Distributed Data Analytics

Exercise Sheet 6

Sebastian Pineda Arango Mtr. Nr. 246098

In this notebook, we aim to implement and explain the algorithm for multi-dimensional series classification proposed by Yi Zheng et. al "Time Series Classification Using Multi-Channels Deep Convolutional Neural Networks" [1]. The notebook comprises the following sections:

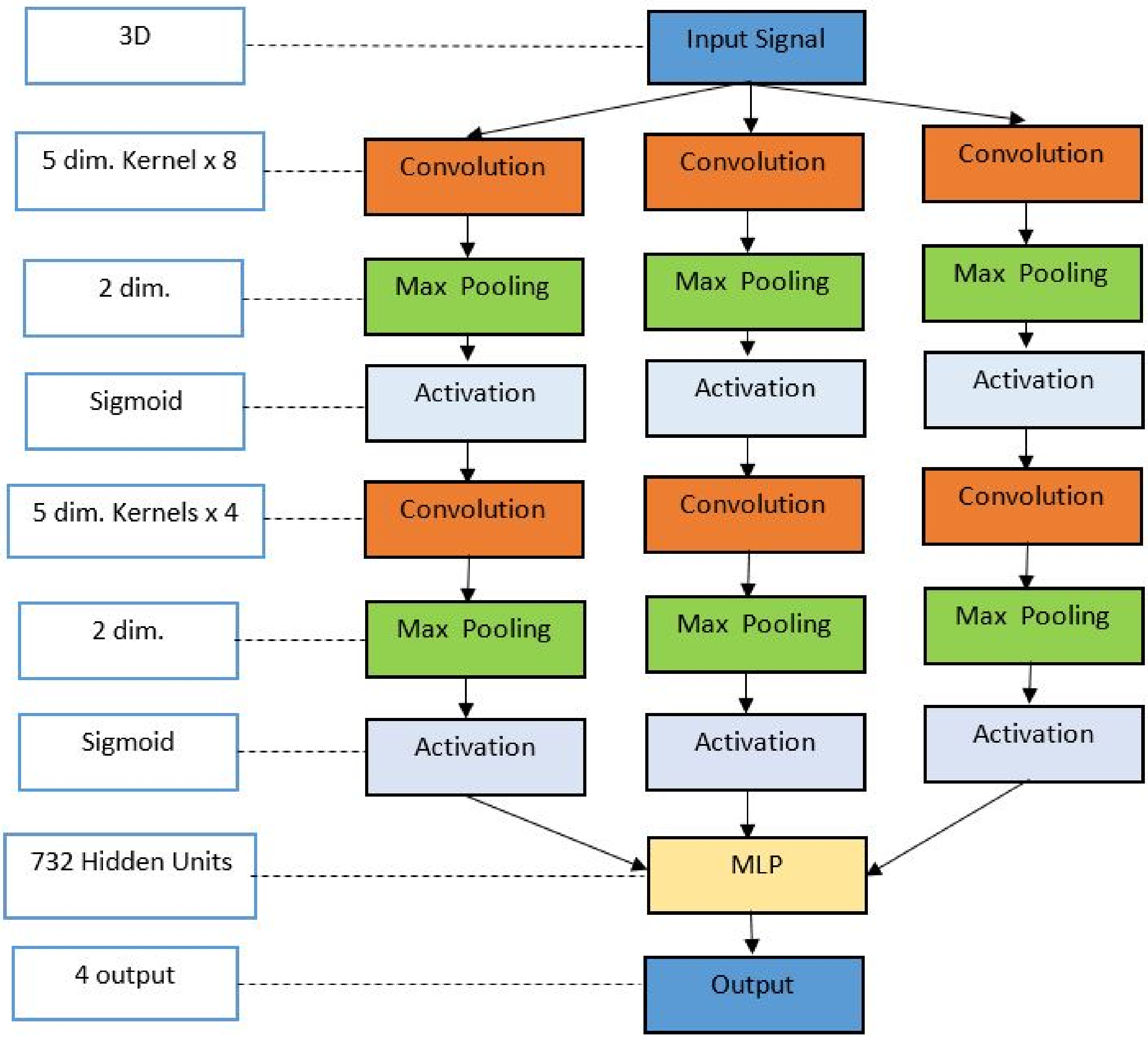
- Explanation of the architecture
- Explanation of the back-propagation
- Implementation of the paper

Explanation of the architecture

The architecture design can be summarized in the following main points:

- The input signal may have multiple channel (multiple time series). They depict an image for a signal with three time series.
- There are three main layer stages: 2 convolutional feature extractyor and a fully connected neural network.
- The first convolutional layer comprises: 8 filter extrators of size 5, a subsampling layer (max-pooling) of size 2 and a sigmoid activation function.
- The second convolutional layer compriceses: 4 filter extractor of size 5, a subsampling layer and a sigoid activation layer.
- The final fully onnected layer comprises a hidden connecter layer with 732 neurons and output layer with 4 output neurons.

The previous stages are represented graphically in the following image.



Explanation of the back-propagation

The back-propagation look for minimize the cost function, which for the paper is the cross entropy function (associated to four classes). To minimize this function, we find the derivative of the cost function respect to the parameters of the network (gradient of the paraemters) and then update the parameters according to gradient descent algorithm (plus using momentum and weight decay). Therefore we can summarize the gradient-based learning (as explained in section 3.2.) in the following way:

- Feedforward pass: computation of the outputs of all the layers given the training data
- Backpropagation pass: computing the gradient using the chain rule
- Gradients applied: update of the parameters based on the gradients and using momentum and weight decay

The parameters of the netowrk are the kernel values and the weights of the fully connected neural network. In total, there are: 40 parameters for the first stage, 20 parameters for the second stage and 2928 parameters for the hidden layer. The updating of the weights of the neural netowrk are a standard procedure, already exhaustively explained in the literature. However, the auhtos show explicitely the derivation of the gradient for the kernel parameters. Applying the chain rule over the whole network, it is possible also then to find the gradient of the kernel parameters.

Implementation of the paper

We want to implement the paper in Tensorflow, however, first we mut do a preprocessing of the raw data so that we can create the learning algorithm.

The steps to carry out the implementation are:

1. Load the data and select the activity and time series of interest (suggested in the paper):

    - Standing
    - Walking
    - Ascending stairs
    - Descending staris

    Focus only on the 3D signal of the IMU hand acceleromenter in this notebook. In the paper, however, it is not clear which signal they use in the paper. Furthermore, we also delete those row with at least one missing value.
2. Create the subsequences through a sliding window which stores subsets of the data (256 timestamps) as a single sample for training. The step-size for the sliding window is fixed to 32, while in the paper they try several. The smaller the step size, the more data we have to train. The sliding window size (=256) is the same as in the paper.
3. Standarize every dimension of the sliding windows from the training and test set.
4. Create the graph implementing the architecture shown in the first section.
5. Suscribe the plots to tensorboard to visualize results. To make the tensorboard plots, we base on code from [4].
6. Train the network with mini-batch gradiend descent. To create the minibatches, we use the useful code from [2].

Loading, selecting and cleaning the data

In [1]:

```
#importing libraries
import os
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import math
from sklearn.preprocessing import StandardScaler

%matplotlib inline

#reading data
path = "PAMAP2_Dataset\Protocol"
data_files = os.listdir(path)
```

```
C:\Users\User\Anaconda3\lib\site-packages\h5py\__init__.py:34: FutureWarning:
Conversion of the second argument of issubdtype from `float` to `np.floating`
is deprecated. In future, it will be treated as `np.float64 == np.dtype(floa
t).type`.
  from ._conv import register_converters as _register_converters
```

In [2]:
```
def preprocess_file(file_name):

    """This function preprocesses the subject file which is given as input"""

    #reasing the data
    data_path = os.path.join(path, file_name)
    file = open(data_path, 'r')
    text = file.read()
    lines = text.split("\n")
    lines_data = [l.split(" ") for l in lines]
    data = np.array(lines_data[:-1]) #discarding last element because it is fr
ee
    data[data=="NaN"]= np.nan #renaming NaN
    data_tf = data.astype(float)

    #Activities ID of interest:
    # - Standing (3)
    # - Walking (4)
    # - Ascending stairs (12)
    # - Descending staits (13)

    #Filtering activities
    data_fil = np.vstack((data_tf[data_tf[:,1] == 3,],
                          data_tf[data_tf[:,1] == 4,],
                          data_tf[data_tf[:,1] == 12,],
                          data_tf[data_tf[:,1] == 13,]))

    #filtering only sensor measurements of interest: subject108 and subject109
    #are left apart since they don't have data usful for the current applicati
on
    X = data_fil[:, 4:7] #3D-acceleartion data from IMU hand
    Y = data_fil[:,1 ].astype(int).reshape(-1,1) #label: activity ID

    return X, Y

#selecting files of interest - splitting in train and test
data_files_train = data_files[:6]
data_files_test = data_files[7]


#preprocessing all the files and merging in a single file
X_train, Y_train = preprocess_file(data_files_train[0])

for file in data_files_train[1:]:

    temp_X, temp_Y = preprocess_file(file)
    print("File name:", file, " got ", temp_X.shape)
    X_train = np.vstack((X_train, temp_X))
    Y_train = np.vstack((Y_train, temp_Y))

#preprocessing the data for test set
X_test, Y_test = preprocess_file(data_files_test)
```

```
File name: subject102.dat  got  (90664, 3)
File name: subject103.dat  got  (75233, 3)
File name: subject104.dat  got  (87617, 3)
File name: subject105.dat  got  (81173, 3)
File name: subject106.dat  got  (74640, 3)
```

In [3]:
```
#selecting the rows f the data with at least one NA
select_train = np.isnan(X_train).any(axis=1)
select_test = np.isnan(X_test).any(axis=1)

print("Shape before...")
print("Train:", X_train.shape)
print("Test:", X_test.shape)

#deleting rows with at least one NA in train and test set
X_train = X_train[ ~select_train, :]
Y_train = Y_train[ ~select_train, :]
X_test = X_test[ ~select_test, :]
Y_test = Y_test[ ~select_test, :]

print("Shape before...")
print("Train:", X_train.shape)
print("Test:", X_test.shape)
```

```
Shape before...
Train: (484086, 3)
Test: (78031, 3)
Shape before...
Train: (479208, 3)
Test: (77521, 3)
```

Creating subsequences

In [4]:
```
#contructing subsequences (trianing samples)
step = 32
size_subseq = 256

def construct_tensor(X, Y, step, size_subseq):

    """Construct the subsequences of data. At the end, every subsequence is
    a tensor of shape (number of steps, size_subseq, 3).  The number of steps,
    therefore, is the number training samples and the 3 means that there are
    three channels(because it is a 3D signal)."""

    size_t = X.shape[0]
    X_pre = np.zeros(((size_t//step)-1, size_subseq,3))
    Y_pre = np.zeros((size_t//step-1))

    print(X_pre.shape)
    for i, j in enumerate(range(0,size_t, step)):
        y_i = Y[j]
        y_f = Y[min(j+size_subseq-1, size_t-1)]
        temp = X[j:(j+size_subseq),]
        if(y_i==y_f and temp.shape[0]==size_subseq):
            X_pre[i,:,:]= temp
            Y_pre[i] = Y[j]
    Y_pre = np.array(Y_pre)
    X_pre = np.delete(X_pre, np.where(Y_pre==0), 0)
    Y_pre = np.delete(Y_pre, np.where(Y_pre==0))

    return X_pre, np.array(pd.get_dummies(Y_pre))

#constructing sequences for train and test set
X_train_pre, Y_train_pre = construct_tensor(X_train, Y_train, step, size_subse
q)
X_test_pre, Y_test_pre = construct_tensor(X_test, Y_test, step, size_subseq)

train_size = X_train_pre.shape[0]
test_size = X_test_pre.shape[0]
n_classes = Y_train_pre.shape[1]
print("Train size:", train_size)
print("Test size:", test_size)
print("Num. classes:", n_classes)
```

```
(14974, 256, 3)
(2421, 256, 3)
Train size: 14784
Test size: 2391
Num. classes: 4
```

Standarizing the data

In [6]:
```
depth = X_train_pre.shape[2]

#creating the objects to scale
scalers = [StandardScaler() for i in range(depth)]

x_train = np.zeros(X_train_pre.shape)
x_test = np.zeros(X_test_pre.shape)

for i in range(depth):
    x_train[:,:,i] = scalers[i].fit_transform(X_train_pre[:,:,i])
    x_test[:,:,i] = scalers[i].transform(X_test_pre[:,:,i])

y_train = Y_train_pre
y_test = Y_test_pre
```

Creating the network and training the model

In [7]:
```
#defining functions
def random_mini_batches(X, Y, mini_batch_size = 64, seed = 0):
    """
    Creates a list of random minibatches from (X, Y)

    Arguments:
    X -- input data, of shape (input size, number of examples)
    Y -- true "label" vector (containing 0 if cat, 1 if non-cat), of shape (1, 
number of examples)
    mini_batch_size - size of the mini-batches, integer
    seed -- this is only for the purpose of grading, so that you're "random mi
nibatches are the same as ours.

    Returns:
    mini_batches -- list of synchronous (mini_batch_X, mini_batch_Y)
    """

    m = X.shape[0]                  # number of training examples
    mini_batches = []
    np.random.seed(seed)

    # Step 1: Shuffle (X, Y)
    permutation = list(np.random.permutation(m))
    shuffled_X = X[permutation,:]
    shuffled_Y = Y[ permutation,:]

    # Step 2: Partition (shuffled_X, shuffled_Y). Minus the end case.
    num_complete_minibatches = math.floor(m/mini_batch_size)
    # number of mini batches of size mini_batch_size in your partitionning
    for k in range(0, num_complete_minibatches):
        mini_batch_X = shuffled_X[k * mini_batch_size : k * mini_batch_size + 
mini_batch_size,:]
        mini_batch_Y = shuffled_Y[ k * mini_batch_size : k * mini_batch_size + 
mini_batch_size,:]
        mini_batch = (mini_batch_X, mini_batch_Y)
        mini_batches.append(mini_batch)

    # Handling the end case (last mini-batch < mini_batch_size)
    if m % mini_batch_size != 0:
        mini_batch_X = shuffled_X[num_complete_minibatches * mini_batch_size : 
m,:]
        mini_batch_Y = shuffled_Y[num_complete_minibatches * mini_batch_size : 
m, :]
        mini_batch = (mini_batch_X, mini_batch_Y)
        mini_batches.append(mini_batch)

    return mini_batches
```

In [8]:
```
def variable_summaries(var):
    """Attach a lot of summaries to a Tensor (for TensorBoard visualizatio
n)."""
    with tf.name_scope('summaries'):
        mean = tf.reduce_mean(var)
        tf.summary.scalar('mean', mean)
        with tf.name_scope('stddev'):
            stddev = tf.sqrt(tf.reduce_mean(tf.square(var - mean)))
        tf.summary.scalar('stddev', stddev)
        tf.summary.scalar('max', tf.reduce_max(var))
        tf.summary.scalar('min', tf.reduce_min(var))
        tf.summary.histogram('histogram', var)
```

In [10]:
```
#initializing hyperparameters
n_iterations = 100
lr = 0.01
batch_size = 64
momentum= 0.9
reg_weight = 0.00025

#initializing graph
tf.reset_default_graph()

#creating placeholders
x = tf.placeholder(tf.float32, shape=(None, size_subseq, depth))
y = tf.placeholder(tf.float32, shape=(None, n_classes))

#creating variables for the convolutional layers of three channels
conv1_set = [tf.Variable(tf.truncated_normal(shape=[5, 1, 8], mean=0, stddev=
0.1)) for i in range(depth)]
conv2_set = [tf.Variable(tf.truncated_normal(shape=[5, 8, 4], mean=0, stddev=
0.1)) for i in range(depth)]

#creating variables for the fully connected
W = tf.Variable(tf.truncated_normal(shape=[depth*4*61, 732], mean=0, stddev=0.
1))
bias = tf.Variable(tf.truncated_normal(shape=[732], mean=0, stddev=0.1))
W2 = tf.Variable(tf.truncated_normal(shape=[732, n_classes], mean=0, stddev=0.
1))
bias2 = tf.Variable(tf.truncated_normal(shape=[n_classes], mean=0, stddev=0.1
))


#creating graph for the first convolutional layer
conv11 = [tf.nn.conv1d(tf.reshape(x[:,:,i], (-1, size_subseq, 1)),
                       conv1_set[i], stride=1, padding='VALID') for i in range
(depth)]
pool11 = [tf.layers.max_pooling1d(conv11[i], pool_size=2, strides=2, padding=
'valid') for i in range(depth)]
activation11 = [tf.nn.sigmoid(pool11[i]) for i in range(depth)]

#creating graph for the second convolutional layer
conv22 = [tf.nn.conv1d(activation11[i], conv2_set[i], stride=1, padding='VALI
D') for i in range(depth)]
pool22 = [tf.layers.max_pooling1d(conv22[i], pool_size=2, strides=2, padding=
'valid') for i in range(depth)]
activation22 = [tf.nn.sigmoid(pool22[i]) for i in range(depth)]

#creating graph for the fully connected layer
flat1 = [tf.contrib.layers.flatten(activation22[i]) for i in range(depth)]
stack = tf.concat(flat1, axis=1)
h = tf.matmul(stack, W)+bias #first fully connected layer
conv_net_output = tf.matmul(h, W2)+bias2 #second fully connected layer

#creating graph for regularizer
regularizer = tf.nn.l2_loss(W) + tf.nn.l2_loss(W2)

for i in range(depth):
    regularizer += tf.nn.l2_loss(conv1_set[i])
```

```
    regularizer += tf.nn.l2_loss(conv2_set[i])

#loss and train step
cost =  tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits( labels=y, logi
ts=conv_net_output)) + regularizer*reg_weight
train_step = tf.train.AdamOptimizer(lr).minimize(cost)

#computing accuracy
correct_pred = tf.equal(tf.argmax(conv_net_output, 1), tf.argmax(y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32), name='accuracy')

#obtaining gradients for debug in tensorboard
grads = tf.train.AdamOptimizer(lr).compute_gradients(loss=cost)

#suscribing tensors to plot in tensorboard
for i in range(depth):
    with tf.name_scope('conv1'):
        variable_summaries(conv1_set[i])

    with tf.name_scope('activation1'):
        variable_summaries(activation11[i])

    with tf.name_scope('conv2'):
        variable_summaries(conv2_set[i])

    with tf.name_scope('activation2'):
        variable_summaries(activation22[i])

with tf.name_scope('performance'):
    tf.summary.scalar('accuracy', accuracy)
    tf.summary.scalar('cost', cost)

with tf.name_scope('output'):
    variable_summaries(conv_net_output)

for grad in grads:
    with tf.name_scope('grads'):
        variable_summaries(grad)

#initializin graph and tensorboard writers
init = tf.global_variables_initializer()
sess = tf.Session()

summ_writer_train = tf.summary.FileWriter(os.path.join('summaries','train'), s
ess.graph)
summ_writer_test = tf.summary.FileWriter(os.path.join('summaries','test'), ses
s.graph)

merged = tf.summary.merge_all()

sess.run(init)

#minibatch gradient descent
for i in range(n_iterations):

    minibatches = random_mini_batches(x_train, y_train, batch_size, 1)
```

```
    for j, minibatch in enumerate(minibatches):
            idx = np.random.randint(0,train_size)
            batch_X, batch_Y = minibatch
            summ, _, cost_, accuracy_ = sess.run([merged, train_step, cost, ac
curacy],
                                  feed_dict = {x:batch_X, y:batch_Y})


    summ_train, cost_train, accuracy_train = sess.run([ merged, cost, accuracy
],
                                  feed_dict = {x:x_train, y:y_train})

    summ_test, cost_test, accuracy_test = sess.run([ merged, cost, accuracy],
                                  feed_dict = {x:x_test, y:y_test})

    if (i%20 == 0):
        print("Iteration ",i)
        print("cost train:", cost_train)
        print("acc train:", accuracy_train)
        print("cost test:", cost_test)
        print("acc test:", accuracy_test)

    # Write the obtained summaries to the file, so it can be displayed in the
 TensorBoard
    summ_writer_train.add_summary(summ_train, i)
    summ_writer_test.add_summary(summ_test, i)
    summ_writer_train.flush()
    summ_writer_test.flush()
```

```
WARNING:tensorflow:From <ipython-input-10-6c3ab86c6927>:51: softmax_cross_ent
ropy_with_logits (from tensorflow.python.ops.nn_ops) is deprecated and will b
e removed in a future version.
Instructions for updating:

Future major versions of TensorFlow will allow gradients to flow
into the labels input on backprop by default.

See `tf.nn.softmax_cross_entropy_with_logits_v2`.

Iteration  0
cost train: 1.3394886
acc train: 0.67376894
cost test: 2.3113377
acc test: 0.49853617
Iteration  20
cost train: 0.5494436
acc train: 0.80654764
cost test: 1.6182616
acc test: 0.58678377
Iteration  40
cost train: 0.51511633
acc train: 0.8321834
cost test: 0.86081266
acc test: 0.84190714
Iteration  60
cost train: 0.530075
acc train: 0.83630955
cost test: 0.9031895
acc test: 0.8318695
Iteration  80
cost train: 0.52028954
acc train: 0.8475379
cost test: 1.1499143
acc test: 0.8209954
```

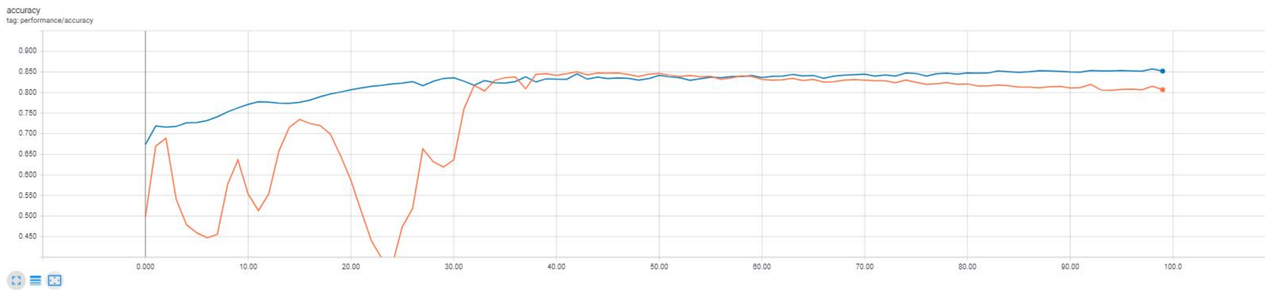
In [15]:
```
conv1_, activation1_, conv2_, activation2_,  full_, flat_, stack_= sess.run([c
onv11[0], activation11[0],
                conv22[0], activation22[0], conv_net_output, flat1[0], stack],
feed_dict = {x:x_train, y:y_train})
print("conv1:", conv1_.shape) #output of the first convolution (one channel)
print("conv2:", conv2_.shape) #output of the second convolution (one channel)
print("activation1:", activation1_.shape) #size of the first activation (one c
hannel)
print("activation2:", activation2_.shape) #size of the first activation (one c
hannel)
print("flat:", flat_.shape) #size after flatten (one channel)
print("concatenated:", stack_.shape) #size after concatenating
print("full:", full_.shape) #size of the output
```

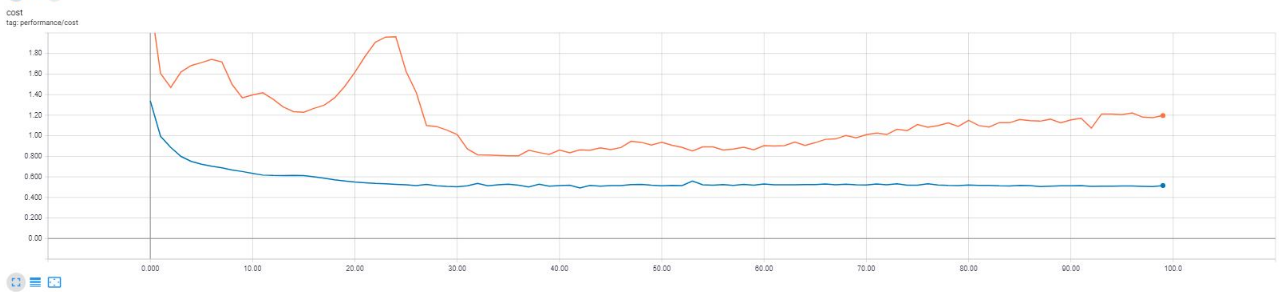
```
conv1: (14784, 252, 8)
conv2: (14784, 122, 4)
activation1: (14784, 126, 8)
activation2: (14784, 61, 4)
flat: (14784, 244)
concatenated: (14784, 732)
full: (14784, 4)
```

Plots in tensorboard

In the following picture, we show some of the plots made in TensorBoard:

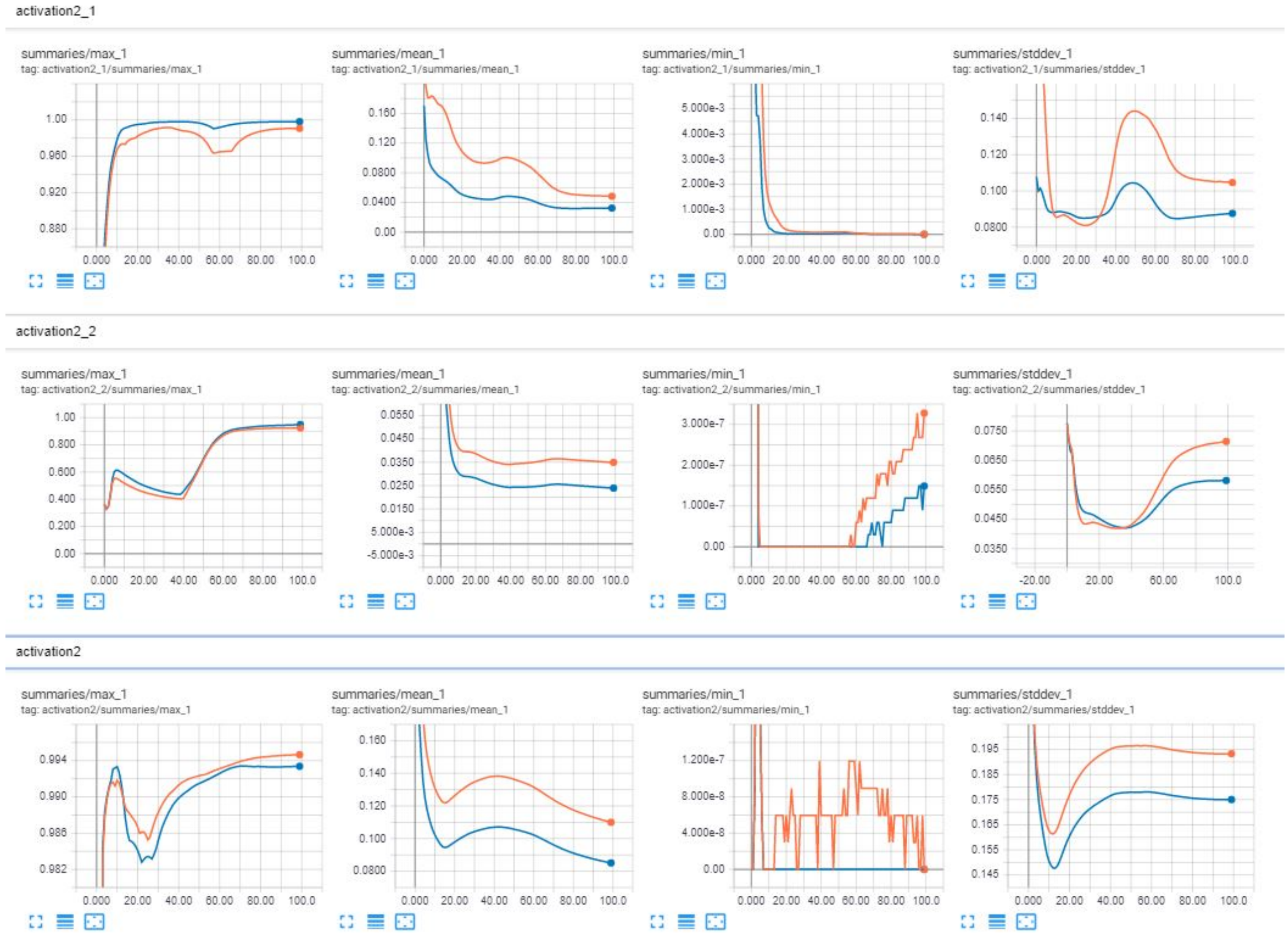
- Cost and accuracy (blue= train, red=test)





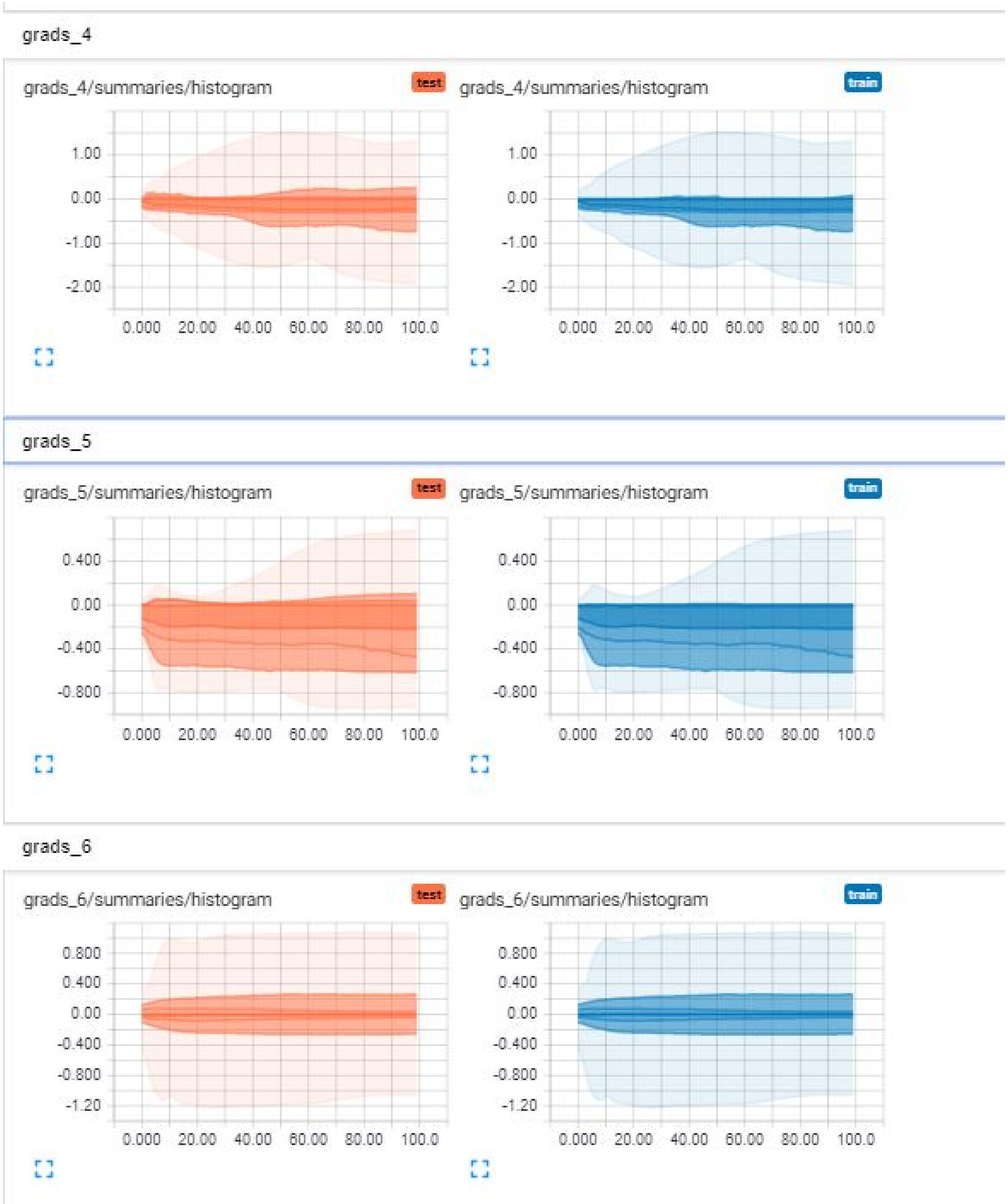
- Statisics of activations in the fist stage (different channels)

- Statisics of activations in the second stage (different channels)

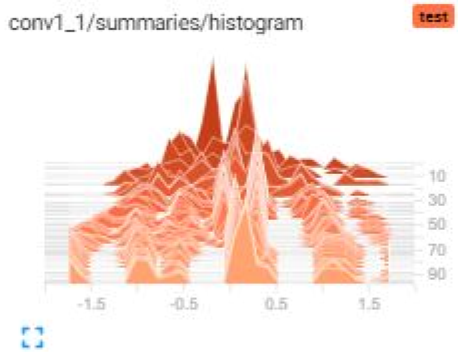


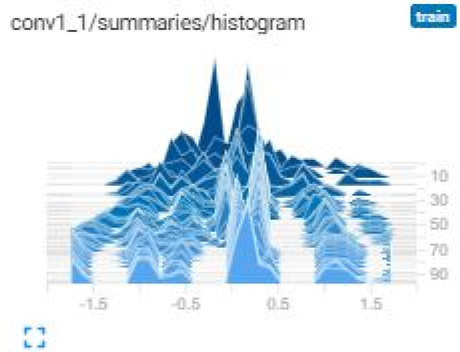
- Statisics of gradients in the fist stage (different channels)

- Distributions of some gradients

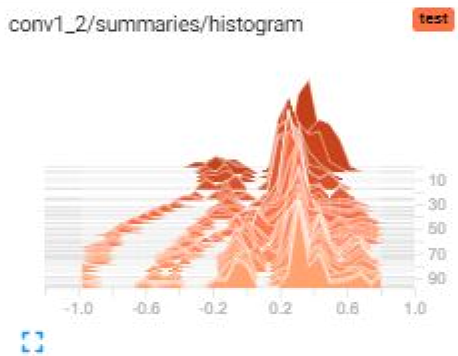


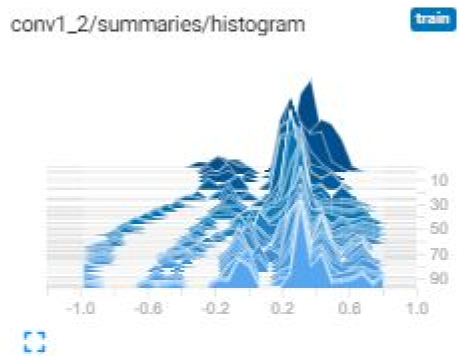
- Histograms of first convolutional filters conv1_1



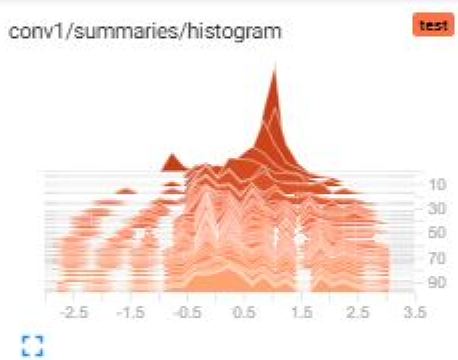


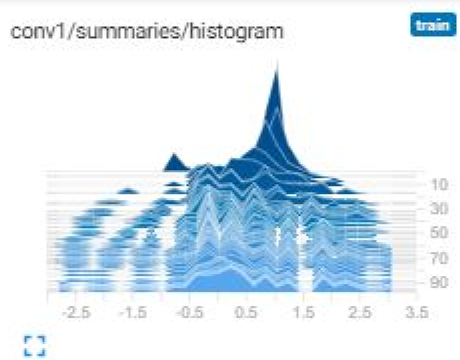
conv1_2





conv1





Conclusions and comments about implementation

In the paper there are some points that are not clear, and therefore we make some assumptions that may not meet the original ideas.

- The authors don't specify which 3D times series they use (which columns from the data file). There are several possibles subsets from the original dataset [5]. For simplicity, and speed in the preprocessing, we onyl work with the accelerometer signals from the hand (IMU hand).
- The implementation of the gradient update referenced in the paper could be done with *tf.train.MomentumOptimizer(learning_rate=lr, momentum= momentum, use_nesterov=True)*. However, we achieved better results with *AdamOptimizer* (which is the state of the art optimizer and widley used). To emualte the weight decay (=0.0005), we use L2-regularization (=0.00025).
- In the paper they specify that they use SGD. However, we use mini-batch (batch size= 64), since we were able to see softer convergence plots.
- We hiphotetize that because of the unexact specification of the columns to use, we are not able to reproduce completely the results. In our case, as we can see in above plots, we achieved an accuracy of 85.03% in test while the paper report an accuracy of 91.14% using step=32.

References:

[1] Zheng Y., Liu Q., Chen E., Ge Y., Zhao J.L. (2014) *Time Series Classification Using Multi-Channels Deep Convolutional Neural Networks.* In: Li F., Li G., Hwang S., Yao B., Zhang Z. (eds) Web-Age Information Management. WAIM 2014. Lecture Notes in Computer Science, vol 8485. Springer, Cham

[2] Code for mini-batches taken from: https://github.com/andersy005/deep-learning-specialization-coursera/blob/master/02-Improving-Deep-Neural-Networks/week3/Programming-Assignments/tf_utils.py (https://github.com/andersy005/deep-learning-specialization-coursera/blob/master/02-Improving-Deep-Neural-Networks/week3/Programming-Assignments/tf_utils.py)

[3] Regularization: https://markojerkic.com/build-a-multi-layer-neural-network-with-l2-regularization-with-tensorflow/ (https://markojerkic.com/build-a-multi-layer-neural-network-with-l2-regularization-with-tensorflow/)

[4] Tensorboard: https://www.tensorflow.org/guide/summaries_and_tensorboard (https://www.tensorflow.org/guide/summaries_and_tensorboard)

[5] PAMP2 Dataset: http://archive.ics.uci.edu/ml/datasets/pamap2+physical+activity+monitoring (http://archive.ics.uci.edu/ml/datasets/pamap2+physical+activity+monitoring)