

FINAL YEAR PROJECT DOCUMENTATION

TExNet: TUNING EXISTENTIAL NETWORK FOR HIGH RESOLUTION EDGE DETECTION

DONE BY: ZAID BAIG (160071601156)

S SREYAS (160071601137)

TABLE OF CONTENTS

1. Abstract
2. Introduction
3. Literature Survey
4. Algorithm
 - a. Callbacks
 - b. Convolutions
 - c. Data Augmentation
 - d. Transfer Learning
 - e. Multiclass Classification
5. Model Motivation
6. TExNet Image Generator Code
7. TExNet Architecture Code
 - a. Program Code
 - b. Program Summary Output
8. TExNet Learning Rate Definition Code
9. TExNet Learning Rate Definition Output
10. TExNet Model Compilation/Training Code
 - a. Training Methodology
11. Test Analysis
 - a. Model History Log 1
 - b. Model History Log 2
 - c. Model History Log 3
 - d. Model History Log 4
 - e. Model History Log 5
 - f. Model History Log 6
 - g. Model History Log 7
 - h. Model History Log 8
12. Comparison
13. Future Work
14. Conclusion
15. Reference

Abstract

In video-based object detection for either in self-driving or in video surveillance, the issue we practically face in this area is to deal with very deep convolutional neural networks (CNN). Very deep CNNs do a great job in detecting feature sets at different layers of abstractions, right from the edges (close to the shallow layer) to very complex features of the object (nearing the output layer). So deeper the neural network results in the risk of an issue known as vanishing gradients problem, to tackle this issue the concept of inclusion of residual blocks to the neural network architecture was proposed. This helped give rise to the concept of skip connections which help in preventing the gradients to go to zero during the training process. Another issue with deep CNNs is computational cost, which normally occurs due to the number of hyperparameters which the neural network has to train, with inception network we would be able to reduce the number of hyperparameters by 10 folds if we were to work with a 1×1 convolutional feature set. In this proposed work, TExNet is aimed to incorporate the advantages of both, the Inception model and the ResNet model, and come up with a nested Resnet-Inception model which will be able to produce results with much higher accuracy and can work with a much deeper CNN model. It makes use of the concept of concatenating multiple CNN feature set results and adding a residual block on top of it. The architecture of the model will comprise of a nested Residual block which will be applied both, inside the inception cell and outside running parallelly between 2 inception cells. The proposed work is aimed to preserve the most minute features, which may be lost in traditional architecture during the training process and help in increasing the number of class labels it would be able to detect. The proposed model will be tested mainly under GPU computing environment and tested using TensorFlow 1.1x.

Introduction

There has been a significant rise to the number of neural network architectures which the deep learning community had offered us in the field of computer vision. The most common feature which is present in most of today's well known neural network architecture is the inclusion of the concept of using convolutional in their respective architecture, for the concept of convolution was first introduced in 1998 in the LeNet architecture where the paper proposed the concept of reducing the image size with the help of filters and extracting primary features like edges, lines and other predominant features of the object to be detected. The LeNet architecture was mainly developed for scanning of handwritten documents. The convolutional layer mainly comprises these feature matrices that help in extracting the predominant features from the image and leaving behind the lesser ones. This concept of the LeNet architecture inspired the creation of all other neural network architecture which have been made today. The same concept is applied on the AlexNet architecture where it consists of a deeper neural network and helps in recognizing more objects. The Google Net architecture used the same concept of convolutions but instead of computing in a sequential order, the convolutional and reduction phase were done parallelly and the result of those were concatenated and sent to the next step. The overall neural network size of the Google Net architecture was significantly larger than the AlexNet or the VGG network, as VGG-16 just represents a neural network architecture with 16 layers of convolution layers. Now as the introduction of much deeper neural networks are being developed, there is a rising problem of what is known as vanishing gradients. To further explain what vanishing gradient, as we train a neural network model there are two major steps involved, i.e, the forward propagation step and the backward propagation step. The forward propagation step will initially compute the result of the given input and then this initial result is compared with the original result using a loss function, this function then tells the neural network of whether the parameter values should increase or decrease in accordance to the learning rate model and when the parameters of the neural network. Usually the parameter values of the neural network are mostly between 0 and 1, hence the vanishing gradients issue arises when these parameters go ever so close to 0 that it becomes meaningless when being computed during the forward propagation step and when it hits zero the future training process of gradient descent will become meaningless as there will be zero change in the parameters that have hit zero, as the learning rate to directly dependent on the parameter being non-zero.

Literature Survey

Perceiving Motion from Dynamic Memory for Vehicle Detection in Surveillance Videos - Wei Liu, Shengcai Liao*, Senior Member, IEEE, and Weidong Hu (2018)

This paper proposed in the improvement of how memory modules can affect the process of object detection in videos.

Single Image Super-Resolution Using Deep CNN with Dense Skip Connections and Inception-ResNet - Chao Chen and Feng Qi (2018)

The paper shows that the performance of single-image super-resolution methods can be significantly boosted by using deep convolutional neural networks. In this study, it presents a novel single-image super-resolution method by introducing dense skip connections in a very deep network. In the proposed network, the feature maps of each layer are propagated into all subsequent layers, providing an effective way to combine the low-level features and high-level features to boost the reconstruction performance. In addition, the dense skip connections in the network enable short paths to be built directly from the output to each layer, alleviating the vanishing-gradient problem of very deep networks. Moreover, deconvolution layers are integrated into the network to learn the up-sampling filters and to speed up the reconstruction process. The proposed method substantially reduces the number of parameters, enhancing the computational efficiency. It evaluates the proposed method using images from four benchmark datasets and sets a new state of the art.

Deep Residual Learning for Image Recognition - Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun (2017)

This paper introduced a new technique or a new tool kit which developers and researchers can use to build an effective neural network architecture. It solves the traditional problems of vanishing gradients by making sure the output from each layer is either greater than or lower than the previous result of the previous activation layer or equal to the result of the previous activation layer.

Going Deeper with Convolutions - Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich (2014)

The origin of this paper was inspired by the movie named inception. It proposes a simple twist to the already existing convolutional neural network architecture by combining and running multiple filters parallelly at the same step and concatenating the result and repeating the process. This method had a positive impact on the accuracy of the model and also gave the needed stability to support a very deep neural network architecture. But this result came at the cost of an increase in the number of hyperparameters and ultimately increasing the computation cost overall. To bring down the hyperparameters the concept of one by one convolutional filters is used, which brought the number of hyperparameters to be tuned by about ten folds.

Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning- Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, Alex Alemi (2017).

The following paper was aimed at combining the perks of the inception network and the Resnet network and came up with a neural network architecture which is supposedly the best architecture for running very deep convolutional neural networks.

Network in Network - Min Lin, Qiang Chen, Shuicheng Yan (2014)

This paper proposed an efficient way of computing with gradients and helped in the reduction of the number of hyperparameters being used. The main idea was basically to make an unorthodox approach to the traditional method of calculating the result of a convolutional filter. By making use of a one by one convolutional filter the model will not drop performance, but will in turn have a massive improvement in computational cost. A slight note to be taken in this is that, the filter will not help in reducing the size of input layer but for each filter step should be followed by a pooling step which will help in the reduction of the input size.

Gradient Based Learning Applied to Document Recognition - Yann LeCun Leon Bottou Yoshua Bengio and Patrick Haffner (1998)

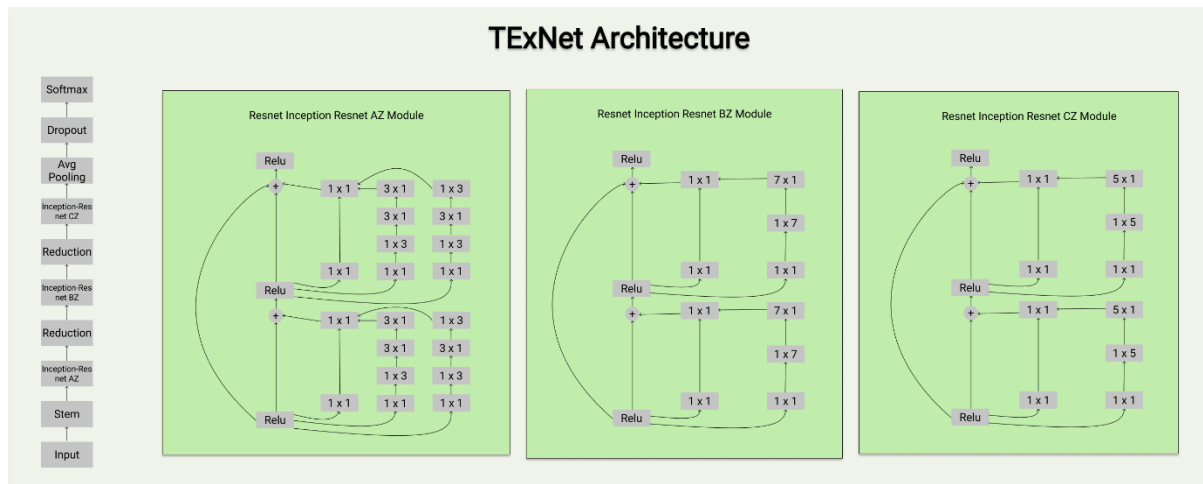
This paper is regarded as the beginning of the field of computer vision. It is the first paper that introduced the concept of feature detection using convolutional neural networks, where the input is convolved and passed through a learning step involving a forward step and a backward step. The forward step tries to compute and get the result for the given filter value and input, then depending on the result a loss is computed to indicate the direction of change that has to be made on the filter values by a constant learning rate. The change in the gradients is performed in the backward step. This process is repeated several times till the model gives a satisfying result.

Residual Inception - Xingpeng Zhang Sheng Huang Xiaohong Zhang Wei Wang Qiuli Wang Dan Yang (2018)

The core idea of ResNet is introducing a so-called “identity shortcut connection” that skips one or more layers. convolutional neural network (CNN) models emerged, several tasks in computer vision have actively deployed CNN models for feature extraction. However, the conventional CNN models have a high computational cost and require high memory capacity, which is impractical and unaffordable for commercial applications such as real-time on-road object detection on embedded boards or mobile platforms. To tackle this limitation of CNN models, this paper proposes a wide-residual-inception (WR-Inception) network, which constructs the architecture based on a residual inception unit that captures objects of various sizes on the same feature map, as well as shallower and wider layers, compared to state-of-the-art networks like ResNet.

Algorithm

The proposed TExNet neural network model is built on a non-conventional custom inheritance method that iteratively builds on the original algorithm. The following is a functional structure of the proposed TExNet model.



Algorithm explanation:

1. TExNet model arguments
 - a. A 3 channel 299x299 pixel image data

```
def Texnet(input_shape=(299,299,3), classes=3):
    X_input = Input(input_shape)
    X = Conv2D(32,(7,7),strides = (2,2))(X_input)
    X = BatchNormalization(axis = 3)(X)
    X = Activation('relu')(X)
    X = MaxPooling2D((3,3),strides=(2,2))(X)
    X = Inception_Resnet_AZ(X,[16,32,64,128])
    X = MaxPooling2D((3,3),strides=(2,2))(X)
    X = Inception_Resnet_BZ(X,[64,128,128,256])
    X = MaxPooling2D((3,3),strides=(2,2))(X)
    X = Inception_Resnet_CZ(X,[64,128,256,256])
    X = AveragePooling2D((4,4))(X)
    X = Flatten()(X)
    X = Dense(classes, activation = 'softmax')(X)
    model = Model(inputs=X_input,outputs = X,name='TExNet')
    return model
```

- b. An integer dimension of the output space
2. Algorithm structure
 - a. Initial convolution layer

```

X_input = Input(input_shape)
X = Conv2D(32, (7,7), strides = (2,2))(X_input)
X = BatchNormalization(axis = 3)(X)
X = Activation('relu')(X)
X = MaxPooling2D((3,3), strides=(2,2))(X)

```

b. Residual block between inception cells

```

def Inception_Resnet_CZ(X, filters):
    F1, F2, F3, F4 = filters
    print(X.shape)
    X_shortcut_1 = X
    X_shortcut_2 = X

    x1, x2 = Inception_resnet_cz_block(X, filters)

    X = tf.keras.layers.Concatenate(axis=3)([x1, x2])
    X_shortcut_1 = Conv2D(filter_temp_, (1,1), activation = 'relu')(X_shortcut_1)

    X = Add()(X, X_shortcut_1)

    x1, x2 = Inception_resnet_cz_block(X, filters)
    X = tf.keras.layers.Concatenate(axis=3)([x1, x2])

    X_shortcut_2 = Conv2D(filter_temp_, (1,1), activation = 'relu')(X_shortcut_2)

    X = Add()(X, X_shortcut_2)
    return X

```

c. Residual block within inception cells

```

def Inception_resnet_cz_block(X, filters):
    F1, F2, F3, F4 = filters

    x1a = Conv2D(F1, (1,1))(X)
    x1a = BatchNormalization(axis=3)(x1a)
    x1a = Activation('relu')(x1a)
    x2a = Conv2D(F1, (1,1))(X)
    x2a = BatchNormalization(axis=3)(x2a)
    x2a = Activation('relu')(x2a)

    x2b = Conv2D(F2, (1,7), padding='same')(x2a)
    x2b = BatchNormalization(axis=3)(x2b)
    x2b = Activation('relu')(x2b)

    x2c = Conv2D(F3, (7,1), padding='same')(x2b)
    x2c = BatchNormalization(axis=3)(x2c)
    x2c = Activation('relu')(x2c)

    x1 = Conv2D(F4, (1,1), activation = 'relu')(x1a)
    x2 = Conv2D(F4, (1,1), activation = 'relu')(x2c)

    return x1, x2

```

d. Final classification step


```

for i in range(5):
    X = Inception_Resnet_AZ(X,[8,16,32,64])
X = MaxPooling2D((3,3),strides=(2,2))(X)
for i in range(10):
    X = Inception_Resnet_BZ(X,[8,16,32,64])
X = MaxPooling2D((3,3),strides=(2,2))(X)
for i in range(5):
    X = Inception_Resnet_CZ(X,[8,16,32,64])
X = AveragePooling2D((4,4))(X)
X = Flatten()(X)
X = Dense(classes, activation = 'softmax')(X)
model = Model(inputs=X_input,outputs = X,name='TExNet')

```

Callbacks

The training process of the model does not incorporate termination of the training process when the model attains a human level accuracy in both the training set and the validation set. As the resulting curve of the training and validation accuracy is not exponential or the curve does not produce a standard linear function. The callback termination functional call, will disrupt the training process and will not be the optimal way through which the model's efficiency can be determined. The only callback function call used during the model analysis is the csv callback function which records the model's accuracy and loss function score in every epoch into a csv file. These files are then saved to get further analysis of the benefactor of the model and what are the downsides of it.

Convolutions

The usage of Residual networks has not only made the deep neural network a bit better but has also increased the overall path length of the neural network, in order to compute and produce one epoch the model takes a whopping 70-80 seconds. Even though the model has only 20 million parameters which is far less than the inception v4 neural network, the inception neural network takes 35-45 seconds at each epoch for the same dataset. By path increase it technically means that there are more computations taking place at each layer than when compared to the computation on a relatively huge trainable matrix.

Data Augmentation

The datasets used in evaluation of the TExNet model are all in .jpg files which undergo data augmentation via ImageDataGenerator function of Keras library.

rescale = 1./255, each image is rescaled in 255 pixel ratio, the resolution of the image is multiplied by 1/255. The reason why we convert to 1/255 basis is because the pixel value in gray scale ranges from 0 to 255, where 0 is black and 255 is white. The same applies for RGB images where each color has a 0 to 255 range for each individual color and the dimension is 3, as in (R(0-255),G(0-255),B(0-255)).

Rotation_range=40, the rotation function rotates the image at a particular angle to give the dataset some randomness in the viewpoint of the data in multiple angles.

Width_shift_range=0.2, this function basically shifts the image by a fixed value in the horizontal direction. Does similar functionality in providing randomness by shifting the image in the horizontal direction.

Height_shift_range=0.2, Similar to the width shift range function

Shear_range=0.2, turns the image by clockwise or anti clockwise direction by a certain angle depending on the value specified between -1 to 1.

Zoom_range=0.2, It zooms the image by the zoom percent, in this case the zoom percent mentioned is 0-20% of the overall image. At max 20% of the image is zoomed in to provide an enlarged/different view of the image.

Horizontal_flip=True, this functionality will generate the mirror reflection of the image

Fill_mode='nearest'. Here we have different options we can experiment on how the void portions of the image should be filled. There are 4 options, ['constant', 'nearest', 'reflect', 'wrap']; the fill mode selected is nearest where the pixel closest to the void is naturally replicated. For example, 'xxx|xyz|'zzz'; where the quoted letters are

target size = 299,299, This basically reshapes the image resolution to our preference. It is an extremely useful functionality in the basis of applying transfer learning where the model is pre trained with a standard image pixel

batch size = 2. This helps in providing the given data set in the form of batches that can be sent to the model for training and depending on the size of the number of batches it would have a direct impact on the training of the model.

The above functionalities help in increasing the number of images by a certain amount. It helps in training the model to train certain images at various different angles and perspectives.

Transfer Learning

It is mainly used in training and testing other existing models, as recreating those existing models is extremely time consuming and a waste of effort. Through the process of transfer learning, the model's tail nodes can be modified to adapt to the dataset that is being used for the testing and validation.

Multiclass classification

The model is tested using 2 data sets, one of them being the rock-paper-scissors dataset where the dataset consists of almost 300 images of 100 of it from each category. The TExNet model works fairly better for this dataset by showing a 1% improvement when compared with the inception v4 model. The other dataset used is the Digits MNIST dataset where the data is classified into 10 categories ranging from 0 to 9 handwritten images where each category consists of almost 4000 images adding up a total of 32,000 training images and 8,000 validation/testing images.

Model motivation:

1: ResNet has been a rising model in the field of deep neural networks considering its approach towards combining layers of the neural network (NN). The ResNet model uses a concept called residual block which bypasses or in other words fast-forwards the output of the previous layer by skip connection method and passes it to deeper layers of the neural network. This approach of ResNet model has been inherited for the proposed TExNet model, which uses the concept of residual blocks for inclusion of deeper neural layers without the loss of information that usually degrades over layers in conventional large deep neural network models, where the backpropagation of the gradient degrades to infinitesimally small values that the model fails to capture resulting in a saturated or overfitted model.

2: Secondly, the success of the Inception model has been in the introduction of 1x1 convolution preceding the conventional NxN convolution, which reduces the number of input channels required to tune the model altogether. This reduction in dimension hugely succeeded in reducing the parameter requirement for the model as a whole by 10-20 folds. This idea of Inception model has also

been inherited into the proposed TExNet model which acts as a catalyst for the aim of achieving higher accuracy models with finer performance.

Model structure and function:

The proposed TExNet model comprises 2 primary components, an independent residual block that connects multiple inception cells together and a transformed inception cell that consists of residual block(s) within it. The purpose of having a residual block and inception cell have already been explained in earlier parts. This part will explain the purpose of this specific order of arrangement. As residual blocks bypass the neural layers that are connected around it, wrapping the inception cells around a residual block will help in faster propagation of information with lesser loss factor. Similarly, for better time-based performance, individual inception cells are embedded with residual blocks theoretically increasing the performance in terms of training time and accuracy for a given number of epochs. The proposed TExNet model is expected to have a decreased gradient factor for larger deep neural networks for better accuracy with particularly a trade-off against run-time caused by relatively deeper networks.

TExNet Image Generator Code:

```
train_dir          =          "D:/Program          Files          -
Development/Data/trainingSet/trainingSet/"
train_datagen = ImageDataGenerator(rescale = 1./255,
                                   rotation_range=40,
                                   width_shift_range=0.2,
                                   height_shift_range=0.2,
                                   shear_range=0.2,
                                   zoom_range=0.2,
                                   horizontal_flip=True,
                                   fill_mode='nearest')

train_generator = train_datagen.flow_from_directory(train_dir,
                                                    batch_size=8,

target_size=(299,299))

test_dir          =          "D:/Program          Files          -
Development/Data/trainingSample/trainingSample/"
test_datagen = ImageDataGenerator(rescale = 1./255,
                                   rotation_range=40,
                                   width_shift_range=0.2,
                                   height_shift_range=0.2,
                                   shear_range=0.2,
                                   zoom_range=0.2,
                                   horizontal_flip=True,
                                   fill_mode='nearest')

test_generator = test_datagen.flow_from_directory(test_dir,
                                                  batch_size=8,

target_size=(299,299))
Found 32800 images belonging to 10 classes.
Found 9200 images belonging to 10 classes.
```

TExNet Architecture Code

Program Code

```
filter_temp = 192 #398
filter_temp_ = 128 #512

def Inception_Resnet_AZ(X, filters, stage=1, block=1):

    F1, F2, F3, F4 = filters
    print(X.shape)
    X_shortcut_1 = X
    X_shortcut_2 = X

    x1, x2, x3 = Inception_resnet_az_block(X, filters)

    X = tf.keras.layers.Concatenate(axis=3) ([x1, x2, x3])
    X_shortcut_1 = Conv2D(filter_temp, (1, 1), activation =
'relu')(X_shortcut_1)

    X = Add() ([X, X_shortcut_1])

    x1, x2, x3 = Inception_resnet_az_block(X, filters)
    X = tf.keras.layers.Concatenate(axis=3) ([x1, x2, x3])

    X_shortcut_2 = Conv2D(filter_temp, (1, 1), activation =
'relu')(X_shortcut_2)

    X = Add() ([X, X_shortcut_2])

    return X

def Inception_resnet_az_block(X, filters):

    F1, F2, F3, F4 = filters

    x1a = Conv2D(F1, (1, 1)) (X)
    x1a = BatchNormalization(axis=3) (x1a)
    x1a = Activation('relu') (x1a)
    x2a = Conv2D(F1, (1, 1)) (X)
    x2a = BatchNormalization(axis=3) (x2a)
    x2a = Activation('relu') (x2a)
    x3a = Conv2D(F1, (1, 1)) (X)
    x3a = BatchNormalization(axis=3) (x3a)
    x3a = Activation('relu') (x3a)

    x2b = Conv2D(F2, (1, 3), padding='same') (x2a)
    x2b = BatchNormalization(axis=3) (x2b)
    x2b = Activation('relu') (x2b)
    x3b = Conv2D(F2, (1, 3), padding='same') (x3a)
    x3b = BatchNormalization(axis=3) (x3b)
    x3b = Activation('relu') (x3b)

    x2c = Conv2D(F3, (3, 1), padding='same') (x2b)
    x2c = BatchNormalization(axis=3) (x2c)
    x2c = Activation('relu') (x2c)
    x3c = Conv2D(F3, (3, 1), padding='same') (x3b)
    x3c = BatchNormalization(axis=3) (x3c)
    x3c = Activation('relu') (x3c)
```

```

x3d = Conv2D(F4, (3,1),padding='same')(x3c)
x3d = BatchNormalization(axis=3)(x3d)
x3d = Activation('relu')(x3d)
x3d = Conv2D(F4, (1,3),padding='same')(x3d)
x3d = BatchNormalization(axis=3)(x3d)
x3d = Activation('relu')(x3d)

x1 = Conv2D(F4, (1,1),activation = 'relu')(x1a)
x2 = Conv2D(F4, (1,1),activation = 'relu')(x2c)
x3 = x3d
return x1,x2,x3

def Inception_Resnet_BZ(X,filters):
    F1,F2,F3,F4 = filters
    print(X.shape)
    X_shortcut_1 = X
    X_shortcut_2 = X

    x1,x2 = Inception_resnet_cz_block(X,filters)

    X = tf.keras.layers.Concatenate(axis=3)([x1,x2])
    X_shortcut_1 = Conv2D(filter_temp_, (1,1),activation =
'relu')(X_shortcut_1)#512

    X = Add()([X,X_shortcut_1])

    x1,x2 = Inception_resnet_cz_block(X,filters)
    X = tf.keras.layers.Concatenate(axis=3)([x1,x2])

    X_shortcut_2 = Conv2D(filter_temp_, (1,1),activation =
'relu')(X_shortcut_2)#512

    X = Add()([X,X_shortcut_2])
    return X

def Inception_resnet_bc_block(X,filters):
    F1,F2,F3,F4 = filters

    x1a = Conv2D(F1, (1,1))(X)
    x1a = BatchNormalization(axis=3)(x1a)
    x1a = Activation('relu')(x1a)
    x2a = Conv2D(F1, (1,1))(X)
    x2a = BatchNormalization(axis=3)(x2a)
    x2a = Activation('relu')(x2a)

    x2b = Conv2D(F2, (1,5),padding='same')(x2a)
    x2b = BatchNormalization(axis=3)(x2b)
    x2b = Activation('relu')(x2b)

    x2c = Conv2D(F3, (5,1),padding='same')(x2b)
    x2c = BatchNormalization(axis=3)(x2c)
    x2c = Activation('relu')(x2c)

    x1 = Conv2D(F4, (1,1),activation = 'relu')(x1a)
    x2 = Conv2D(F4, (1,1),activation = 'relu')(x2c)

```

```

        return x1,x2

def Inception_Resnet_CZ(X, filters):
    F1,F2,F3,F4 = filters
    print(X.shape)
    X_shortcut_1 = X
    X_shortcut_2 = X

    x1,x2 = Inception_resnet_cz_block(X, filters)

    X = tf.keras.layers.Concatenate(axis=3) ([x1,x2])
    X_shortcut_1 = Conv2D(filter_temp_, (1,1), activation =
'relu')(X_shortcut_1)

    X = Add() ([X,X_shortcut_1])

    x1,x2 = Inception_resnet_cz_block(X, filters)
    X = tf.keras.layers.Concatenate(axis=3) ([x1,x2])

    X_shortcut_2 = Conv2D(filter_temp_, (1,1), activation =
'relu')(X_shortcut_2)

    X = Add() ([X,X_shortcut_2])
    return X

def Inception_resnet_cz_block(X, filters):

    F1,F2,F3,F4 = filters

    x1a = Conv2D(F1, (1,1)) (X)
    x1a = BatchNormalization(axis=3) (x1a)
    x1a = Activation('relu') (x1a)
    x2a = Conv2D(F1, (1,1)) (X)
    x2a = BatchNormalization(axis=3) (x2a)
    x2a = Activation('relu') (x2a)

    x2b = Conv2D(F2, (1,7),padding='same') (x2a)
    x2b = BatchNormalization(axis=3) (x2b)
    x2b = Activation('relu') (x2b)

    x2c = Conv2D(F3, (7,1),padding='same') (x2b)
    x2c = BatchNormalization(axis=3) (x2c)
    x2c = Activation('relu') (x2c)

    x1 = Conv2D(F4, (1,1),activation = 'relu') (x1a)
    x2 = Conv2D(F4, (1,1),activation = 'relu') (x2c)

    return x1,x2

def Texnet(input_shape=(299,299,3), classes=3):
    X_input = Input(input_shape)
    X = Conv2D(32, (7,7),strides = (2,2)) (X_input)
    X = BatchNormalization(axis = 3) (X)
    X = Activation('relu') (X)
    X = MaxPooling2D((3,3),strides=(2,2)) (X)
    for i in range(5):
        X = Inception_Resnet_AZ(X, [8,16,32,64])
    X = MaxPooling2D((3,3),strides=(2,2)) (X)

```

```

for i in range(10):
    X = Inception_Resnet_BZ(X, [8,16,32,64])
X = MaxPooling2D((3,3), strides=(2,2))(X)
for i in range(5):
    X = Inception_Resnet_CZ(X, [8,16,32,64])
X = AveragePooling2D((4,4))(X)
X = Flatten()(X)
X = Dense(classes, activation = 'softmax')(X)
model = Model(inputs=X_input, outputs = X, name='TEXNet')
return model

```

Program Summary Output:

Model: "TEXNet"

Layer (type)	Output Shape	Param #	Connected to
=====			
input_1 (InputLayer)	[(None, 299, 299, 3)]	0	
conv2d (Conv2D)	(None, 147, 147, 32)	4736	input_1[0][0]
batch_normalization (BatchNormaliza	(None, 147, 147, 32)	128	conv2d[0][0]
activation (Activation)	(None, 147, 147, 32)	0	batch_normalization[0][0]
max_pooling2d (MaxPooling2D)	(None, 73, 73, 32)	0	activation[0][0]
conv2d_3 (Conv2D)	(None, 73, 73, 8)	264	max_pooling2d[0][0]
batch_normalization_3 (BatchNor	(None, 73, 73, 8)	32	conv2d_3[0][0]
activation_3 (Activation)	(None, 73, 73, 8)	0	batch_normalization_3[0][0]
conv2d_5 (Conv2D)	(None, 73, 73, 16)	400	activation_3[0][0]
batch_normalization_5 (BatchNor	(None, 73, 73, 16)	64	conv2d_5[0][0]
conv2d_2 (Conv2D)	(None, 73, 73, 8)	264	max_pooling2d[0][0]
activation_5 (Activation)	(None, 73, 73, 16)	0	batch_normalization_5[0][0]
.			
.			
.			
conv2d_329 (Conv2D)	(None, 17, 17, 64)	2112	activation_210[0][0]
concatenate_39 (Concatenate)	(None, 17, 17, 128)	0	conv2d_328[0][0] conv2d_329[0][0]
conv2d_330 (Conv2D)	(None, 17, 17, 128)	16512	add_37[0][0]
add_39 (Add)	(None, 17, 17, 128)	0	concatenate_39[0][0] conv2d_330[0][0]
average_pooling2d (AveragePooli	(None, 4, 4, 128)	0	add_39[0][0]
flatten (Flatten)	(None, 2048)	0	average_pooling2d[0][0]
dense (Dense)	(None, 10)	20490	flatten[0][0]
=====			
Total params: 1,437,562			
Trainable params: 1,428,698			
Non-trainable params: 8,864			

TExNet Learning Rate Definition Code:

```
lr_schedule = tf.keras.callbacks.LearningRateScheduler(  
    lambda epoch: 1e-4 * 10**(epoch / 20))  
model.compile(optimizer = RMSprop(lr = 1e-4),  
              loss='categorical_crossentropy',  
              metrics=['acc'])  
  
history = model.fit_generator(train_generator,  
                             validation_data = test_generator,  
                             steps_per_epoch = 100,  
                             epochs=100,  
                             validation_steps = 50,  
                             verbose = 2,  
                             callbacks = [lr_schedule])
```

TExNet Learning Rate Definition Code Output:

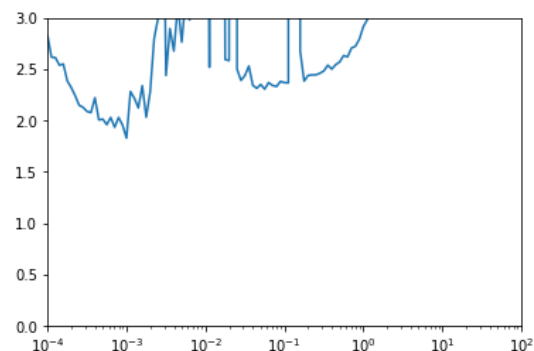
```
Epoch 1/100  
Epoch 1/100  
100/100 - 207s - loss: 2.8227 - acc: 0.1250 - val_loss: 2.3170 - val_acc: 0.0900  
Epoch 2/100  
Epoch 1/100  
100/100 - 72s - loss: 2.6151 - acc: 0.1550 - val_loss: 2.3936 - val_acc: 0.0800  
Epoch 3/100  
Epoch 1/100  
100/100 - 76s - loss: 2.6097 - acc: 0.0950 - val_loss: 2.3775 - val_acc: 0.1200
```

.
.
.

```
Epoch 1/100  
100/100 - 74s - loss: 12.2229 - acc: 0.0700 - val_loss: 12.3626 - val_acc: 0.1200  
Epoch 99/100  
Epoch 1/100  
100/100 - 74s - loss: 13.1570 - acc: 0.1000 - val_loss: 14.8024 - val_acc: 0.1300  
Epoch 100/100  
Epoch 1/100  
100/100 - 75s - loss: 15.0299 - acc: 0.0800 - val_loss: 8.8443 - val_acc: 0.0800
```

```
import matplotlib.pyplot as plt  
plt.semilogx(history.history["lr"], history.history["loss"])  
plt.axis([1e-4, 1e2, 0, 3])
```

[0.0001, 100.0, 0, 3]



TEXNet Model Compilation/Training Code:

```
model.compile(optimizer = RMSprop(lr = 1e-3),
              loss='categorical_crossentropy',
              metrics=['acc'])
```

```
import datetime
from tensorflow.keras.callbacks import CSVLogger
csv_logger = CSVLogger("model_history_log_texnet_test_7.csv", append=True)
callbacks = myCallback()
history = model.fit_generator(train_generator,
                             validation_data = test_generator,
                             steps_per_epoch = 50,
                             epochs=500,
                             validation_steps = 25,
                             verbose = 2,
                             callbacks = [csv_logger])
```

```
Epoch 1/500
Epoch 1/500
50/50 - 173s - loss: 2.4937 - acc: 0.1375 - val_loss: 2.3088 - val_acc: 0.0900
Epoch 2/500
Epoch 1/500
50/50 - 40s - loss: 2.3562 - acc: 0.1400 - val_loss: 2.3089 - val_acc: 0.1050
Epoch 3/500
Epoch 1/500
50/50 - 41s - loss: 2.2865 - acc: 0.1450 - val_loss: 2.3075 - val_acc: 0.0900
Epoch 4/500
Epoch 1/500
50/50 - 41s - loss: 2.1630 - acc: 0.2375 - val_loss: 2.3102 - val_acc: 0.1050
Epoch 5/500
Epoch 1/500
50/50 - 41s - loss: 2.1131 - acc: 0.2575 - val_loss: 2.3082 - val_acc: 0.1450

:

:

:

Epoch 496/500
Epoch 1/500
50/50 - 44s - loss: 0.2283 - acc: 0.9425 - val_loss: 0.4554 - val_acc: 0.8900
Epoch 497/500
Epoch 1/500
50/50 - 44s - loss: 0.2420 - acc: 0.9400 - val_loss: 0.4652 - val_acc: 0.8750
Epoch 498/500
Epoch 1/500
50/50 - 47s - loss: 0.2412 - acc: 0.9300 - val_loss: 0.3371 - val_acc: 0.9100
Epoch 499/500
Epoch 1/500
50/50 - 42s - loss: 0.2225 - acc: 0.9475 - val_loss: 10.1277 - val_acc: 0.2600
Epoch 500/500
Epoch 1/500
50/50 - 45s - loss: 0.2218 - acc: 0.9450 - val_loss: 8.7724 - val_acc: 0.1050
```











Training Methodology

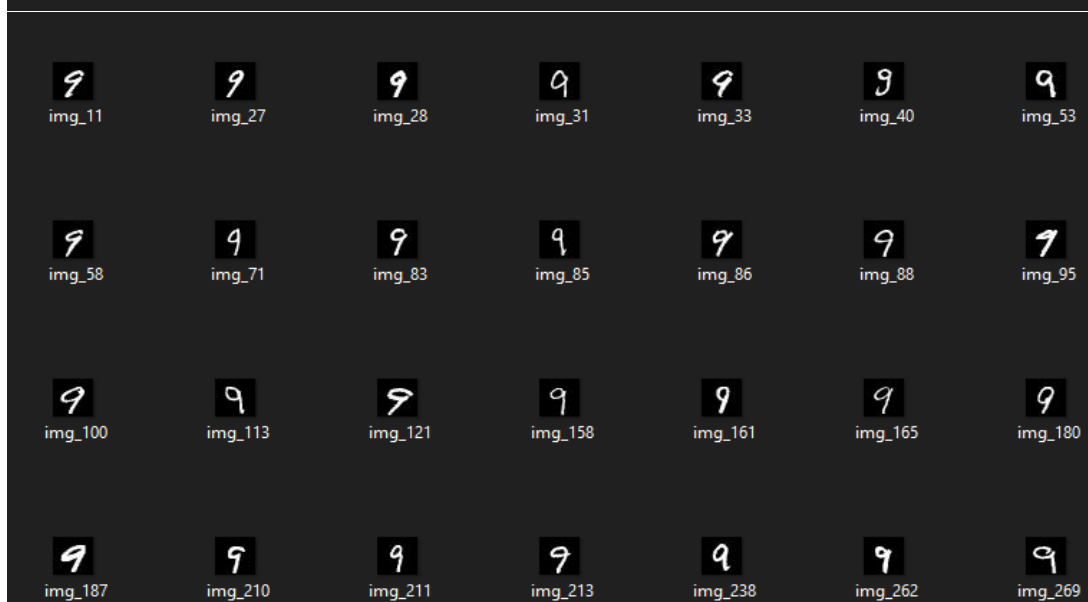
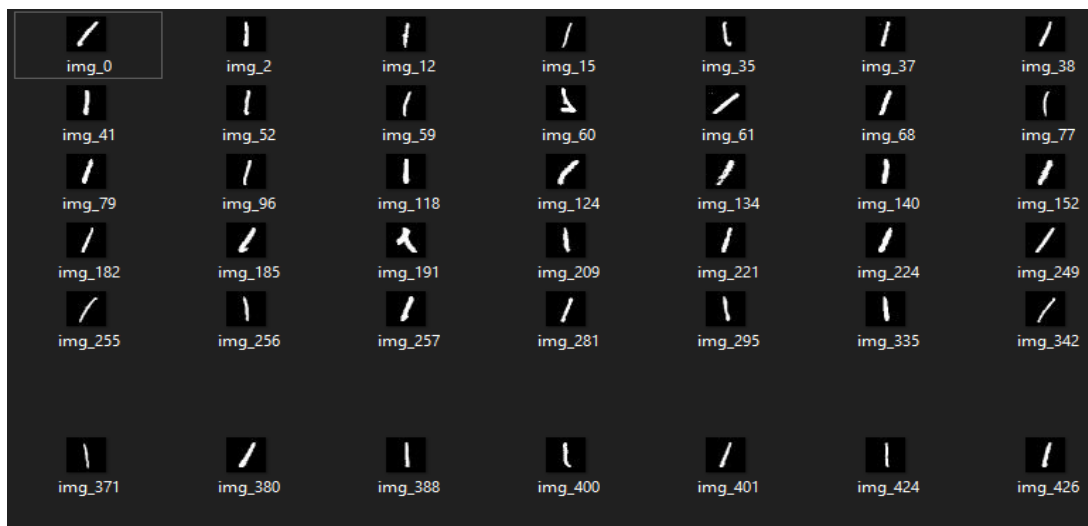
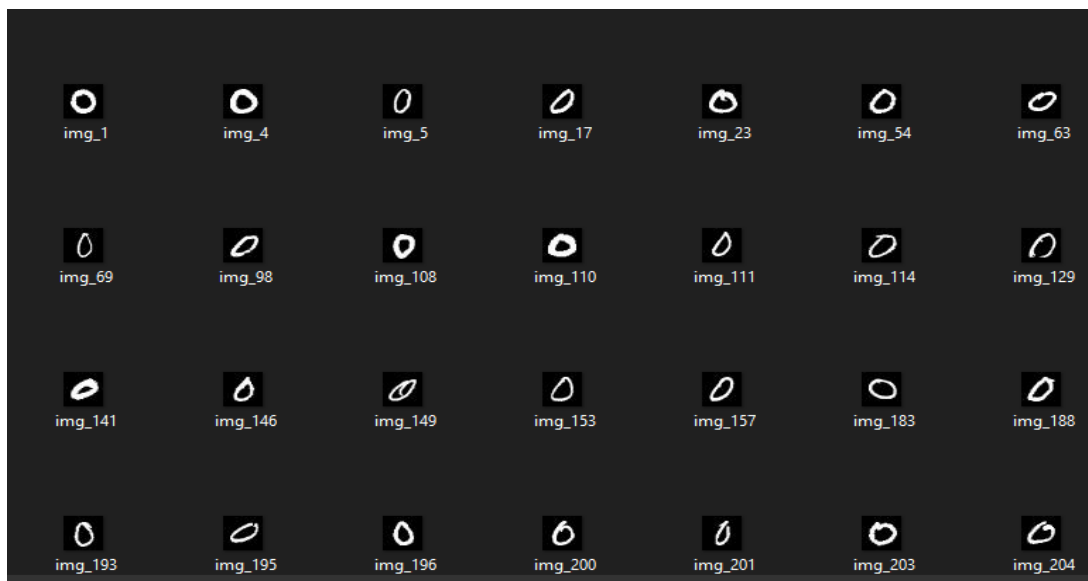
Our networks were trained using very limited computing hardware, comprising the market standard NVIDIA GTX 1050Ti (Portable). The full extent of the network's capability is limited as the training takes at least 70-80 seconds per epoch, for the steps per epoch set at 300. Compared to the Inception V4, the training per epoch is twice of it. The hyper-parameter tuning we have opted is to hard code and hard test the model to find the most optimal parameters for which the network can be best trained on for any given dataset. The whole network is trained within Jupyter notebook, using TensorFlow 1.15 – GPU and CUDNN. All test comparison is made initially with the Rock-Paper-Scissors dataset which are stored in terms of their respective directory. The dataset consists of 300 images of each category of dataset, where 50 of the images are used for validation and rest of the images are used for training. The number of epochs set for training the network for the given RPS dataset is set at 200.

The learning rate plays one of the most imported roles for the training of the network. Even the slightest change in the learning rate either improves the training by making it training it faster or very slowly. The aim of this network is to be able to train faster for images that have many features within it. The method applied to find the most optimal parameters for the model is not the best, but by the method of trial and error the model is subjected to give out more information of where the model lacks and what are the parameters that need improving. The training results for each model apart from TExNet take at least 4-5 hrs to get a stabilized result, but for TExNet takes at least 6 hours. The results for the RPS dataset indicate that the TExNet model is 1% better than the inception v4 model.

The model will next be trained with the most standard dataset most used for testing out any new network.

Digits_MNIST_Dataset: comprises 60,000 images for training and 10,000 images for training. Each individual image consists of a single digit from the range 0-9. (The result of the networks trained with this dataset will be updated, and any additional changes to the model will also be noted and updated in the document.

Name	Date modified	Type	Size
 0	30-04-2020 23:28	File folder	
 1	30-04-2020 23:28	File folder	
 2	30-04-2020 23:30	File folder	
 3	30-04-2020 23:30	File folder	
 4	30-04-2020 23:31	File folder	
 5	30-04-2020 23:31	File folder	
 6	30-04-2020 23:31	File folder	
 7	30-04-2020 23:32	File folder	
 8	30-04-2020 23:32	File folder	
 9	30-04-2020 23:33	File folder	

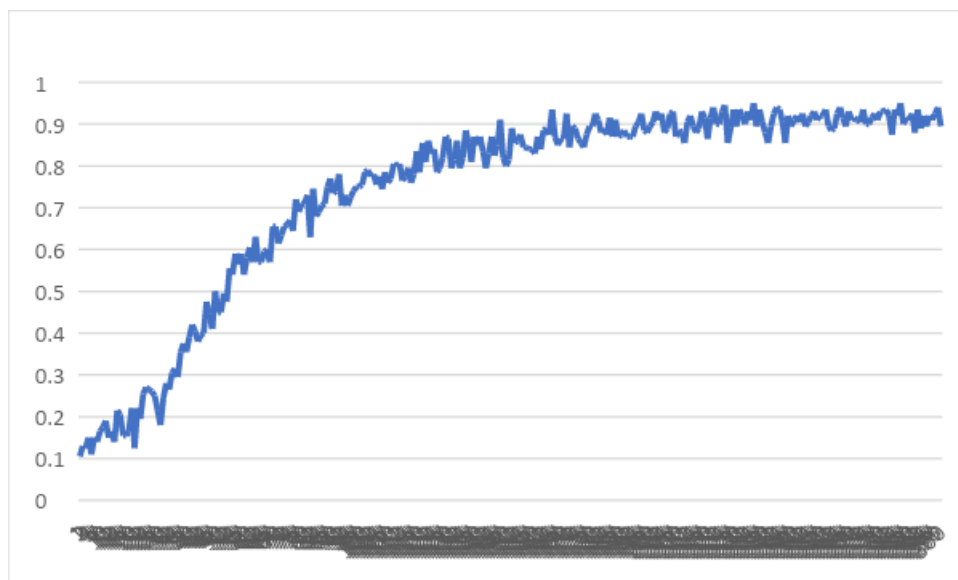


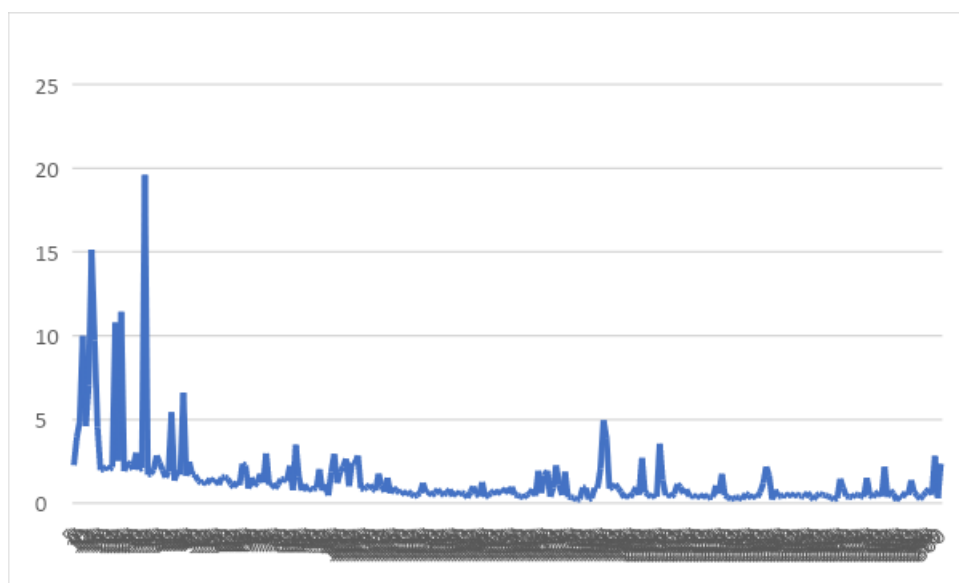
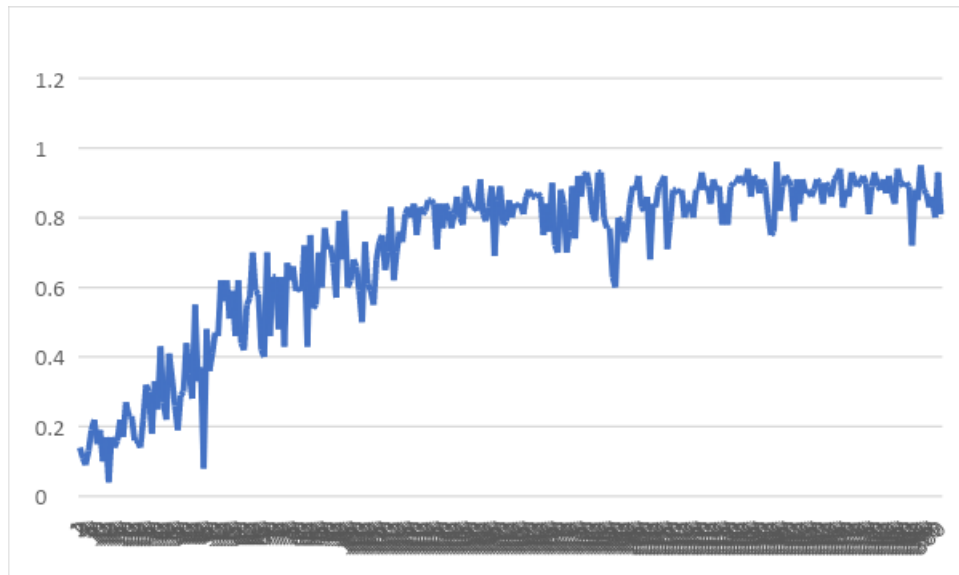
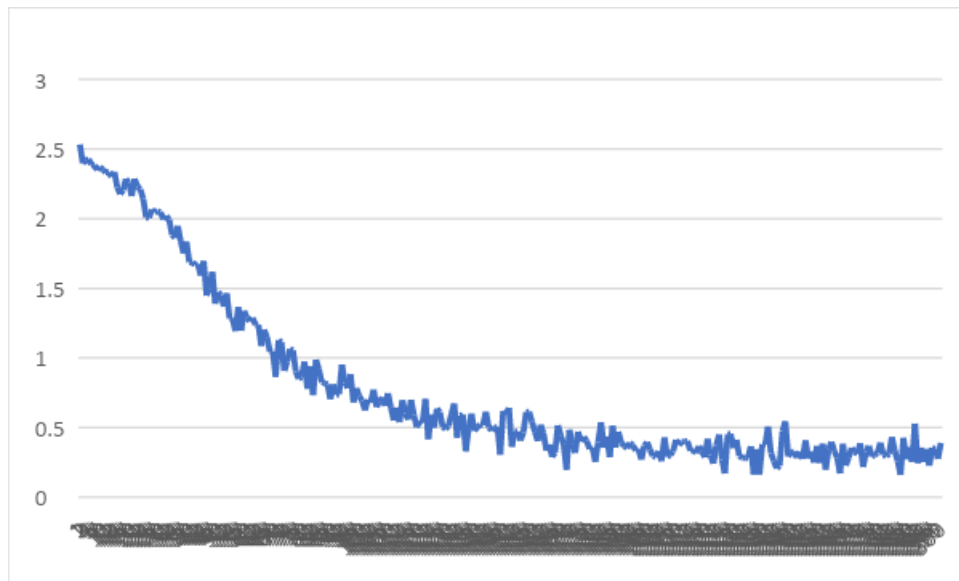
Test Analysis

The test analysis for each case is taken in such a way that the training and validation scores are taken for every epoch and the average of all is taken into consideration. The significant part of this test is to analyse the difference between the use of 50 million parameters as in the inception v4 model to that of 20 million of the TExNet model. The main takeaway through these tests will be to find ways to tweak the TExNet model in order to perform better than the inception model for the given dataset. The dataset used in this analysis is the Digit MNIST dataset which is the most common dataset used in comparing the performance of models and distinguishing them.

model_history_log_inception_resnet_v2_Mnist_Digits_1

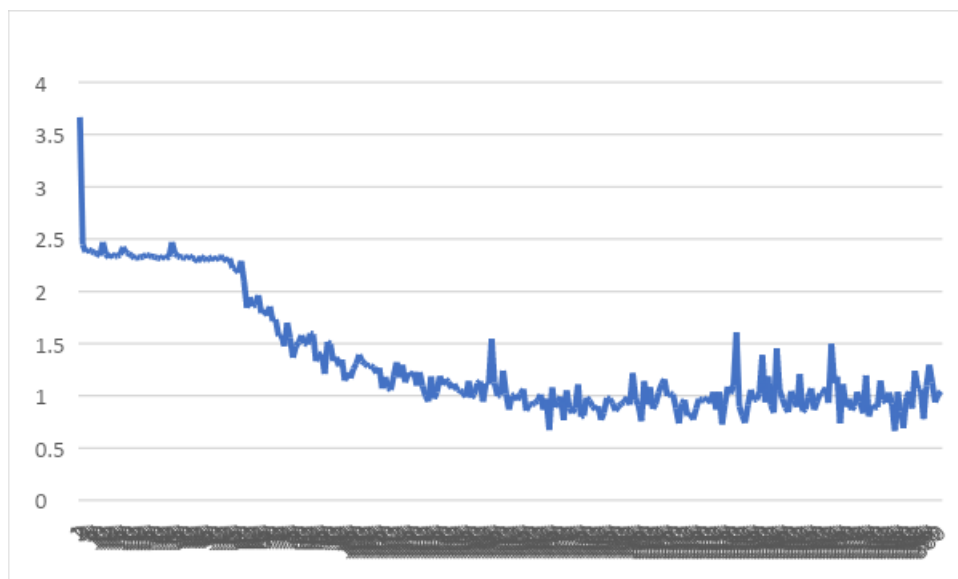
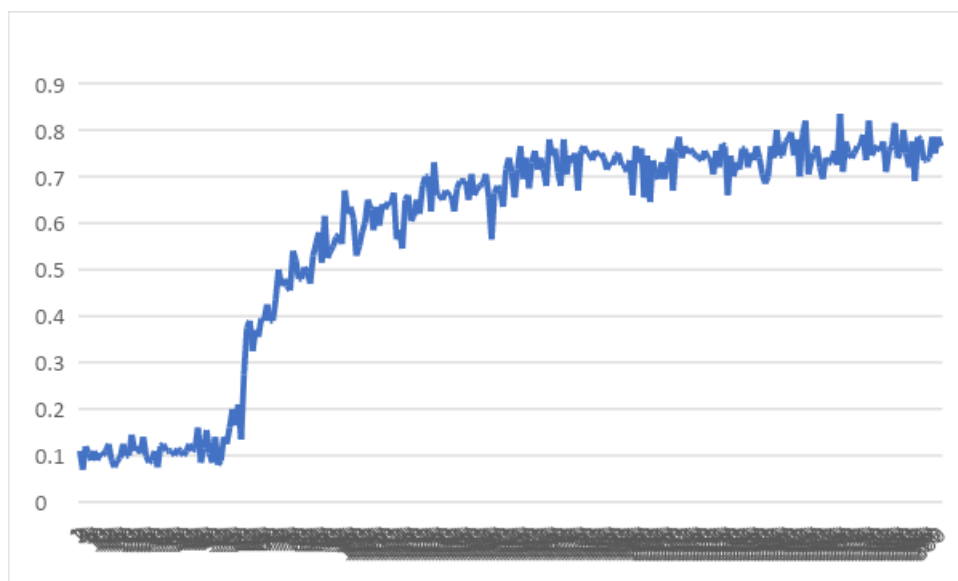
- Model used is Inception Resnet or inception v4.
- Learning rate applied is $e-3$ or 10^{-3} .
- Training accuracy achieved is 89.5%.
- Validation accuracy achieved is 81%.
- Analysis: the model stabilizes between 85-90% after 200 epochs, and the data collected is for 300 epochs. The time taken for each epoch is approx. 45-50 seconds. Even though the model consists of more than 50 million parameters the forward and backward propagation step is cut short for the number of computation steps involved in this. Making it important to rework the TExNet model and reduce the computation step to a significant amount.

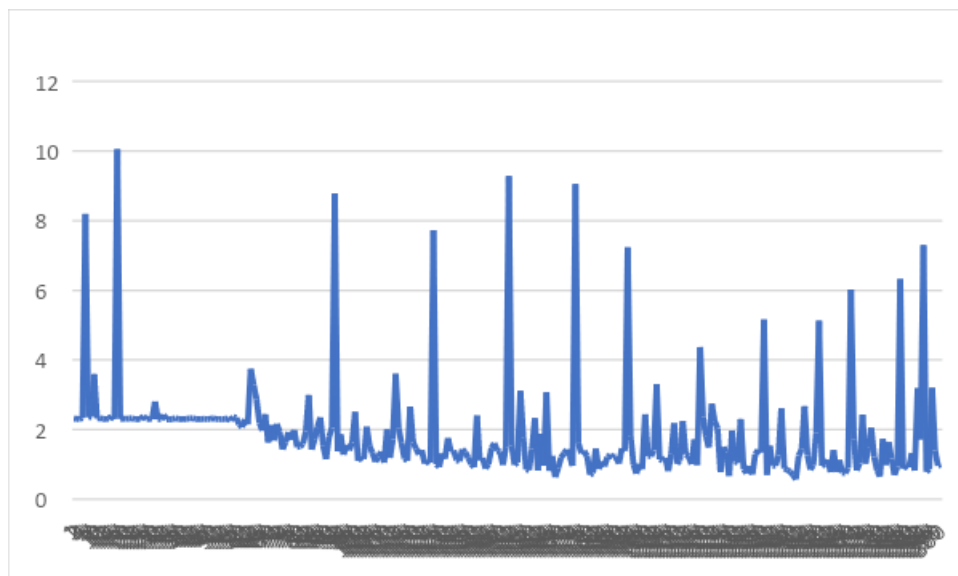
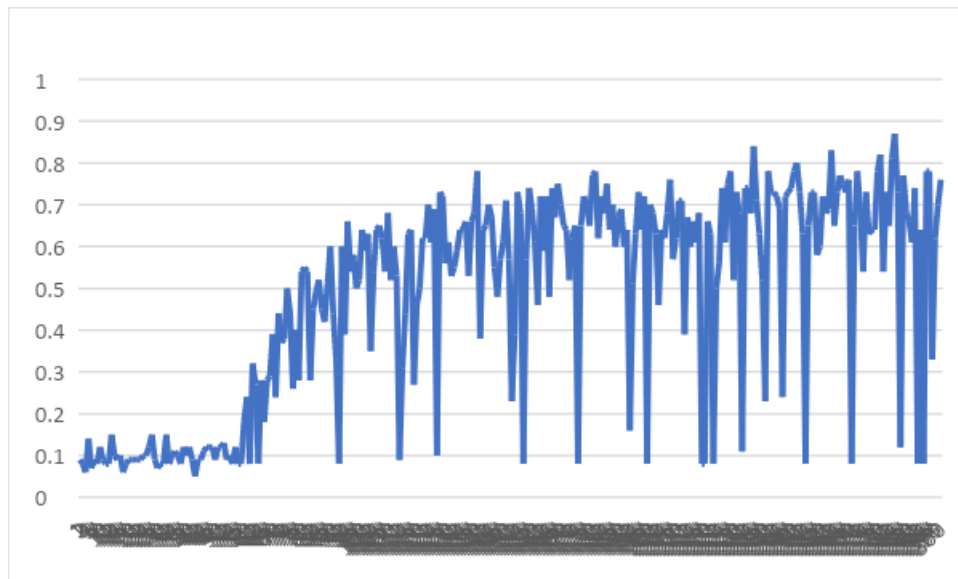




model_history_log_texnet_test_2

- Model used is TExNet.
- Learning rate applied is $e-3$ or 10^{-3} .
- Training accuracy achieved is 57%
- Validation accuracy achieved is 47%.
- Analysis: the model stabilizes between 75-80% after 200 epochs, and the data collected is for 300 epochs. The time taken for each epoch is approx. 70-80 seconds. Even though the model consists of more than 20 million parameters the forward and backward propagation step takes a lot of time to execute as the number of computation and merging steps is almost twice as much as inception v4. The learning portion of this step is that the learning accuracy is the only portion that gets stabilized and the validation portion goes haywire in terms of having a constant increase. The problem raised in the dataset split where only 5% of the dataset is used for validation and 95% is used for training.

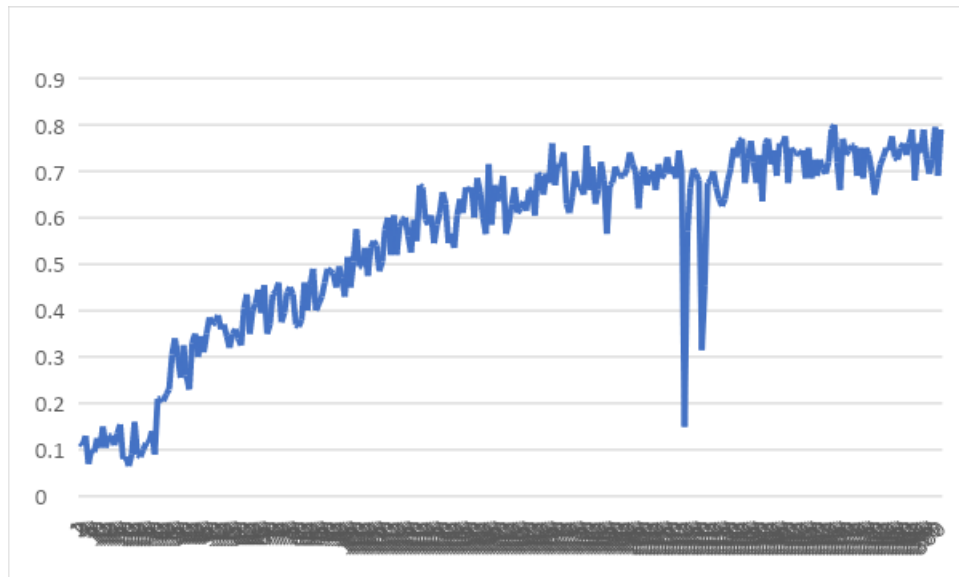


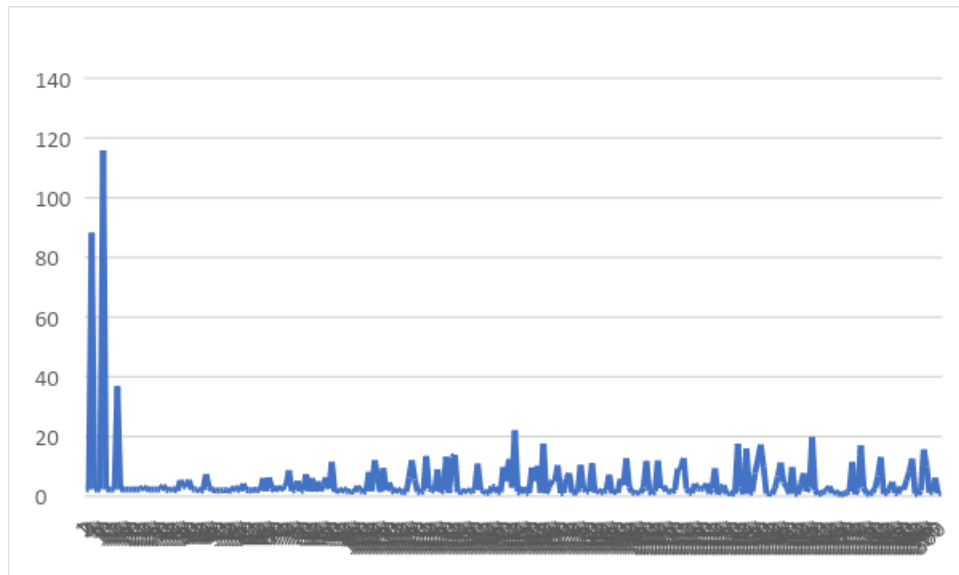
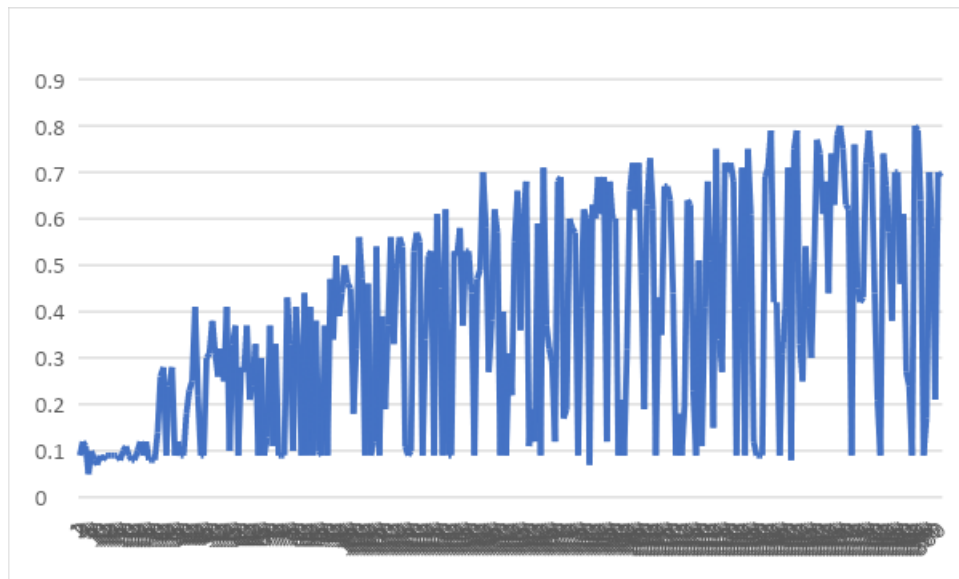


model_history_log_texnet_test_3

- Model used is TExNet.
- Learning rate applied is $e-3$ or 10^{-3} .
- Training accuracy achieved is 55%
- Validation accuracy achieved is 35%.
- Analysis: the model stabilizes between 75-80% after 250 epochs, and the data collected is for 300 epochs. The time taken for each epoch is approx. 70-80 seconds. This data is collected after solving the dataset split issue and turns out that the overall difference between the training accuracy average and the validation accuracy average is significantly higher which is about 20% when compared to the 10%. As this can be explained through the use of the 5% of the data for validation which internally resulted in the accuracy for few of the epochs to go to human level accuracy of 99%. This rise in the accuracy eventually led to the increase in the average accuracy for the validation. Rest assured that no such human level accuracy was achieved during this test as the max percentage reached by the model during the training was 79% in the validation accuracy. Now this brings the situation as to where the model can be tweaked, for both these tests, there is a window where the validation and the training scores

is increasing pretty slowly when compared to the previous test for the inception v4. By this the conclusion achieved and the next test to be analysed will check the most optimal learning rate that is to be applied on the model to get a fairly sophisticated and natural learning and validation curve for the given MNIST dataset.

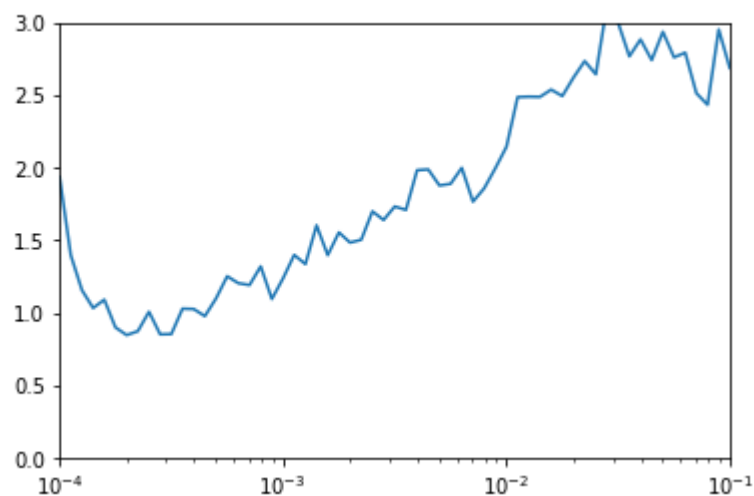




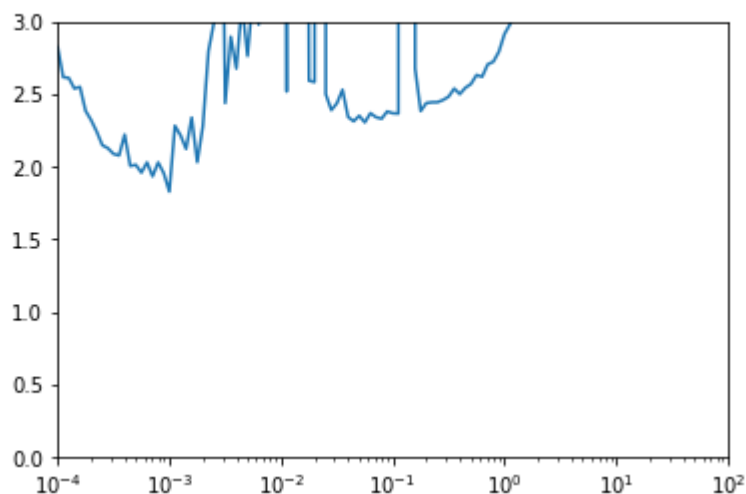
model_history_log_texnet_test_4

- Model used is TExNet.
- Learning rate applied is $7e-4$ or 70^{-4} .
- Training accuracy achieved is 55%
- Validation accuracy achieved is 35%.
- Analysis: After readjusting the learning rate to an even smaller amount than the previous, the resulting learning curve is such that when compared with the previous test, the validation loss score and the training loss score are very close to each other. The average score in the training and validation score is respectively.

Out[14]: [0.0001, 0.1, 0, 3]



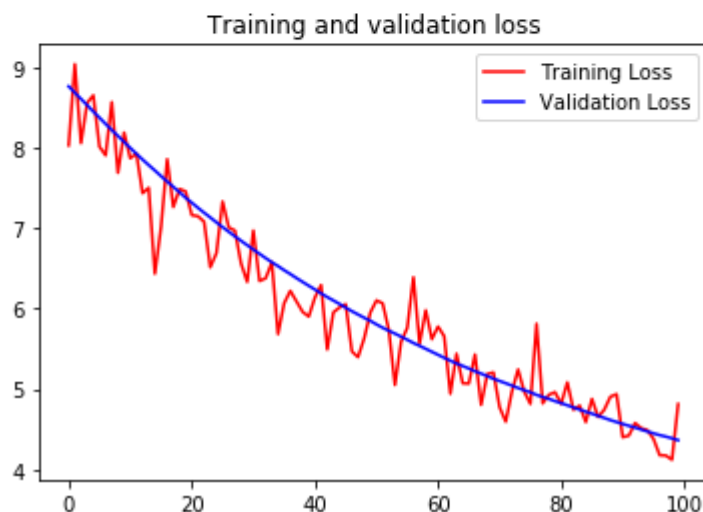
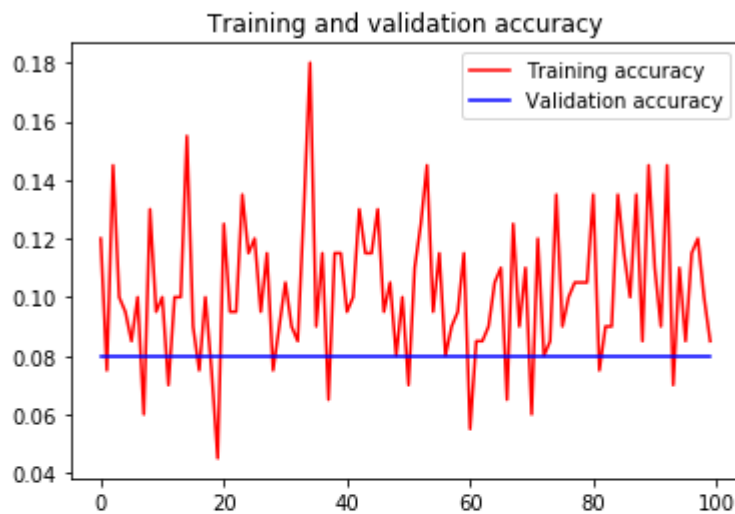
Out[25]: [0.0001, 100.0, 0, 3]



The top graph is that of the learning rate analysis of the Inception v4 model and the bottom is that of the TExNet. Both models seem to perform better with the learning rate being in-between $e-4$ and $e-3$. When tests were performed on the TExNet model with $7e-4$ learning rate, the model performed slowly in terms of loss reduction. The next test to be performed will be having to tweak with the steps per epoch parameter and to find an optimal parameter to benefit the model in general case and as well as to perform better than the inception v4.

Topping it off with the main highlight is that the difference between the mean training loss and the mean testing loss, the model has achieved a significant result in this test by having both the curves extremely close to each other. This is possible only by finding the optimal learning rate. The optimal learning rate is directly dependent on the dataset than the actual model itself. So, the learning rate will differ for any other given dataset and it is highly considered to test and find the optimal learning rate by using the learning rate call-back.

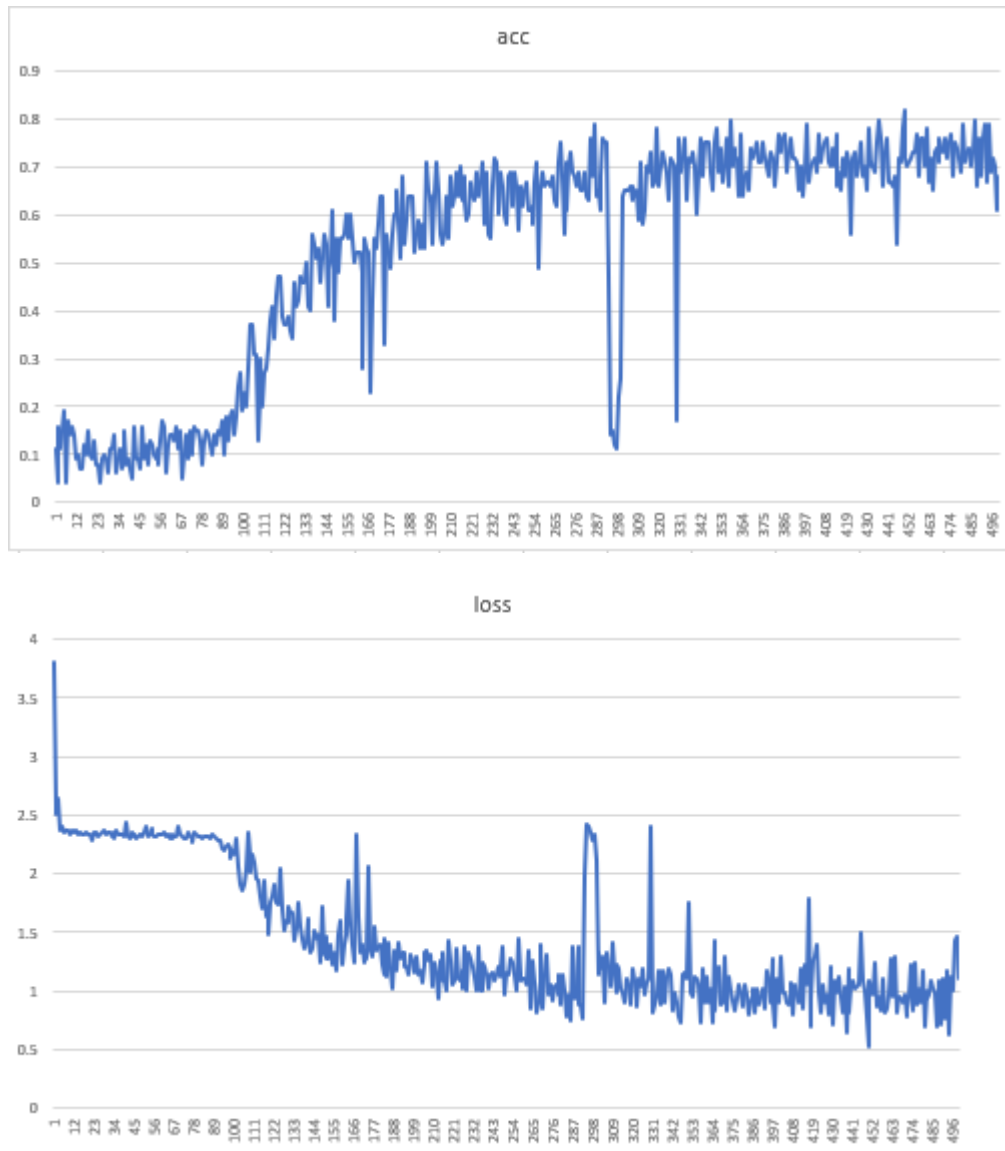
Mean Training Accuracy = 0.10210000045597553
Mean Testing Accuracy = 0.07999999821186066
Mean Training Loss = 5.990787440602109
Mean Testing Loss = 6.075742799624055



[model_history_log_texnet_test_5](#)

- Model used is TExNet.
- Learning rate applied is $9e-4$ or 9^{-3} .
- Training accuracy achieved is 53%
- Validation accuracy achieved is 47%.
- Analysis: Learning rate is readjusted and the results have been a slight improvement in general. The steps per epoch parameter was reduced to half and that resulted in reducing the time taken to finish one epoch by half. Reducing the steps per epoch resulted in a very jumpy curve and was looking less likely than the curve generated in the inception v4 model. This time the number of epochs for the training process was set to 500 epochs, this was done prior to the discovery of the reduced time taken to finish one epoch. The next test will be having to test out any form of layer tweaking or reduction of the overall layers used so that the time taken for one computation/epoch is as close to the inception v4 model. Another finding in this

test was that the system used to test the model is not highly capable in discovering the true potential of the TExNet model. The model's main system requirement to generate proper and smooth curve is the need to use at least 12GB Ram, with this in mind necessary changes will be made to the model where the layers that require high demand of RAM will be tweaked/reduced so that the model can generate the training and validation sets with a batch size of 32. Batch size in general plays a significant role in how good the model will perform during the training process.



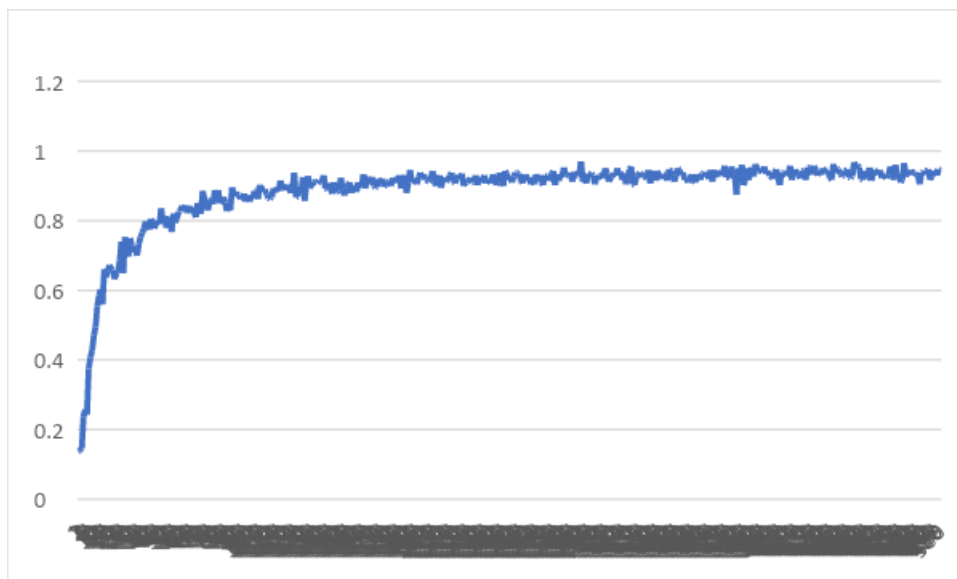
[model_history_log_texnet_test_6](#)

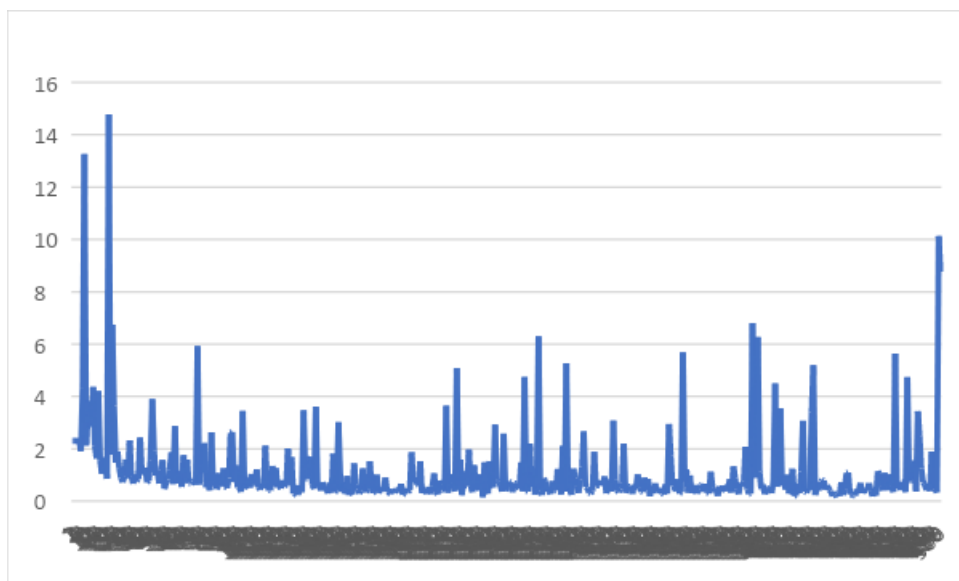
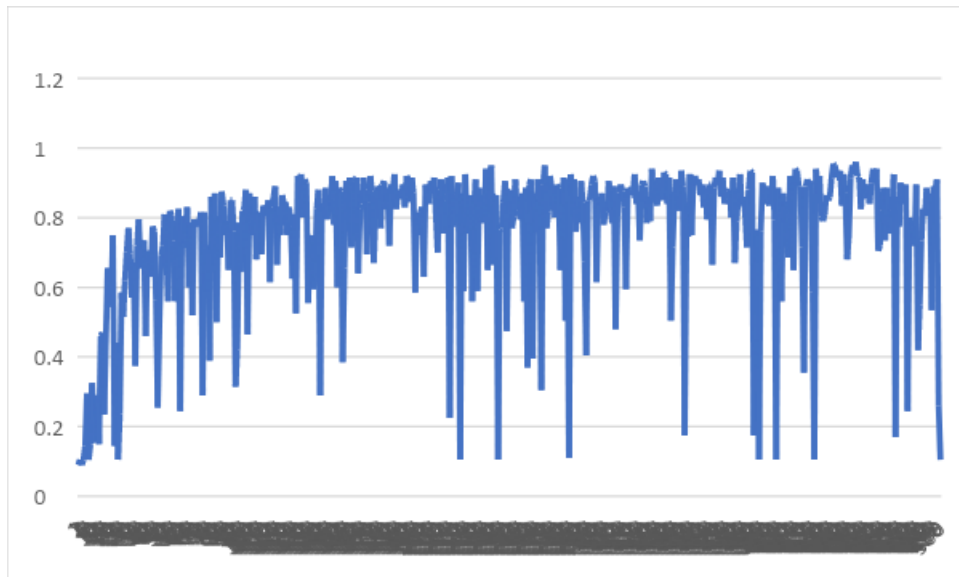
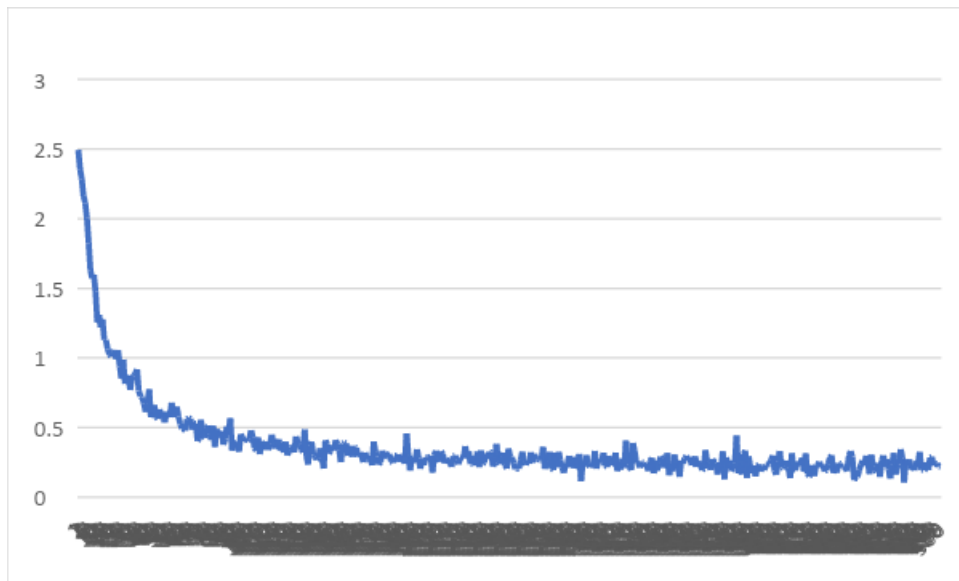
Test resulted in failure as the model training crashed at epoch 56 and the main cause being the memory allocation in the RAM of the system. Overall, the whole system crashed in the process even though the model was optimized to support the current system specification.

[model_history_log_texnet_test_7](#)

- Model used is TExNet.
- Learning rate applied is 10^{-3} .
- Training accuracy achieved is 88.4%
- Validation accuracy achieved is 75.6%.

- Analysis: Learning rate is readjusted and also the batch size for both training and validation has been increased to 8 due to system limitations, and the results have skyrocketed. The training has been done for 500 epochs where it elapsed for almost 6 hours in total training time where the steps per epoch for training is set to 50 and the steps per epoch for validation has been set to 25. The average time taken for one epoch was estimated to be around 41 seconds which is again a little more than the inception v4 model. But this is due to the fact that the TExNet model has been updated in terms of the number of trainable parameters set, estimating to be around 1.5 million. When compared with the inception v4 model this model consists of lesser trainable parameters but consists of more computation steps in a single forward propagation step. Now coming to the difference in the training and validation steps, where the difference is around 13% in average which signifies that the model does not have issues in regards with overfitting and underfitting. This issue may not rise due to the fact of the number of training samples being used for training and validation per epoch. The MNIST dataset being used consists of about 32,000 training set images and 8,000 validation set images and this is sent to the image generator function which in turn produces approx. 30+ images which get generated at random through the means of random rotation and other geometrical applications implied to it. So, judging by the number of samples the model is using and for the training time of about 6 hours for 500 epochs the model is able to get an average validation score of 88% in the training set. As this will help in training more complex images and give us more satisfying and less time-consuming results when applied to other real-life applications.



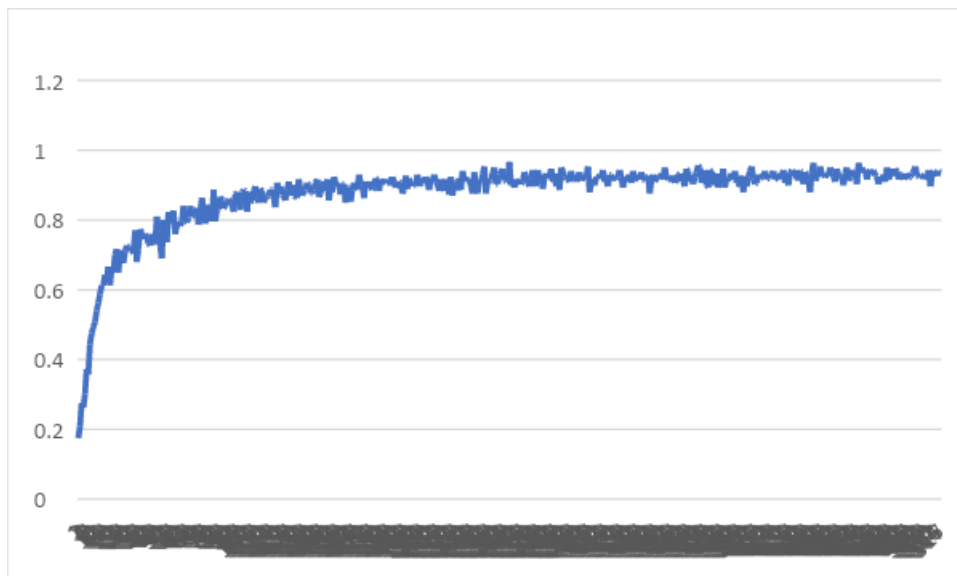


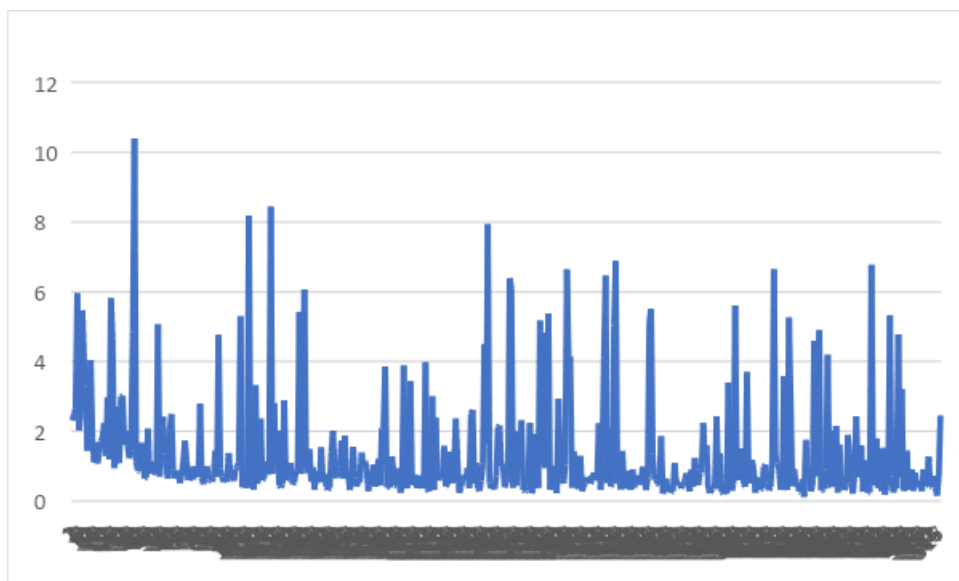
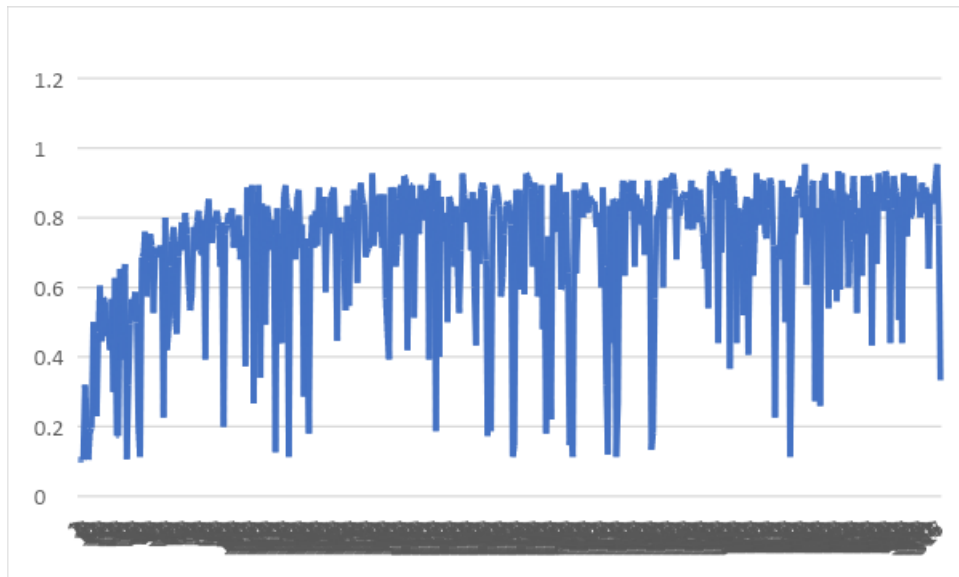
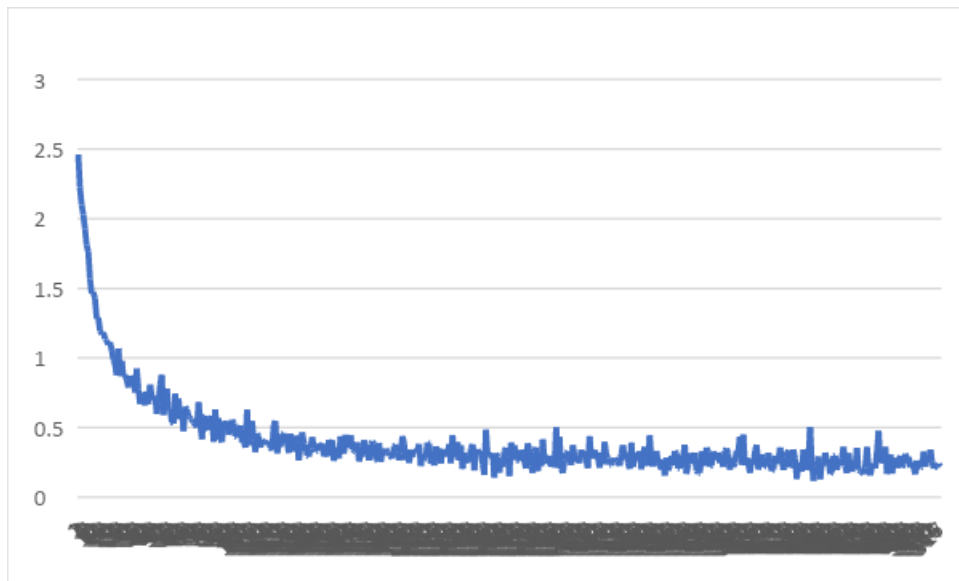
At epoch 437, the model reached its peak in both training and validation set. Where the training accuracy reached to almost 96.5% and the validation accuracy reached 95.5%. The next test will be adjusting the steps per epochs with the standard equation used by the deep learning community, this is mainly to reduce the jumpy effects seen in the validation accuracy curve

model_history_log_texnet_test_7

- Model used is TExNet.
- Learning rate applied is 10^{-3} .
- Training accuracy achieved is 87.6%
- Validation accuracy achieved is 71%.

Analysis: The batch size for this test is reduced to 6 to help counter the memory issue when the inclusion of a saved model call-back is initiated. With the help of this call back, the best epoch is noted and saved. Then the model weights are saved with regards to the best epoch so when applied to a real-life application the model will be able to perform efficiently better than some random epoch value or average weight of it. Since the model does not face issues with overfitting and underfitting, this approach seems to be the most optimal. The best epoch found is to be performing 93% accuracy on the training set and 95% on the validation set, where the difference is 2% which would be acceptable as the number of training sets is almost more than 35-40 million images when compared to the use of 1-2 million validation images. The large variation in the curve of both the validation loss and accuracy can be reduced by increasing the batch size but due to system limitation this is not possible for the moment. The next test will be having to load and reuse the model and test it out on a real-life application and check its performance.





model_history_log_texnet_test_8

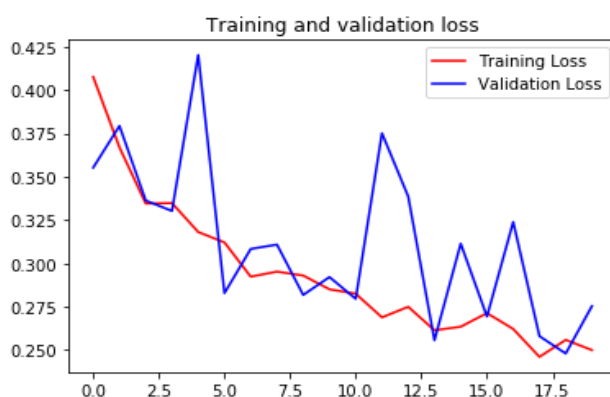
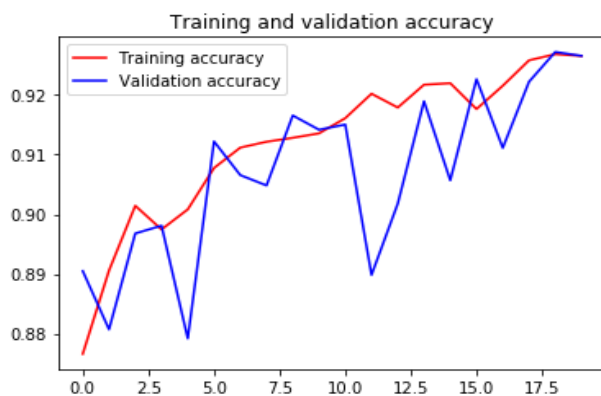
- Model used is TExNet.
- Learning rate applied is 10^{-3} .
- Training accuracy achieved is 91.2%
- Validation accuracy achieved is 90.7%.

Analysis: The batch size is set to the same value as the previous test and the only changed made was in the `steps_per_epoch` parameter where the equation used is,

$$steps_{per_epoch} = \frac{len(training_{set})}{len(batchSize_{trainingset})}$$

The results produced an even better line curve and a constant learning rate, but at the cost of the time taken for each epoch. The time for each epoch elapsed to around 30 mins and the total training time elapsed to around 10 hours for 20 epochs. But the model now predicts the handwritten digits with an accuracy of about 91% on average which can be termed to human level performance if were to take in to consideration of the randomness of how the image is sent across the model to t

```
Mean Training Accuracy = 0.9120282471179962
Mean Testing Accuracy = 0.9070295572280884
Mean Training Loss = 0.29383670197820644
Mean Testing Loss = 0.3116270608867512
```

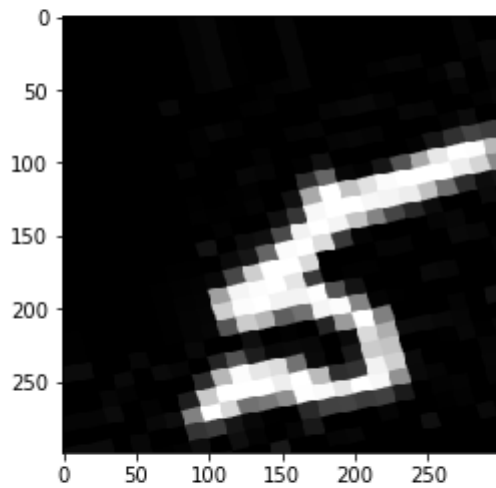


```

1 #model.predict(test_generator)
2 x = image.img_to_array(train_generator[0][0][2])
3 plt.imshow(x)
4 x = np.expand_dims(x, axis=0)
5 #print(x.shape)
6 result = model.predict(x)

```

(1, 299, 299, 3)

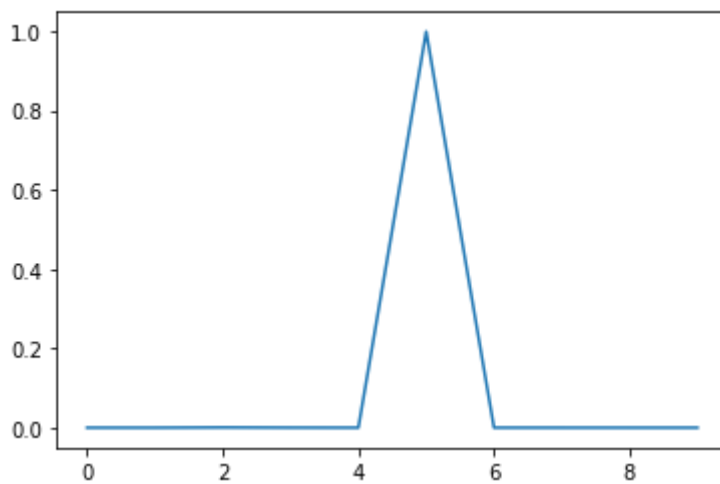


```

1 print(result[0])
2 plt.plot(result[0])
3 plt.show()

```

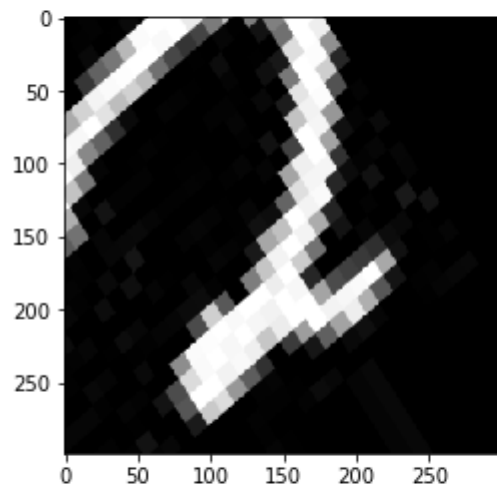
[7.6234130e-10 8.4771939e-10 6.5414986e-04 1.6724506e-04 4.9725742e-07
9.9914801e-01 2.4874125e-05 5.0499007e-06 3.2750169e-08 2.2594106e-07]



```

1 #model.predict(test_generator)
2 x = image.img_to_array(train_generator[0][0][3])
3 plt.imshow(x)
4 x = np.expand_dims(x, axis=0)
5 #print(x.shape)
6 result = model.predict(x)

```



```

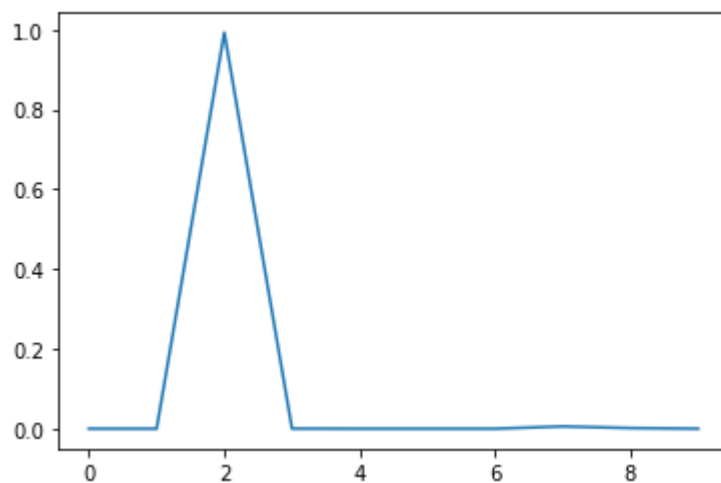
1 print(result[0])
2 plt.plot(result[0])
3 plt.show()

```

```

[1.0443025e-05 6.3112402e-08 9.9244279e-01 2.3580731e-04 3.7860653e-07
 1.3339870e-06 5.5940522e-08 5.7180161e-03 1.5630276e-03 2.8147702e-05]

```



Comparison

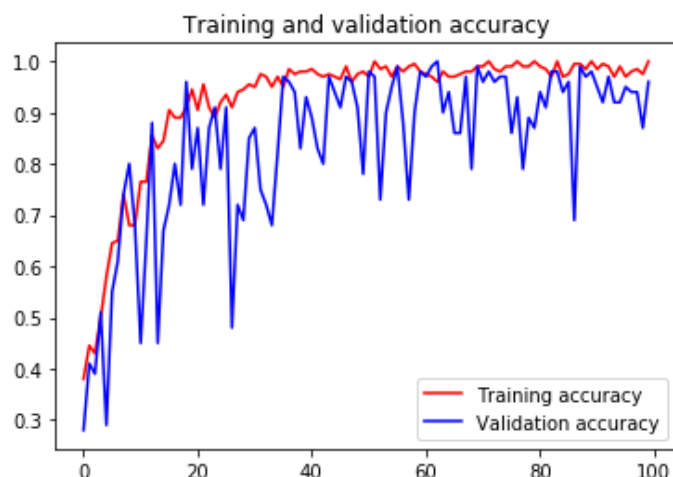
This section will particularly discuss about the already existing model independently with its advantages and disadvantages along with the proposed model and how it takes advantages of the existing models and tries to eliminate their disadvantage

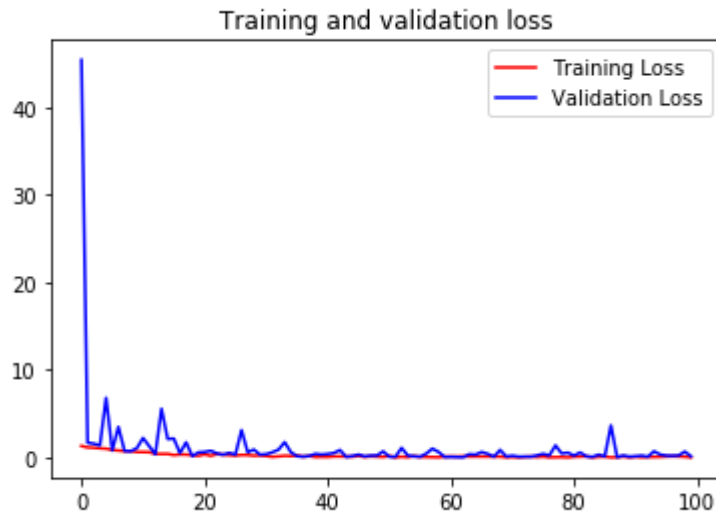
ResNet model:

Universal approximation theorem states that given enough capacity, a feed-forward neural network consisting of a single layer is just sufficient enough to represent any function. However, the system can be very large which can end up overfitting the data but simply stacking up the layers on top of each other wouldn't work. Hence, when the neural network goes deeper and deeper, the back propagation of gradient becomes insignificant. The ResNet model uses the identity connection method to skip layers as discussed briefly in the overview section, which allows a deeper network by allowing the gradient to travel a large number of layers. This also results in having the error rates for deeper layers just about the same compared to their shallower counterparts. A fitting hypothesis is that letting the stacked layers fit a residual mapping is easier than letting them directly fit the desired underlying mapping. Here, although deeper neural networks can be trained, it still loses its accuracy by a small percentage.

Inception model:

The inception model as discussed in the overview section uses the 1x1 convolution before passing through each of the other NxN convolution and also uses it in the post-max-pooling stage. This helps in reducing the number of parameters to be tuned by many folds helping in better accuracy but increases the training-time for the model by a small proportion.



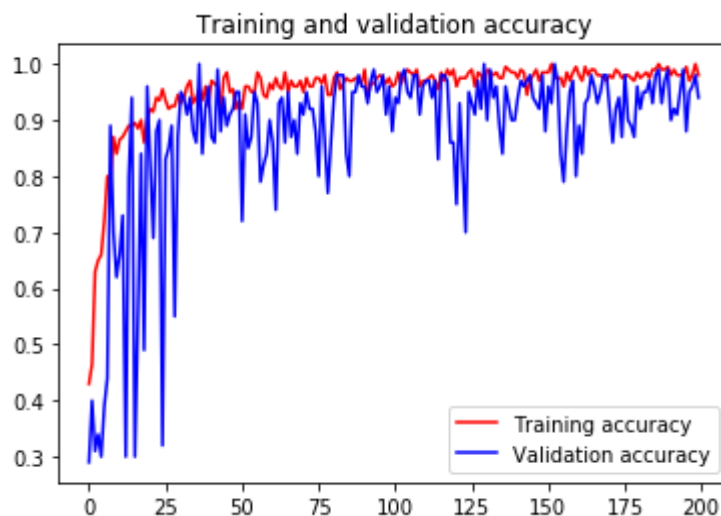


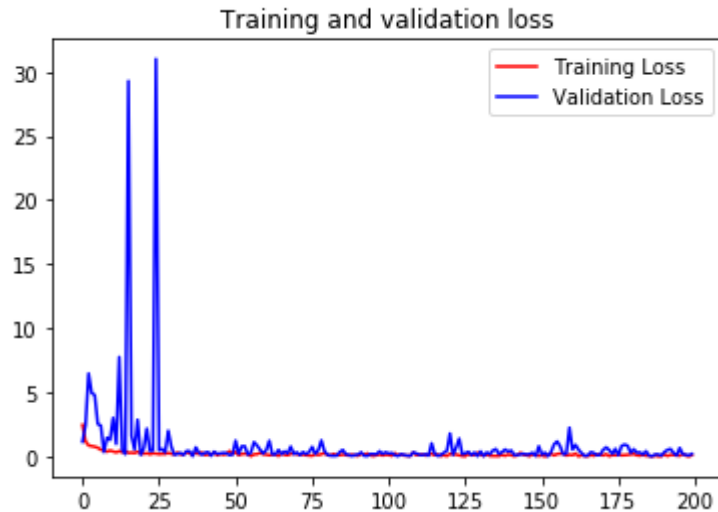
Mean Training Accuracy = 0.9179000082612038
Mean Testing Accuracy = 0.8071000019460917
Mean Training Loss = 0.21761294682582974
Mean Testing Loss = 15.065904694341336

TExNet model [proposed]:

Total params: 20,313,315
Trainable params: 20,277,731
Non-trainable params: 35,584

The proposed model incorporates the advantages of both the model, bypassing neural layers for faster computation in deeper networks and introduction of 1x1 convolution for relaxing the model complexity and increasing performance. The first stage optimization consists of residual blocks between two inception cells, this lets the model pass more accurate and detailed data to deeper layers without more loss of generality, helping stacking up deeper neural layers for finer detection. Secondly, the usage of residual blocks within the inception cell themselves allows the model to cost cut on processing time with lesser performance trade-off.





Mean Training Accuracy = 0.9510750071704388

Mean Testing Accuracy = 0.8764000023901463

Mean Training Loss = 0.18173162590011338

Mean Testing Loss = 0.862837291185465

Comparing the Inception Resnet V4 model with the proposed TExNet model shows that the proposed model doesn't just theoretically improvise over the original but practically gives performance improvement which can be seen from the above given graphs.

Firstly, the proposed TExNet model provides a significant training accuracy of 95.1%, that is nearly 4% greater than the 91.7% training accuracy of the original Inception Resnet V4 model.

Secondly, the proposed TExNet model provides a significant testing accuracy of 87.6%, that is significantly 7% more accurate than the 80.7% testing accuracy of the original Inception Resnet V4 model.

Future work:

1. Incorporating/Replacing better performing models in the future which can result in increased performance.
2. Usage of custom-built hardware systems for proposed model specific optimization.
3. Test the model in an environment with higher computational specification.
4. Adding DNNs to the tail to further enhance the model.
5. Calculation and comparison of the 1% and 5% error rate of the model to that of the other existing models.

Conclusion

Thus, the proposed model is a theoretical improvement or an incremental model that leverages the ResNet and Inception models, specifically their advantages. The proposed model has also gone through custom improvements in parameter setting for making the performance and complexity relaxation more significant. The accuracy and the loss rate of the model performs significantly better given the high amount of randomness that is being generated in the validation set. The jumpy curves in the validation set can be of the sign of the random positioning and random rotations that occur for every time the image is called for validation, basically every time the image is called for validation the image changes its position and rotation of where the features are present and the original form of the image is not broadcasted for validation. The model produces a descent linear drop in the loss value for every epoch and a descent linear increase in the accuracy even though the addition of the DNN does not exist in the tail. Normally DNNs are added to the tail to further enhance the decision making of the model, but even the absence of this element does not affect the performance of the TExNet model.

On a conclusive remark, the proposed TExNet model is 1.2% faster than the original Inception ResNet V4 model on the basis of overall training-testing performance measure. Also, the original Inception ResNet V4 model and the proposed TExNet model has been tested on naive datasets and the published results are for the same, further testing and training on large datasets will be performed for a more conclusive and elaborate training-testing performance measure. With the general MNIST dataset the model falls a bit short of about 1-5% on average as this may be due to the fact that the overall filter size is reduced and the restricted computational hardware used in the evaluation of the model.

Reference

- [1] Srivastava R K, Greff K, Schmidhuber J. Training very deep networks[C]//Advances in neural information processing systems. 2015: 2377-2385.
- [2] He K, Zhang X, Ren S, et al. Deep residual learning for image recognition[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2016: 770-778.
- [3] B. Li and Y. He, "An Improved ResNet Based on the Adjustable Shortcut Connections," IEEE Access, vol. 6, pp. 18967–18974, 2018.
- [4] Szegedy C, Ioffe S, Vanhoucke V, et al. Inception-v4, inception resnet and the impact of residual connections on learning[C]//AAAI. 2017, 4: 12.
- [5] Dong C, Loy C C, He K, et al. Learning a deep convolutional network for image super-resolution[C]//European Conference on Computer Vision. Springer, Cham, 2014: 184-199.
- [6] X. Zhang, S. Huang, X. Zhang, W. Wang, Q. Wang, and D. Yang, "Residual Inception: A New Module Combining Modified Residual with Inception to Improve Network Performance," 2018 25th IEEE International Conference on Image Processing (ICIP), 2018.