Deep Learning - Median House Value Assesment Activity

April 9th, 2019

Daniel Mínguez Camacho: dani.min.cam@gmail.com
Javier de la Rúa Martínez: javierdlrm@outlook.com

Submitted to
Martin Molina Gonzalez
Daniel Manrique Gamo

1 Introduction

This document is created as an assignment for Deep Learning course at Universidad Politécnica de Madrid. The aim of this report is describing the actions performed during the implementation of a deep network using tensorflow.

The first section contains a description of the design process, how we installed the programming environment and the required libraries as well as how we built the network. Then we present the results and finally, we give a short conclusion of our experience during the work performed.

We won´t introduce formulas and other proofs of the different techniques because they are described in the class slides and can be easily found on the internet. We wanted also to reduce the extension of the document

1.1 Problem

This California Housing Prices dataset has been downloaded from StatLib repository (http://lib.stat.cmu.edu/datasets/). It is based on data from the 1990 California census, what is not important for deep learning. The original dataset appeared in R. Kelley Pace and Ronald Barry, “Sparse Spatial Autoregressions,” Statistics & Probability Letters 33, no. 3 (1997): 291–297.

The problem proposed to solve in this assignment is to predict the house value based on the others variables present in the dataset.

1.1 Data used

We have used the files MedianHouseValuePreparedCleanAttributes.csv and MedianHouseValueOneHotEncodedClasses.csv as described in the assignment description.

MedianHouseValuePreparedCleanAttributes.csv
The original dataset contained 20,640 instances corresponding to districts in california ranging from 600 to 3.000 people. This dataset has been previously cleaned, preprocessed and prepared with the following operations (described in class notes 2.1.1, slide 22):

- Total_bedrooms attribute has 207 missing values, (na or nan), which are removed since they are very little compared to the whole dataset.
- Dataset is randomized.
- Classes are encoded: first discretized and then one-hot encoded.
- Attributes are individually re-scaled, normalized with a min-max scaling within the range [-1,1]: x-(max/2) / (max-min)/2.
- The correlation matrix between all pairs of attributes has been calculated to visualize their dependencies. The results achieved report that total_rooms, total_bedrooms, population and households are highly (positive) correlated.

After this phase of data preparation, a final dataset of 20,433 instances are obtained with 8 attributes (InputsMedianHouseValueNormalized.csv): $longitude$ and $latitude$ (location), $medianage$, $totalrooms$, $totalbedrooms$, $population$, $households$ and $medianincome$. The last will be our label.

From this data, the classification problem consists on estimating the median house value, categorized into the following 10 clases (price intervals in thousand dollards): [15.0, 82.3], [82.4, 107.3], [107.4, 133.9], [134.0, 157.3], [157.4, 179.7], [179.8, 209.4], [209.5, 241.9], [242.0, 290.0], [290.1, 376.6] and [376.7, 500.0]. Each class is labelled from 0 (the cheapest) to 9 (the most expensive), and one-hot encoded in **MedianHouseValueOneHotEncodedClasses.csv** file.

2 Design process

2.1 Installation

We have decided to use Anaconda as our programming environment. The installation process was as follows:

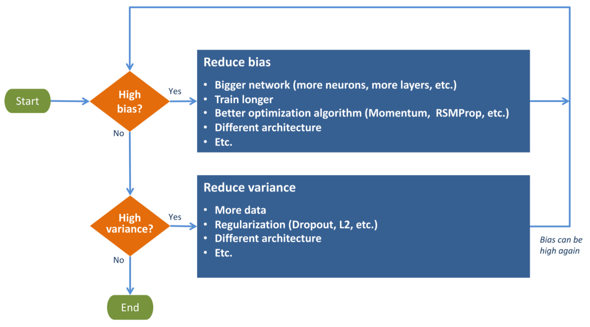
1. **Anaconda download and installation** , we downloaded Anaconda from the official webpage (https://www.anaconda.com/distribution/) and installed it following the steps of the program.
2. **Environment creation** we created a separated environment called tf-gpu for installing the libraries and we added it to jupyter
3. **Libraries installation** , after that, we installed the following libraries used in the notebook:

```
import sys
!conda install --yes --prefix {sys.prefix} tensorflow-gpu
!conda install --yes --prefix {sys.prefix} keras-gpu
```

4. **Test of tensorflow**, after the installation we executed the test program provided in class (matrix multiplication) in order to check that tensorflow was working properly

2.2 Process followed

During the development process we focused on using the approach discussed in class:



In order to achieve this, we structured the develoment process in three stages:

1. **Models search**, in this stage, we executed around 200 models with different architectures and parameters, in order to compare which approaches give better results. We analyzed the outputs using tensorboard.
2. **Models choice**, we selected the 5 best models according to the test accuracy and we re-executed them with a higher number of epochs. After that we saved the models.
3. **Ensemble method**, using the models obtained in the previous step, we constructed a ensemble network in order to check if it gave a better result

2.2.1 Models search

We explored different combinations of models with different architectures and hyperparameters:

- **Activation functions tried**: softmax, sigmoid, Hyperbolic tangent, ReLu, ELU, Leaky Relu
- **Loss functions tried**: Sigmoid cross-entropy, Softmax cross-entropy
- **Training techniques tried**: Gradient descent, Momentum, RMSprop
- **Normalization techniques tried** (without taking into account the changes made to the inputs): Dropout, Weights inizialization (Zero, random uniform, random normal, Xavier uniform, Xavier normal)
- **DL Architectures tried**: (500, 300, 300, 150,75,25,10),(300,150,75,25,10), (100,300,500,400,200,100,50,10), (150,75,25,10), (300,150,75)
- **Parameters tried**: Learning rate (0.1,0.01,0.001,0.0001), batch size (16-32-100,200) and random
- **Metric used**: loss and accuracy

We used the following libraries for the code developed:

```
import numpy as np
import pandas as pd
import tensorflow as tf
```

The main function used is below, we input the different paramters to create_run_model, then we create the graph for the network and run the model. The inputs are:

- **Inputs**: Number of inputs of the network (integer).
- **Outputs**: Number of outputs of the network (integer).
- **Learning rate**: for the training algorithm (float).
- **n_neurons**: Array of integers with the number of neurons per layer (the number of layers is the lenght of this array) (Array).
- **batch_norm**: boolean, if true batch normalization is applied on each layer.
- **dropout**: boolean, if true batch normalization is applied on each layer.
- **optimizer**: string, type of optimizer to use on each layer. Possible values ("relu", "elu", "leakyrelu", "softmax", "sigmoid", "tanh").
- **initb**: string, way of initialize the bias, possible values ("zero", "const"), the const is with value 0.1.
- **initw**: strig, way of initialize the weights, possible values ("Xavier_Normal", "Xavier_Uniform","RUniform", "RNormal", "TNormal"). The Uniform is between (-0.5, 0.5) and the normal has mean 0 and stdev 0.5.
- **l2**: boleean, if true l2 normalization is applied.
- **writter_train**: writter for recording the train values for display them in tensorboard.
- **writter_test**: writter for recording the test values for display then in tensorboard.

```
def create_run_model(inputs, outputs, learning_rate, n_neurons,
                     batch_norm, activation, loss_fun,
                     dropout,optimizer, initb, initw,
                     l2, writter_train, writter_test):

    g = create_graph(inputs, outputs, learning_rate, n_neurons,batch_norm,
                     activation, loss_fun, dropout, optimizer, initb, initw, l2)

    run_model(writer_train, writer_test, g)
```

The graph is created with the function below, the structure is the following:

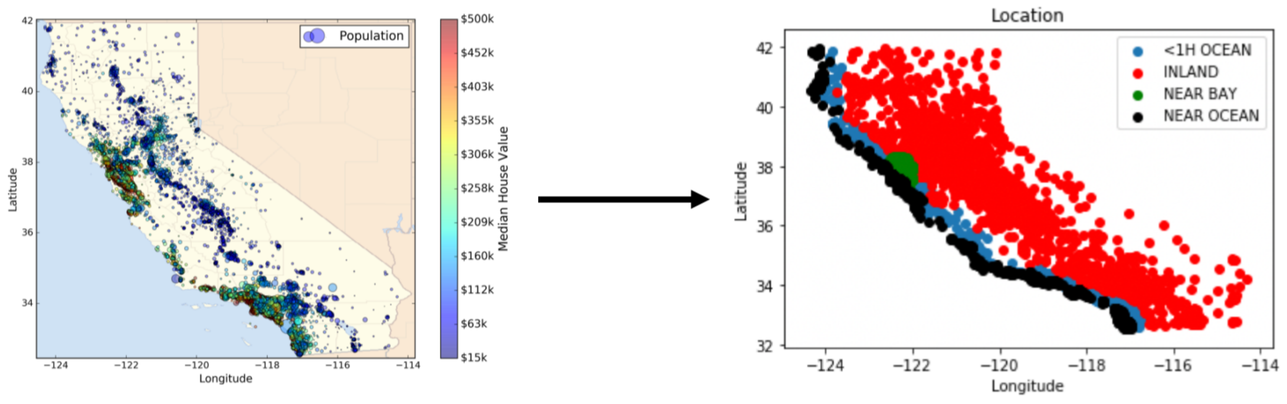
1. Create an empty graph.
2. Define the inputs to the graph.
3. Define the hidden layers and its activations functions.
4. Define loss function.
5. Define optimizer.
6. Define the accuracy.
7. Define logs for tensorboard and variables to use in the train step.

```
def create_graph(inputs, outputs, learning_rate, n_neuron,batch_norm, activation,
                 loss_fun, dropout, optimizer, initb, initw, l2):
    # 1. Create an empty graph

    g1 = tf.Graph()
    with g1.as_default() as g:

        # 2. Define the inputs to the graph

        training = tf.placeholder_with_default (False, shape=(), name = "training") #for batch nor
m.
        X = tf.placeholder (dtype=tf.float32, shape=(None,inputs),name="X")  #Data
        t = tf.placeholder (dtype=tf.float32, shape=(None,outputs), name="Labels")  #Labels

        # 3. Define the hidden layers and its activations functions

        w = []
        hidden_layers = []
        layers_temp, w_temp = dense_layer(X,inputs, n_neurons[0], batch_norm, activation, dropout,
\
                                          training, initb, initw)

        hidden_layers.append(layers_temp)
        w.append(w_temp)

        for layer in range(1,len(n_neurons)):
            layers_temp, w_temp = dense_layer(hidden_layers[layer-1],
                                              n_neurons[layer-1],
                                              n_neurons[layer],
                                              batch_norm, activation, dropout, \
                                              training, initb, initw)
            hidden_layers.append(layers_temp)
            w.append(w_temp)

        layers_temp, w_temp = dense_layer(hidden_layers[len(n_neurons)-1],
                                          n_neurons[-1],
                                          outputs,False, "None", False, training, initb, initw)
        net_out = layers_temp
        w.append(w_temp)

        y = tf.nn.softmax(logits=net_out, name="y")

        # 4. Define loss function

        beta = 0.01
        with tf.name_scope("Cost"):
            if loss_fun == "softmax":
                cross_entropy = tf.nn.softmax_cross_entropy_with_logits_v2(labels=t,logits=net_out
)
            elif loss_fun == "sigmoid":
                cross_entropy = tf.nn.sigmoid_cross_entropy_with_logits(labels=t, logits=net_out)
            # Loss function with L2 Regularization with beta=0.01
            if l2 == True:
                regularizers =tf.nn.l2_loss(w[0])
                for i in range(1,len(w)):
                    regularizers =  regularizers + tf.nn.l2_loss(w[i])
                mean_log_loss=tf.reduce_mean(cross_entropy+beta*regularizers,name="mean_log_loss")
            else:
                mean_log_loss = tf.reduce_mean (cross_entropy, name="mean_log_loss")

        # 5. Define optimizer

        with tf.name_scope("train"):
            if optimizer == "Adam":
                train_step = tf.train.AdamOptimizer(learning_rate).minimize(mean_log_loss)
            elif optimizer == "RMSPROP":
                train_step = tf.train.RMSPropOptimizer(learning_rate).minimize(mean_log_loss)
            elif optimizer == "momentum":
                train_step = tf.train.MomentumOptimizer(learning_rate,0.9).minimize(mean_log_loss)
            elif optimizer == "gradient_desc":
                train_step =tf.train.GradientDescentOptimizer(learning_rate).minimize(mean_log_los
s)

        # 6 Define Accuracy

        with tf.name_scope("Evaluation"):
            correct_predictions = tf.equal(tf.argmax(y,1),tf.argmax(t,1))
            accuracy = tf.reduce_mean(tf.cast(correct_predictions,tf.float32))
        # 7 Define logs for tensorboard and variables to use in the train step

        tf.summary.scalar("cross_entropy", mean_log_loss)
        tf.summary.scalar("accuracy", accuracy)

        merged_summary = tf.summary.merge_all()

        g.add_to_collection("elements", training)
        g.add_to_collection("elements", X)
        g.add_to_collection("elements", t)
        g.add_to_collection("elements", y)
        g.add_to_collection("elements", mean_log_loss)
        g.add_to_collection("elements", train_step)
        g.add_to_collection("elements", correct_predictions)
        g.add_to_collection("elements", accuracy)
        g.add_to_collection("elements", merged_summary)

    return g1
```

We use the following function for create each hidden layer cell. It uses the inputs already explained in the previous functions. The structure is the following:

1. Weigts definition
2. Bias definition
3. Activation definition
4. Regularization techiques. We have tried batch normalization before the activation function
5. Logs creation

We use batch normalization and then dropout (both after the activation function after researching about the topic.



```python
def dense_layer(input, channels_in, channels_out, batch_norm, activation, dropout, training, init
b, initw, name="dense"):
    with tf.name_scope(name):

        # 1. Weights definition

        if initw == "RUniform":
            w = tf.Variable(tf.random_uniform([channels_in, channels_out],\
                                              minval=-0.5, maxval = 0.5),name = "W")
        elif initw == "Xavier_Normal":
            w = tf.Variable(tf.glorot_normal_initializer()((channels_in, channels_out)),name =
"W")
        elif initw == "Xavier_Uniform":
            w = tf.Variable(tf.glorot_uniform_initializer()((channels_in, channels_out)),name =
"W")
        elif initw == "RNormal":
            w = tf.Variable(tf.random_normal([channels_in, channels_out], stddev=0.5),name = "W")
        elif initw == "TNormal":
            w = tf.Variable(tf.truncated_normal([channels_in, channels_out], stddev=0.5),name =
"W")

        # 2. Bias definition

        if initb == "zero":
            b = tf.Variable(tf.constant(0.1, shape = [channels_out]), name = "b")
        elif initb == "const":
            b = tf.Variable(tf.zeros(shape = [channels_out]), name = "b")

        # 3. Actication definition

        prev_layer = input

        if batch_norm:
            prev_layer = tf.layers.batch_normalization(prev_layer, training=training)

        if activation == "relu":
            prev_layer = tf.nn.relu(tf.matmul(prev_layer, w) + b)
        elif activation == "elu":
            prev_layer = tf.nn.elu(tf.matmul(prev_layer, w) + b)
        elif activation == "leakyrelu":
            prev_layer = tf.nn.leaky_relu(tf.matmul(prev_layer, w) + b)
        elif activation == "softmax":
            prev_layer = tf.nn.softmax(tf.matmul(prev_layer, w) + b)
        elif activation == "sigmoid":
            prev_layer = tf.nn.sigmoid(tf.matmul(prev_layer, w) + b)
        elif activation == "tanh":
            prev_layer = tf.nn.tanh(tf.matmul(prev_layer, w) + b)

        # 4. Regularization techniques

        if dropout:
            prev_layer = tf.nn.dropout(prev_layer, rate = 0.25)

        # 4. Logs creation

        tf.summary.histogram("weights", w)
        tf.summary.histogram("bias", b)
        tf.summary.histogram("act", prev_layer)

        return prev_layer, w
```

Finally we run the model for training with the following function:

1. Paramaters definition
2. Graph initialization
3. Retrieve elements used from graph
4. Define the feeds for each call
5. Loop for training
6. Logs for tensorboard
7. Evaluation metrics

We used different values for epochs and batch size for the different models (and fixed batch vs random batch selection also), although here they are fixed.

```python
def run_model(writer_test, writer_train, g):

    # 1. Parameters definition

    accuracy_train_history = []
    n_epochs = 30000
    batch_size = 400


    # 2. Graph initialization

    with tf.Session(graph = g) as sess:


        sess.run(tf.global_variables_initializer())
        # This parameter is activated when we want ot save the session for load it later
        saver = tf.train.Saver()

        writer_train.add_graph(sess.graph)
        writer_test.add_graph(sess.graph)

        # 3. Retrieve elements used from graph

        training, X, t, y, mean_log_loss, train_step, \
        correct_predictions, accuracy, merged_summary = g.get_collection("elements")

        # 4. Define the feeds for each call

        test_feed = {training: False, X: x_test[:NUM_TEST_EXAMPLES],\
                                      t: t_test[:NUM_TEST_EXAMPLES]}
        train_feed = {training: False, X: x_train[:NUM_TRAINING_EXAMPLES], \
                                       t: t_train[:NUM_TRAINING_EXAMPLES]}

        # 5. Loop for training

        for epoch in range(n_epochs):

            offset = np.random.randint(0,NUM_TRAINING_EXAMPLES,300)

            feed = {training: True, X: x_train[offset], t: t_train[offset]}
            feed_eval = {training: False, X: x_train[offset], t: t_train[offset]}

            # 6. Logs for tensorboard

            if epoch % 10 == 0:
                s_train = sess.run(merged_summary, feed_dict=feed_eval)
                s_test = sess.run(merged_summary, feed_dict=test_feed)
                writer_train.add_summary(s_train,epoch)
                writer_test.add_summary(s_test,epoch)

            if epoch % 500 == 0:
                train_accuracy = sess.run(accuracy, feed_dict=feed_eval)
                test_accuracy = sess.run(accuracy, feed_dict=test_feed)
                print("step %d, training accuracy %g, test accuracy %g" % \
                      (epoch, train_accuracy, test_accuracy))

            sess.run (train_step, feed_dict=feed)

        # 7. Evaluation metrics

        accuracy_test = accuracy.eval(feed_dict=test_feed)
        accuracy_train = accuracy.eval(feed_dict=train_feed)

        test_predictions = y.eval(feed_dict={X: x_test[:NUM_TEST_EXAMPLES]})
        test_correct_preditions = correct_predictions.eval (feed_dict=test_feed)

        train_mean_log_loss = mean_log_loss.eval(feed_dict = test_feed)
        test_mean_log_loss = mean_log_loss.eval(feed_dict = test_feed)

        # This is activated when we want to save the model
        saver.save(sess, "/TF_Models/Batch/Batch.ckpt")
        sess.close()
```

With this function now we launch our search, first we read the data:

```
run 1.ReadingData.py
```

```
x_train: (16346, 8)
t_train: (16346, 10)
x_dev: (2043, 8)
t_dev: (2043, 10)
x_test: (2044, 8)
t_test: (2044, 10)
```

```
INPUTS = x_train.shape[1]
OUTPUTS = t_train.shape[1]
NUM_TRAINING_EXAMPLES = int(round(x_train.shape[0]/1))
NUM_DEV_EXAMPLES = int (round (x_dev.shape[0]/1))
NUM_TEST_EXAMPLES = int (round (x_test.shape[0]/1))
```

We define one more function for creating an string for naming the folders created for tensorflow (and so be able to query them using regex expressions), we basically input all the parameters used and concatenate them:

```
def make_hparam_string(initb, initw, learning_rate, n_neurons, batch_norm, optimizer, activation,
loss_fun, dropout, l2):

    output = "learning_rate = " + str(learning_rate) + ", neurons = " + str(n_neurons) + \
            ", batch_norm =" + str(batch_norm) + ", opt = " + str(optimizer) + \
            ", act = " + str(activation) + \
            ", loss = " + str(loss_fun) + ", drop = " + str(dropout) + \
            ", binit = " + str(initb) + \
            ", winit = " + str(initw) + ", l2 = " + str(l2)

    return  output
```

We also have to define the parameters, we include all the parameters we tried below, although we didn´t tried all the combinations for computing time reasons:

```
opt = ["Adam", "RMSPROP", "momentum", "gradient_desc"]
act = ["relu", "elu", "leakyrelu", "softmax", "sigmoid", "tanh"]
loss = ["softmax", "sigmoid"]
BInit = ["zero", "const"]
WInit = ["Xavier_Normal", "Xavier_Uniform","RUniform",  "RNormal", "TNormal"]
batch_norm = [True, False]
dropout = [True, False]
l2 = [True, False]
rates = [1E-1, 1E-2, 1E-3, 1E-4]
neurons = [[500, 300, 300, 150,75,25,10],[300,150,75,25,10], \
           [100,300,500,400,200,100,50,10], [150,75,25,10],[300,150,75]]
```

Finally we run the different models (again although its displayed like these, we selected different combinations of the parameters above for the different runs we did), basically we:

1. Initialize the loop
2. Create the folder string
3. Print the model for log purposes
4. Create the tensorboard writters
5. Create and run the model

```python
def run()
    # 2. Create the folder string

    hparam_str = make_hparam_string(initb, initw, \
                                    learning_rate,\
                                    n_neurons, batch_norm, \
                                    optimizer, activation, \
                                    loss_fun, dropout, l2)
    i = i + 1

    # 3. Print the model for log purposes

    print("-----------------------------")
    print("MODEL " + hparam_str)
    print("-----------------------------")

    # 4. Create the tensorboard writters

    writer_train = tf.summary.FileWriter("/TF_Logs/20190904/"\
                                         + str(i) + " Train " + \
                                         hparam_str)
    writer_test = tf.summary.FileWriter("/TF_Logs/20190904/" \
                                        + str(i) + " Test " + \
                                        hparam_str)

    # 5. Create and run the model

    create_run_model(INPUTS,OUTPUTS, learning_rate,n_neurons, \
                     batch_norm,activation,loss_fun,dropout,optimizer, \
                     initb, initw, l2, writer_test,writer_train)
```

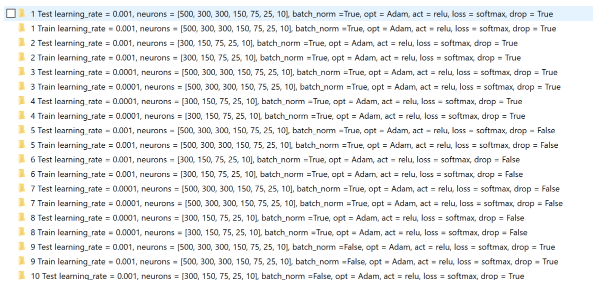
```python
# This variable is for ordering the folders that are created
i = 0

# 1. Initialize the loop

for optimizer in ["Adam"]:
    for activation in ["elu"]:
        for loss_fun in ["softmax"]:
            for initb in BInit:
                for initw in WInit:
                    for batch_norm in [False]:
                        for dropout in [False]:
                            for learning_rate in rates:
                                for n_neurons in neurons:
                                    run(INPUTS,OUTPUTS, \
                                        learning_rate,n_neurons, \
                                        batch_norm,activation, \
                                        loss_fun,dropout,\
                                        optimizer,initb, \
                                        initw, l2, \
                                        writer_test,writer_train)
```

The strategy followed during the search was executing a large number of models with low epochs and non-random batch size at the beggining in order to check if the model improve and how quickly. After that we selected the parameters that worked better for a second execution with less models but more epochs. Finally we executed a small number of models with more epochs and specific parameters.

The cmd code used to execute tensorboard is (remember that the logs are stored in the path were jupyter is executed):

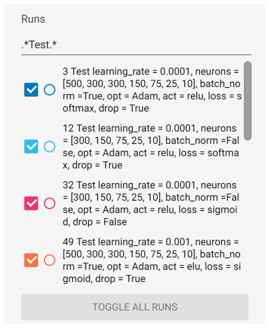
```
tensorboard --logdir /logs/1
```

So as we stated in the first execution we obtained arond 120 models:



```
1 Test learning_rate = 0.001, neurons = [500, 300, 300, 150, 75, 25, 10], batch_norm =True, opt = Adam, act = relu, loss = softmax, drop = True
1 Train learning_rate = 0.001, neurons = [500, 300, 300, 150, 75, 25, 10], batch_norm =True, opt = Adam, act = relu, loss = softmax, drop = True
2 Test learning_rate = 0.001, neurons = [300, 150, 75, 25, 10], batch_norm =True, opt = Adam, act = relu, loss = softmax, drop = True
2 Train learning_rate = 0.001, neurons = [300, 150, 75, 25, 10], batch_norm =True, opt = Adam, act = relu, loss = softmax, drop = True
3 Test learning_rate = 0.0001, neurons = [500, 300, 300, 150, 75, 25, 10], batch_norm =True, opt = Adam, act = relu, loss = softmax, drop = True
3 Train learning_rate = 0.0001, neurons = [500, 300, 300, 150, 75, 25, 10], batch_norm =True, opt = Adam, act = relu, loss = softmax, drop = True
4 Test learning_rate = 0.0001, neurons = [300, 150, 75, 25, 10], batch_norm =True, opt = Adam, act = relu, loss = softmax, drop = True
4 Train learning_rate = 0.0001, neurons = [300, 150, 75, 25, 10], batch_norm =True, opt = Adam, act = relu, loss = softmax, drop = True
5 Test learning_rate = 0.001, neurons = [500, 300, 300, 150, 75, 25, 10], batch_norm =True, opt = Adam, act = relu, loss = softmax, drop = False
5 Train learning_rate = 0.001, neurons = [500, 300, 300, 150, 75, 25, 10], batch_norm =True, opt = Adam, act = relu, loss = softmax, drop = False
6 Test learning_rate = 0.001, neurons = [300, 150, 75, 25, 10], batch_norm =True, opt = Adam, act = relu, loss = softmax, drop = False
6 Train learning_rate = 0.001, neurons = [300, 150, 75, 25, 10], batch_norm =True, opt = Adam, act = relu, loss = softmax, drop = False
7 Test learning_rate = 0.0001, neurons = [500, 300, 300, 150, 75, 25, 10], batch_norm =True, opt = Adam, act = relu, loss = softmax, drop = False
7 Train learning_rate = 0.0001, neurons = [500, 300, 300, 150, 75, 25, 10], batch_norm =True, opt = Adam, act = relu, loss = softmax, drop = False
8 Test learning_rate = 0.0001, neurons = [300, 150, 75, 25, 10], batch_norm =True, opt = Adam, act = relu, loss = softmax, drop = False
8 Train learning_rate = 0.0001, neurons = [300, 150, 75, 25, 10], batch_norm =True, opt = Adam, act = relu, loss = softmax, drop = False
9 Test learning_rate = 0.001, neurons = [500, 300, 300, 150, 75, 25, 10], batch_norm =False, opt = Adam, act = relu, loss = softmax, drop = True
9 Train learning_rate = 0.001, neurons = [500, 300, 300, 150, 75, 25, 10], batch_norm =False, opt = Adam, act = relu, loss = softmax, drop = True
10 Test learning_rate = 0.001, neurons = [300, 150, 75, 25, 10], batch_norm =False, opt = Adam, act = relu, loss = softmax, drop = True
```

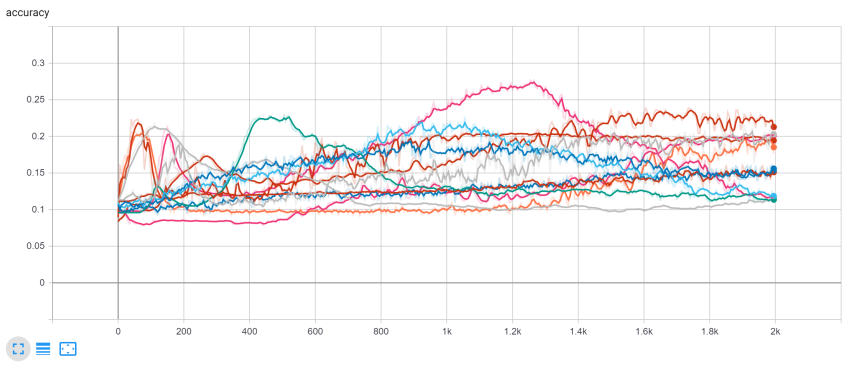
We analyzed the results with tensorflow, using regex to filter the results



```
.*Test.*
```

- ☑ 3 Test learning_rate = 0.0001, neurons = [500, 300, 300, 150, 75, 25, 10], batch_norm =True, opt = Adam, act = relu, loss = softmax, drop = True
- ☑ 12 Test learning_rate = 0.0001, neurons = [300, 150, 75, 25, 10], batch_norm =False, opt = Adam, act = relu, loss = softmax, drop = True
- ☑ 32 Test learning_rate = 0.0001, neurons = [300, 150, 75, 25, 10], batch_norm =False, opt = Adam, act = relu, loss = sigmoid, drop = False
- ☑ 49 Test learning_rate = 0.001, neurons = [500, 300, 300, 150, 75, 25, 10], batch_norm =True, opt = Adam, act = elu, loss = sigmoid, drop = True

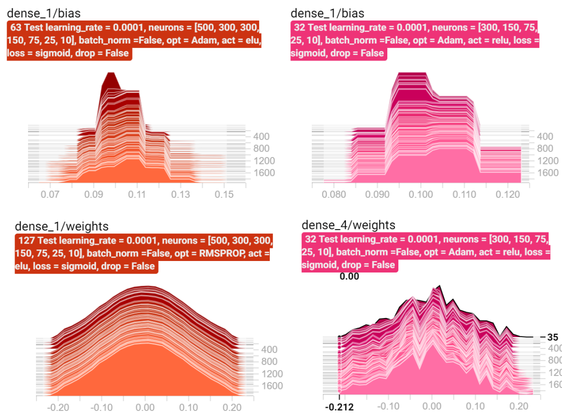
here we can see the best 11 models with 2000 epochs (showing only test results):



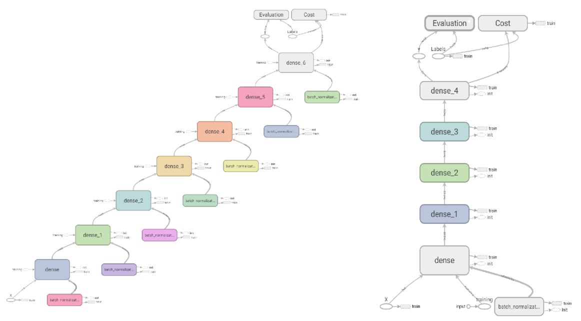
The best results were obtained with the following parameters with a test accuracy of 0.2236 and train accuracy 0.2233 after 2.000 epochs:

- learning_rate = 0.0001
- neurons = [500, 300, 300, 150, 75, 25, 10]
- batch_norm =False
- opt = Adam
- act = elu
- loss = sigmoid
- drop = False

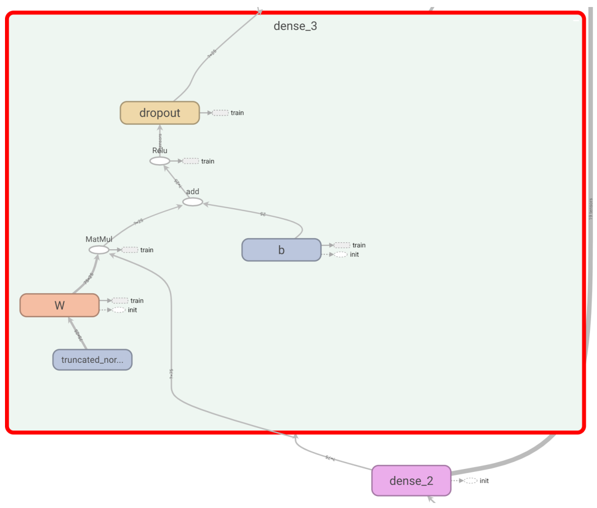
Using tensorboard we can also see how the weights evolve over time and the difference between different inizialization methods. For example, below we can see how the bias of two different models evolve over time (initialized in 0.1) and the difference in the weights of the last layer and the first (initialized with random normal), we can see how the last layer is more updated and the first layer remains normal (due to the low epochs applied):



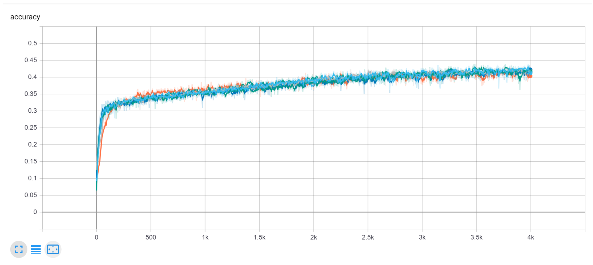
We can also see the graphs structures of the different models (left example with batch normalization and right without):



And we can zoom to see inside (for example to see the dropout):



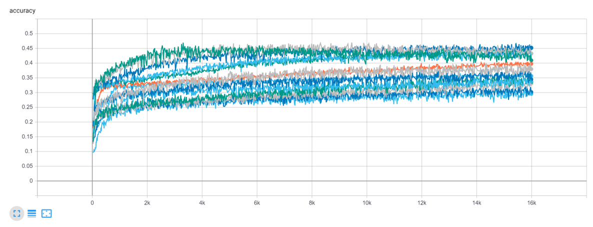
In summary, after that, we executed again using 4000 epochs, the parameters that gave better results and some more. We obtained for the best model a test accuracy of 0.4193 and train accuracy of 0.38 (we display only test results)



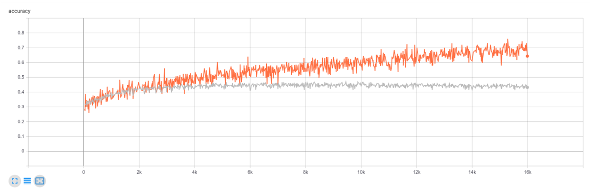
The parameters used for the best model were:

- learning_rate = 0.001
- neurons = [300, 150, 75, 25, 10]
- batch_norm =False
- opt = Adam
- act = elu
- loss = softmax
- drop = False
- binit = const
- drop = Xavier_Uniform

And finally we executed some models for a much large number of epochs (16.000), obtaining the following test resutls:



The best model regarding test achieved 0.4697 in test set and 0.6367 in train. The best model regarding train achieved 0.76 in train, although 0.4525 in test set. If we have a look at the graph of this model we can see that its overfitting mor or less since epoch 4.000:



2.2.2 Models Choice

After that, we selected the 5 models that gave us better results and we reexecuted them again, saving the results

```
def model_execution(optimizer,activation,loss_fun,initb,
                    initw,batch_norm,dropout,l2,learning_rate,n_neurons,i)

    hparam_str = make_hparam_string(initb, initw, \
                                    learning_rate,\
                                    n_neurons, batch_norm, \
                                    optimizer, activation, \
                                    loss_fun, dropout, l2)

    print("----------------------------")
    print("MODEL " + hparam_str)
    print("----------------------------")

    writer_train = tf.summary.FileWriter("/TF_Logs/20190904/"\
                                         + str(i) + " Train " + \
                                         hparam_str)
    writer_test = tf.summary.FileWriter("/TF_Logs/20190904/" \
                                        + str(i) + " Test " + \
                                        hparam_str)

    create_run_model(INPUTS,OUTPUTS, learning_rate,n_neurons, \
                     batch_norm,activation,loss_fun,dropout,optimizer, \
                     initb, initw, l2, writer_test,writer_train)
```

We repeated the above code for the following parameters:

```
## Model 1 - Overfitted best result (train 0.96 - 50K epochs - Train 0.94 - Test 0.42)
optimizer = "Adam"
activation = "elu"
loss_fun = "softmax"
initb = "const"
initw = "Xavier_Normal"
batch_norm = False
dropout = False
l2 = False
learning_rate = 1E-3
n_neurons = [500, 300, 300, 150,75,25,10]
i = 0

model_execution(optimizer,activation,loss_fun,initb,
                    initw,batch_norm,dropout,l2,learning_rate,n_neurons,i)
```

```
## Model 2 - Best test result (without overfitting) (9.5K epochs - Train 0.61 - Test 0.44)
optimizer = "Adam"
activation = "elu"
loss_fun = "softmax"
initb = "const"
initw = "Xavier_Normal"
batch_norm = False
dropout = False
l2 = False
learning_rate = 1E-3
n_neurons = [500, 300, 300, 150,75,25,10]

i = 2

model_execution(optimizer,activation,loss_fun,initb,
                    initw,batch_norm,dropout,l2,learning_rate,n_neurons,i)
```

```
## Model 3 - 2nd best test result (20K epochs - Train 0.53 - Test 0.45)
optimizer = "Adam"
activation = "elu"
loss_fun = "softmax"
initb = "const"
initw = "Xavier_Uniform"
batch_norm = False
dropout = False
l2 = False
learning_rate = 1E-3
n_neurons = [300, 150, 75, 25, 10]

i = 3

model_execution(optimizer,activation,loss_fun,initb,
                    initw,batch_norm,dropout,l2,learning_rate,n_neurons,i)
```

```
## Model 4 - Best dropout result (120K epochs - Train 0.35 - Test 0.36)
optimizer = "Adam"
activation = "relu"
loss_fun = "softmax"
initb = "const"
initw = "Xavier_Normal"
batch_norm = False
dropout = True
l2 = False
learning_rate = 1E-4
n_neurons = [300, 150, 75, 25, 10]

i = 4

model_execution(optimizer,activation,loss_fun,initb,
                    initw,batch_norm,dropout,l2,learning_rate,n_neurons,i)
```

```
## Model 5 - Best L2 result (30K epochs - Train - Test)
optimizer = "RMSPROP"
activation = "elu"
loss_fun = "softmax"
initb = "const"
initw = "Xavier_Normal"
batch_norm = False
dropout = False
l2 = True
learning_rate = 1E-3
n_neurons = [300, 150, 75, 25, 10]

i = 5

model_execution(optimizer,activation,loss_fun,initb,
                    initw,batch_norm,dropout,l2,learning_rate,n_neurons,i)
```

2.2.3 Ensemble model

After saving all the models we reload them and perform the ensemble method:

```
def get_pred(Path, name):
    tf.reset_default_graph()
    # Later, launch the model, use the saver to restore variables from disk, and
    # do some work with the model.
    with tf.Session() as sess:
        new_saver = tf.train.import_meta_graph( Path + name + '.ckpt.meta')
        new_saver.restore(sess, tf.train.latest_checkpoint(Path))
        g = tf.get_default_graph()
        training, X, t, y, mean_log_loss, train_step, \
        correct_predictions, accuracy, merged_summary = g.get_collection("elements")

        # Check the values of the variables
        predictions_test = y.eval(feed_dict={X: x_test[:NUM_TEST_EXAMPLES]})
        predictions_dev = y.eval(feed_dict={X: x_dev[:NUM_TEST_EXAMPLES]})
        predictions_train = y.eval(feed_dict={X: x_train[:NUM_TEST_EXAMPLES]})

        sess.close()
    return predictions_test, predictions_dev, predictions_train
```

```
p_test1, p_dev1, p_train1 = get_pred("/TF_Models/Overfit/", "Overfit")
p_test2, p_dev2, p_train2 = get_pred("/TF_Models/Best/", "Best")
p_test3, p_dev3, p_train3 = get_pred("/TF_Models/2ndBest/", "2ndBest")
p_test4, p_dev4, p_train4 = get_pred("/TF_Models/Dropout/", "Dropout")
p_test5, p_dev5, p_train5 = get_pred("/TF_Models/Batch/", "Batch")
```

```
INFO:tensorflow:Restoring parameters from /TF_Models/Overfit/Overfit.ckpt
INFO:tensorflow:Restoring parameters from /TF_Models/Best/Best.ckpt
INFO:tensorflow:Restoring parameters from /TF_Models/2ndBest/2ndBest.ckpt
INFO:tensorflow:Restoring parameters from /TF_Models/Dropout/Dropout.ckpt
INFO:tensorflow:Restoring parameters from /TF_Models/Batch/Batch.ckpt
```

Option 1: Average of the results

We calculate the average of the results as the final result

```
avg_train = np.mean([p_train1, p_train1, p_train1, p_train1, p_train1], axis=0)
avg_test = np.mean([p_test1, p_test1, p_test1, p_test1, p_test1], axis=0)
avg_dev = np.mean([p_dev1, p_dev1, p_dev1, p_dev1, p_dev1], axis=0)
```

```
test = np.array([t_test[i].argmax() for i in range(avg_test.shape[0])])
train = np.array([t_train[i].argmax() for i in range(avg_train.shape[0])])
dev = np.array([t_dev[i].argmax() for i in range(avg_dev.shape[0])])
```

```
result_train = np.array([avg_train[i].argmax() for i in range(avg_train.shape[0])])
result_test = np.array([avg_test[i].argmax() for i in range(avg_test.shape[0])])
result_dev = np.array([avg_dev[i].argmax() for i in range(avg_dev.shape[0])])
```

```
train_result = np.equal(train, result_train)
test_result = np.equal(test, result_test)
dev_result = np.equal(dev, result_dev)
print("Train accuracy: " + str(np.average(train_result)))
print("Test accuracy: " + str(np.average(test_result)))
print("Dev accuracy: " + str(np.average(dev_result)))
```

```
Train accuracy: 0.8918786692759295
Test accuracy: 0.4207436399217221
Dev accuracy: 0.44591287322564854
```

Option 2: Voting system

We calculate the resul of each network and perform a voting system, choosing model 3 or model 2 in case of draw.

```
train1 = np.array([p_train1[i].argmax() for i in range(avg_train.shape[0])])
train2 = np.array([p_train2[i].argmax() for i in range(avg_train.shape[0])])
train3 = np.array([p_train3[i].argmax() for i in range(avg_train.shape[0])])
train4 = np.array([p_train4[i].argmax() for i in range(avg_train.shape[0])])
train5 = np.array([p_train5[i].argmax() for i in range(avg_train.shape[0])])

test1 = np.array([p_test1[i].argmax() for i in range(avg_test.shape[0])])
test2 = np.array([p_test2[i].argmax() for i in range(avg_test.shape[0])])
test3 = np.array([p_test3[i].argmax() for i in range(avg_test.shape[0])])
test4 = np.array([p_test4[i].argmax() for i in range(avg_test.shape[0])])
test5 = np.array([p_test5[i].argmax() for i in range(avg_test.shape[0])])

dev1 = np.array([p_dev1[i].argmax() for i in range(avg_dev.shape[0])])
dev2 = np.array([p_dev2[i].argmax() for i in range(avg_dev.shape[0])])
dev3 = np.array([p_dev3[i].argmax() for i in range(avg_dev.shape[0])])
dev4 = np.array([p_dev4[i].argmax() for i in range(avg_dev.shape[0])])
dev5 = np.array([p_dev5[i].argmax() for i in range(avg_dev.shape[0])])
```

```
train_voting = np.array([train1, train2, train3, train4, train5])
test_voting = np.array([test1, test2, test3, test4, test5])
dev_voting = np.array([dev1, dev2, dev3, dev4, dev5])
```

```
def voting_calc(inp_array):
    final = np.array([])

    for i in range(inp_array.shape[1]):

        occurances = np.bincount(inp_array[:,i])

        if len(np.where(occurances == np.max(occurances)))>1:
            if np.isin(inp_array[:,i][3],np.where(occurances == 2)):

                final = np.append(final,inp_array[:,i][3])

            else:

                final = np.append(final,inp_array[:,i][2])

        else:

            final = np.append(final,np.argmax(occurances))

    return final
```

```
final_train = voting_calc(train_voting)
final_test = voting_calc(test_voting)
final_dev = voting_calc(dev_voting)
```

```
print("Train accuracy: " + str(np.mean(np.equal(final_train, result_train))))
print("Test accuracy: " + str(np.mean(np.equal(final_test, result_test))))
print("Dev accuracy: " + str(np.mean(np.equal(final_dev, result_dev))))
```

```
Train accuracy: 0.6643835616438356
Test accuracy: 0.6453033268101761
Dev accuracy: 0.6441507586882036
```

We can see that we improved the results in a considerable way.

3 Final results

Since we were working with the test set since the beggining when developing the models, we use the t_test and x_test as development sets and x_dev as final test sets.

In this section we are going to present the final results (Train-Test-Dev) for each of the 5 models selected and the ensemble.

3.1 Model 1 - Overfitted

```
result_train =  np.array([p_train1[i].argmax() for i in range(p_train1.shape[0])])
result_test = np.array([p_test1[i].argmax() for i in range(p_test1.shape[0])])
result_dev = np.array([p_dev1[i].argmax() for i in range(p_dev1.shape[0])])

train_result = np.equal(train, result_train)
test_result = np.equal(test, result_test)
dev_result = np.equal(dev, result_dev)

print("Train accuracy: " + str(np.average(train_result)))
print("Test accuracy: " + str(np.average(test_result)))
print("Dev accuracy: " + str(np.average(dev_result)))
```

```
Train accuracy: 0.8918786692759295
Test accuracy: 0.4207436399217221
Dev accuracy: 0.44591287322564854
```

3.2 Model 2 - Best Model

```
result_train =  np.array([p_train2[i].argmax() for i in range(p_train2.shape[0])])
result_test = np.array([p_test2[i].argmax() for i in range(p_test2.shape[0])])
result_dev = np.array([p_dev2[i].argmax() for i in range(p_dev2.shape[0])])

train_result = np.equal(train, result_train)
test_result = np.equal(test, result_test)
dev_result = np.equal(dev, result_dev)

print("Train accuracy: " + str(np.average(train_result)))
print("Test accuracy: " + str(np.average(test_result)))
print("Dev accuracy: " + str(np.average(dev_result)))
```

```
Train accuracy: 0.5758317025440313
Test accuracy: 0.4344422700587084
Dev accuracy: 0.4522760646108664
```

3.3 Model 3 - 2nd Best Model

```
result_train =  np.array([p_train3[i].argmax() for i in range(p_train3.shape[0])])
result_test = np.array([p_test3[i].argmax() for i in range(p_test3.shape[0])])
result_dev = np.array([p_dev3[i].argmax() for i in range(p_dev3.shape[0])])

train_result = np.equal(train, result_train)
test_result = np.equal(test, result_test)
dev_result = np.equal(dev, result_dev)

print("Train accuracy: " + str(np.average(train_result)))
print("Test accuracy: " + str(np.average(test_result)))
print("Dev accuracy: " + str(np.average(dev_result)))
```

```
Train accuracy: 0.48238747553816047
Test accuracy: 0.4691780821917808
Dev accuracy: 0.4845814977973568
```

3.4 Model 4 - Dropout

```
result_train =  np.array([p_train4[i].argmax() for i in range(p_train4.shape[0])])
result_test = np.array([p_test4[i].argmax() for i in range(p_test4.shape[0])])
result_dev = np.array([p_dev4[i].argmax() for i in range(p_dev4.shape[0])])

train_result = np.equal(train, result_train)
test_result = np.equal(test, result_test)
dev_result = np.equal(dev, result_dev)

print("Train accuracy: " + str(np.average(train_result)))
print("Test accuracy: " + str(np.average(test_result)))
print("Dev accuracy: " + str(np.average(dev_result)))
```

```
Train accuracy: 0.38454011741682975
Test accuracy: 0.3703522504892368
Dev accuracy: 0.3607440039158101
```

3.5 Model 5 - Batch Normalization

```
result_train =  np.array([p_train5[i].argmax() for i in range(p_train5.shape[0])])
result_test = np.array([p_test5[i].argmax() for i in range(p_test5.shape[0])])
result_dev = np.array([p_dev5[i].argmax() for i in range(p_dev5.shape[0])])

train_result = np.equal(train, result_train)
test_result = np.equal(test, result_test)
dev_result = np.equal(dev, result_dev)

print("Train accuracy: " + str(np.average(train_result)))
print("Test accuracy: " + str(np.average(test_result)))
print("Dev accuracy: " + str(np.average(dev_result)))
```

```
Train accuracy: 0.313600782778865
Test accuracy: 0.32093933463796476
Dev accuracy: 0.3176700930004895
```

3.6 Ensemble

```
train_voting = np.array([train1, train2, train3, train4, train5])
test_voting = np.array([test1, test2, test3, test4, test5])
dev_voting = np.array([dev1, dev2, dev3, dev4, dev5])

final_train = voting_calc(train_voting)
final_test = voting_calc(test_voting)
final_dev = voting_calc(dev_voting)

print("Train accuracy: " + str(np.mean(np.equal(final_train, result_train))))
print("Test accuracy: " + str(np.mean(np.equal(final_test, result_test))))
print("Dev accuracy: " + str(np.mean(np.equal(final_dev, result_dev))))
```

```
Train accuracy: 0.6643835616438356
Test accuracy: 0.6453033268101761
Dev accuracy: 0.6441507586882036
```

4 Conclusions

Working with this assignment has been quite a challenge, since the dataset didn´t give us a way to easily train our model. We have spent much time figuring out how we could improve the test results we were obtaining because they were quite low (around 0.45). This is the reason why we decided to launch a big amount of models and analyse them using TensorFlow, we wanted to choose the best hyperparameters we could.

The use of Tensorboard was really helpful this purpose. We were able to see also how the weights and other elements of the network evolved over time and this was quite interesting. We have kept record of all the logs and models in case they were needed to complete this work.

Regarding the low test accuracy we obtained in the models tried, we think this can be due to the fact that the dataset is small and there are 10 different labels. Also the dataset could had been randomized according to the description and this can be another cause of the poor results and the early overfitting.

With our approach we have tried to use the different methods we have studied in class, nevertheless there are more combinations we didn´t explore and that could have lead us to better results like decreasing progressively the learning rate, tuning more the initialization parameters of the weights and bias, trying CNN approach by training layers in order, using other approaches like Adanet to automatically train ensembles or retaining always the best model of each run.

We decided to optimize time and this is why we did not try the methods above mentioned, we wanted to focus in have an overview of how the different models behaved because we thought this would help us to find the best model (instead of tuning concrete hyperparameters or automatic methods), we also wanted to learn about the use of tensorboard.

Finally, we could see in this approach the power of the ensemble methods, particularly of the voting system method, with which we were able to come up with a relatively better model coming from average ones.