CapStone Project ¶

This project will try to create a convolution Neural network, train it with Street view house numbers and calculate the accuracy of predicting of house numbers

Data set http://ufldl.stanford.edu/housenumbers/ (http://ufldl.stanford.edu/housenumbers/)

1. Definition

Project Overview

In this world, it is very important fora computer to understand images. I mean, if a computer is able to make out a person from a image or easily learn how to do edge detection in order to find out objects, it will help solve a lot of real world issues as will automate a lot of things which are now manual for instance

1) Recognizing a dress from an image will help in recommending other dresses

2) A self driving car to understand images and objects from images

3) Understanding numbers or text from images will help people to understand languages which they do not understand

and there are numberous domain in which object recognition can be applied.

In this project, I will train a computer to understand house numbers (0-9) from images.

Project Origin

I took the dataset from the stanford site. There are a whole lot of house number images there. Seperate datasets are provided for training and test data. I will use that for training and testing my nueral network

DataSets

One can find the dataset here : http://ufldl.stanford.edu/housenumbers/ (http://ufldl.stanford.edu/housenumbers/) Training set : http://ufldl.stanford.edu/housenumbers/train_32x32.mat (http://ufldl.stanford.edu/housenumbers/train_32x32.mat) Test set : http://ufldl.stanford.edu/housenumbers/test_32x32.mat (http://ufldl.stanford.edu/housenumbers/test_32x32.mat)

Mat files are serialized files of images which can be loaded pretty quickly in python using scypy package. These images are 32x32x3 images Loading the .mat files creates 2 variables: X which is a 4-D matrix containing the images, and y which is a vector of class labels. To access the images, X(:,:,:,i) gives the i-th 32-by-32 RGB image, with class label y(i)

Problem Statement

I have 2 sets, one is the training set and the other is the test set of images. These have low resolution image files which have photos of house numbers in them. The numbers are in the range from 0-9. I will create a neural network which will identify these numbers from photos.

Task

Here are a set of task I will perform on the set of data

1) Read the image file(.mat) file in memory [both training and test set]

2) This will give me a dictionary of 2 variables, a 4D matrix of image and the other the class label

3) Extract the image and the class label in seperate variables

4) The image is a hXwXcxt [height, width, channel, total images].

5) I will transpose or reshape the image for simplifying calculation to a tXhXwXc [total images, height, width, channel

6) The class label will be from 0-9. It is seen that models do not give better results if numbers are used as it is very difficult for a computer to differentiate if use numbers. This is when one hot encoding comes into play.

7) Create a neural network [options, fullyconnected, convolutional]

8) Train it with train data

9) Find accuracy with the test data

10) Optimize the model to improve accuracy of the model with repeated training and tests.

Metrics

Accuracy will be the percentage of the success rate of predicting correctly. This is well good metrics as model will either correctly predict the house number or not. This is how I will be finding out the accuracy of the model

1) There are 10 output neurons to predict numbers from range 0-9

2) Since the neuron activation function is the softmax function, it will give output from range 0-1

3) All the output neurons with have a number from 0-1

4) The one with the max value, is the neuron that we will choose. Since each neuron represents a number, the number will be derieved from this.

5) Thus success is represented as 1 and failiure as 0 and we average it out to get the accuracy.

II. Analysis & III. Methodology

I will be explaining these 2 parts going through the code which will make us better understand it.

In [1]:

```
#import all the necessary libraries
import tensorflow as tf
import scipy.io as scp
import numpy as np
import random
from scipy import misc
import matplotlib.pyplot as plt
import datetime

#To plot the graphs in jupyter
%matplotlib inline
```

Load the training and the test data using the mat file. Also define the log path for tensorboard log files.

In [2]:

```
# Load the data
testData = scp.loadmat('../../data/svhn/test_32x32.mat')
trainData = scp.loadmat('../../data/svhn/train_32x32.mat')

logs_path = '/home/ubuntu/tensorFlowLogs/'
```

Extract the image data and the labels.

Issues and rectification

Since there are 3 channels the matrix values have a range from 0-255. The max value is 255. Initially when I did not normalize my training data, my model did not gave me good accuracy. Also the neural network converge much faster with normalization. To normalize I divided the data with 128 and subtracted it with 1 which will give me data in the range from (-1,-1)..

I do this for both, training and test data

In [3]:

```
testDataX = testData['X'].astype('float32') / 128.0 - 1
testDataY = testData['y']

trainDataX = trainData['X'].astype('float32') / 128.0 - 1
trainDataY = trainData['y']
```

Data exploration

Image size : 32x32x3 Total Training data : 73257 Total Test data : 26032

Shape of input data : 32x32x32x73257 Shape of output label : 73257x1

In [4]:

```
print type(trainDataX)
print type(trainDataY)

print 'train Data image shape : ', trainDataX.shape
print 'train data output shape : ', trainDataY.shape
print 'test data image shape : ', testDataX.shape
print 'test data output shape : ', testDataY.shape
```

```
<type 'numpy.ndarray'>
<type 'numpy.ndarray'>
train Data image shape :  (32, 32, 3, 73257)
train data output shape :  (73257, 1)
test data image shape :  (32, 32, 3, 26032)
test data output shape :  (26032, 1)
```

It will be very convinient for me to reshape the input data to a more convinient form. For this I will reshape the input data from 32x32x3x73257 to a form 73257x32x32x3

Though this is not necessary, for my convinience I am doing this.

In [5]:

```
# try tansposing the array
def transposeArray(data):
    xtrain = []
    trainLen = data.shape[3]
    for x in xrange(trainLen):
        xtrain.append(data[:,:,:,x])

    xtrain = np.asarray(xtrain)
    return xtrain
```

In [6]:

```
trainDataX = transposeArray(trainDataX)
testDataX = transposeArray(testDataX)


print 'New train data image shape : ', trainDataX.shape
```

```
New train data image shape :  (73257, 32, 32, 3)
```

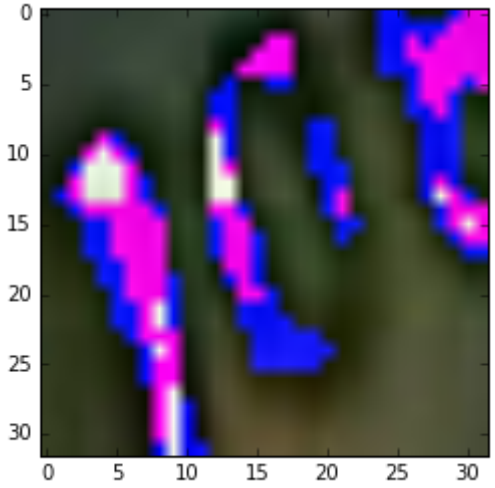
Randomly Chosing and showing images

If you look at the images, some of the images are so bad that even humans cannot correctly identify them

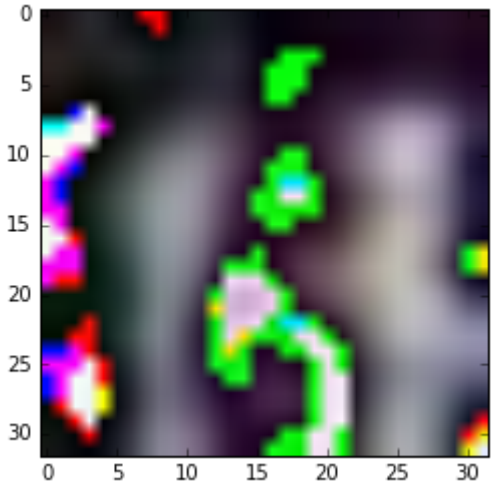
In [7]:

```
p = np.random.permutation(range(len(trainDataX)))
count = 0
for i in p:
    print 'Number : ', (trainData['y'])[i]
    plt.imshow(trainDataX[i])
    plt.show()
    count = count + 1
    if count >= 5:
        break;
```

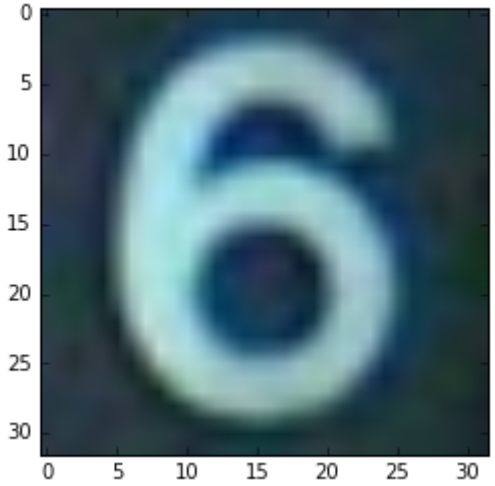
```
Number :  [10]
```



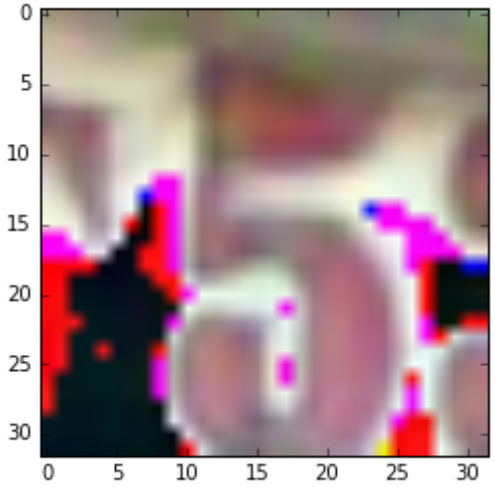

```
Number :  [6]
```



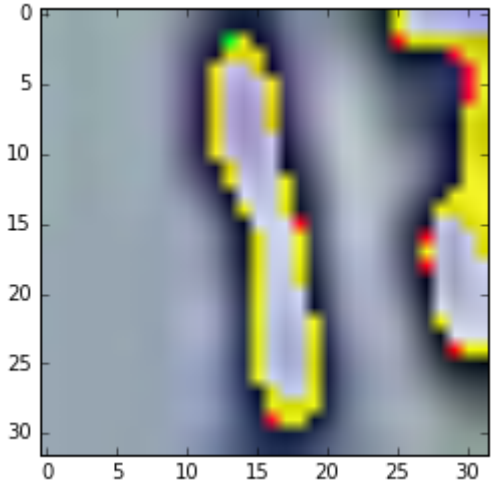

```
Number :  [6]
```



```
Number :  [5]
```



```
Number :  [1]
```



As I have already explained I will be using one hot encoding for better accuracy. I tried predicting without one hot encoding and got a poor accuracy. Thus using it

In [8]:

```
def OnehotEndoding(Y):
    Ytr=[]
    for el in Y:
        temp=np.zeros(10)
        if el==10:
            temp[0]=1
        elif el==1:
            temp[1]=1
        elif el==2:
            temp[2]=1
        elif el==3:
            temp[3]=1
        elif el==4:
            temp[4]=1
        elif el==5:
            temp[5]=1
        elif el==6:
            temp[6]=1
        elif el==7:
            temp[7]=1
        elif el==8:
            temp[8]=1
        elif el==9:
            temp[9]=1
        Ytr.append(temp)

    return np.asarray(Ytr)
```

Converting the label to one hot encoding as the prediction really improves. This make sense as well.

In [9]:

```
# convert y to one hot encoding
trainDataY = OnehotEndoding(trainDataY)
testDataY = OnehotEndoding(testDataY)
print 'train data output shape : ', trainDataY.shape
print 'test data output shape : ', testDataY.shape
```

```
train data output shape :  (73257, 10)
test data output shape :  (26032, 10)
```

Here comes the inputs to the neural network that I have designed. I wll explain the neural network at the later stage where I have defined that in tensorflow. These are the parameters I will be using. I have tried with different options and these are the ones with which I got a good accuracy.

Iamge height : 32

Width : 32

Channel : 3 (R,G,B)

tags : number of output labels : (0-9) = 10

patch : Convolution kernel size, I will use a 5x5 kernel

Depth : Total number of kernels I will use in a convolution : 16

num_hidden : These are the number of hidden neurons I will use in the last hidden layer : 128 [FYI : I have used 64 earlier but with 128 got a better accuracy]

dropout : To prevent overfitting, I am using dropout probability: .75 [FYI : Used .50 as well, but this gave better accuracy]

Learning rate : This is the rate at which the weight parameters learn for the neurons. [FYI : :Large learning rate was giving me nan, reducing the learning rate helped in converging the model]

In [10]:

```
#Neural network parameters
height = 32
width = 32
channel = 3
tags = 10
patch = 5
depth = 16
num_hidden = 128
dropout = 0.75 # Dropout, probability to keep units

learning_rate = 1e-4
```

In [11]:

```
stddev = 1e-1
tf_X = tf.placeholder("float", shape=[None, height, width, channel], name = "X-Input")
tf_Y = tf.placeholder("float", shape=[None, tags], name = "LabeledData")

convW1 = tf.Variable(tf.random_normal([patch, patch, channel, depth], stddev=stddev), name=
bias1 = tf.Variable(tf.random_normal([depth], stddev=stddev), name="Bias1")

convW2 = tf.Variable(tf.random_normal([patch, patch, depth, depth], stddev=stddev), name="C
bias2 = tf.Variable(tf.random_normal([depth], stddev=stddev), name = "Bias2")

w3 = tf.Variable(tf.random_normal([height // 4 * width // 4 * depth, num_hidden], stddev=st
bias3 = tf.Variable(tf.random_normal([num_hidden]), name="bias3")

w4 = tf.Variable(tf.random_normal([num_hidden, tags], stddev=stddev), name="w4")
bias4 = tf.Variable(tf.random_normal([tags], stddev=stddev), name="bias4")

keep_prob = tf.placeholder(tf.float32) #dropout (keep probability)
```

Here is the model I have tried to build

Input Image : 32x32x3

Reason for choosing Convolution Network

when I started this project I used a fully connected model which was giving me a accuracy of arround 40% taking hude time to learn. So why does a fully connected network gave such a bad accuracy. I found out that for images, there is a strong correlation between pixels in images. Now for a fully connected network, if we convert it to a 1D input, it will somewhat break this correlation. Here is my question on correlation of images on stackoverflow http://stackoverflow.com/questions/37546491/how-does-a-neural-network-work-with-correlated-image-data?noredirect=1#comment62601217_37546491 (http://stackoverflow.com/questions/37546491/how-does-a-neural-network-work-with-correlated-image-data?noredirect=1#comment62601217_37546491)

Reading a lot of articles, I found out that for images, convolution netowrk is most suited. This is because it filters out the image for edge detection at the initial layers before converting it to a 1D array of pixels.

Let me define each layer one by one first Convolution Hidden layer : 5x5x3x16

Edge Detection

Here the kernel size is 5x5. I can either use a 3x3 or a 5x5. 5x5 did a better job. It will filter out the data for edge detection. With 16 kernel size it will give a 16x16x16 output with SAME padding and 1x1 stride.

Padding : Same, Stride : [1,1,1,1]

Also using Relu which will give me output from 0-x where x is the input.

Output of first Convolution Hidden layer : 32x32x16

Pooling : Pooling is important because it will recude the size of the image, taking the most highlighted pixel from the set of pixel. From a set of pixel we will take the most outstanding pixel using max pool.

Pooling : max, Stride : [2,2] : Output : 16x16x16

Second convolution works the same ways as the first one reducing the height and width of the image further.

Second Convolution Hidden layer : 5x5x16x16

Padding : Same, Stride : [1,1,1,1]

Output of Second Convolution Hidden layer : 16x16x16

Pooling : max, Stride : [2,2] : Output : 8x8x16

This is the fully connected layer which will reshape the output of the earlierlayer to 1 dimentional.

third Hidden Layer fully connected : 8x8x16

Output of third Hidden layer : 128

This layer will be connected to the output neurons.

Fourth Hidden Layer : 128 x 10

In [12]:

```
#model

def model(X):

    #first layer : Convolution
    conv = tf.nn.conv2d(X, convW1, [1,1,1,1], padding='SAME')
    hidden1 = tf.nn.relu(conv + bias1)

    #second layer : pooling
    hidden2 = tf.nn.max_pool(hidden1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SA

    #third layer : convolution
    conv2 = tf.nn.conv2d(hidden2, convW2, [1,1,1,1], padding='SAME')
    hidden3 = tf.nn.relu(conv2 + bias2)

    #fourth layer : pooling
    hidden4 = tf.nn.max_pool(hidden3, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SA

    #reshape it to a single Dimensional
    shape = hidden4.get_shape()

    #5th layer : fully connected
    newInput = tf.reshape(hidden4, [-1, shape[1].value * shape[2].value * shape[3].value])
    hidden5 = tf.nn.relu(tf.matmul(newInput, w3) + bias3)

    dp5 = tf.nn.dropout(hidden5, keep_prob)

    return tf.matmul(dp5, w4) + bias4
```

In [13]:

```
with tf.name_scope('Model'):
    pred = model(tf_X)
with tf.name_scope('loss'):
    loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(pred, tf_Y))


# Optimizer.
with tf.name_scope('AdamOptimizer'):
    optimizer = tf.train.AdamOptimizer(learning_rate).minimize(loss)

with tf.name_scope('accuracy'):
    accuracy = tf.reduce_mean(tf.cast(tf.equal(tf.argmax(tf.nn.softmax(pred),1),tf.argmax(t

# Create a summary to monitor cost tensor
tf.scalar_summary("loss", loss)
# Create a summary to monitor accuracy tensor
tf.scalar_summary("accuracy", accuracy)
# Merge all summaries into a single op
merged_summary_op = tf.merge_all_summaries()
```

In [14]:

```
def Accuracy(X, Y, message, sess):
    print message, sess.run(accuracy, feed_dict= {tf_X: X, tf_Y: Y, keep_prob:1.0})
```

In [ ]:

```
print  'started at : ', str(datetime.datetime.now())
with tf.Session() as sess:
    tf.initialize_all_variables().run()

    # op to write logs to Tensorboard
    summary_writer = tf.train.SummaryWriter(logs_path, graph=tf.get_default_graph())

    epoch = 20000
    batch_size = 64
    print('Initialized')

    p = np.random.permutation(range(len(trainDataX)))
    trX, trY = trainDataX[p], trainDataY[p]
    start = 0
    end = 0

    for step in range(epoch):
        start = end
        end = start + batch_size

        if start >= len(trainDataX):
            start = 0
            end = start + batch_size

        if end >= len(trainDataX):
            end = len(trainDataX) - 1
        if start == end:
            start = 0
            end = start + batch_size

        #print step, start, end

        #batch = np.random.choice(len(trainDataX) - 1, batch_size)
        inX, outY = trX[start:end], trY[start:end]
        _, summary = sess.run([optimizer, merged_summary_op], feed_dict= {tf_X: inX, tf_Y:
        summary_writer.add_summary(summary, step)

        if step % 500 == 0:
            print 'cost at each step :', step, 'is :', sess.run(loss, feed_dict={tf_X: inX,

    #Accuracy(trX, trY, 'accuracy of training data : ', sess)
    Accuracy(testDataX, testDataY, 'accuracy of test data : ', sess)

print  'Ended at : ', str(datetime.datetime.now())
```

In [17]:

```
def DrawResult(loss, accuracy):
    fig = plt.figure(figsize=(10,10))
    lossImg = misc.imread(loss)
    accuracyImg = misc.imread(accuracy)

    plt.imshow(lossImg)
    plt.show()

    fig = plt.figure(figsize=(10,10))
    plt.imshow(accuracyImg)
    plt.show()
```

Results

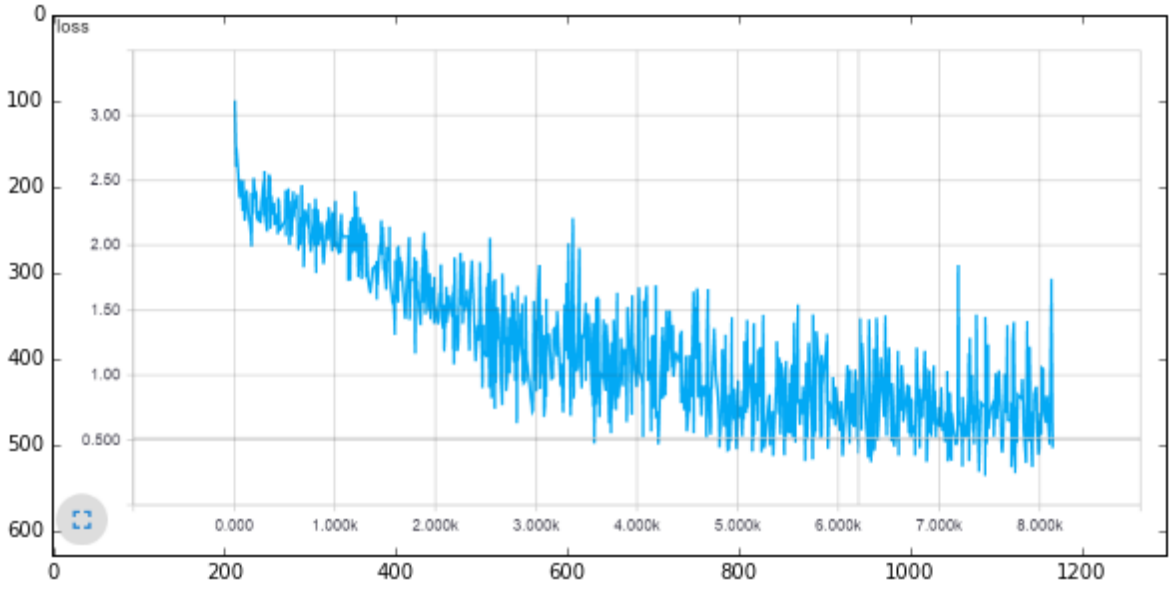
8 core 32 gb ram machine

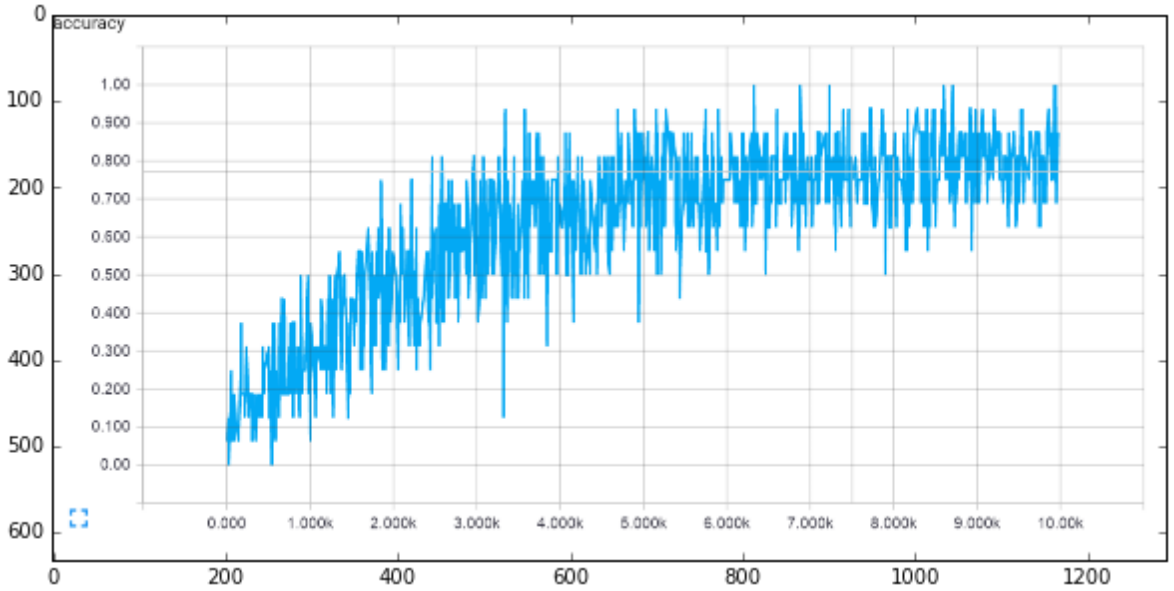
Epoch : 10000 Batch : 16

Total Time : 7 min Accuracy on Test set : 82%

In [18]:

```
DrawResult('loss1.PNG', 'accuracy1.PNG')
```





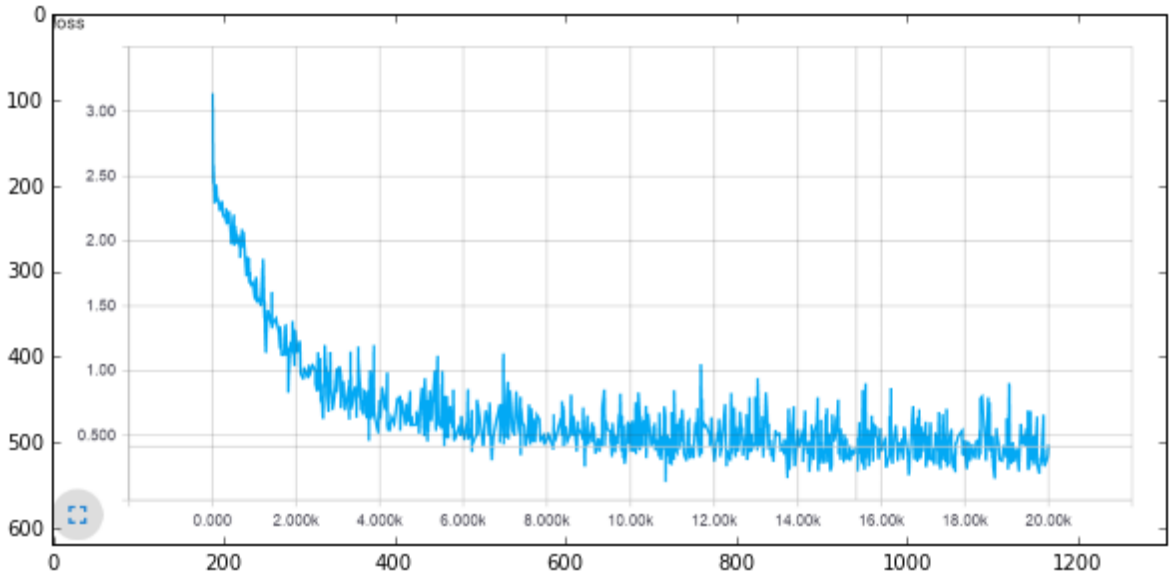
Epoch : 20000 Batch : 64

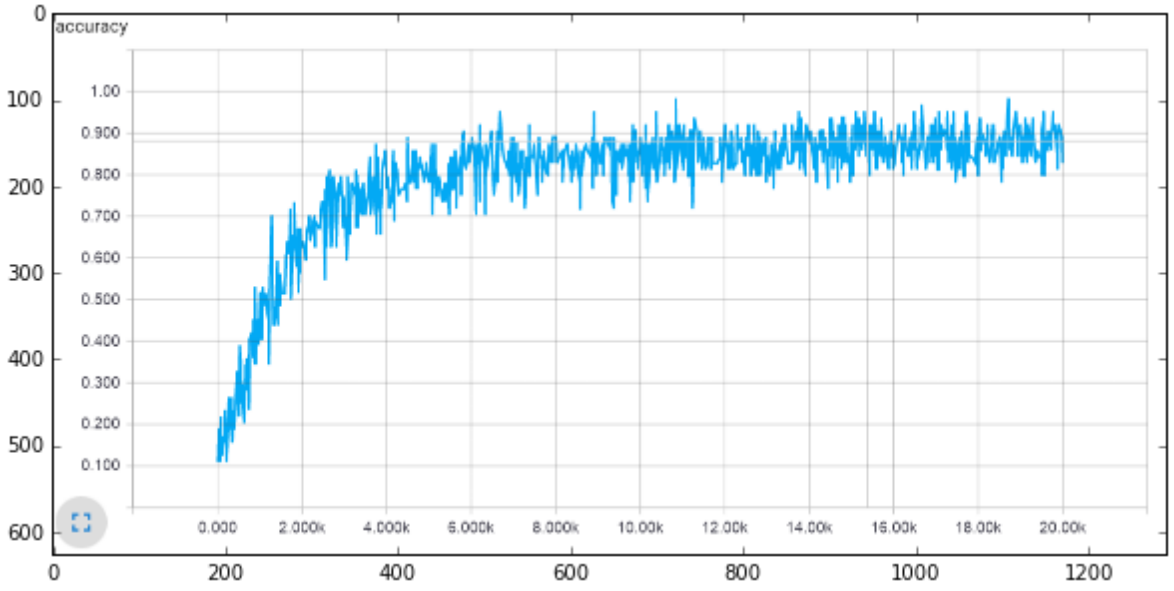
Total Time : 38 min

Accuracy on Test set : 87%

In [20]:

```
DrawResult('loss2.PNG', 'accuracy2.PNG')
```



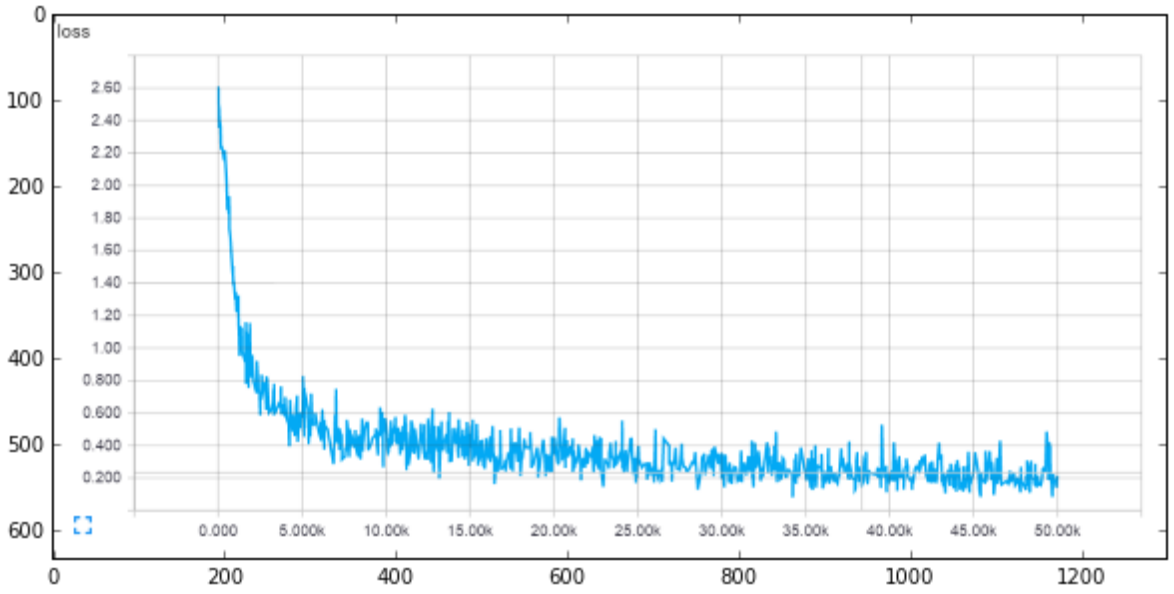


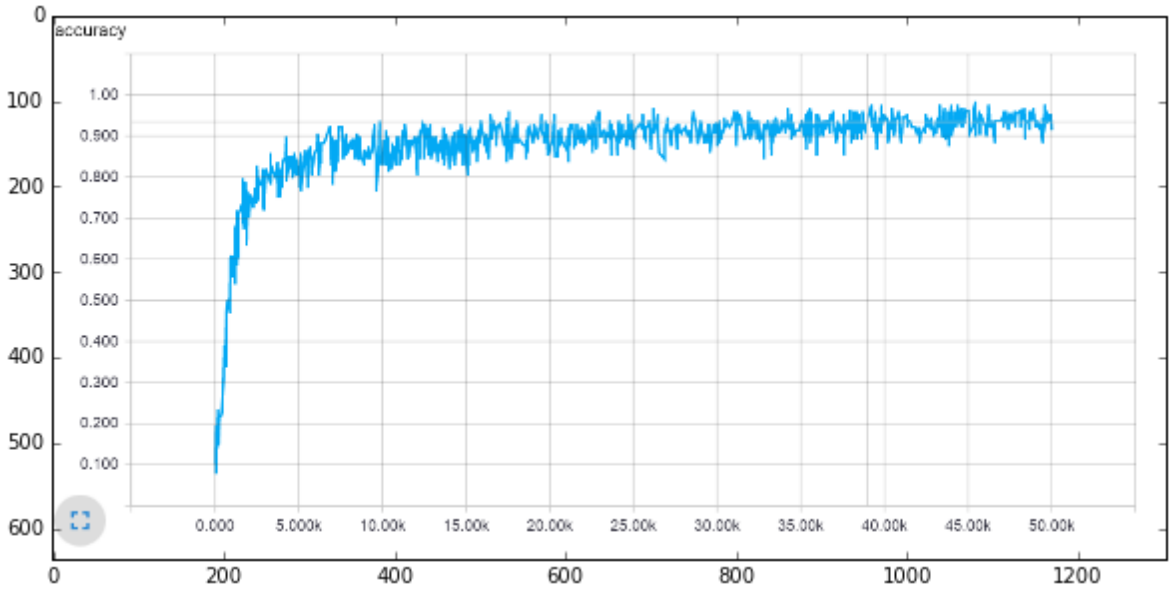
Epoch : 50000 Batch : 128

Total Time : 120 min Accuracy on Test set : 90%

In [22]:

```
DrawResult('loss3.PNG', 'accuracy3.PNG')
```





Conclusion

I am using stochoistic gradient decent. Since I do not have a GPU, therefore small batch is used to train the model. Batch gradient desent will take days on my machine to learn.

Even then, Stochoistic gradient decent is giving a very good accuracy. With the increase in batch size, accuracy will also increase, but will not be in that proportion.

Also I tried increasing the hidden layers, to see if I can get a better accuracy, accuracy increased marginal and took a large time to train.

This is what I conclude and learned from this project

1) Stochoistic gradient decent give comparable accuracy as batch one

2) More the hidden layers, more training time for the model.

3) Convolutions are the models to choose when correlation data is there like images, audio clips etc

In [ ]:

```
```