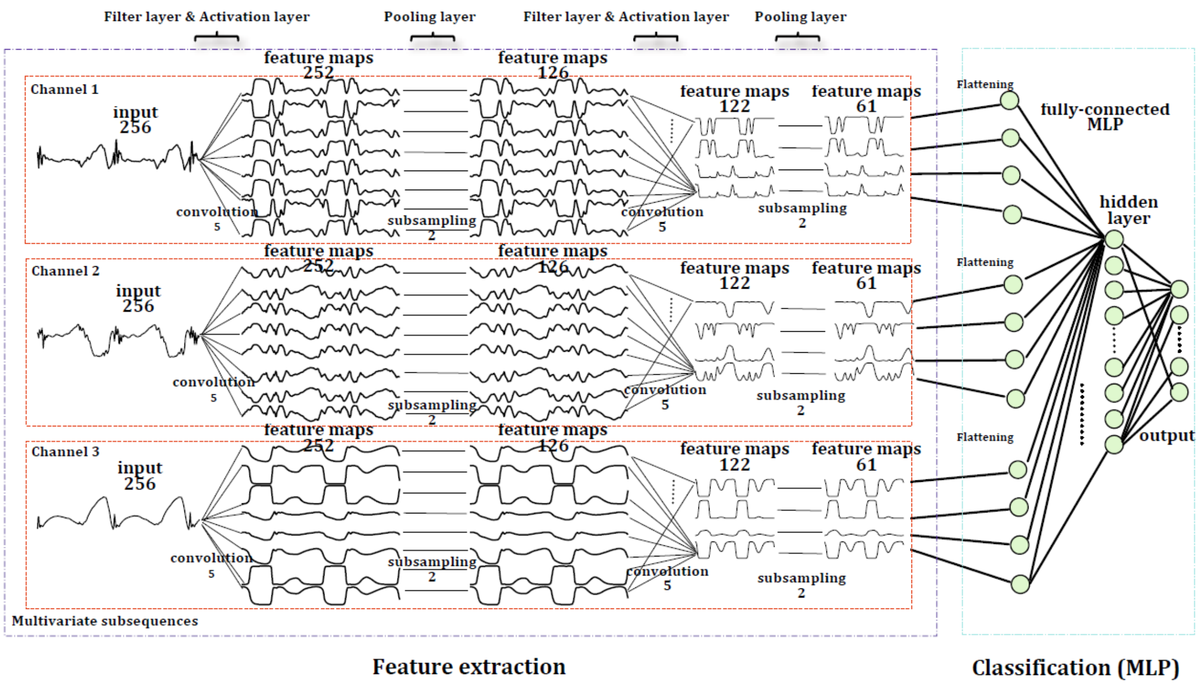
Lab Course - Distributed Data Analytics
Exercise 6

Part 1: Research Paper:

Architecture Design:



The paper states that the experiments are carried out in such a way that each physical activity corresponds to a 3D time series. Said so, the three channels shown in the architecture corresponds to each dimension of the time series. In each channel, 2 convolutions are 2 pooling are performed and the output are fully connected to the output layer with which the physical activity is predicted using softmax function. Each channel has an input time series of length 256 which is convolved with 8 layers of 1x5 Kernel with sigmoid activation layer to form a feature map of size 8x252. The feature maps are average pooled to size 8x126 with pool size of 2 and strides 2. The feature maps in stage 2 are convoluted with sigmoid activation layer to produce feature maps of size 4x122 in each channel. These are again average pooled to 4x61. Now, the feature maps from all the 3 channels are flattened to form a full-connected MLP of 732 neurons which is then connected to the output layer of 4 which is same as the number of target classes. The result from the output layer is fed into a softmax function to get the probability of each class. The one with highest value is taken the class for the time series.

The explained architecture is used in the following part of the assignment to predict the human physical activities. While training the model, the parameter updating procedure includes forward pass, backpropagation and the gradient applied. As during the forward pass, the layers are convoluted and average pooled which makes the size to shrink, during back propagation, the weights of the pooled layer are updated in such a way that the layer is up sampled using methods like transpose convolution and the gradients are calculated.

$$\frac{\partial E}{\partial \mathbf{x}_j^{l-1}} = up(\frac{\partial E}{\partial \mathbf{x}_j^{l}})$$

Similarly, in the filter layer, the gradients of weights are calculated based on the gradient of sigmoid and the gradient of up sampled pooled layer. The gradients of biases are the summation of the all the gradients.

$$\frac{\partial E}{\partial \mathbf{z}_j^l} = \frac{\partial E}{\partial \mathbf{x}_j^l}\frac{\partial \mathbf{x}_j^l}{\partial \mathbf{z}_j^l} = sigmoid^{'}(\mathbf{z}_j^l) \circ up(\frac{\partial E}{\partial \mathbf{x}_j^{l+1}})$$

$$\frac{\partial E}{\partial b_j^l} = \sum_u (\boldsymbol{\delta}_j^l)_u$$

The difference in the Kernel weights and MLP weights is that in Kernel, the weights are less dependent where as in MLP the weights are dependant to each other and only the weights that are similar. The gradient of the kernel weights propagate over all the layers and the quantities. As, the size of the feature maps are reduced by kernels, they are padded with zero while finding the gradients of the same.

$$\frac{\partial E}{\partial \mathbf{k}_{ij}^l} = \frac{\partial E}{\partial \mathbf{z}_j^l}\frac{\partial \mathbf{z}_j^l}{\partial \mathbf{k}_{ij}^l} = \boldsymbol{\delta}_j^l * reverse(\mathbf{x}_i^{l-1})$$

$$\frac{\partial E}{\partial \mathbf{x}_i^{l-1}} = \sum_j \frac{\partial E}{\partial \mathbf{z}_j^l}\frac{\partial \mathbf{z}_j^l}{\partial \mathbf{x}_i^{l-1}} = \sum_j pad(\boldsymbol{\delta}_j^l) * reverse(\mathbf{k}_{ij}^l)$$

Part 2: Research Paper Implementation:

In order to implement the research paper, I've used one of the dataset used by the author, PAMPA2 dataset. The dataset is a open-source for medical applications and has 19 different physical activities focussed on 9 subjects. Of which, the authors in the paper have chosen walking, standing, ascending stairs and descending stairs which are activities: 3, 4, 12 and 13. All the subject data are loaded iteratively and only the instances with required activities are selected and processed further.

Function name : data_split
Parameter : data to select required instances

```
def data_split(data):
    data_req = data[(data[1] == 3) | (data[1] == 4) | (data[1] == 12) | (data[1] == 13)]
    data_channel = data_req[[1,38,39,40]]
    data_time = data_req[[0]]
    data_channel.columns = [0,1,2,3]
    return data_time,data_channel
```

```
directory = "G:/DA - Hildeshim/DDA Lab/Exercise 6/Dastet/PAMAP2_Dataset/Protocol/"
for i in os.listdir(directory):
    data = pd.read_csv(directory + i,header=None,sep=" ")
    data_time,data_channel = data_split(data)
    data_channel = data_channel.reset_index()
    data_channel = data_channel.drop(columns="index")
    data_channel.fillna(method="ffill",inplace=True)
    data_channel.to_csv(directory+i[:-4]+".csv",index=None)
```

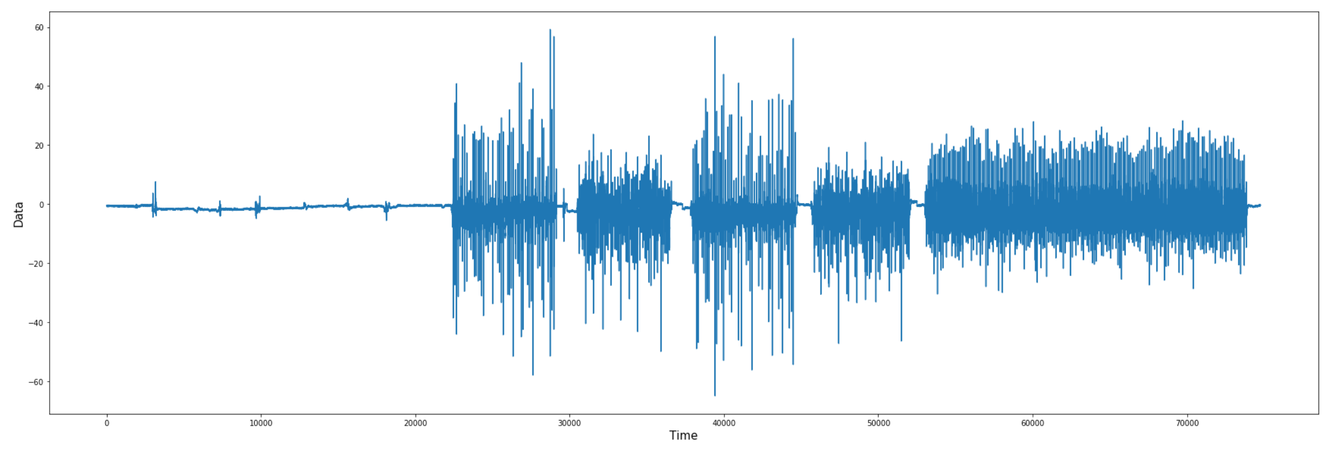
Once the required instances are selected, I used the 3D Acceleration data from IMU – Ankle sensor to predict the human activity from the given time-series. To prove this, Konzeptionierung eines Aktivit ¨ atsmonitoring-Systems f ¨ ur medizinische Applikationen mit dem 3D- ¨ Accelerometer der Sendsor GmbH. Master's thesis, Technische Universitat Munchen" are taken as reference.

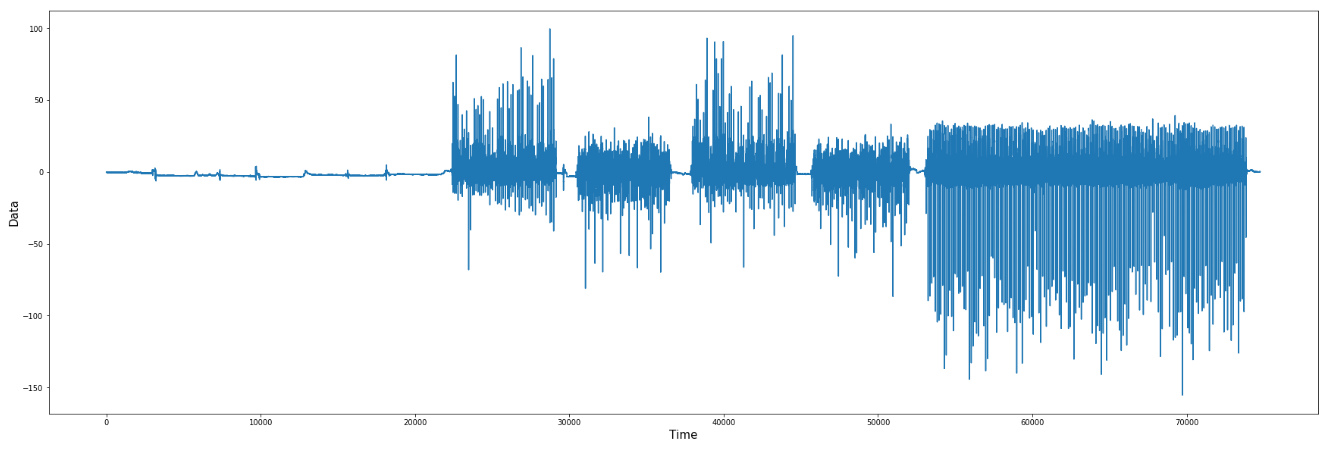
The following analysis was made on the 3D Acceleration data of the IMU sensors and based on the analysis, the ankle data seemed to be the best for classification among different activities.

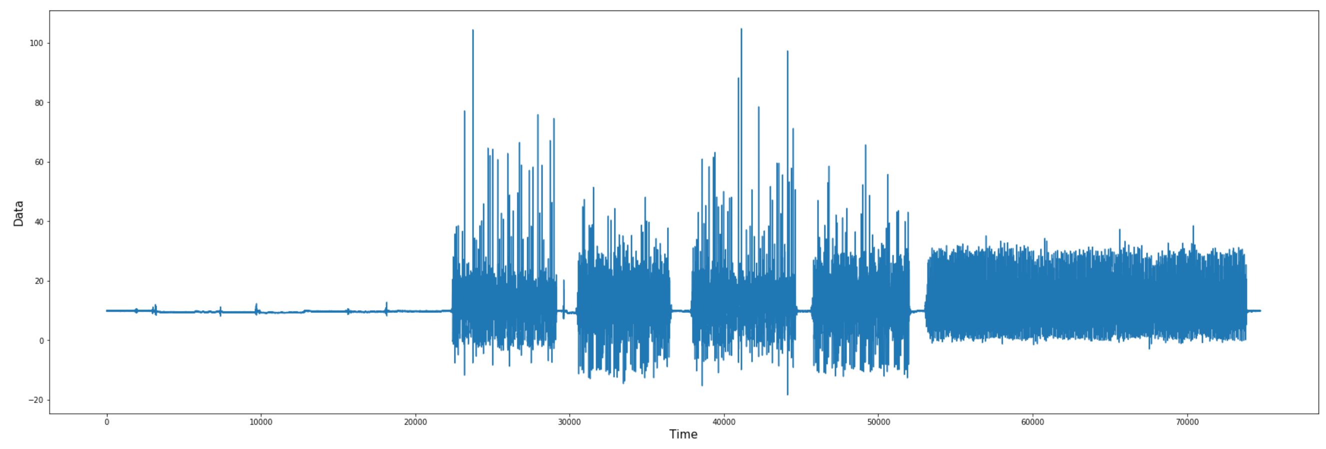
	start	end	Activity
0	0	21716	3
1	21716	29836	12
2	29836	37316	13
3	37316	45086	12
4	52505	74759	4

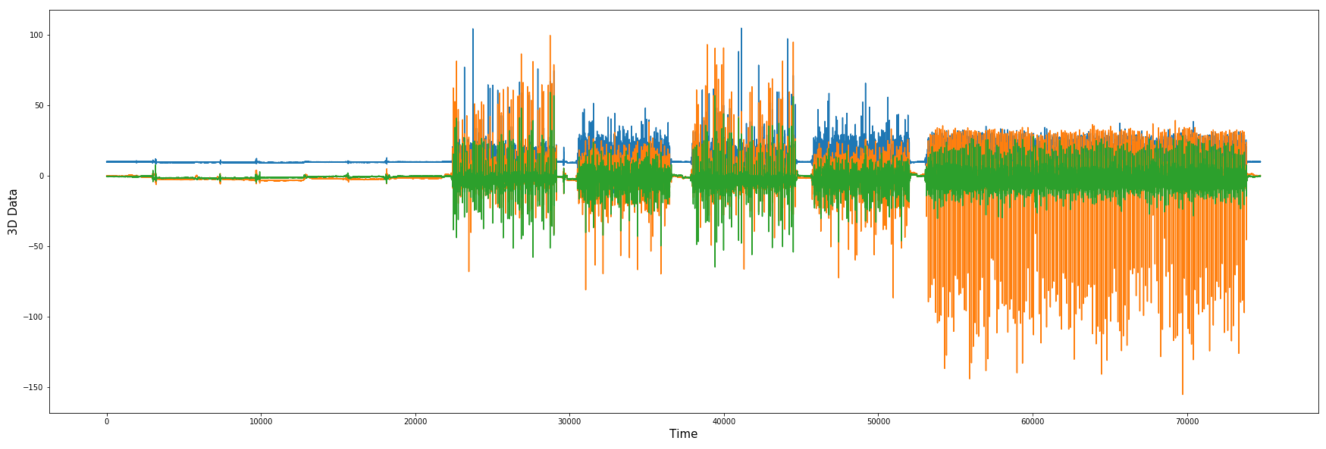
Activity statistics of Subject101

3D Acceleration Data from IMU Ankle of Subject 1

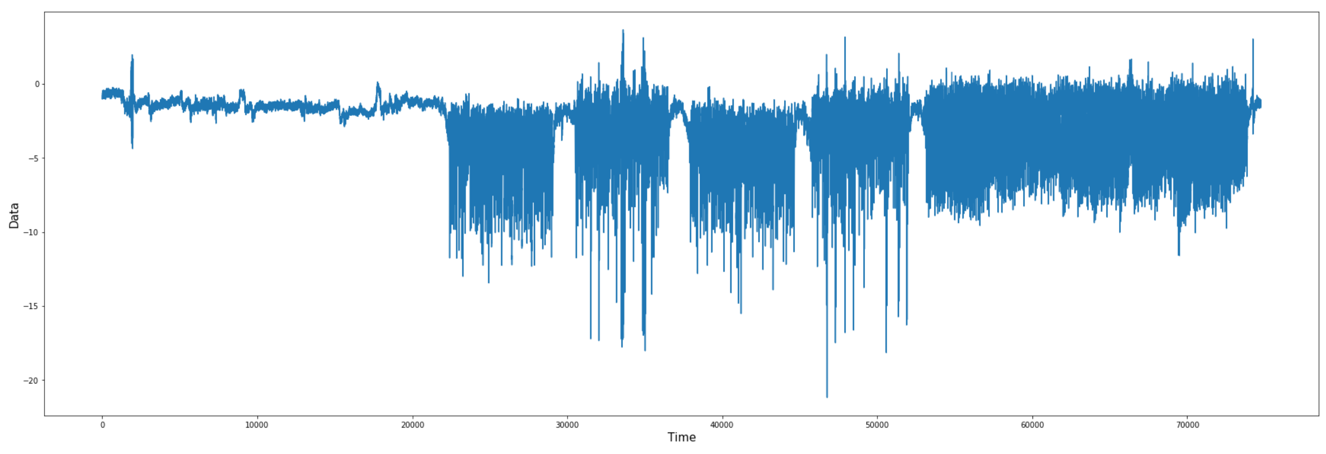


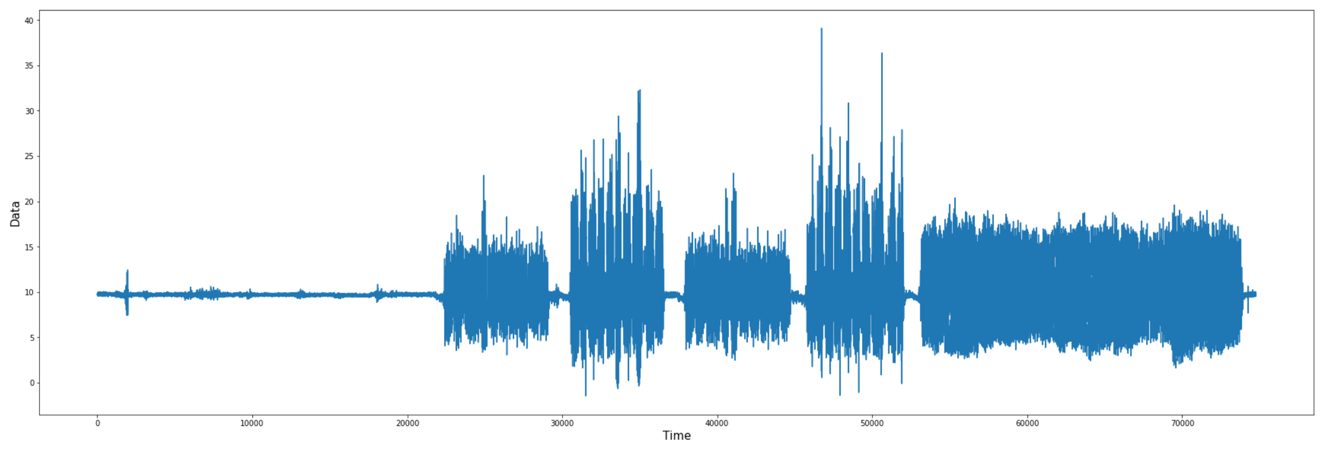


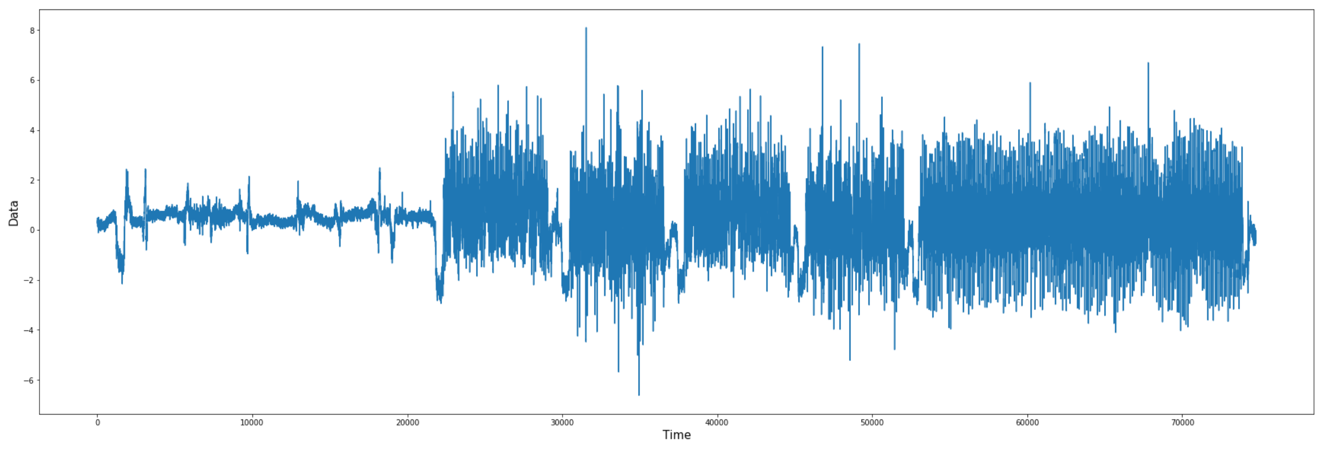


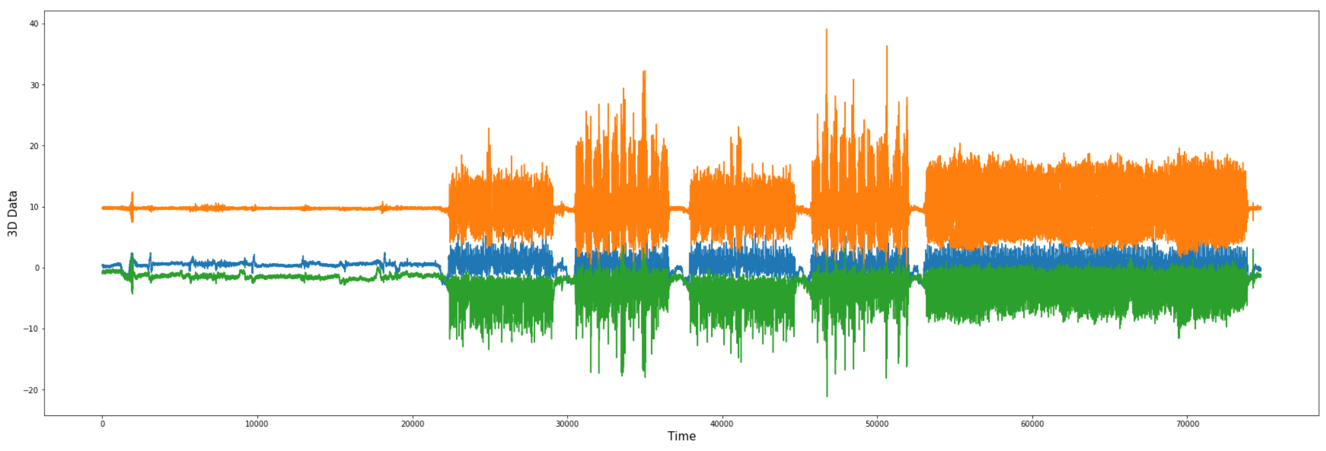


3D Acceleration Data from IMU Chest of Subject 1

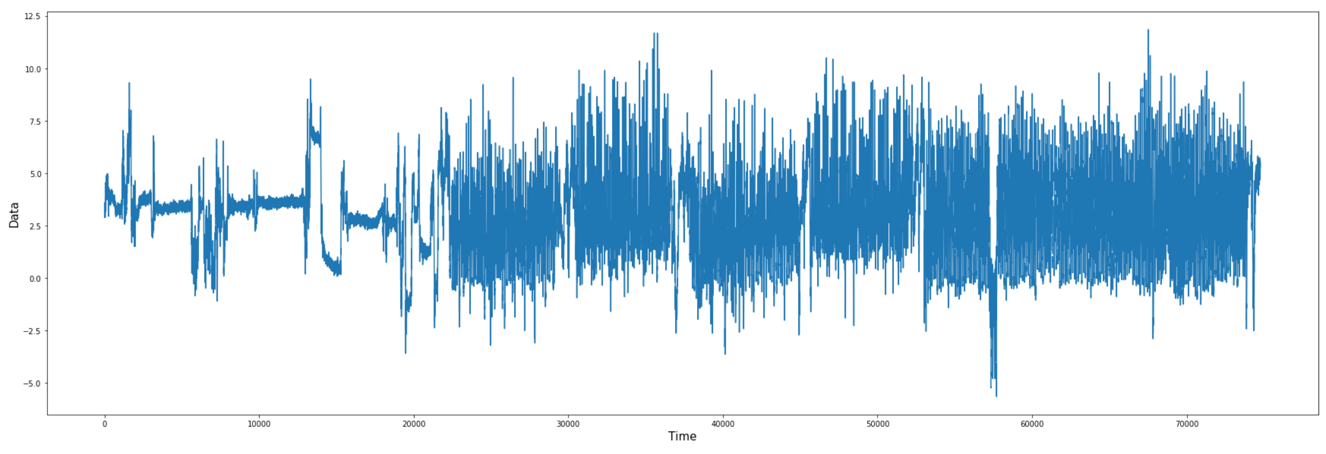


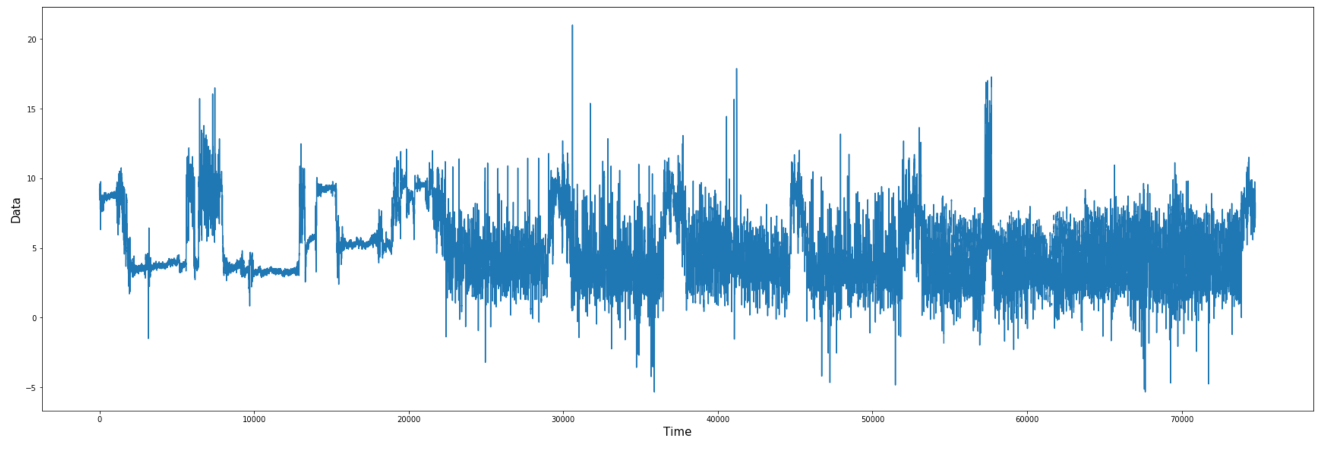


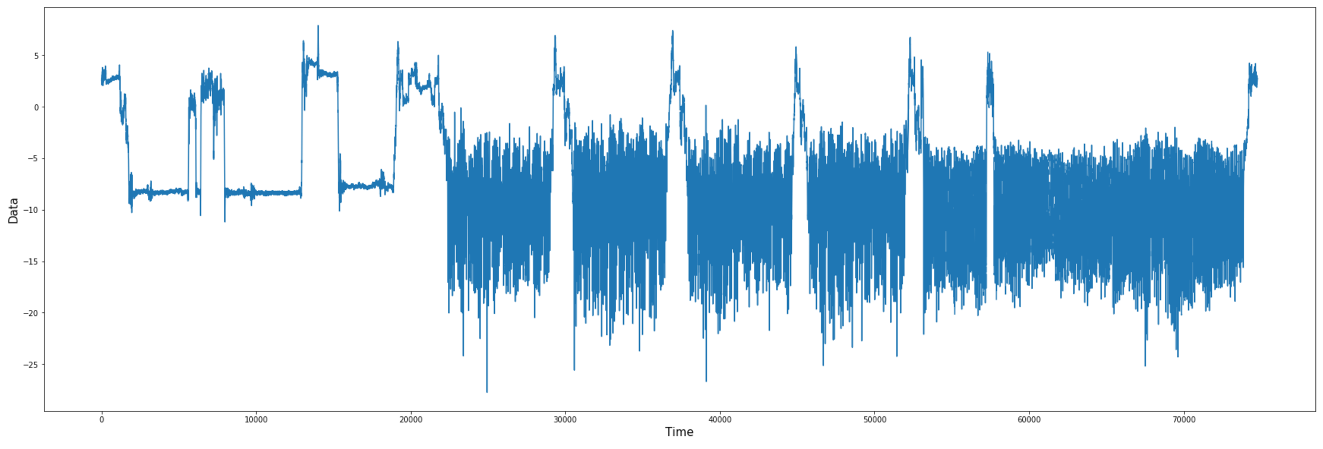


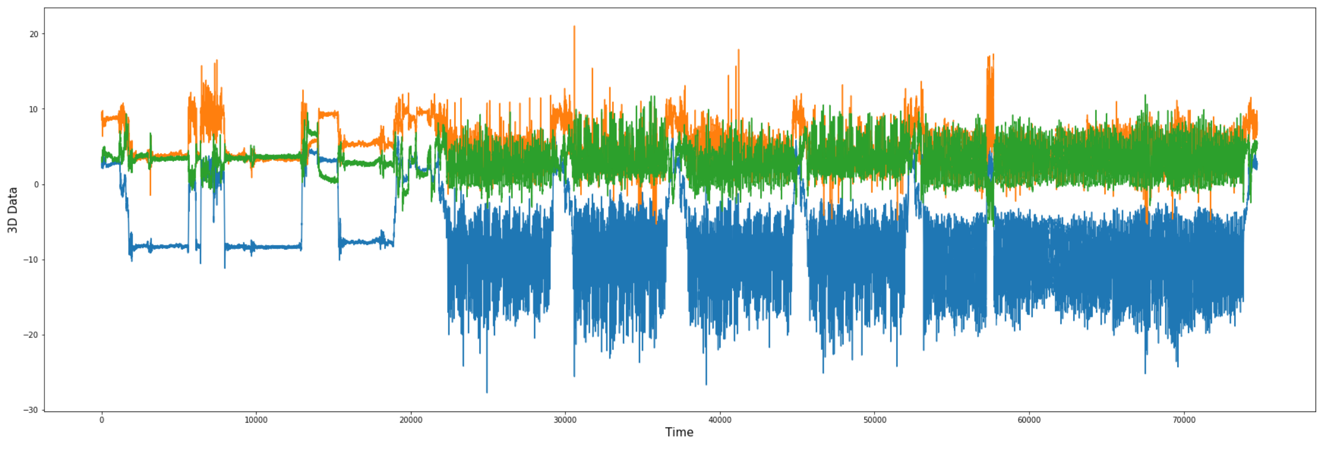


3D Acceleration Data from IMU Hand of Subject 1





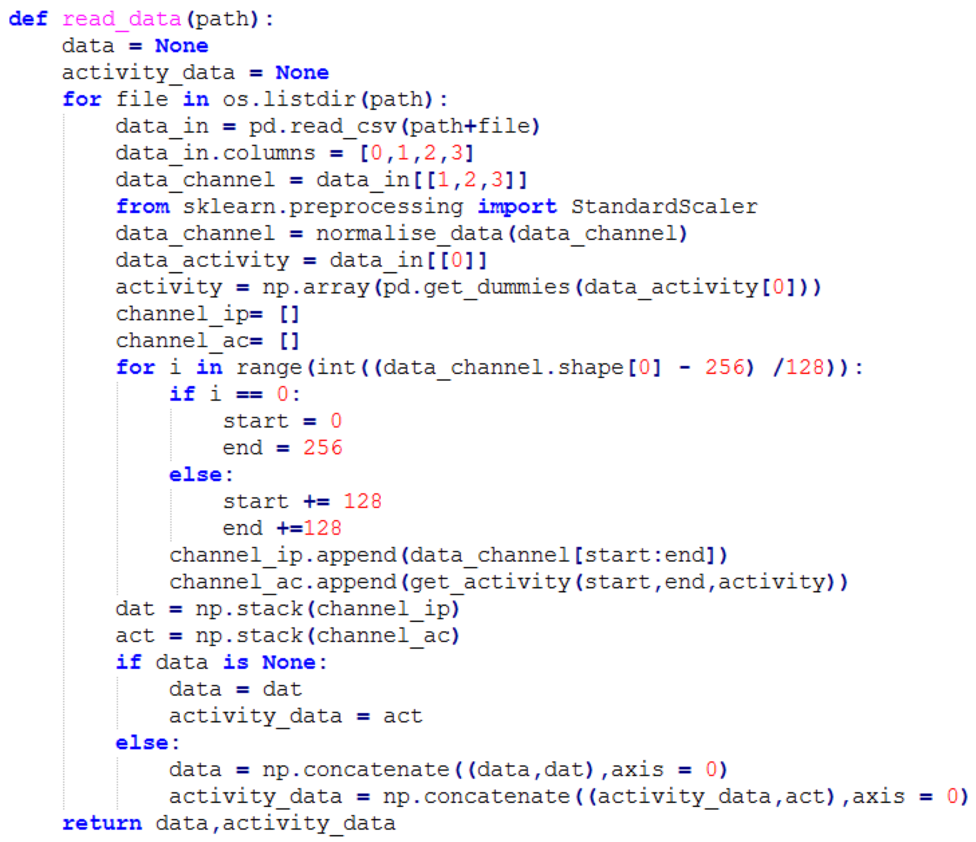




As a result of the analysis, only the instances with the 3D Acceleration data of IMU Ankle are considered and are saved in a separate csv file for quick retrieval.

Function name : read_data
Parameter : path to read the data

This function reads the csv file from the directory and converts the multivariate time-series to univariate ones. This is achieved by using a sliding window of size 256 and a sliding step of 128. Hence, once the 256 instances are taken as a time-series, the next time-series will be of stride 128. Also the maximum number of activities occurred in a time-series is taken as the activity of the time-series. The data is then normalised using StandardScaler and the target values are one-hot coded for classification.

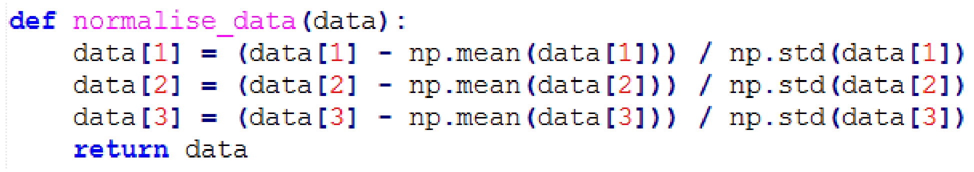


```python
def read_data(path):
    data = None
    activity_data = None
    for file in os.listdir(path):
        data_in = pd.read_csv(path+file)
        data_in.columns = [0,1,2,3]
        data_channel = data_in[[1,2,3]]
        from sklearn.preprocessing import StandardScaler
        data_channel = normalise_data(data_channel)
        data_activity = data_in[[0]]
        activity = np.array(pd.get_dummies(data_activity[0]))
        channel_ip= []
        channel_ac= []
        for i in range(int((data_channel.shape[0] - 256) /128)):
            if i == 0:
                start = 0
                end = 256
            else:
                start += 128
                end +=128
            channel_ip.append(data_channel[start:end])
            channel_ac.append(get_activity(start,end,activity))
        dat = np.stack(channel_ip)
        act = np.stack(channel_ac)
        if data is None:
            data = dat
            activity_data = act
        else:
            data = np.concatenate((data,dat),axis = 0)
            activity_data = np.concatenate((activity_data,act),axis = 0)
    return data,activity_data
```

Function name : normalise_data
Parameter : data to be normalised

This function is used to normalise the data as specified by the author by subtracting the mean and dividing by the standard deviation. It's also known as the StandardScaler.

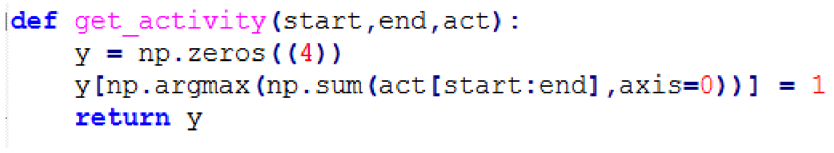
$$\frac{x-\mu}{\sigma}$$



```python
def normalise_data(data):
    data[1] = (data[1] - np.mean(data[1])) / np.std(data[1])
    data[2] = (data[2] - np.mean(data[2])) / np.std(data[2])
    data[3] = (data[3] - np.mean(data[3])) / np.std(data[3])
    return data
```

Function name : get_activity
Parameter : activity column

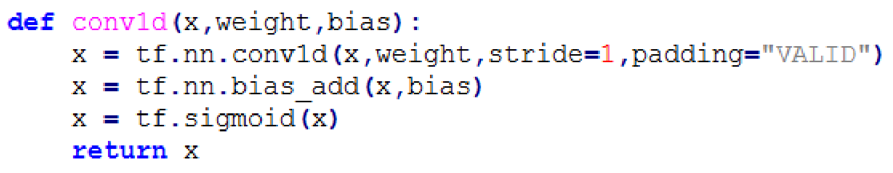
This function creates a one-hot code based on the maximum number of activities in a uni-variate time-series and returns the one-hotted values of the activity.



```
def get_activity(start,end,act):
    y = np.zeros((4))
    y[np.argmax(np.sum(act[start:end],axis=0))] = 1
    return y
```

Function name : conv1d
Parameter : input,weights,bias

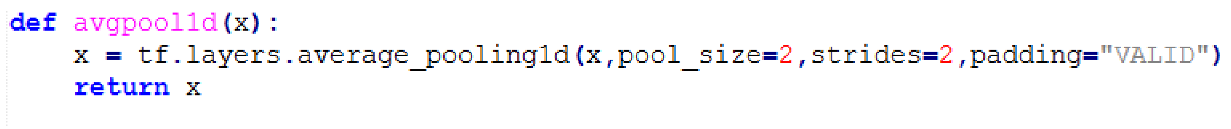
This function performs a 1-dimensional convolution on the input time-series with the given weights and biases. A stride of length 1 is set and the padding is made valid. Once the convolution is performed, bias is added to it and sigmoid activation function is used.



```
def conv1d(x,weight,bias):
    x = tf.nn.conv1d(x,weight,stride=1,padding="VALID")
    x = tf.nn.bias_add(x,bias)
    x = tf.sigmoid(x)
    return x
```

Function name : avgpool1d
Parameter : input

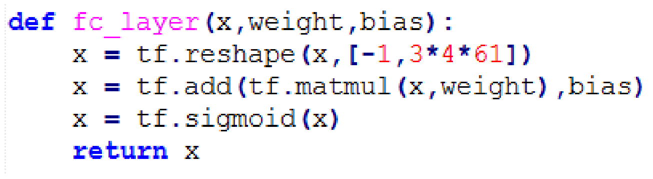
This function performs a down sampling on the convoluted layer. The pooling is based on average of the values with a kernel of size 2 and stride of 2. The padding used here is valid.



```
def avgpool1d(x):
    x = tf.layers.average_pooling1d(x,pool_size=2,strides=2,padding="VALID")
    return x
```

Function name : fc_layer
Parameter : inputs, weights and biases

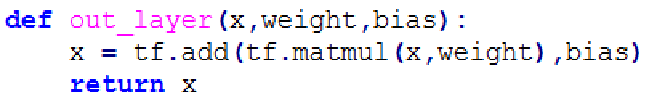
This function flattens the feature map to a one dimension vector and makes MLPs to connect itself to the output layer. Once the feature map is flattened, weights are multiplied and bias are added. Afterwhich, sigmoid activation map is used to get the result of the layer.



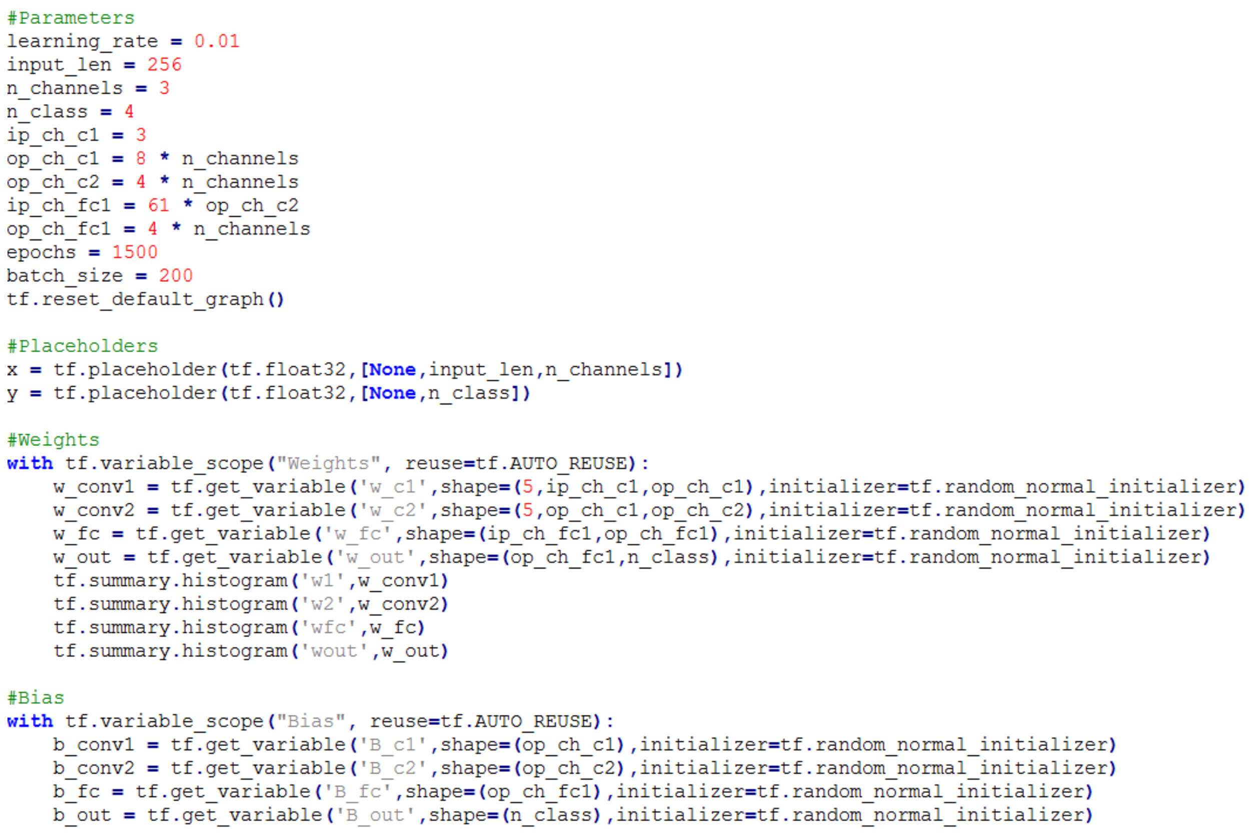
```
def fc_layer(x,weight,bias):
    x = tf.reshape(x,[-1,3*4*61])
    x = tf.add(tf.matmul(x,weight),bias)
    x = tf.sigmoid(x)
    return x
```

Function name : out_layer
Parameter : inputs, weights and biases

This function is the output layer and has the number of neurons equal to the number of classes. This provides the feature map by multiplying the weights with input feature map and adding the bias to it. The output of this layer is sent to softmax for predicting the activity.

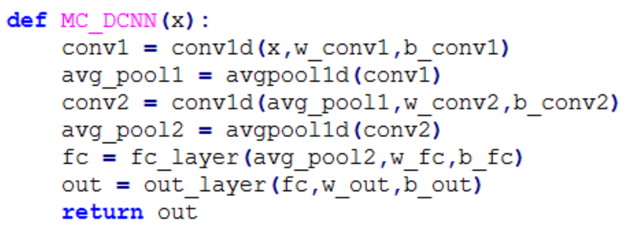

```
def out_layer(x,weight,bias):
    x = tf.add(tf.matmul(x,weight),bias)
    return x
```

The following are the parameters set, variables, weights and bias initialised for the networks to train the MC-DCNN model.


```
#Parameters
learning_rate = 0.01
input_len = 256
n_channels = 3
n_class = 4
ip_ch_c1 = 3
op_ch_c1 = 8 * n_channels
op_ch_c2 = 4 * n_channels
ip_ch_fc1 = 61 * op_ch_c2
op_ch_fc1 = 4 * n_channels
epochs = 1500
batch_size = 200
tf.reset_default_graph()

#Placeholders
x = tf.placeholder(tf.float32,[None,input_len,n_channels])
y = tf.placeholder(tf.float32,[None,n_class])

#Weights
with tf.variable_scope("Weights", reuse=tf.AUTO_REUSE):
    w_conv1 = tf.get_variable('w_c1',shape=(5,ip_ch_c1,op_ch_c1),initializer=tf.random_normal_initializer)
    w_conv2 = tf.get_variable('w_c2',shape=(5,op_ch_c1,op_ch_c2),initializer=tf.random_normal_initializer)
    w_fc = tf.get_variable('w_fc',shape=(ip_ch_fc1,op_ch_fc1),initializer=tf.random_normal_initializer)
    w_out = tf.get_variable('w_out',shape=(op_ch_fc1,n_class),initializer=tf.random_normal_initializer)
    tf.summary.histogram('w1',w_conv1)
    tf.summary.histogram('w2',w_conv2)
    tf.summary.histogram('wfc',w_fc)
    tf.summary.histogram('wout',w_out)

#Bias
with tf.variable_scope("Bias", reuse=tf.AUTO_REUSE):
    b_conv1 = tf.get_variable('B_c1',shape=(op_ch_c1),initializer=tf.random_normal_initializer)
    b_conv2 = tf.get_variable('B_c2',shape=(op_ch_c2),initializer=tf.random_normal_initializer)
    b_fc = tf.get_variable('B_fc',shape=(op_ch_fc1),initializer=tf.random_normal_initializer)
    b_out = tf.get_variable('B_out',shape=(n_class),initializer=tf.random_normal_initializer)
```

Function name : out_layer
Parameter : inputs, weights and biases

This is the network architecture and this function performs the training operation by calling the convolution, pooling, fully connected and output layers. The returns the feature map as output.


```
def MC_DCNN(x):
    conv1 = conv1d(x,w_conv1,b_conv1)
    avg_pool1 = avgpool1d(conv1)
    conv2 = conv1d(avg_pool1,w_conv2,b_conv2)
    avg_pool2 = avgpool1d(conv2)
    fc = fc_layer(avg_pool2,w_fc,b_fc)
    out = out_layer(fc,w_out,b_out)
    return out
```

The following snippet of code shows the declaration of the optimizers and other operations. The activation function used in the output layer is softmax. The loss function used here is softmax cross entropy loss and the optimizer used is RMSProp with learning rate 0.01 and parameters decay = 0.0005, momentum = 0.9, epsilon = 0.01 as specified in the research paper. Also, the scalar and histogram summaries are recorded to visualise with tensorboard.

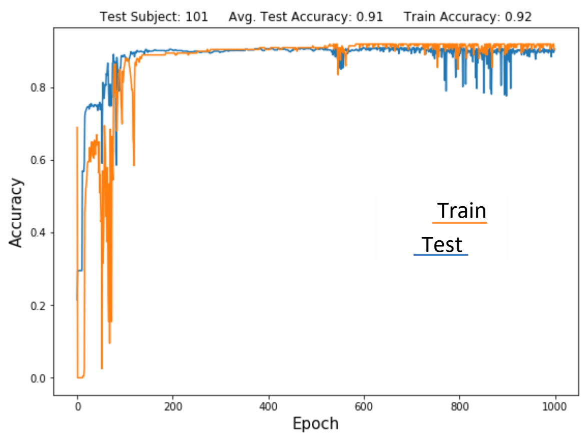
```
model = MC_DCNN(x)

predictions = tf.nn.softmax_cross_entropy_with_logits(logits=model,labels=y)
tf.summary.histogram("predictions", predictions)

loss = tf.reduce_mean(predictions)
tf.summary.scalar("Loss_Scalar",loss)
tf.summary.histogram("loss",loss)

optimizer_RMS = tf.train.RMSPropOptimizer(learning_rate=learning_rate,
                                          decay=0.0005,
                                          momentum=0.9,
                                          epsilon=0.01).minimize(loss)

optimizer_Adam = tf.train.AdamOptimizer(learning_rate=learning_rate,
                                          epsilon=0.01).minimize(loss)

correct_pred = tf.equal(tf.argmax(model, 1), tf.argmax(y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
tf.summary.scalar("Accuracy",accuracy)
tf.summary.histogram("Accuracy_histogram",accuracy)
```

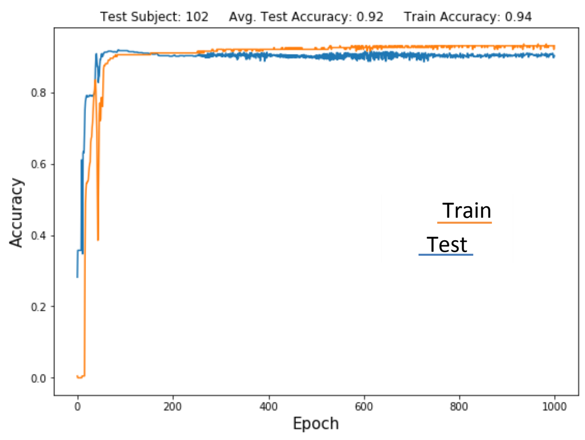
Once the variables are initialised and optimizers are defined, the variables are globally initialised and the session for the tensorflow is started. For each epoch, a batch of size 200 is made on the input train data. The testing is done on LOOCV technique. Where, in each pass, one of the subjects is used for testing to better predict the stability of the model. Also, in this part the training and testing summary and results are recorded for tensorboard visualisations.

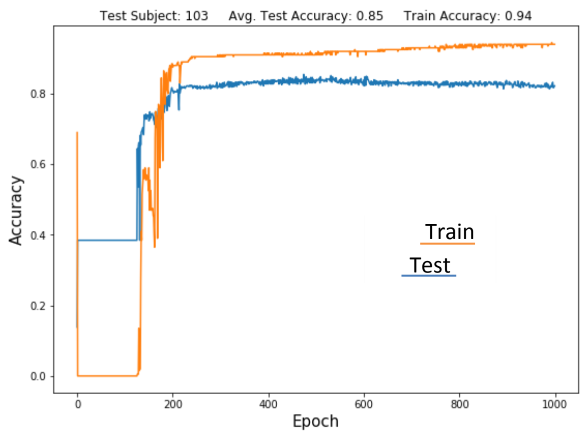
```
init = tf.global_variables_initializer()

with tf.Session() as sess:
    train_writer = tf.summary.FileWriter("G:/DA - Hildeshim/DDA Lab/Exercise 6/log/1/train",sess.graph)
    test_writer = tf.summary.FileWriter("G:/DA - Hildeshim/DDA Lab/Exercise 6/log/1/test",sess.graph)
    merge = tf.summary.merge_all()
    sess.run(init)
    counter = 0
    for ep in range(epochs):
        counter += 1
        for batch in range(len(train)//batch_size):
            batch_x = train[batch*batch_size:min((batch+1)*batch_size,len(train))]
            batch_y = activity_train[batch*batch_size:min((batch+1)*batch_size,len(activity_train))]
            assert not np.any(np.isnan(batch_x))
            assert not np.any(np.isnan(batch_y))
            feed = {x : batch_x, y : batch_y}
            summary, cost, _ , acc = sess.run([merge, loss, optimizer_RMS, accuracy], feed_dict = feed)
            train_writer.add_summary(summary,counter)
            if ep%100 == 1:
                print("\nEpoch: "+str(ep)+"\tTraining loss: "+str(cost)+"\tTraining Accuracy: "+str(acc))
                feed = {x : test, y : activity_test}
                su, cost, acc = sess.run([merge, loss, accuracy], feed_dict = feed)
                test_writer.add_summary(su,counter)
                print("\t\tTesting loss: "+str(cost)+"\tTesting Accuracy: "+str(acc))
```

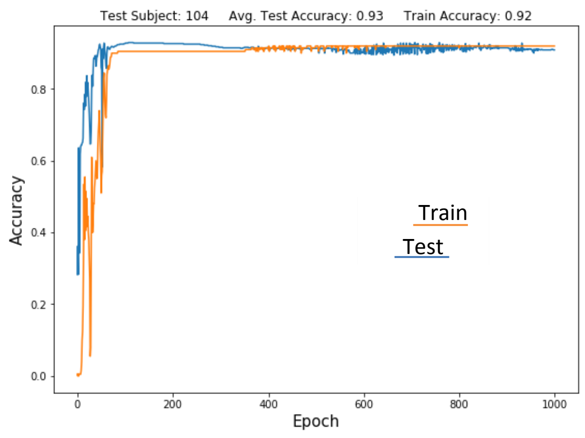
The accuracy and loss of testing and training are recorded and plotted as shown below.

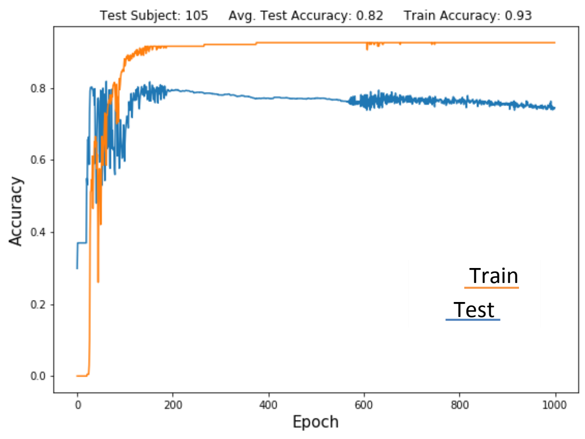
Average Testing and Training Accuracy using LOOCV technique

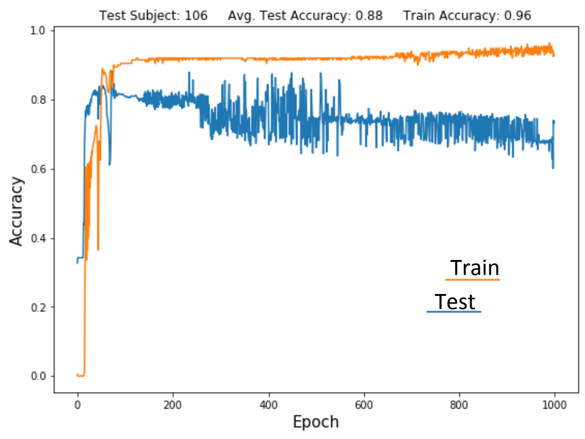


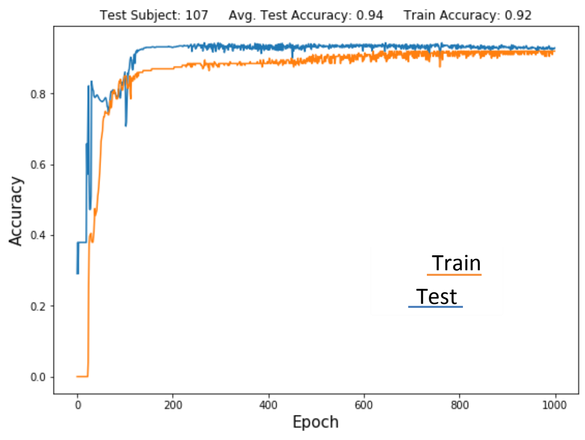


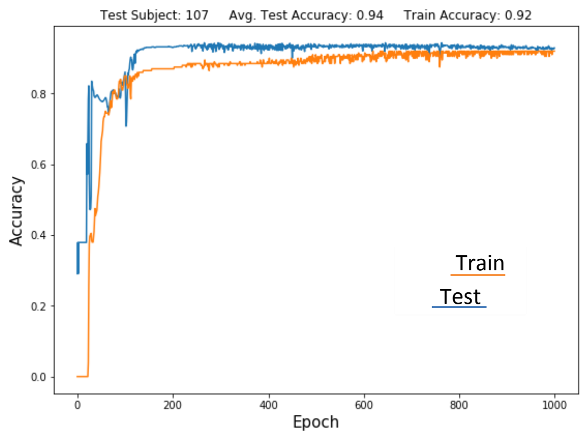




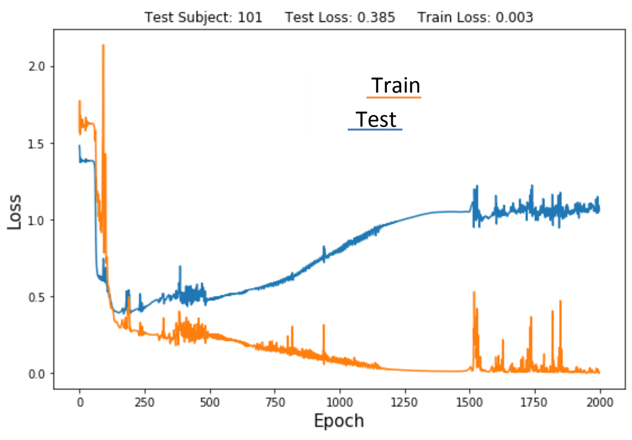


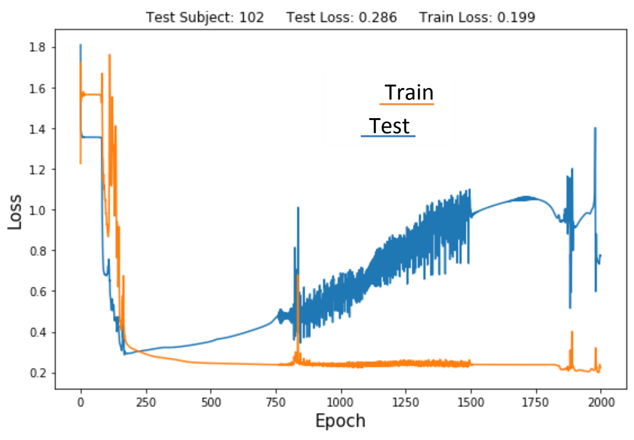


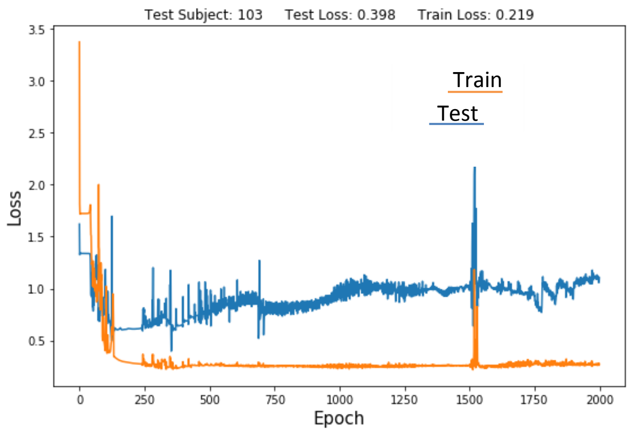


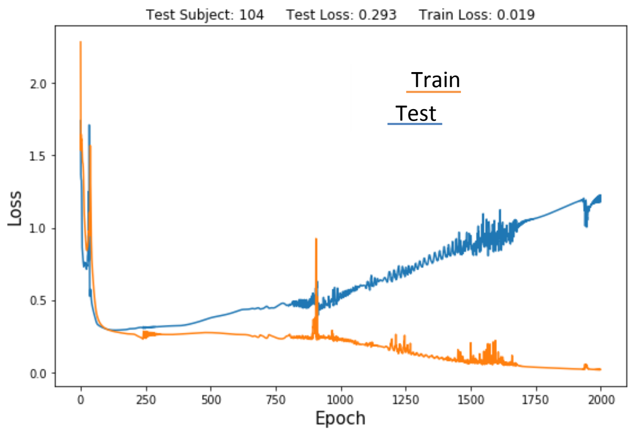


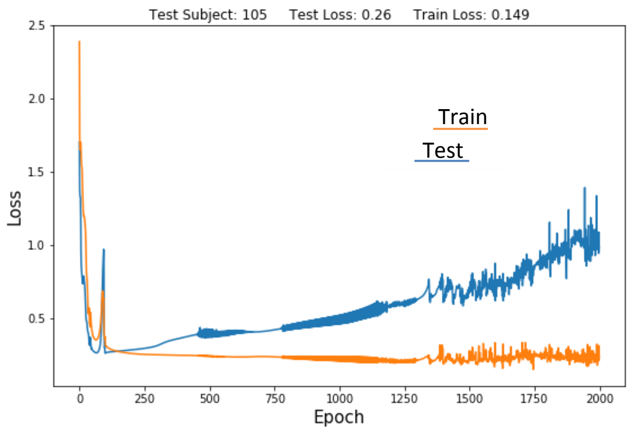
Average Testing and Training Loss using LOOCV technique

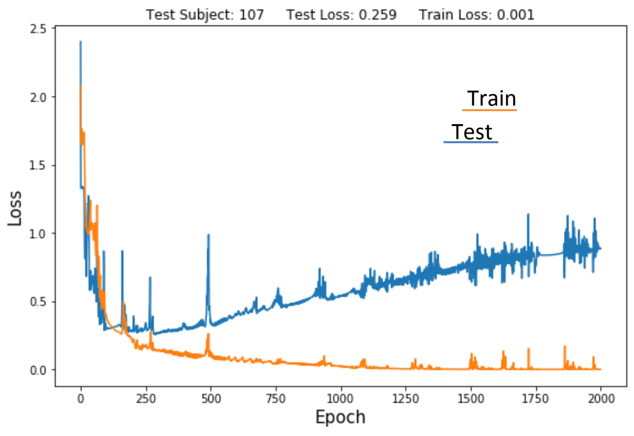










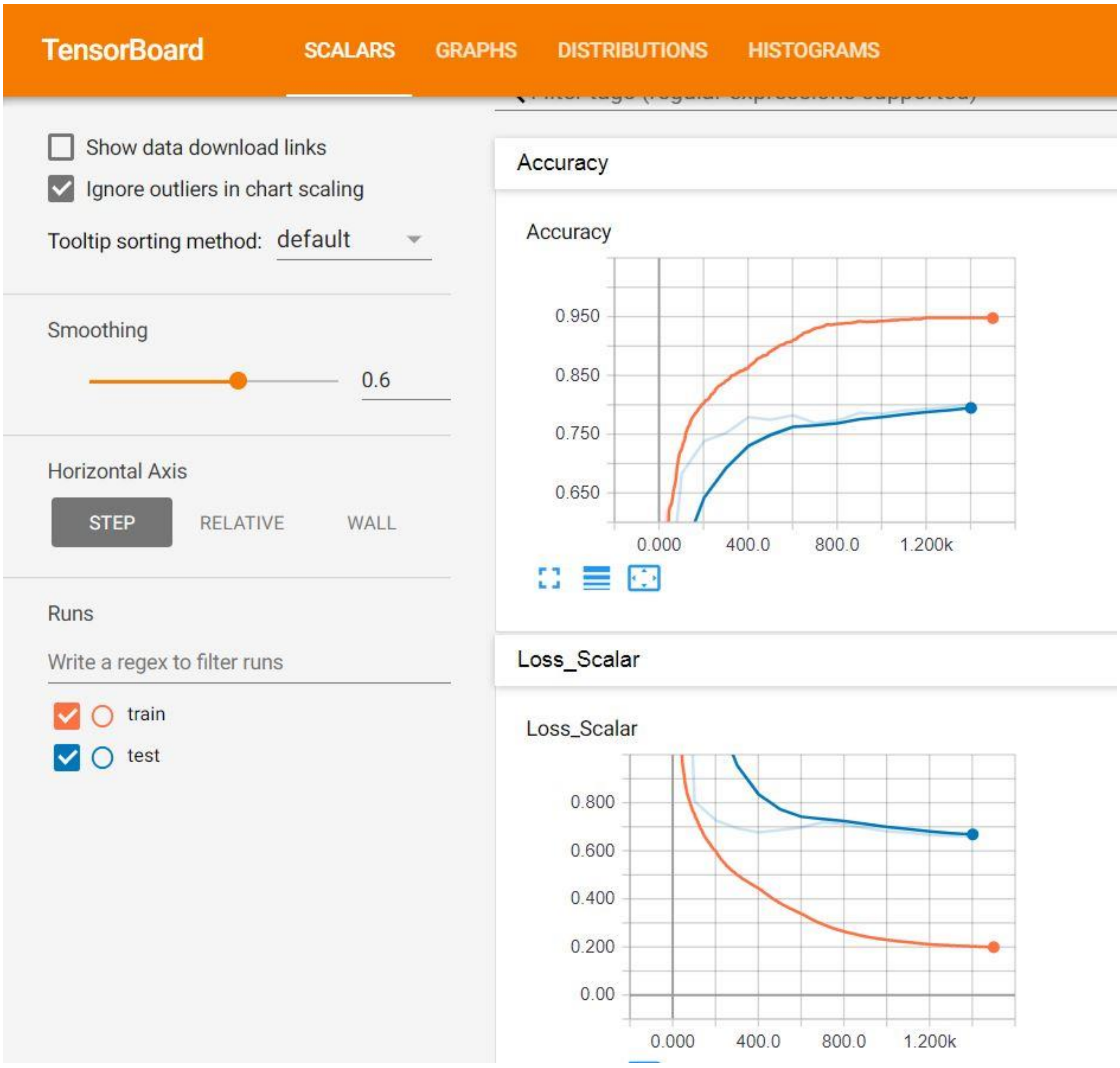


From the results it can be seen that, the model tries to overfit after certain epochs, that the training loss decreases but the testing loss increases. Due to time and memory constraints, the model was tested with sliding step 128 only. Also, for stride 128, the average accuracy achieved by the research paper is 90.34 and in my case, it is 93.28 which is better than the paper's implementation. This may be because of the different hyperparameters and the 3D data used as these are not clearly mentioned in the research paper.

Tensorboard Results:

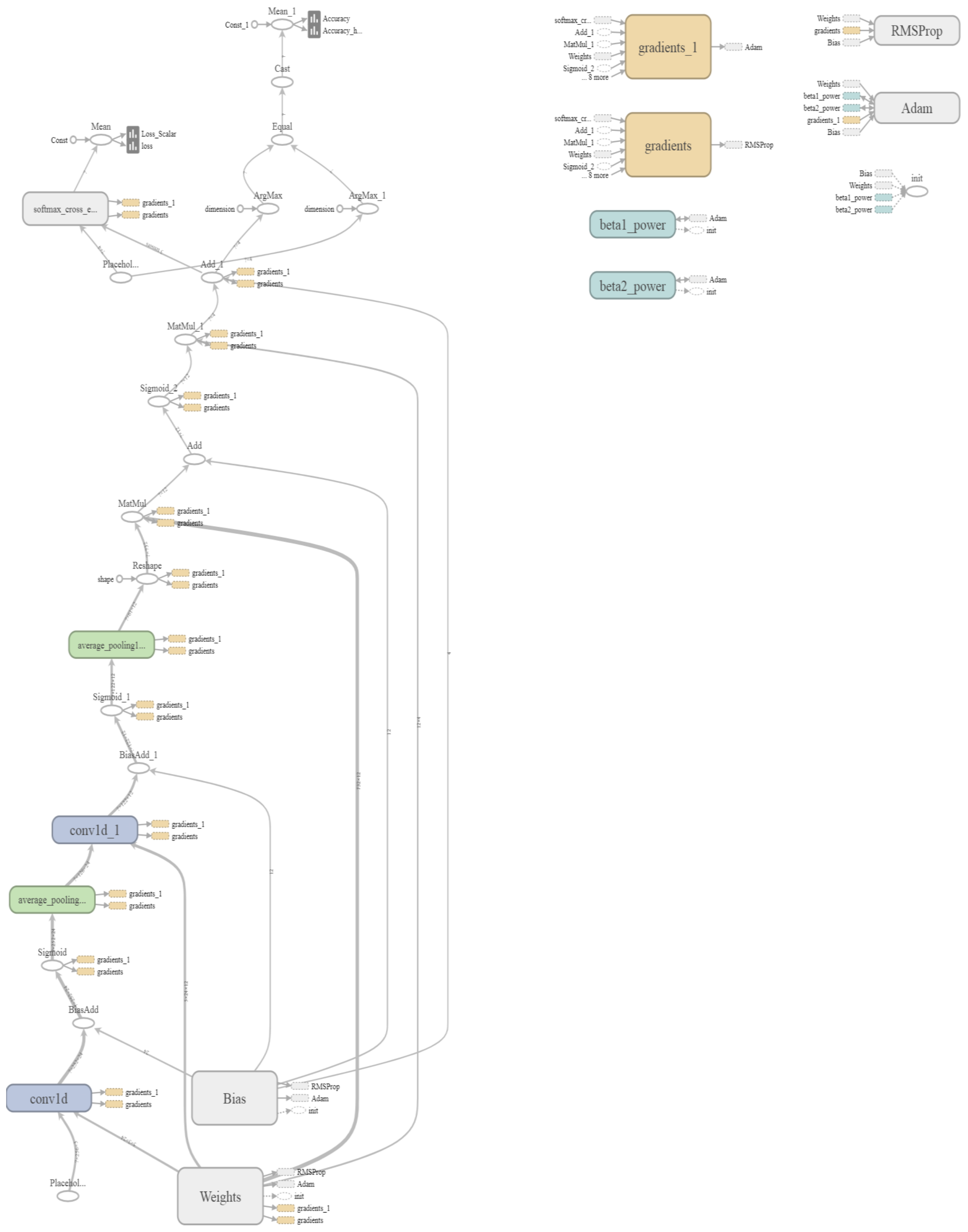
The model results show above are run in co-lab and due to memory constraints, the tensorboard results are shown for a limited dataset i.e., 3 subjects in train and 1 subject in test for the same parameters, optimizer and operations. The results are as shown below.

Scalar: Loss and Accuracy

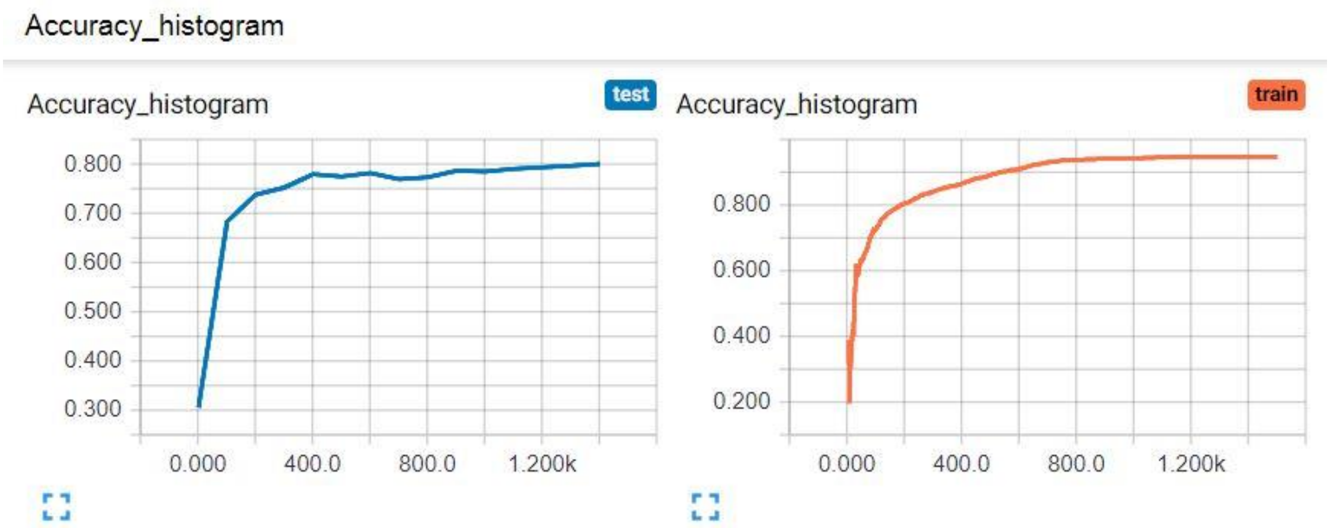


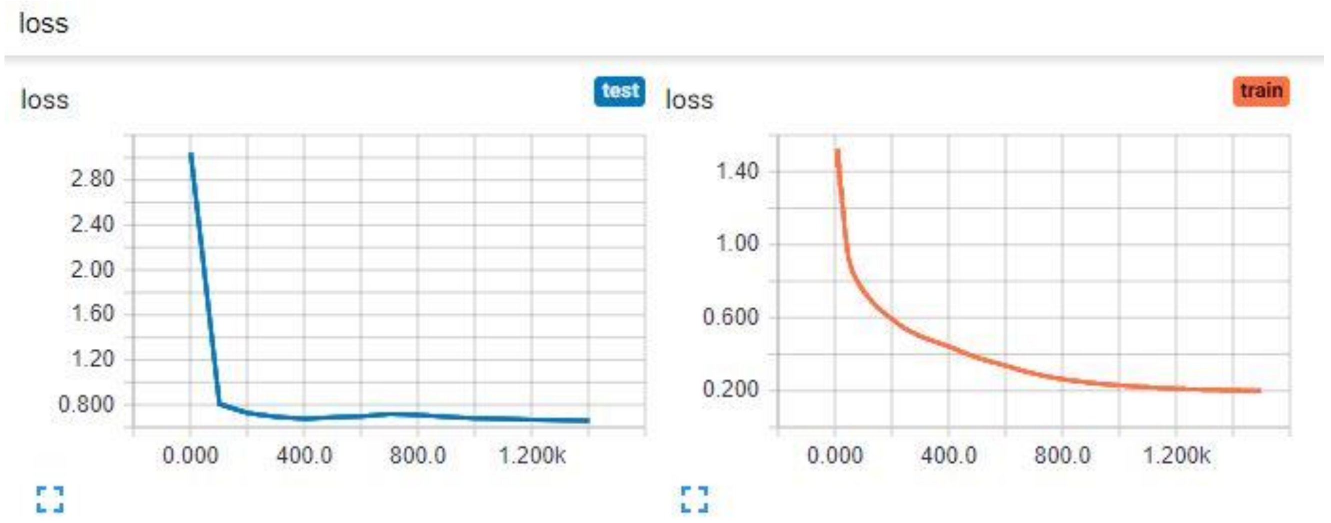
☐ Show data download links
☑ Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing 0.6
Horizontal Axis [✓ STEP] RELATIVE WALL
☑ train
☑ test

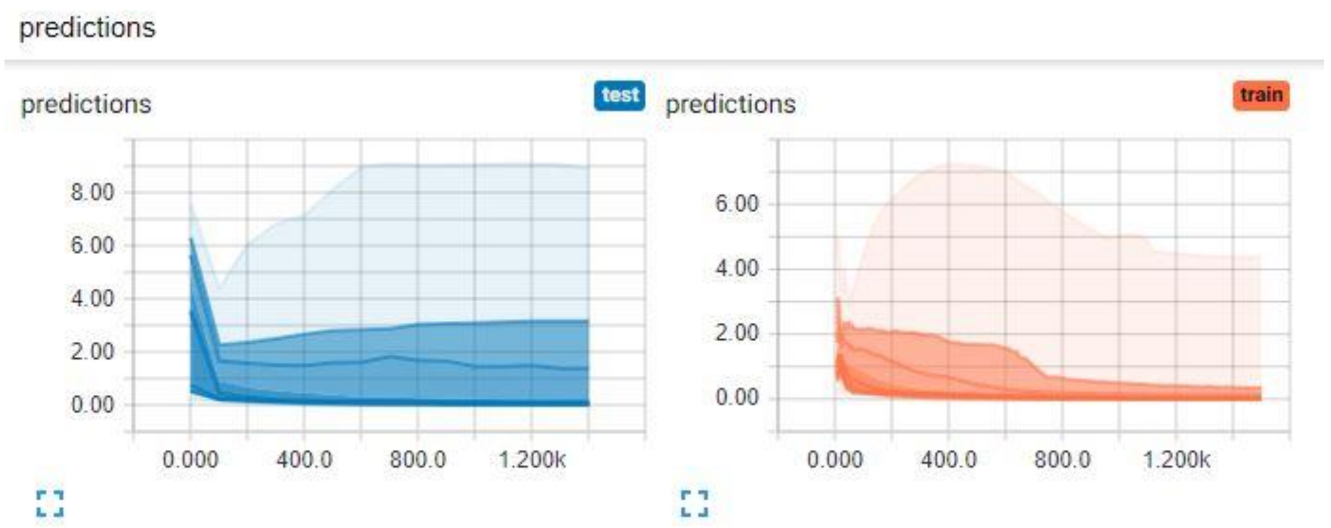
Graph: Model

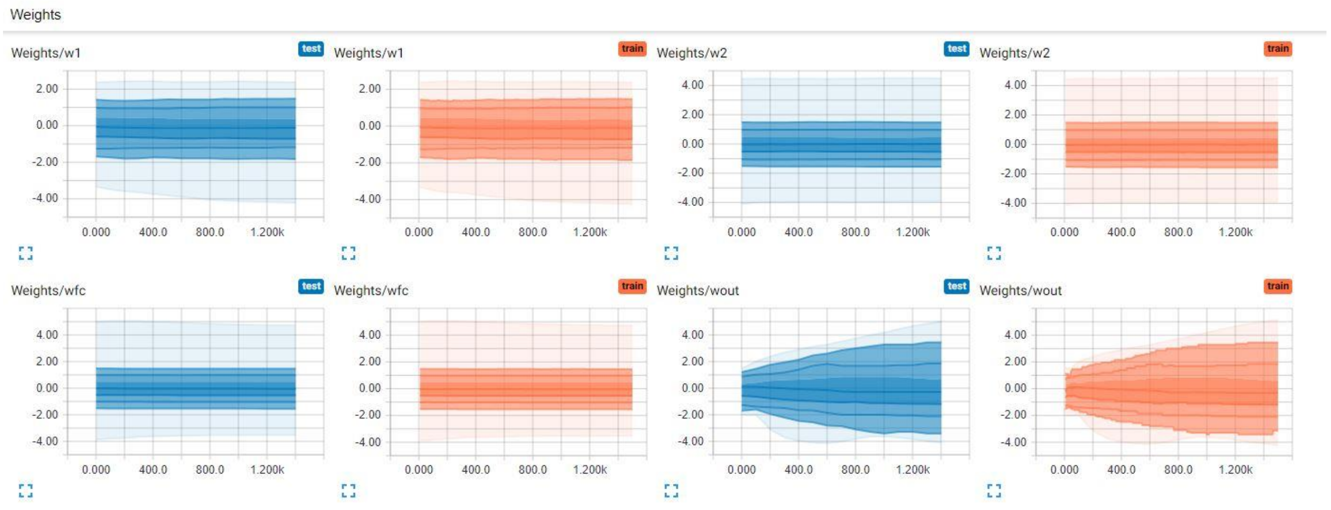

Mean_1
Const_1
Accuracy
Accuracy_h...
Cast
Equal
Mean
Const
Loss_Scalar
loss
softmax_cross_e...
gradients_1
gradients
ArgMax
ArgMax_1
dimension
Placehol...
Add_1
MatMul_1
Sigmoid_2
Add
MatMul
Reshape
shape
average_pooling1...
Sigmoid_1
BiasAdd_1
conv1d_1
average_pooling...
Sigmoid
BiasAdd
conv1d
Bias
Weights
RMSProp
Adam
init
softmax_cr...
Add_1
MatMul_1
Weights
Sigmoid_2
... 8 more
gradients_1
gradients
beta1_power
beta2_power
Bias
Weights
beta1_power
beta2_power

Distributions:

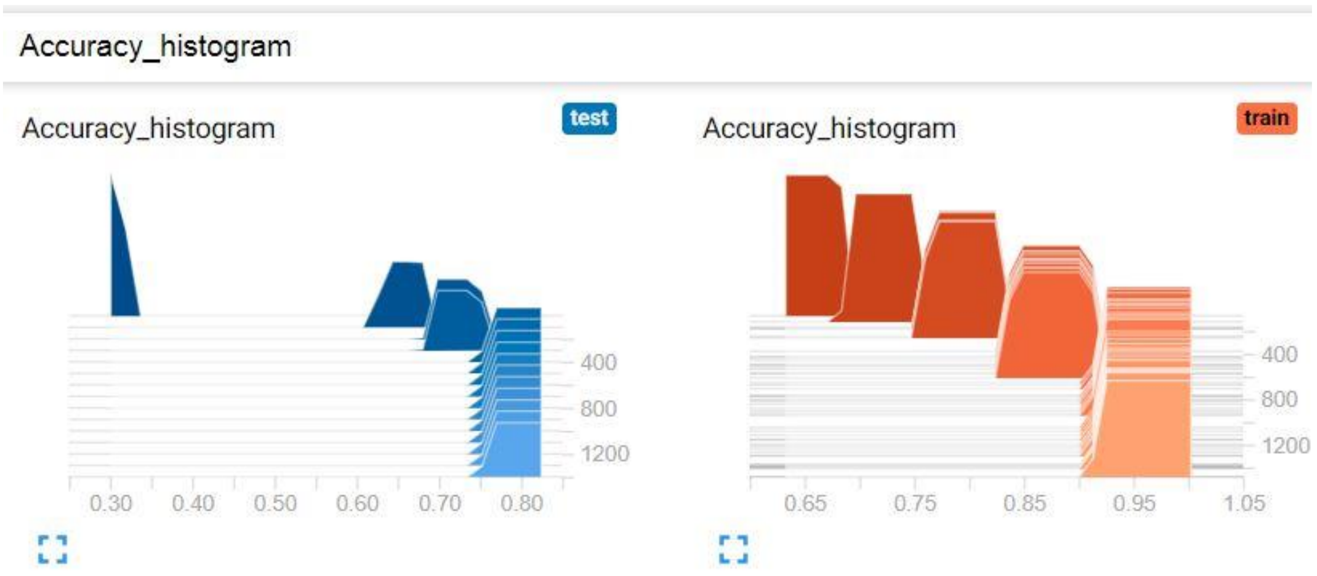


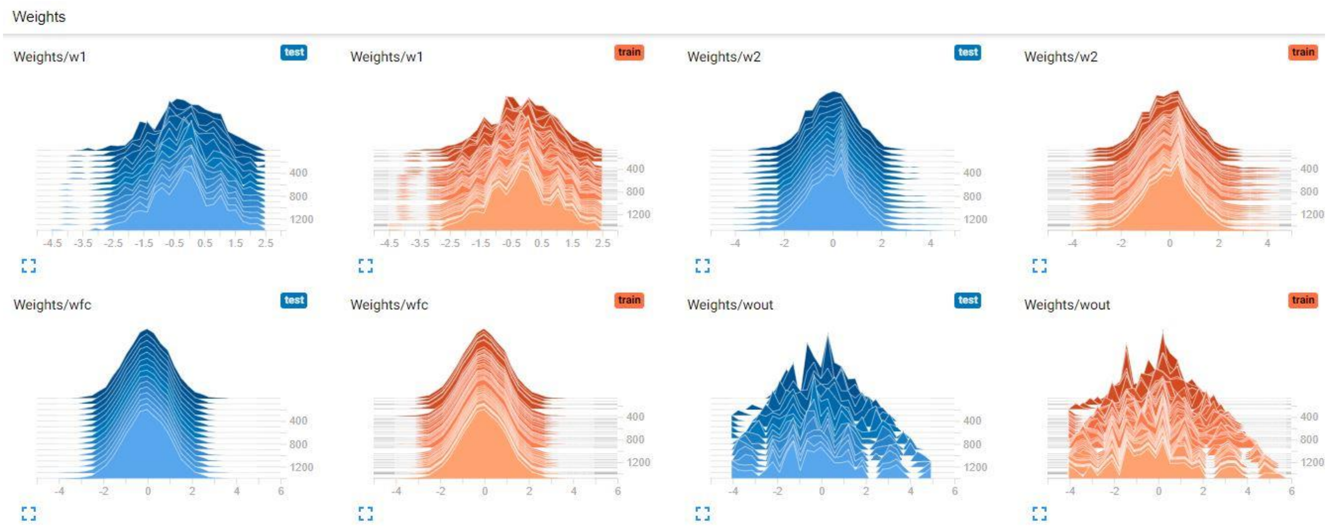


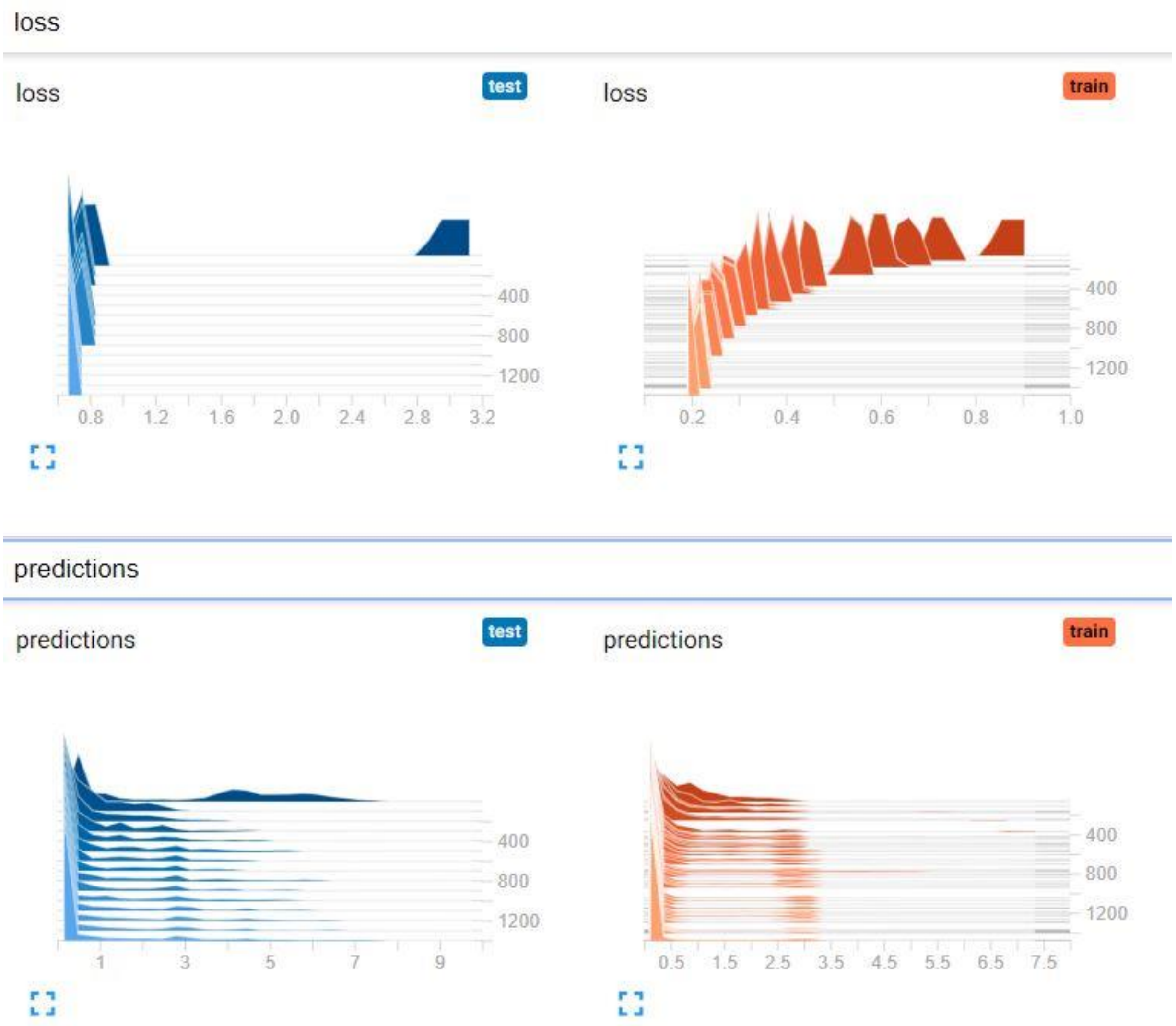



Weights
Weights/w1
test
Weights/w1
train
Weights/w2
test
Weights/w2
train
Weights/wfc
test
Weights/wfc
train
Weights/wout
test
Weights/wout
train

Histograms:


Accuracy_histogram
Accuracy_histogram
test
Accuracy_histogram
train


Weights
Weights/w1
test
Weights/w1
train
Weights/w2
test
Weights/w2
train
Weights/wfc
test
Weights/wfc
train
Weights/wout
test
Weights/wout
train


loss
loss
test
0.8
1.2
1.6
2.0
2.4
2.8
3.2
400
800
1200
loss
train
0.2
0.4
0.6
0.8
1.0
400
800
1200
predictions
predictions
test
1
3
5
7
9
400
800
1200
predictions
train
0.5
1.5
2.5
3.5
4.5
5.5
6.5
7.5
400
800
1200