README.md

Image classification using CNN

1. Dataset

For process of classification Cifar 10 data set is used. This data set contains 10 categories and 6000 images per class. This data is broken in five batches and one test set.

1.1. Reading the data

For reading this data following python function is used

```
def loadCifarData(basePath):

    trainX = []
    testX = []
    trainY = []
    testY = []

    """Load training data"""
    for i in range(1, 6):
        with open(join(basePath, "data_batch_%d" %i), "rb") as f:
            dictionary = pickle.load(f, encoding = 'bytes')
            trainX.extend(dictionary[b'data'])
            trainY.extend(dictionary[b'labels'])

    with open(join(basePath, "test_batch"), "rb") as f:
        dictionary = pickle.load(f, encoding = 'bytes')
        testX.extend(dictionary[b'data'])
        testY.extend(dictionary[b'labels'])

    return trainX, trainY, testX, testY
```

1.2. Image vector transformations

The data present as a vector. To display render the image of this vector some processing is required.

```
def toImage(array, rows = 32, columns = 32):
    return array.reshape(3, rows, columns).transpose([1, 2, 0])

def toData(img, rows = 32, columns = 32):
    return img.transpose([-1, -2, 0]).flatten()
```

The above two methods convert the one dimensional vector to 3 dimensional image vector and 3 dimensional image vector back to one dimensional vector.

1.3 Displaying images

For displaying the images in cifar data set following method is used. This method makes use of matplotlib library.

```
def plotImages(rows, columns, data, convert = True):
    fig, ax = plt.subplots(nrows=rows, ncols=columns)

    if rows == 1:
        ax = [ax]
    if columns == 1:
        ax = [ax]

    index = 0
    for row in ax:
        for col in row:
            if convert:
                col.imshow(toImage(data[index]))
            else:
                col.imshow(data[index])
            index = index + 1
```

```
plotImages(3, 3, trainRawX)
```

2. Preprocessing

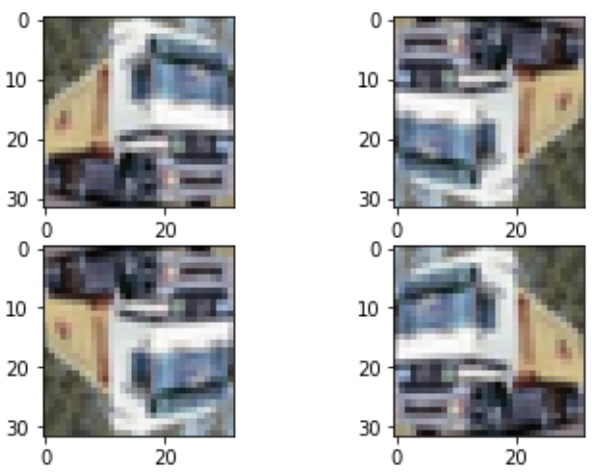
2.1. Data augmentation

While training images it is necessary that the images are augmented, more data is generated from existing data. This regularizes the training model by reducing the variance in data.

2.1.1. Flipping the images

The simplest augmentation method is to flip the provided images. Following method flips the input images and generates three more images.

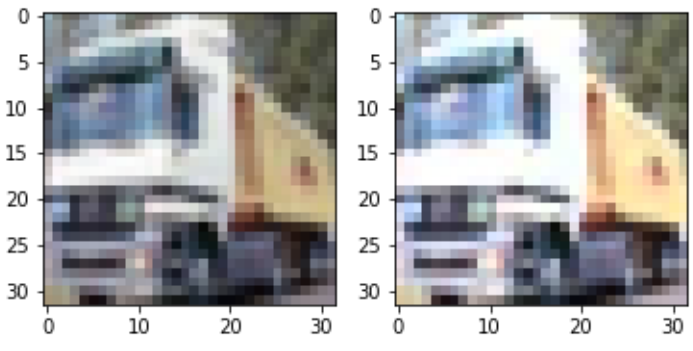
```
def flipImage(srcImage):
    flippedImages = []
    flippedImages.append(np.fliplr(srcImage))
    flippedImages.append(np.flipud(srcImage))
    flippedImages.append(np.flipud(np.fliplr(srcImage)))
    return flippedImages
```



2.1.2 Changing the brightness

Second image augmentation can be to change the brightness of the image. This can be done by adding some random noise to the image. For doing so the following method uses open cv.

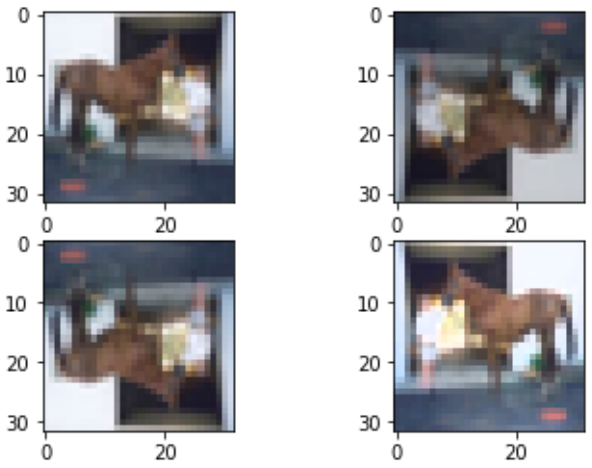
```
def changeBrightness(image):
    image = cv2.cvtColor(image,cv2.COLOR_RGB2HSV)
    image = np.array(image, dtype = np.float64)
    randomBrightness = .5+np.random.uniform()
    image[:,:,2] = image[:,:,2]*randomBrightness
    image[:,:,2][image[:,:,2]>255]  = 255
    image = np.array(image, dtype = np.uint8)
    image = cv2.cvtColor(image,cv2.COLOR_HSV2RGB)
    return image
```



2.1.2 Merging both the methods

```
def augmentImage(imageVector):
    augmentedImages = []
    rawImages = []

    image = toImage(imageVector)
    flippedImages = flipImage(image)
    flippedImages.append(image)

    coinTossOutcome = np.random.binomial(1, 0.5, len(flippedImages))
    for img, toss in zip(flippedImages, coinTossOutcome):
        if toss == 1:
            img = changeBrightness(img)
        augmentedImages.append(img)
        rawImages.append(toData(img))
    return augmentedImages, rawImages
```

The above method flips the provided input image and randomly selects few and changes the brightness of the image. This method uses the two methods mentioned above. In the end for each image three more images are generated.



2.2 One hot encoding

The target values needs to be encoded as one hot vectors for training. For this the sklearn toolkit can be used.

```
encoder = OneHotEncoder()
trainRawY = encoder.fit_transform(np.array(trainRawY).reshape(-1,1)).todense()
testY = encoder.transform(np.array(testY).reshape(-1,1)).todense()

"""Training data augmentation"""
trainX = []
trainY = []

for x, y in zip(trainRawX, trainRawY):
    rawAugmentedImages = augmentImage(x)[0]
    trainX.extend(rawAugmentedImages)
    target = [y for i in range(0, len(rawAugmentedImages))]
    trainY.extend(target)
```

3. Batch creation

Now the data size is too large and neural network cannot train on entire data in one pass. Hence it is necessary that the data is broken into smaller batches.

```
def batchIterator(x, y, batchSize, batchCount):
    size = len(x)
    if batchSize * batchCount > size:
        raise ValueError("Change batch size or change batch count")

    indices = list(range(0, size))
    shuffle(indices)
    indices = indices[0:batchSize * batchCount]
    batches = np.array_split(indices, batchCount)
    for batch in batches:
        yield (x[batch], y[batch])
```

The above method creates an iterator for iterating over training data. One epoch of training data will be one pass over this dataset.

The data needed by tensorflow is not a list but a matrix of 4 dimensions. [imageIndex, rows, columns, channels]. Currently the data is in list format and needs to be converted into this format.

```
trainX = np.stack(trainX, axis=0)
trainY = np.stack(trainY, axis=0)


processedTestX = []
processedTestY = []

for x, y in zip(testX, testY):
    processedTestY.append(y)
    processedTestX.append(toImage(x))

processedTestX = np.stack(processedTestX, axis=0)
processedTestY = np.stack(processedTestY, axis=0)
```

4. Tensorflow helper methods

4.1 Create convolution layer

```
def createConvolutionLayer(inputLayer, kernelHeight,
                kernelWidth, channelSize, kernelCount, strideX, strideY):
    """This will create a four dimensional tensor
    In this tensor the first and second dimension define the kernel height and width
    The third dimension define the channel size. If the input layer is
    first layer in neural network then the channel size will be 3 in case of RGB images
    else 1 if images are grey scale. Furthermore if the input layer is Convolution layer
    then the channel size should be no of kernels in previous layer"""


    weights = tf.Variable(tf.truncated_normal([kernelHeight, kernelWidth,
     channelSize, kernelCount], stddev=0.03))
    bias = tf.Variable(tf.constant(0.05, shape=[kernelCount]))

    """Stride is also 4 dimensional tensor
    The first and last values should be 1 as they represent the image index and
    chanel size padding. Second and Third index represent the X and Y strides"""
    layer = tf.nn.conv2d(input = inputLayer, filter = weights, padding='SAME',
                         strides = [1, strideX, strideY, 1]) + bias
    return layer
```

The input to tensorflow cnn is kernel. This kernel dimensions are

[kernelHeight, kernelWidth, channelSize, kernelCount].

This method will return tensorflow cnn layer.

4.2 Flatten

```
def flattenLayer(inputLayer):
    """Flatten layer. The first component is image count which is useless"""
    flattenedLayer = tf.reshape(inputLayer, [-1,
     inputLayer.get_shape()[1:].num_elements()])
    return flattenedLayer
```

The output of convolution layer and pooling layer will be tensors with four dimensions. The above method flattens such tensors.

4.3 FC layers

```
def fullyConnectedLayer(inputLayer, outputLayerCount):
    weights = tf.Variable(tf.truncated_normal(
                        [int(inputLayer.get_shape()[1]), outputLayerCount], stddev=0.03))
    bias = tf.Variable(tf.constant(0.05, shape=[outputLayerCount]))
    layer = tf.matmul(inputLayer, weights) + bias
    return layer
```

The above method creates standard fully connected layer for neural network.

4.4 Batch Normalization

```
def batchNormalization(inputLayer, isTraining):
    beta = tf.Variable(tf.constant(0.0, shape=[inputLayer.get_shape()[-1]]),
     trainable=True)
    gamma = tf.Variable(tf.constant(1.0, shape=[inputLayer.get_shape()[-1]]),
     name='gamma', trainable=True)
    batchMean, batchVariance = tf.nn.moments(inputLayer, [0,1,2],
```

```
                                        name='moments')
    ema = tf.train.ExponentialMovingAverage(decay=0.5)

    def meanVarianceUpdate():
        emaOp = ema.apply([batchMean, batchVariance])
        with tf.control_dependencies([emaOp]):
            return tf.identity(batchMean), tf.identity(batchVariance)

    mean, var = tf.cond(isTraining, meanVarianceUpdate, lambda:
              (ema.average(batchMean), ema.average(batchVariance)))
    normed = tf.nn.batch_normalization(inputLayer, mean, var, beta, gamma, 1e-3)
    return normed
```

The above method creates batch normalization layer. For this the first task is to create the beta and gamma layer parameters. Gamma scales the values in tensors while beta provides offset that is to be added to the normalized tensor. If the mode is training then the batch mean and batch variance is decayed by decaying factor of 0.3.

4.5 Histogram generation

Tensorboard allows to generate different charts and graphs. However to have a better control, all the charts are generated manually using summary wrapper.

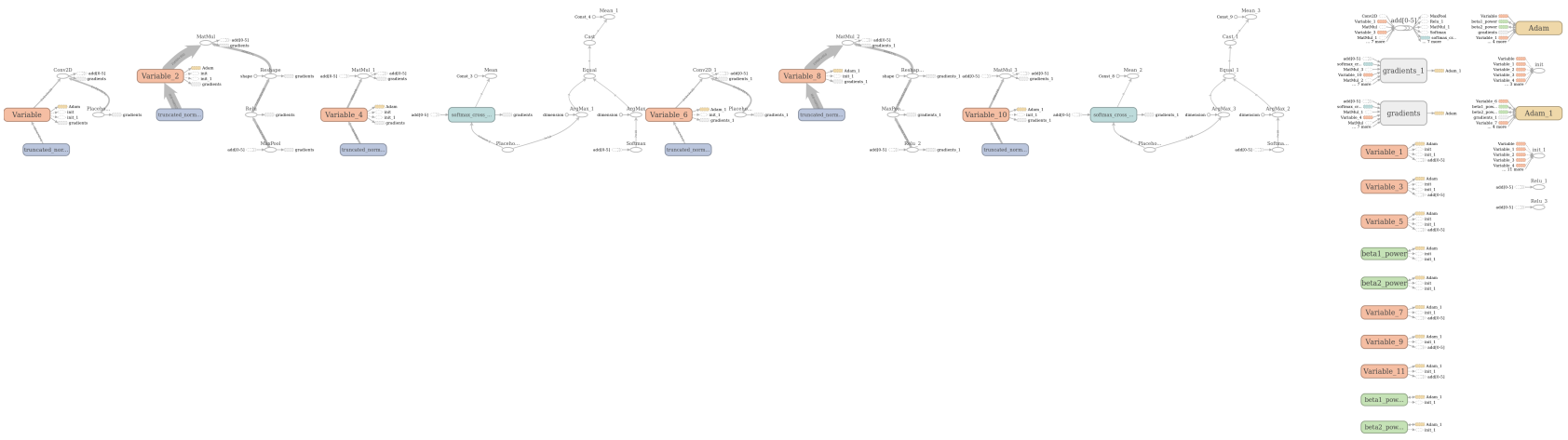
```
def log_histogram(writer, tag, values, step, bins=1000):
    # Convert to a numpy array
    values = np.array(values)

    # Create histogram using numpy
    counts, bin_edges = np.histogram(values, bins=bins)

    # Fill fields of histogram proto
    hist = tf.HistogramProto()
    hist.min = float(np.min(values))
    hist.max = float(np.max(values))
    hist.num = int(np.prod(values.shape))
    hist.sum = float(np.sum(values))
    hist.sum_squares = float(np.sum(values**2))

    bin_edges = bin_edges[1:]

    # Add bin edges and counts
    for edge in bin_edges:
        hist.bucket_limit.append(edge)
    for c in counts:
        hist.bucket.append(c)

    # Create and write Summary
    summary = tf.Summary(value=[tf.Summary.Value(tag=tag, histo=hist)])
    writer.add_summary(summary, step)
    writer.flush()
```

5 Training Model#1

5.1 Structure

The structure of this model includes a convolution layer with relu followed by pooling followed by a fully connected layer with relu activation and output layer with softmax.

```
"""Input is 4 dimensional tensor -1 so that the no of images can be infered on itself"""
inputLayer = tf.placeholder(tf.float32, [None, 32, 32, 3])
yTrue = tf.placeholder(tf.float32, shape=[None, 10])

convolutionLayer1 = createConvolutionLayer(inputLayer, 2, 2, 3, 25, 1, 1)
reluActivatedLayer1 = tf.nn.relu(convolutionLayer1)
poolingLayer1 = tf.nn.max_pool(value=reluActivatedLayer1, ksize=[1, 1, 2, 1],
                strides = [1, 1, 1, 1], padding='SAME')
flattened = flattenLayer(poolingLayer1)
fc1 = fullyConnectedLayer(flattened, 850)
reluActivatedLayer2 = tf.nn.relu(fc1)
fc = fullyConnectedLayer(fc1, 10)
```



5.2 Loss function

For training the neural network softmax is taken as a loss function and Adam is used for optimization process

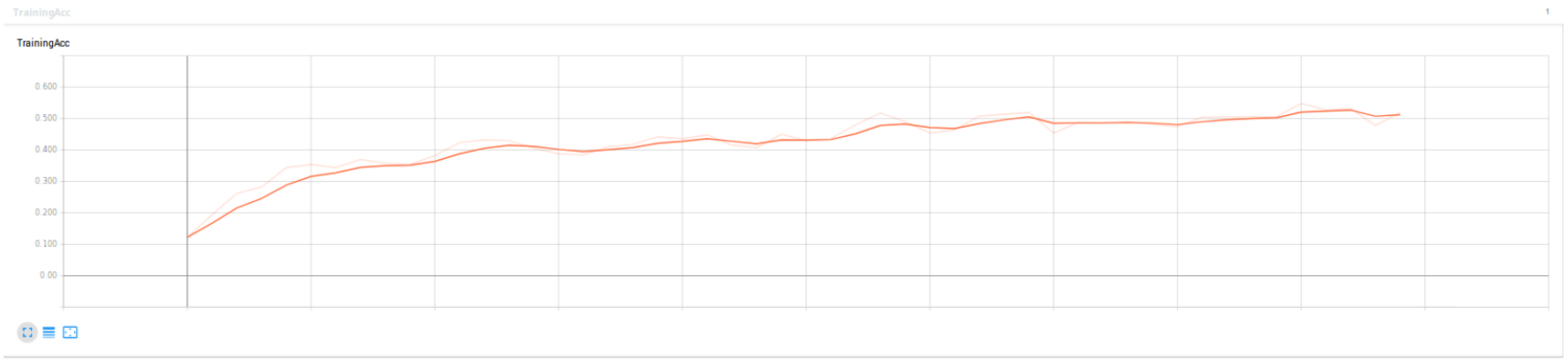
```
predictions = tf.argmax(tf.nn.softmax(fc), axis = 1)
actual = tf.argmax(yTrue, axis = 1)
loss = tf.nn.softmax_cross_entropy_with_logits_v2(logits=fc, labels = yTrue)
costFunction = tf.reduce_mean(loss)
optimizer = tf.train.AdamOptimizer().minimize(costFunction)
accuracy = tf.reduce_mean(tf.cast(tf.equal(predictions, actual), tf.float32))
```

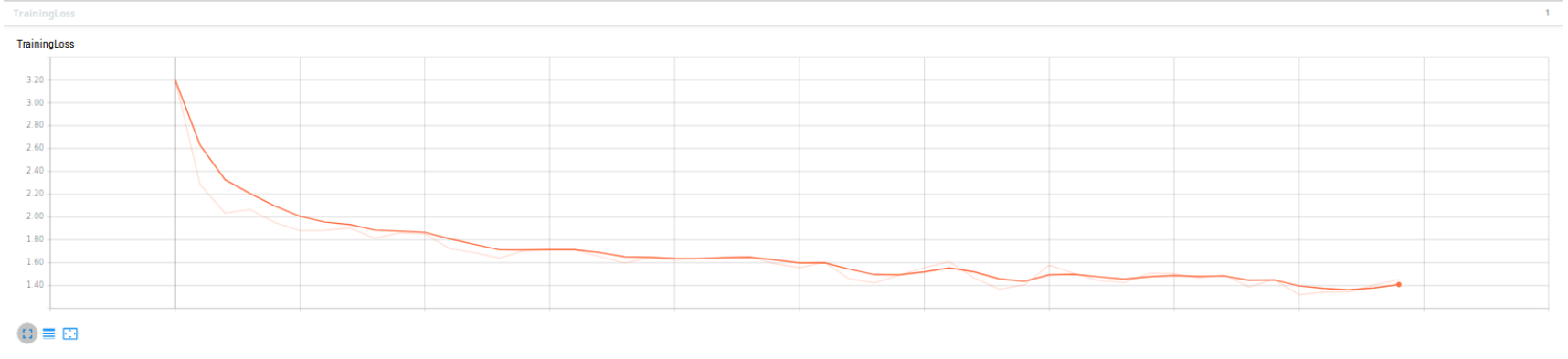
5.3 Training

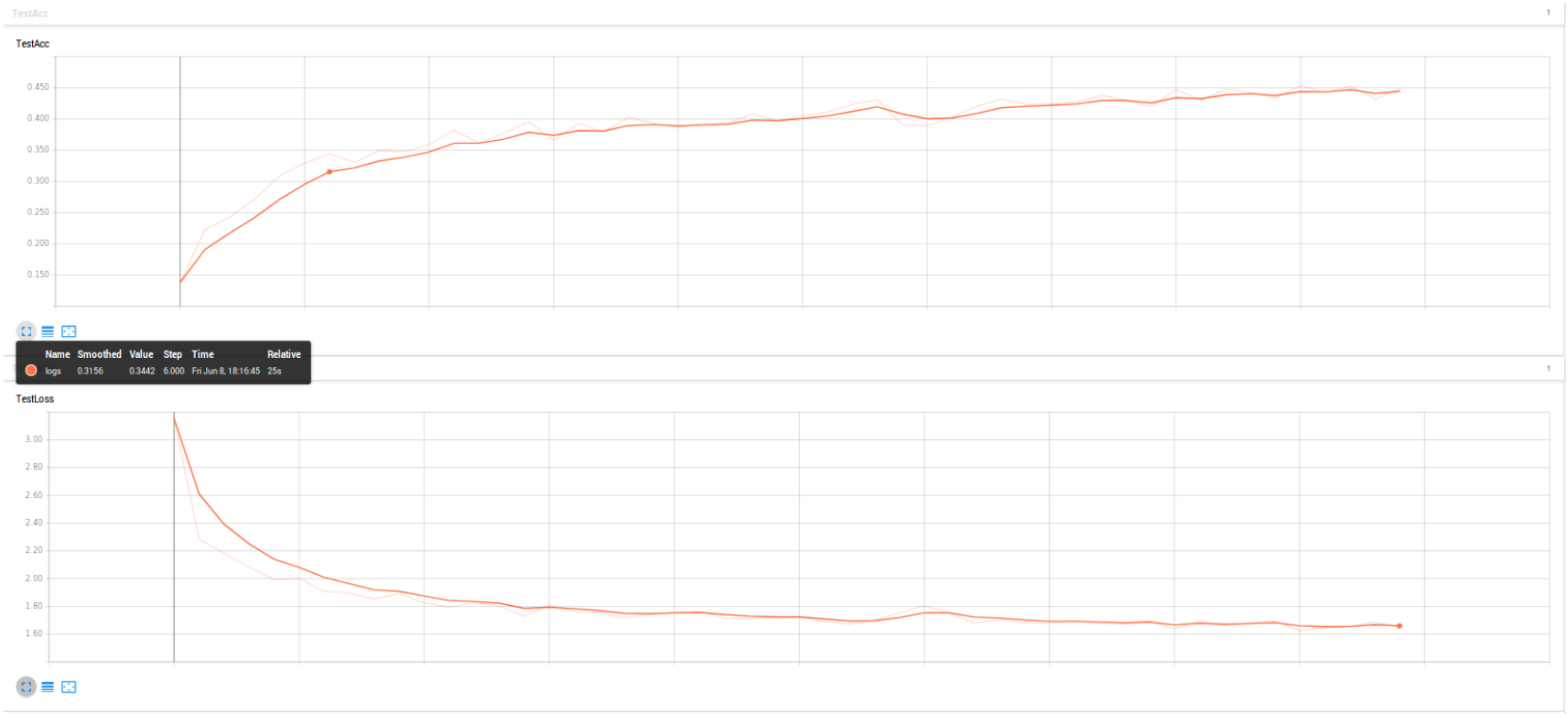
```
session = tf.Session()
"""Initialize the global variables"""
session.run(tf.global_variables_initializer())

summaryWriter =  tf.summary.FileWriter("tensorboard/structure1/logs",
                           graph=tf.get_default_graph())
trainAccList = []
testAccList = []
for i in range(0, 50):
    print("Epoch"+str(i))
    summary = tf.Summary()

    for x, y in batchIterator(trainX, trainY, 500, 50):
        session.run(optimizer, feed_dict={inputLayer:x, yTrue:y})

    loss = session.run(costFunction, feed_dict={inputLayer:x, yTrue:y})
    acc = session.run(accuracy, feed_dict={inputLayer:x, yTrue:y})
    summary.value.add(tag = "TrainingLoss", simple_value = loss)
    summary.value.add(tag = "TrainingAcc", simple_value = acc)
    trainAccList.append(acc)

    lossTestList = []
    accTestList = []
    for x, y in batchIterator(processedTestX, processedTestY, 1000, 5):
        lossTest = session.run(costFunction, feed_dict={inputLayer:x, yTrue:y})
        accTest = session.run(accuracy, feed_dict={inputLayer:x, yTrue:y})
        lossTestList.append(lossTest)
        accTestList.append(accTest)
    summary.value.add(tag = "TestLoss", simple_value = np.mean(lossTestList))
    summary.value.add(tag = "TestAcc", simple_value = np.mean(accTestList))
    testAccList.append(np.mean(accTestList))
    summaryWriter.add_summary(summary, i)
log_histogram(summaryWriter, "TrainAccHist", trainAccList, 50)
log_histogram(summaryWriter, "TestAccHist", testAccList, 50)
session.close()
```

5.4 Performance

The model was trained for 50 epochs and at end of 50 epochs the train accuracy was 52% and test accuracy was 44%







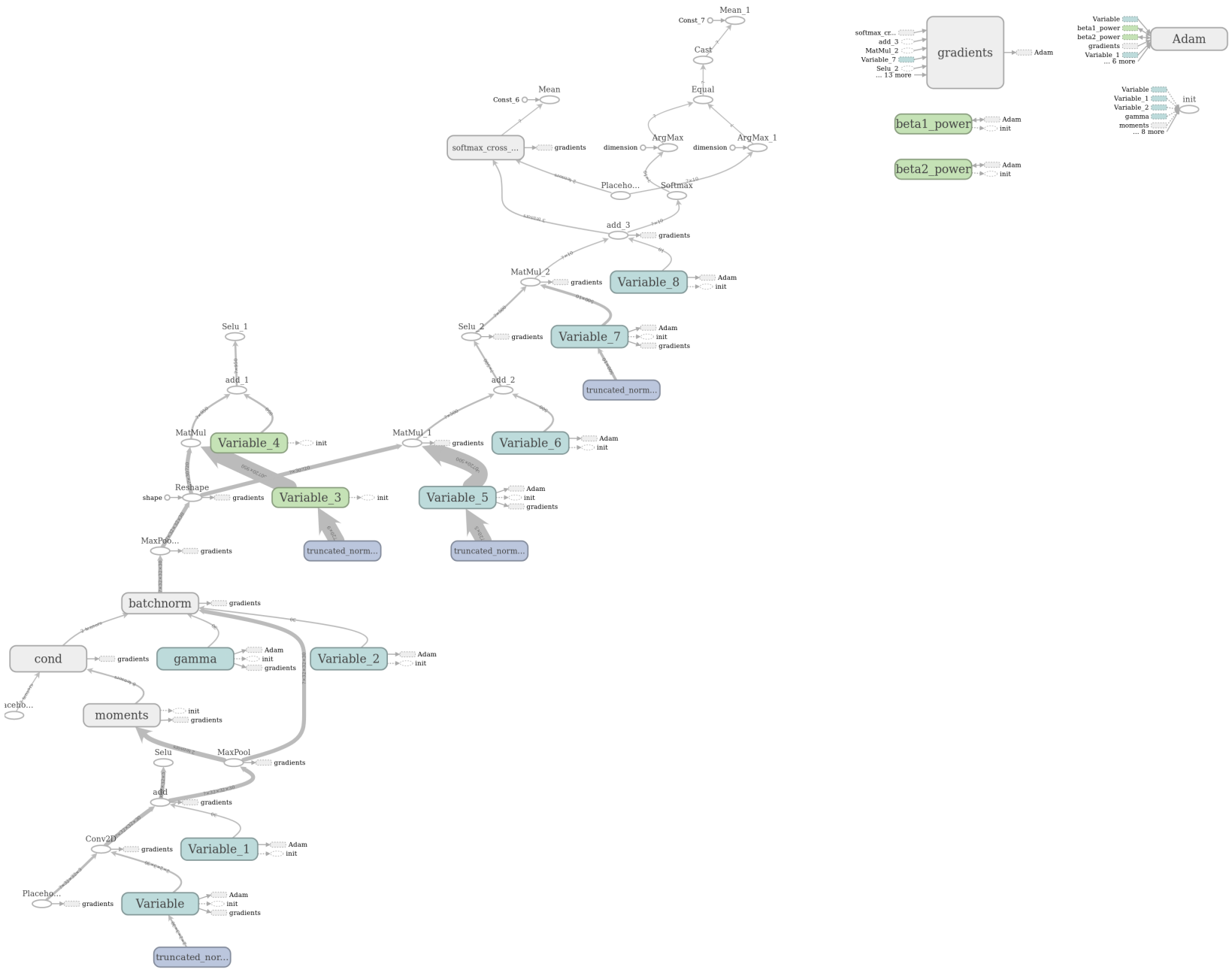
6 Training Model#2

6.1 Structure

For second setup Convolution layer is followed by selu layer followed by pooling followed by batch normalization followed by pooling followed by two fc with selu activation function.

```
"""Input is 4 dimensional tensor -1 so that the no of images can be infered on itself"""
inputLayer = tf.placeholder(tf.float32, [None, 32, 32, 3])
yTrue = tf.placeholder(tf.float32, shape=[None, 10])
isTraining = tf.placeholder(tf.bool, [])


convolutionLayer1 = createConvolutionLayer(inputLayer, 2, 2, 3, 30, 1, 1)
seluActivatedLayer1 = tf.nn.selu(convolutionLayer1)
poolingLayer1 = tf.nn.max_pool(value=convolutionLayer1, ksize=[1, 1, 2, 1],
```

```
                    strides = [1, 1, 1, 1], padding='SAME')
bn1 = batchNormalization(poolingLayer1, isTraining)
poolingLayer2 = tf.nn.max_pool(value=bn1, ksize=[1, 1, 2, 1],
          strides = [1, 1, 1, 1], padding='SAME')

flattened = flattenLayer(poolingLayer2)
fc1 = fullyConnectedLayer(flattened, 950)
seluActivatedLayer2 = tf.nn.selu(fc1)
fc2 = fullyConnectedLayer(flattened, 500)
seluActivatedLayer3 = tf.nn.selu(fc2)
fc= fullyConnectedLayer(seluActivatedLayer3, 10)
```



6.2 Loss function

```
predictions = tf.argmax(tf.nn.softmax(fc), axis = 1)
actual = tf.argmax(yTrue, axis = 1)
loss = tf.nn.softmax_cross_entropy_with_logits_v2(logits=fc, labels = yTrue)
costFunction = tf.reduce_mean(loss)
optimizer = tf.train.AdamOptimizer().minimize(costFunction)
accuracy = tf.reduce_mean(tf.cast(tf.equal(predictions, actual), tf.float32))
```

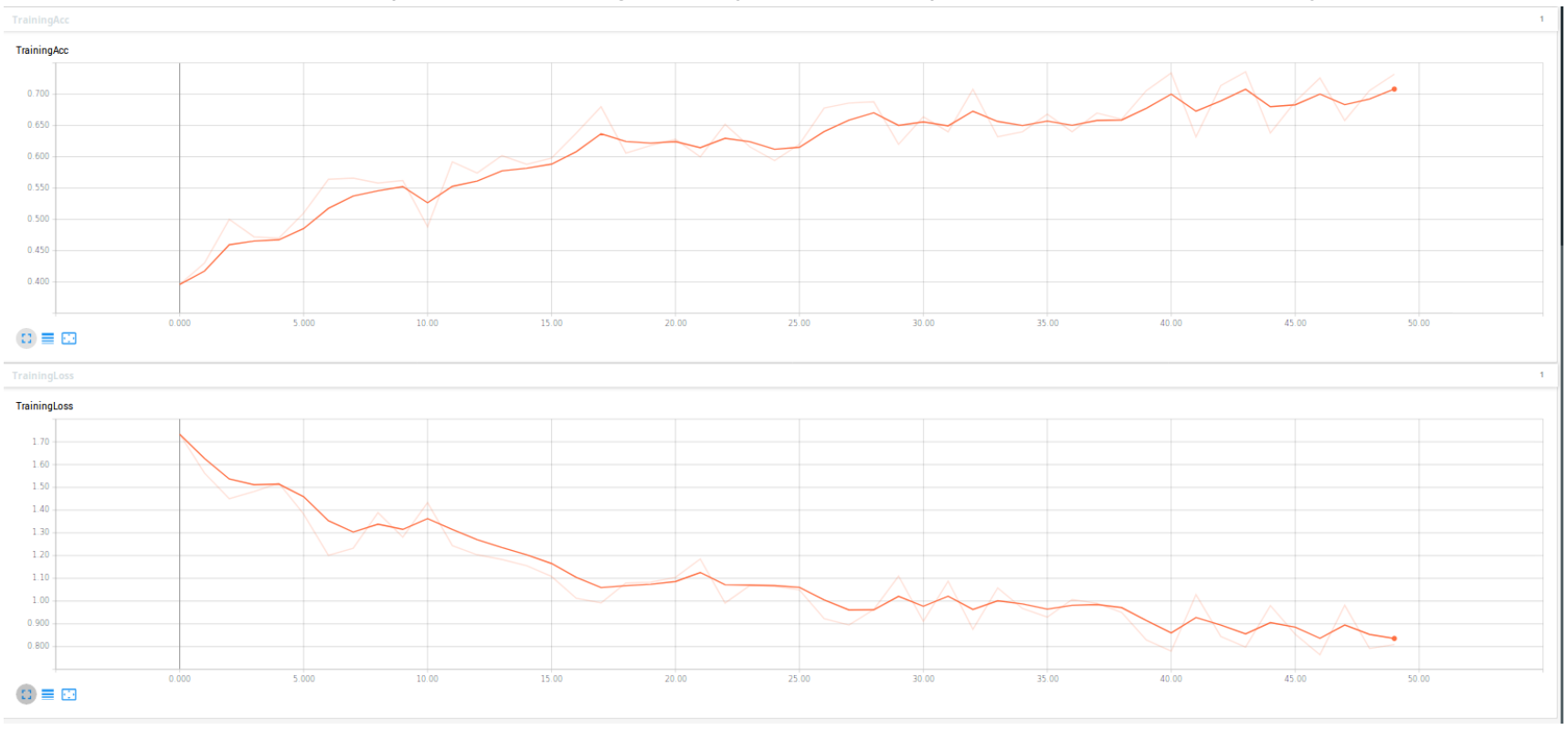
6.3 Training

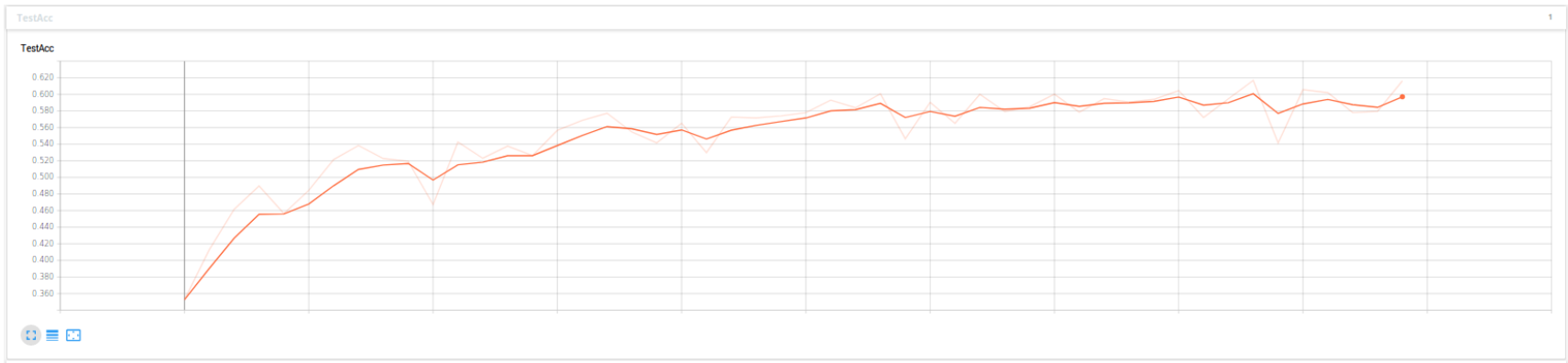
```
summaryWriter =  tf.summary.FileWriter("tensorboard/structure2/logs",
                 graph=tf.get_default_graph())
trainAccList = []
testAccList = []
for i in range(0, 50):
    print("Epoch"+str(i))
    summary = tf.Summary()

    for x, y in batchIterator(trainX, trainY, 500, 50):
        session.run(optimizer, feed_dict={inputLayer:x, yTrue:y, isTraining:True})

    loss = session.run(costFunction, feed_dict={inputLayer:x, yTrue:y, isTraining:False})
    acc = session.run(accuracy, feed_dict={inputLayer:x, yTrue:y, isTraining:False})
    summary.value.add(tag = "TrainingLoss", simple_value = loss)
    summary.value.add(tag = "TrainingAcc", simple_value = acc)
    trainAccList.append(acc)

    lossTestList = []
    accTestList = []
```

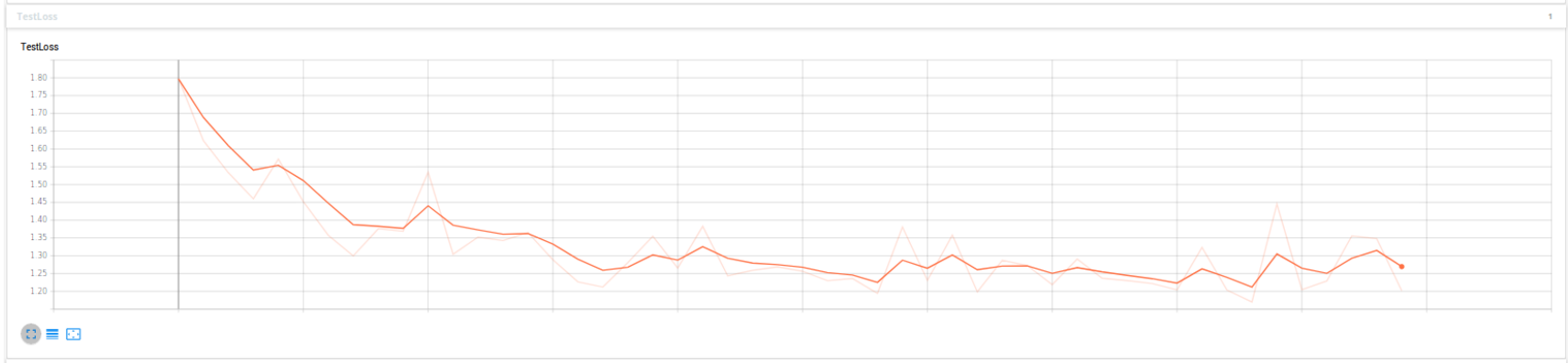
```
        for x, y in batchIterator(processedTestX, processedTestY, 1000, 5):
            lossTest = session.run(costFunction,
                           feed_dict={inputLayer:x, yTrue:y, isTraining:False})
            accTest = session.run(accuracy,
                      feed_dict={inputLayer:x, yTrue:y, isTraining:False})
            lossTestList.append(lossTest)
            accTestList.append(accTest)
        summary.value.add(tag = "TestLoss", simple_value = np.mean(lossTestList))
        summary.value.add(tag = "TestAcc", simple_value = np.mean(accTestList))
        testAccList.append(np.mean(accTestList))
        summaryWriter.add_summary(summary, i)
```

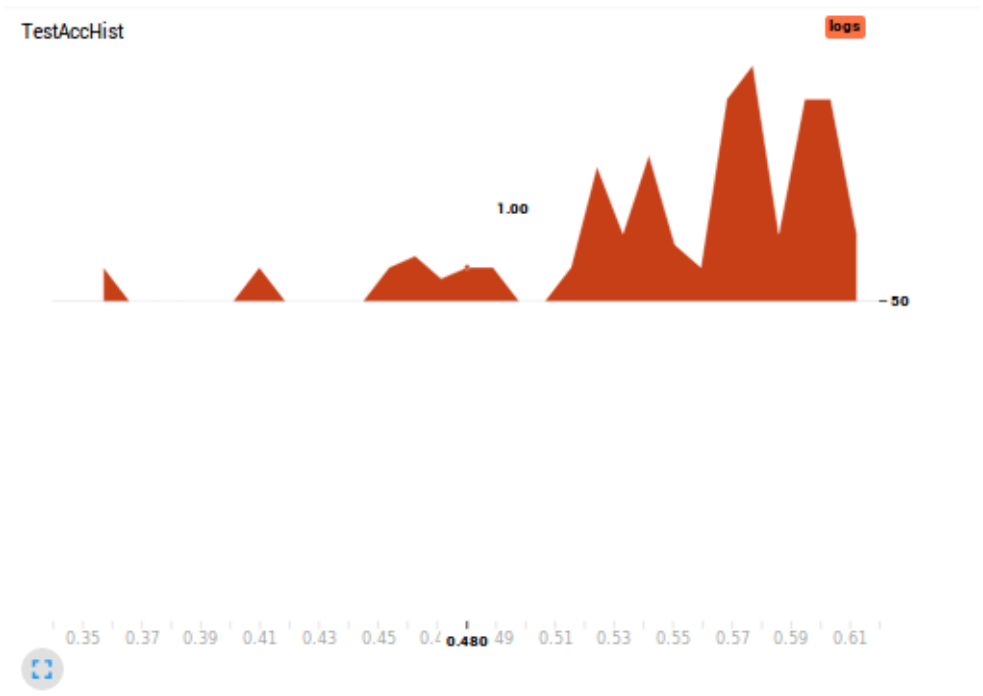
6.1 Performance

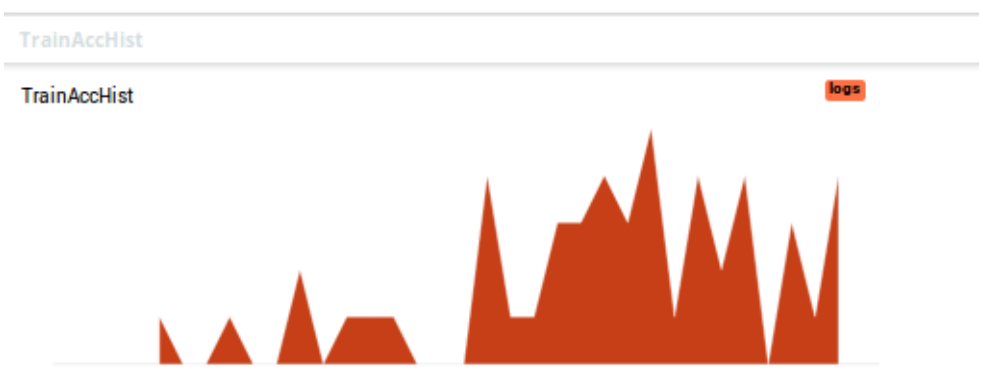
The model was trained for 50 epochs. The training accuracy at end of 50 epochs is 73% and test accuracy is 61%




TestAcc
TestAcc
0.620
0.600
0.580
0.560
0.540
0.520
0.500
0.480
0.460
0.440
0.420
0.400
0.380
0.360


TestLoss
TestLoss
1.80
1.75
1.70
1.65
1.60
1.55
1.50
1.45
1.40
1.35
1.30
1.25
1.20


TestAccHist
logs
1.00
50
0.35 0.37 0.39 0.41 0.43 0.45 0.480 0.51 0.53 0.55 0.57 0.59 0.61


TrainAccHist
TrainAccHist
logs


0.36 0.38 0.40 0.42 0.44 0.46 0.48 0.50 0.52 0.54 0.56 0.58 0.60 0.62 0.64 0.66 0.68 0.70 0.72 0.74