3116 – Lab Distributed Data Analytics – Group 2

Exercise Sheet 6

Yuvaraj Prem Kumar

303384, premyu@uni-hildsheim.de

Part 1: Research Paper

Convolutional neural networks (CNNs) are the current state-of-the-art model architecture for image classification tasks. CNNs apply a series of filters to the raw pixel data of an image to extract and learn higher-level features, which the model can then use for classification. A regular CNN will have a sequence of layers, and every layer of transforms one volume of activations to another through a differentiable function[3]. We use three main types of layers to build CNN architectures: Convolutional Layer, Pooling Layer, and Fully-Connected Layer. These layers to form a full CNN architecture.

- **Convolutional layers**, which apply a specified number of convolution filters to the image. For each subregion, the layer performs a set of mathematical operations to produce a single value in the output feature map.
- **Pooling layers,** which down-sample the image data extracted by the convolutional layers to reduce the dimensionality of the feature map in order to decrease processing time
- **Fully-connected layers,** which perform classification on the features extracted by the convolutional layers and down-sampled by the pooling layers. In a dense layer, every node in the layer is connected to every node in the preceding layer.

In the research paper[4], we are implementing a Multi-Channels Deep Convolutional Neural Network (MC-DCNN). The paper is analysing univariate and multivariate time-series dataset. The authors present their novel approach for deep learning of time-series classifications, through the use of MC-DCNN.

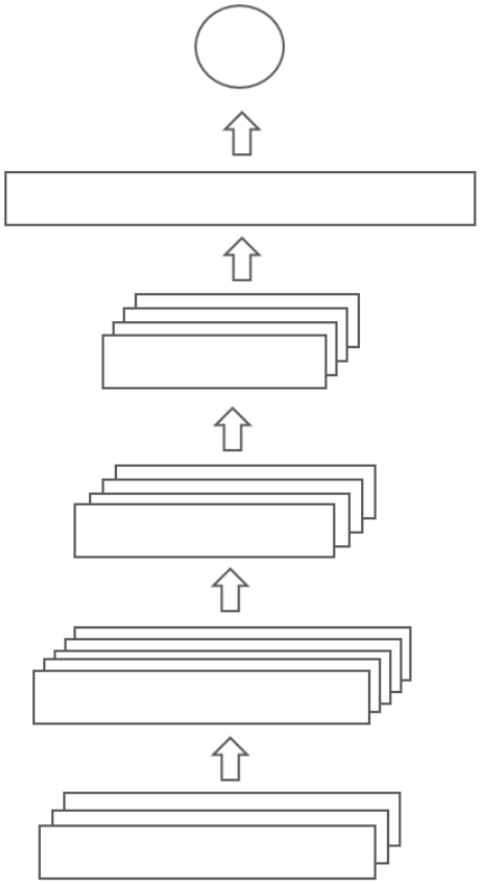
Architecture of MC-DCNN:

Instead of 2D image pixels as per regular use of CNNs, here the classification is of multiple 1D time-series classifications. Hence, there are 3-channels input with length (window size) of 256. Each channel is fed into a 2-stages feature extractor which learns features through filter layer, activation layer, and pooling layer. For example, input channel is passed to the first 1D convolution layer (filter and activation), then goes into the first 1D pooling layer, then into the second 1D convolution layer, into the second 1D pooling layer. After the features are learned, it is passed to the fully-connected layer, after that is flatten and the softmax is computed, giving the final output (classifications). The diagram [5] shows the overall flow, along with the dimensions. The dimensions are from the source dataset and this is shown in Part 2.

This flowchart follows the Figure 3 from the reseach paper and the following template

$$C1(Size) - S1 - C2(Size) - S2 - H - O$$

$$8(5) - 2 - 4(5) - 2 - 732 - 4$$



Output [N,4]
$softmax(X_{FC} * W_{out} + B_{out})$

Fully-connected [732 hidden layers]
$flatten(Pool2)$
$sigmoid(X * W_d + B_3)$

AvgPool2 , C2 (1D Convolution) – [5,8*3,4*3]
$flatten(Pool2)$
$sigmoid(X * W_d + B_3)$

AvgPool1 [C1]

C1 (1D Convolution) – [5,3,8*3]
$Kernel[5], nChannels[3], depth[8]$
$sigmoid(X * W_{wc1} + B_1)$

3-channel input [N,256,3]
x,y,z components of IMU ankle

This is basically Section 3.2 in flowchat form, I added some helper explanations here too. The paper details the approach for the gradient-based learning of this MC-DCNN. The main part is the backpropagation algorithm used to train the model. Here, stochastic gradient descent (SGD) is used (mini-batches) as it could converge faster for this large-scale dataset. The full cycle is Feedforward Pass → Backpropagation Pass → Gradient Applied.

Feedforward Pass: The output feature map is using the sigmoid activation function. Since the data is passed through a sliding window, we are actually operating on the sub-sequences.

Backpropagation Pass: Here we are sending the feedback to the MC-DCNN by computing derivates. After predicted output, $\hat{y}$, the predicted error will be calculated by the set loss function. The paper implements zero padding before sliding over with the Kernel (size=5), but for this implementation, no zero padding is needed.

Gradients Applied: It is clearly defined to used momentum=0.9 and decay=0.005. When training CNNs, it is common to use weight decay. After each update, the weights are multiplied by 0.005. This prevents the weights from growing too large, and can be seen as gradient descent on a quadratic regularization term [6].

Part 2: Research Paper Implementation

Step 1: Data Pre-processing

The most important step, and also the hardest part. There is no clear guideline in the research paper so most of it is left to my intuition. However, a close study of the PAMAP2 readme and in [1], the dataset creator defines which IMU measurement data is relevant.

The paper states; only 4 out of 19 activities are selected 'standing', 'walking', 'ascending stairs, and 'descending stairs'. As in [1], this means that the relevant IMU position is IMU Ankle measurements. Each physical activity is a 3D time series, because of the 3D-acceleration data and corresponding activity. Hence, I am using the "3D-acceleration data (ms-2), scale: ±16g, resolution: 13-bit" measurement as the observations, with "activityID" as target.

- Def function 'generate data' to output xTrain, yTrain, xTest, and yTest
- The train / test split is mentioned as leave-one-out cross validation (LOOCV). However, using the python module implementation (`from sklearn.model_selection import LeaveOneOut`) takes too long to run, and causes RAM overflow for my machine. Hence a more direct approach is chosen – one subject will be the test dataset, and the rest the training dataset. This is rotated.
- Subject 108 and 109 are dropped, because 108 is left-handed, and 109 does not perform the relevant activities. This is also the approached outlined in the research paper.
- Read each file via pandas.read_csv. Fill NaNs with mean of other values, and drop the unwanted rows and columns.
- Normalize the data, using $\frac{x-\mu}{\sigma}$. As recommended in [2], pre-processing should centre the data to have mean of zero, and normalize its scale to [-1, 1] along each feature. The mean must be computed only over the training data and then subtracted equally from all splits (train and test).
- Sliding window is applied, based in the sliding step of 128 and window size of 256. The research paper uses various sliding steps of {128,64,32,16,8} but that is not feasible with my machine. Hence, I could only run it with sliding step of 128, which already takes a very long time to slide across the entire dataset.
- Finally we have two functions to implement all of the above: (1) 'generate_data()' and (2) 'sliding_window'.

Finally, the input for the model can be called via the functions as below example:

```
xTrain, yTrain = generate_data(train_filelist, sliding_step=128)
print('Shape of X_Train: ', xTrain.shape)  # (3891, 256, 3)
print('Shape of Y_Train: ', yTrain.shape)  # (3891, 4)

print('Generating testing dataset')
xTest, yTest = generate_data(test_filelist, sliding_step=128)
print('Shape of X_Test: ',xTest.shape)  # (584, 256, 3)
print('Shape of Y_Test: ',yTest.shape)  # (584, 4)
```

Step 2: Dimensions of model and features

Based on the flowchart in Part 1 and shape of input X, Y ; the following variables are defined.

```
# From shape of input channels (IMU Ankle)
nWidth = 256
nChannels = 3
nLabels = 4
# Stated in research paper
kernel_size = 5
depth = 8
nHidden = 732
```

Step 3: TensorFlow graph parameters

Now that we have the dimensions, we can define the graph X, Y and Weights and Biases dictionary. The paper utilizes decay and momentum to update the weights, this is applied using the related TensorFlow functions (L2 Regularizer).

```
# TensorFlow graph
X = tf.placeholder(tf.float32, [None, nWidth, nChannels])
Y = tf.placeholder(tf.float32, [None, nLabels])

weight_decay = tf.constant(0.0005, dtype=tf.float32)
weights = {
    'wc1': tf.get_variable('W0',shape=([kernel_size,nChannels,depth*nChannels]),initializer=tf.random_normal_initializer,
                           regularizer=tf.contrib.layers.l2_regularizer(weight_decay)),
    'wc2': tf.get_variable('W1',shape=([kernel_size,depth*nChannels,nLabels*nChannels]),initializer=tf.random_normal_initializer,
                           regularizer=tf.contrib.layers.l2_regularizer(weight_decay)),
    'wd1': tf.get_variable('W2',shape=([nHidden,12]),initializer=tf.random_normal_initializer,
                           regularizer=tf.contrib.layers.l2_regularizer(weight_decay)),
    'out': tf.get_variable('W3',shape=([12,nLabels]),initializer=tf.random_normal_initializer,
                           regularizer=tf.contrib.layers.l2_regularizer(weight_decay)),
}

biases = {
    'b1': tf.get_variable('B1',shape=([depth*nChannels]),initializer=tf.random_normal_initializer),
    'b2': tf.get_variable('B2',shape=([nLabels*nChannels]),initializer=tf.random_normal_initializer),
    'b3': tf.get_variable('B3',shape=([12]),initializer=tf.random_normal_initializer),
    'out': tf.get_variable('B4',shape=([nLabels]),initializer=tf.random_normal_initializer),
}
```

Step 4: Helper functions for 1D convolution and 1D average pooling

As outlined in Part 1, we are using 'sigmoid' activation function, and 1D average pooling (instead of maxpool). Unlike the paper, there is no need for zero padding here, hence the parameter is set to 'VALID'.

```
01. def conv1d(X,feature_maps,bias):
02.     conv = tf.nn.conv1d(X,feature_maps,stride=1,padding="VALID")
03.     conv = tf.nn.sigmoid(tf.nn.bias_add(conv, bias))
04.     return conv
05.
06. def avgpool1d(conv):
07.     pooling = tf.layers.average_pooling1d(conv, pool_size=2, strides=2, padding='VALID')
08.     return pooling
```

Step 5: 1D MC-DCNN convolution network

Simply the main function to implement the algorithm in Part 1. The block of the code (and same for step 6) is adapted from the demonstration by the lab tutor Mofassir, for Group 2 students. The inputs are shaped appropriately, and the code block is adapted from conv2d to conv1d.

```
01. def conv_net(X,weights,biases):
02.     conv1 = conv1d(X,weights['wc1'],biases['b1'])  # First layer, feature maps of 252 per input channel
03.     pool1 = avgpool1d(conv1)  # 1-D average pooling, chooses avg. value in the matrix
04.
05.     conv2 = conv1d(pool1,weights['wc2'],biases['b2'])  # Second layer, feature map of 126 per input channel
06.     pool2 = avgpool1d(conv2)  # Avg pooling again
07.
08.     # Fully connected layer
09.     cnn = tf.layers.Flatten()(pool2)  # Reshape conv2 pooling output to fit fully connected layer input
10.     fc1 = tf.add(tf.matmul(cnn, weights['wd1']), biases['b3'])
11.     fc1 = tf.nn.sigmoid(fc1)  # Paper is using sigmoid activation function
12.
13.     out = tf.add(tf.matmul(fc1, weights['out']), biases['out'])
14.     out = tf.nn.softmax(out)  # Final output layer of 4 classes (corresponding to activity ID)
15.     return out
```

Step 6: TensorFlow session and TensorBoard summaries

The summaries operations for TensorBoard are initialized here. The TensorFlow session runs to calculate the training loss. Furthermore, I try to plot the training loss, training accuracy, and testing accuracy as per Lab 5. However, the results are not visually clear as compared to TensorBoard.

The loss function is the normal softmax cross entropy. I chose this as the research paper does not state which to use. For the optimizer, we are using stochastic gradient descent. The prediction and accuracy methodology are not stated explicitly, anyway I use the same as previous labs.

```
1.  pred=conv_net(X,weights,biases)
2.  loss=tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=pred,labels=Y))
3.  optimizer = tf.train.GradientDescentOptimizer(learning_rate = learning_rate).minimize(l
    oss)
4.  correct_prediction = tf.equal(tf.argmax(pred,1), tf.argmax(Y,1))
5.  accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
6.
7.  tf.summary.histogram("Activations", pred)
8.  tf.summary.scalar("Loss_on_train_set", loss)
9.  tf.summary.scalar("Accuracy_on_train_set", accuracy)
10. merged_all_ = tf.summary.merge_all()
```

Step 7: TensorFlow session.

The session is exactly the same as previous lab, with the addition of mini-batches for SGD optimization.

```
with tf.Session() as sess:
    init = tf.global_variables_initializer()
    sess.run(init)
    summary_writer = tf.summary.FileWriter('./tensorboard/train', graph=tf.get_default_graph())
    for epoch in range(training_epochs):
        for batch in range(total_batches):  # Mini-batch for SGD
            batch_x = xTrain[batch * batch_size:min((batch + 1) * batch_size, len(xTrain))]
            batch_y = yTrain[batch * batch_size:min((batch + 1) * batch_size, len(yTrain))]
            _, c, acc,summary = sess.run([optimizer, loss,accuracy, merged_all_], feed_dict={X: batch_x, Y: batch_y})
            training_loss.append(c)
            summary_writer.add_summary(summary, epoch)
        print('Epoch: ', epoch, 'Training Loss =', '{:.9f}'.format(c), 'Training accuracy= {:.5f}'.format(acc))
        train_accuracy.append(acc)
        _, corrtest = sess.run([optimizer, accuracy], feed_dict={X: xTest, Y: yTest})
        test_accuracy.append(corrtest)

    print("Testing Accuracy:", sess.run(accuracy, feed_dict={X: xTest, Y: yTest}))
```

Results:

The following parameters were used:

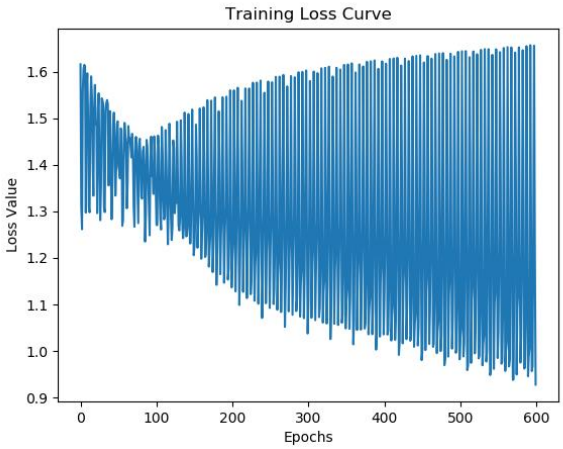
- Learning rate = 0.05
- Number of epochs = 20
- Batch_size = 128 ; this is usually from 10 to 1000 but should be less than 256 here

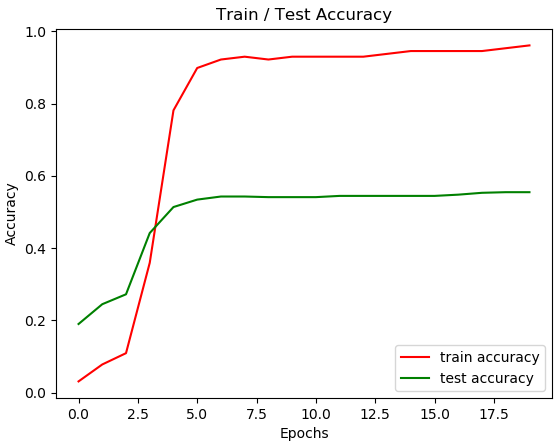
The following output is observed. Since I am using random_normal initializers, I find the result can vary greatly. So we should run a few times to get the average. The full screenshot of code output is shown in Appendix A. The table below shows the sample result-set.

Epoch	Training Loss	Training Accuracy
1	1.463750601	0.07812
2	1.377736449	0.10938
3	1.288345575	0.35938
5	1.142023802	0.89844
10	1.025771976	0.92969
14	0.980600953	0.94531
16	0.959082842	0.94531
18	0.938155472	0.95312
20	0.927849412	0.96094

Testing Accuracy: 0.55479455

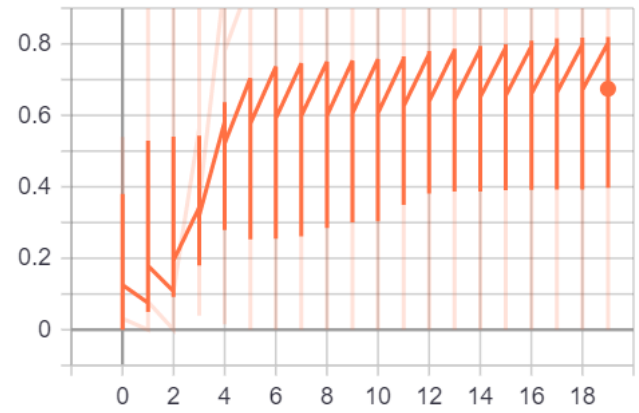
We can observe that the training accuracy reaches a very good value. However, testing accuracy is poor. I have tried changing multiple variables, but the maximum testing accuracy reached to 80++% (Appendix A). I have since been unable to re-create that due to fiddling with the parameters. The plotted graphs better illustrate the results.



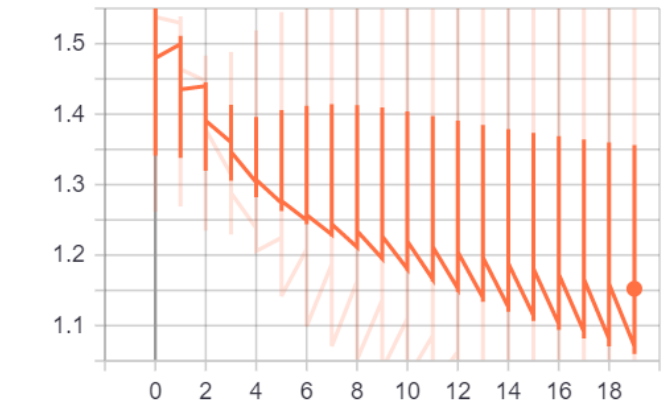


The TensorBoard graphs are as shown:

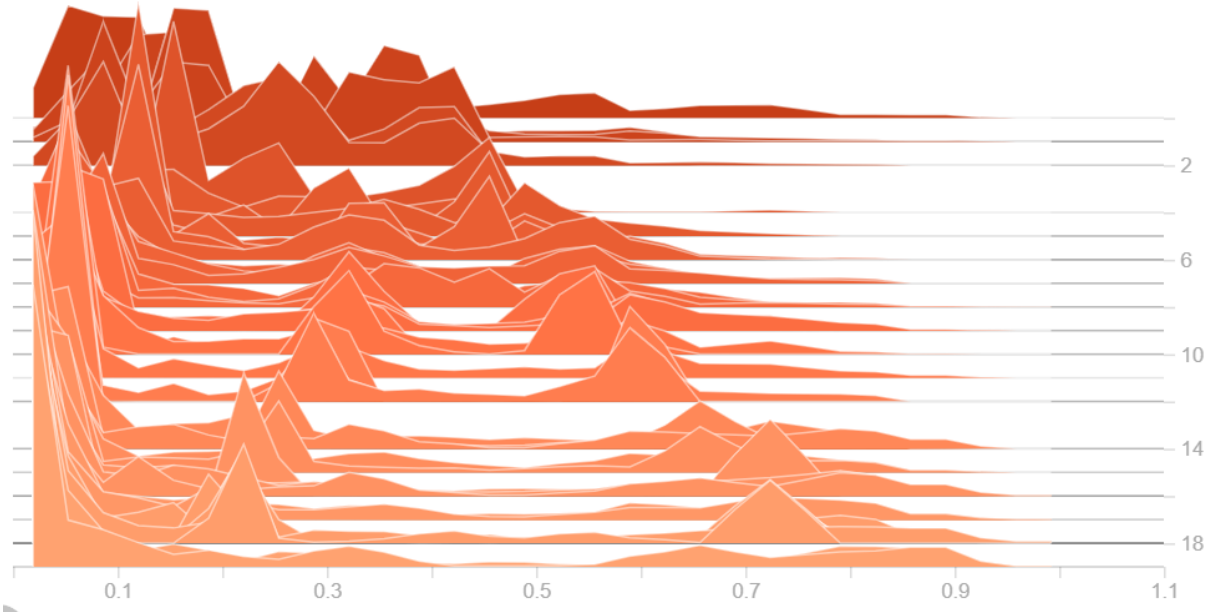
Accuracy_on_train_set



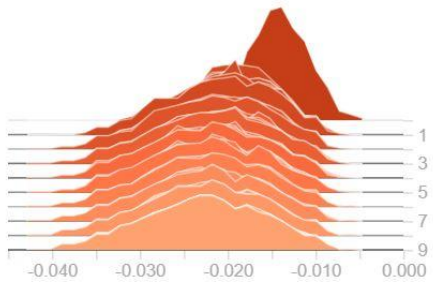
Loss_on_train_set



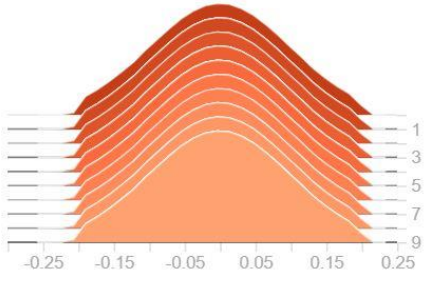
Activations histogram:



Biases



Weights



Discussion:

1. Basically, the paper leaves out many things, hence I took many assumptions, especially when prepossessing the data, and building the loss and prediction functions.
2. This is probably the main reason for the low testing accuracy
3. I was also severely limited by:
    a. My laptop has 8GB ram, and no CPU. Running TensorFlow for even 20 epochs on the MC-DCNN causes it to overheat and reach 100% CPU and 100% RAM usage before crashing.
    b. I have not taken machine learning or deep learning, so I tried my best effort to study about convolutional neural networks. Hence my limited understanding of the paper.
4. The paper can be further improved, for example using ReLU activation function as advised in [2]. Sigmoid function suffers from the problem of "vanishing gradients" as it flattens out at both ends, resulting in very small changes in the weights during backpropagation [3]. This can make the CNN refuse to learn and get 'stuck'.
5. We can further improve the model by regularization and dropout.
6. The PAMAP2 dataset was not really good to implement this MC-DCNN, looks like the BIDMC dataset would have been better as it is well-aligned.
7. This lab was a good exposure to studying and implementing a research paper.

References:

[1] https://kluedo.ub.uni-kl.de/frontdoor/deliver/index/docId/3681/file/_PhDThesis_AttilaReiss.pdf
[2] CS231n Convolutional Neural Networks for Visual Recognition. (n.d.). Retrieved from http://cs231n.github.io/neural-networks-2/
[3] CS231n Convolutional Neural Networks for Visual Recognition. (n.d.). Retrieved from http://cs231n.github.io/convolutional-networks/
[4] Zheng, Y., Liu, Q., Chen, E., Ge, Y., & Zhao, J. L. (2014, June). Time series classification using multi-channels deep convolutional neural networks. In International Conference on Web-Age Information Management (pp. 298-310). Springer, Cham.
[5] Saeed, A. (n.d.). Implementing a CNN for Human Activity Recognition in Tensorflow. Retrieved from http://aqibsaeed.github.io/2016-11-04-human-activity-recognition-cnn/
[6] Metacademy. (n.d.). Retrieved from https://metacademy.org/graphs/concepts/weight_decay_neural_networks

Appendix A – Sample code outputs

```
Generating training dataset
Reading data file:  C:/PythonProjects/PAMAP2_Dataset/Protocol/subject102.dat
Reading data file:  C:/PythonProjects/PAMAP2_Dataset/Protocol/subject103.dat
Reading data file:  C:/PythonProjects/PAMAP2_Dataset/Protocol/subject104.dat
Reading data file:  C:/PythonProjects/PAMAP2_Dataset/Protocol/subject105.dat
Reading data file:  C:/PythonProjects/PAMAP2_Dataset/Protocol/subject106.dat
Reading data file:  C:/PythonProjects/PAMAP2_Dataset/Protocol/subject107.dat
Shape of X_Train:  (3891, 256, 3)
Shape of Y_Train:  (3891, 3)
Generating testing dataset
Reading data file:  C:/PythonProjects/PAMAP2_Dataset/Protocol/subject101.dat
Shape of X_Test:  (584, 256, 3)
Shape of Y_Test:  (584, 3)
2019-06-17 22:43:43.072681: I tensorflow/core/platform/cpu_feature_gu
Epoch:  0 Training Loss = 1.537919283 Training accuracy= 0.03125
Epoch:  1 Training Loss = 1.463750601 Training accuracy= 0.07812
Epoch:  2 Training Loss = 1.377736449 Training accuracy= 0.10938
Epoch:  3 Training Loss = 1.288345575 Training accuracy= 0.35938
Epoch:  4 Training Loss = 1.205718279 Training accuracy= 0.78125
Epoch:  5 Training Loss = 1.142023802 Training accuracy= 0.89844
Epoch:  6 Training Loss = 1.099049091 Training accuracy= 0.92188
Epoch:  7 Training Loss = 1.071051598 Training accuracy= 0.92969
Epoch:  8 Training Loss = 1.051977634 Training accuracy= 0.92188
Epoch:  9 Training Loss = 1.037733078 Training accuracy= 0.92969
Epoch:  10 Training Loss = 1.025771976 Training accuracy= 0.92969
Epoch:  11 Training Loss = 1.014446497 Training accuracy= 0.92969
Epoch:  12 Training Loss = 1.003055096 Training accuracy= 0.92969
Epoch:  13 Training Loss = 0.991712093 Training accuracy= 0.93750
Epoch:  14 Training Loss = 0.980600953 Training accuracy= 0.94531
Epoch:  15 Training Loss = 0.969740212 Training accuracy= 0.94531
Epoch:  16 Training Loss = 0.959082842 Training accuracy= 0.94531
Epoch:  17 Training Loss = 0.948568344 Training accuracy= 0.94531
Epoch:  18 Training Loss = 0.938155472 Training accuracy= 0.95312
Epoch:  19 Training Loss = 0.927849412 Training accuracy= 0.96094
Testing Accuracy: 0.55479455

Process finished with exit code 0
```

```
WARNING:tensorflow:From C:\Python3\lib\site-packages\tensorflow\python\framework\op_def_library.py:263:
Instructions for updating:
Colocations handled automatically by placer.
2019-06-17 03:14:42.502257: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instr
Epoch #:  0  Training Loss =  0.048835546  Training Accuracy =  0.37490436
Epoch #:  1  Training Loss =  0.14880034  Training Accuracy =  0.38676357
Epoch #:  2  Training Loss =  0.41822672  Training Accuracy =  0.40703902
Epoch #:  3  Training Loss =  0.8544191  Training Accuracy =  0.71270084
Epoch #:  4  Training Loss =  0.78904986  Training Accuracy =  0.8117827
Epoch #:  5  Training Loss =  1.0841707  Training Accuracy =  0.85195106
Epoch #:  6  Training Loss =  0.9554863  Training Accuracy =  0.88523334
Epoch #:  7  Training Loss =  0.3678849  Training Accuracy =  0.9158378
Epoch #:  8  Training Loss =  0.17741774  Training Accuracy =  0.92578423
Epoch #:  9  Training Loss =  0.14988624  Training Accuracy =  0.9334353
Testing Accuracy = 0.86130136

Process finished with exit code 0
```

Appendix B – TensorBoard graphmap