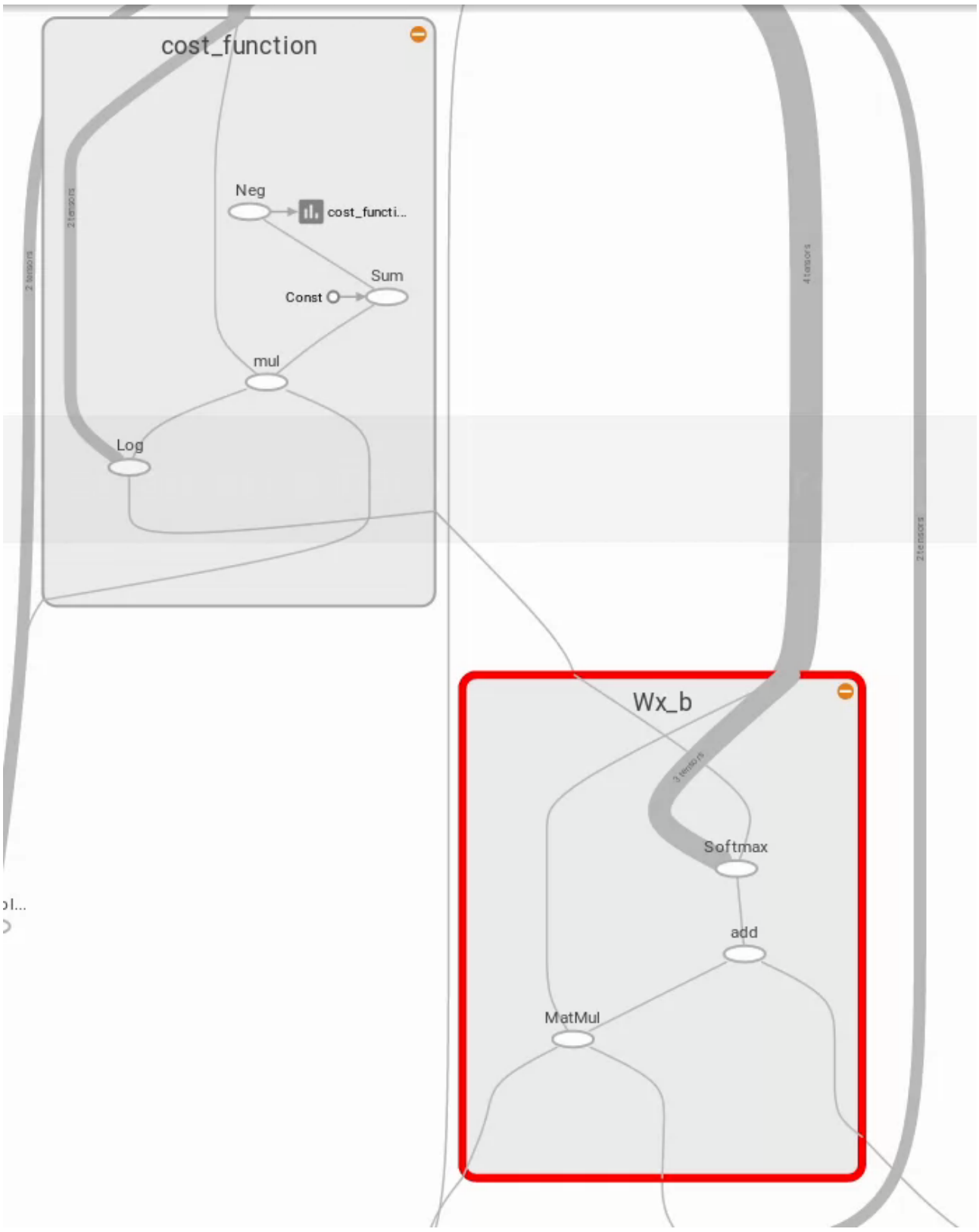
We Can see our graph of cost function over time how it changes as our odel trains.





Here we can see how our model works we can flow of tensors over our edges connecting our nodes or we can say operations.

We can see each detail that how tensors are flowing using tensorgraph.



There is much more in graphs we can see all of them using tensorgraph.

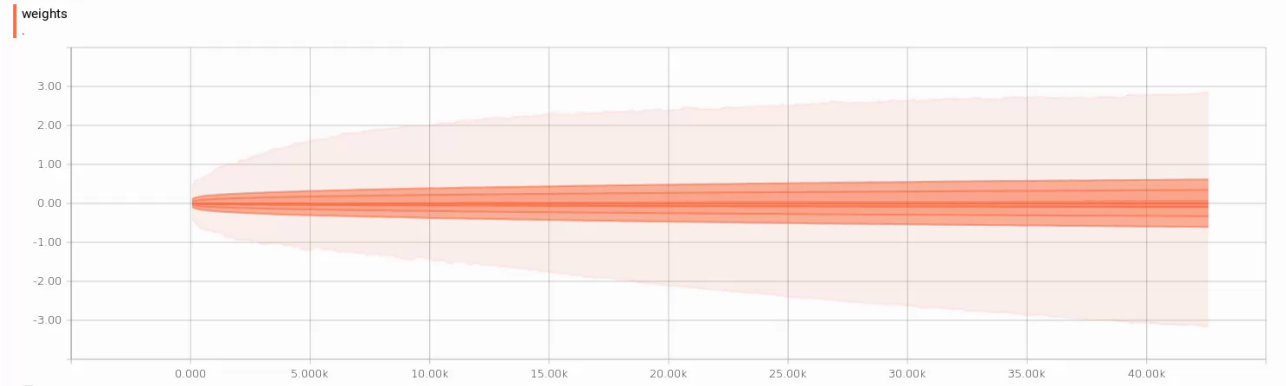
Justification

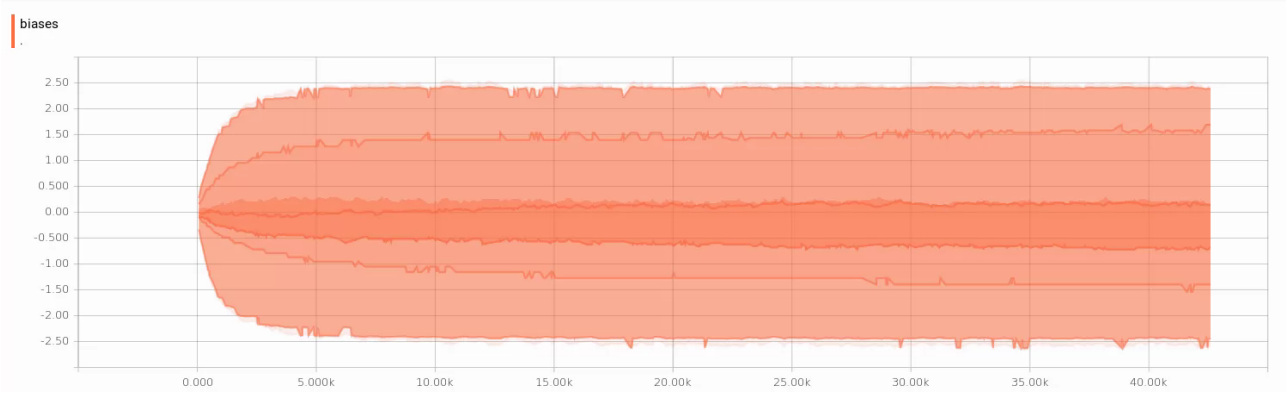
We can see that my benchmark model had very less accuracy less than 90% and that took much more time to train.

My current model has an accuracy of ~92% and it takes very less time to train i.e less then 5 minutes.

V. Conclusion

Free-Form Visualization

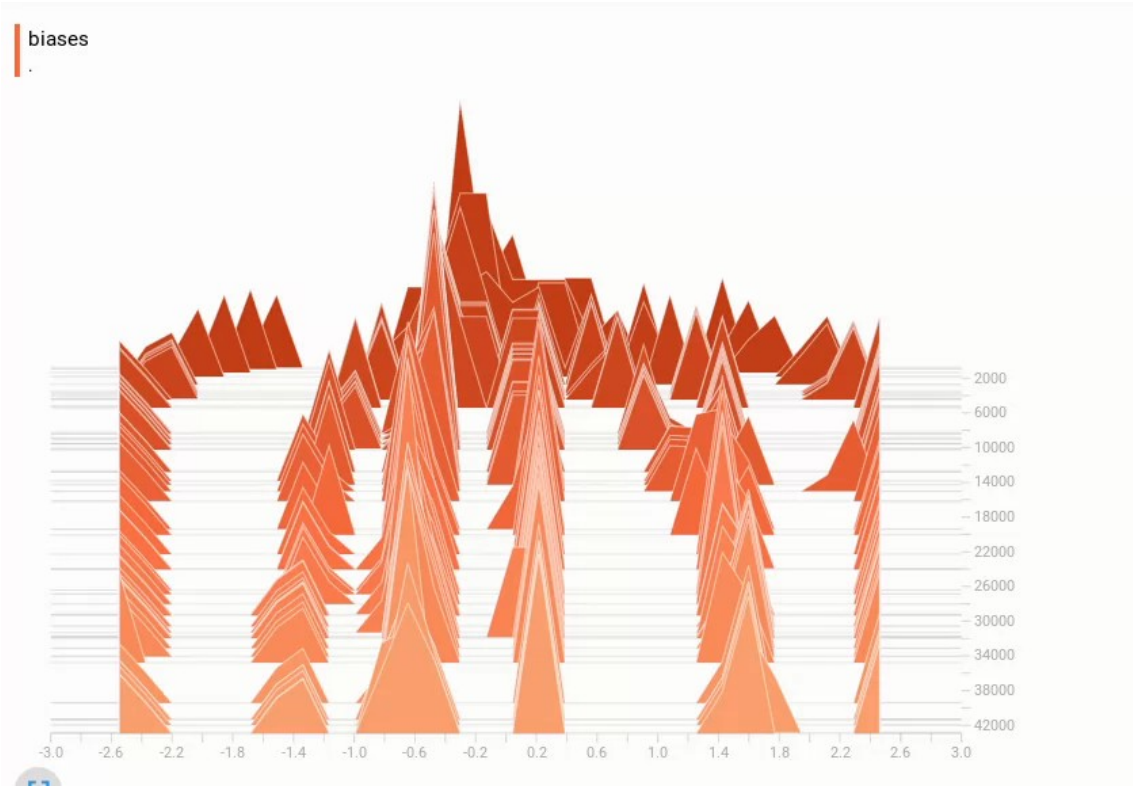




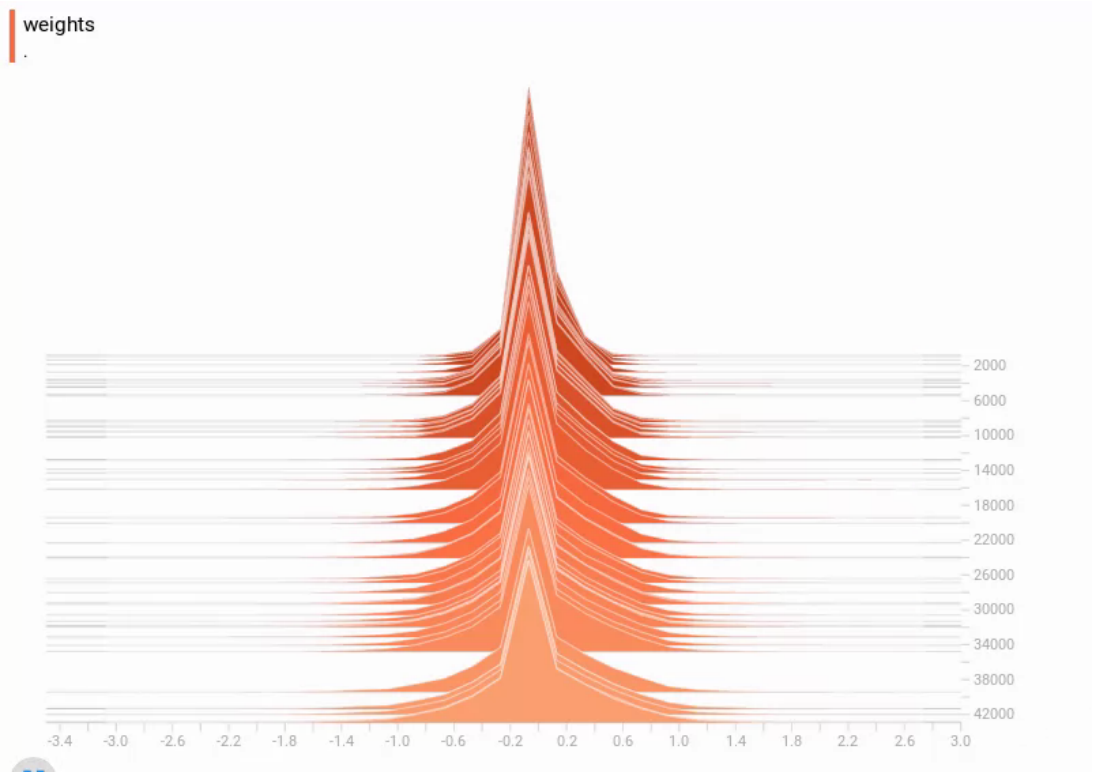
Here we can see the distribution of weights and biases over the data how they change with the change while training and optimise the model.

Histogram

Biases



Weights



Here We can see the histogram of biases and weights while we train the model

Reflection

While preprocessing the data choosing the training and testing set and making different files was quite challenging. Using the tensor graph was quite a fun that we can easily visualise the biases and weighs and also see how our model works in deep. Tuning the hyperparameter and using the softmax function was quite interesting that you can get different results with each and every hyperparamter and tuning them, making the batch size smaller and bigger, changing the learning rate and number of iterations. Choosing the best hyper parameters to gain best accuracy within sufficient time.

Improvement

We can make further improvements to this project we could use SVM with RBF kernels and different parameter values of C and gamma and optimising it using Grid Search and gain accuracy around 92% but we need much more computing power to do it in less time and it takes about a day on a quad-core system.

We could also use CNN with Keras with preprocessing data and gain ~99% accuracy.
