ARGOS project, image classification

Homework 2 - Machine Learning

Mariani Matteo
Matricola ID: 1815188

Contents

Introduction

The **ARGOS system** (Automatic Remote Grand Canal Observation System) has the goal to control the boats' traffic in Venice's Grand Canal, using 13 observation posts (Survey Cells) distribuited along a waterway of about 6 km lenght and 80-150 meters width.
The system should be a deterrent with respect to driver's bad behavior, monitoring the boats' speed with respect to the limit imposed in the laguna or guaranteeing that some type of boats are allowed to circulate in a particular section of the canal.
Considering this scenario, there are a lot of problem to handle like:

- gradual illumination changes;
- motion changes;
- high frequency noise;
- changes in the background geometry (parked boats);
- classification of different type of boats.

For this homework, our scope is limited to the *classification task*: given different screenshots taken by the cells, classify correctly the boats that appear in them. Before analyzing the technologies used, it's better to describe a little bit the ARGOS system's database.

Structure of the dataset

The *MarCDT dataset* [1] is already divided into a training set of 4774 images and a test set of 1969 images. The training set contains images of 24 different categories of boats - the folder "Water" contains false positives. Each of them can be further aggregated into 5 general classes (figure 1). A file containing the pairs $< image-name >; < category >$ is also provided to describe in details the screenshots in the test set. Note that we are interested in the boats' classification into 24 categories.

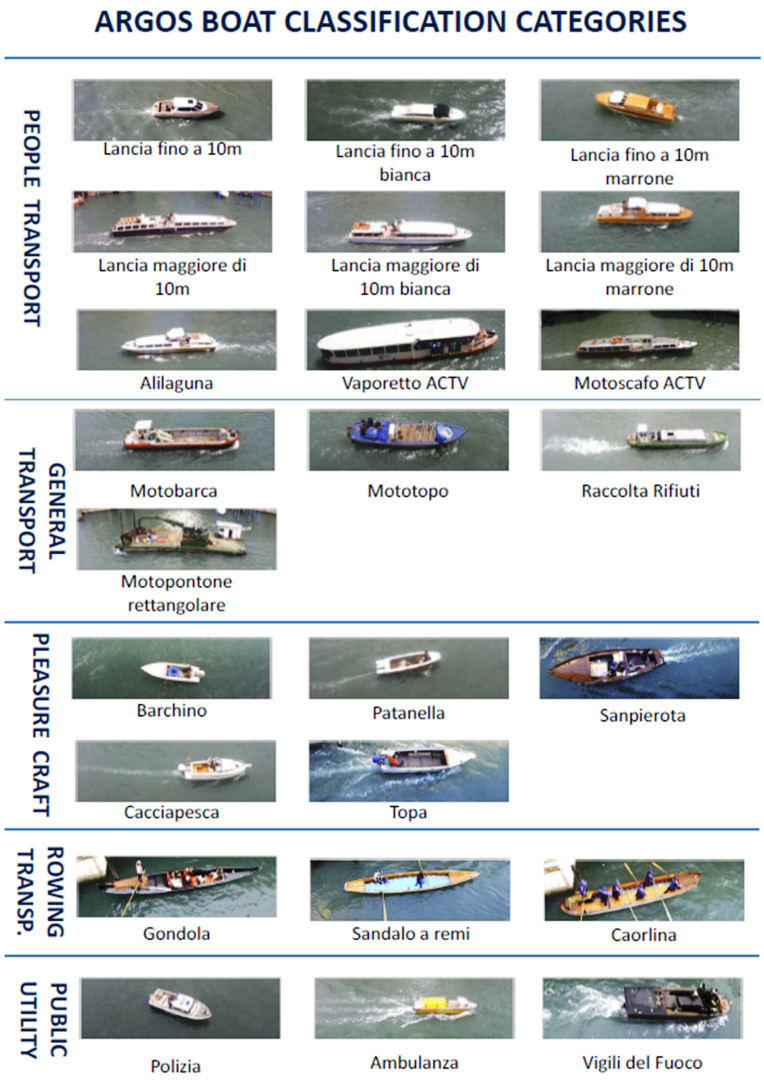

Figure 1: 24 categories of boats divided into 5 general groups

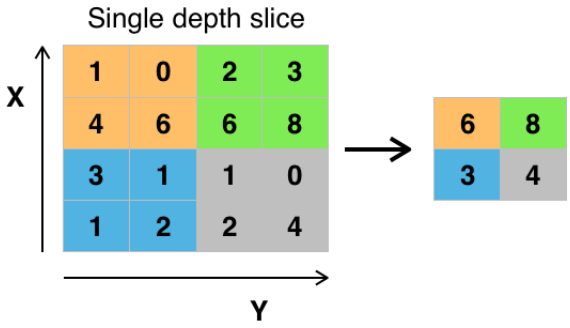

Figure 2: Max pooling with a 2x2 filter and stride=2

Technologies used

The programming language used for the image classification is Python, that guarantee the access to one of the most important open-source library for dataflow programming: *Tensorflow* [2]. This library is also used for machine learning applications such as neural networks. In particular, for this homework is used a *Convolutional Neural Network* (CNN).

A brief inspection into CNNs

Convolutional neural networks (CNNs) are the current state-of-the-art model architecture for image classification tasks. CNNs apply a series of filters to the raw pixel data of an image to extract and learn higher-level features, which the model can then use for classification. CNNs contains three components:

- **Convolutional layers**, which apply a specified number of convolution filters to the image. For each subregion, the layer performs a set of mathematical operations to produce a single value in the output feature map. Convolutional layers then typically apply a ReLU activation function to the output to introduce nonlinearities into the model. ReLU is preferable to other functions, because it trains the neural network several times faster without a significant penalty to accuracy. [3]
- **Pooling layers**, which downsample the image data extracted by the convolutional layers to reduce the dimensionality of the feature map in order to decrease processing time. A commonly used pooling algorithm is *max pooling*, which extracts subregions of the feature map (for example, 2x2 mask), keeps their maximum value, and discards all other values (figure 2). Another example is *average pooling*, which uses the average value from each of a cluster of neurons at the prior layer,
- **Fully connected (or dense) layers**, which perform classification on the features extracted by the convolutional layers and downsampled by the pooling layers. In a dense layer, every node in the layer is connected to every node in the preceding layer.

Typically, a CNN is composed of a stack of convolutional modules that perform feature extraction. Each module consists of a convolutional layer followed by a pooling layer. The last convolutional module is followed by one or more dense layers that perform classification. The final dense layer in a CNN contains a single node for each target class in the model (all the possible classes the model may predict), with a *softmax activation function* to generate a value between 0 and 1 for each node (the sum of all these softmax values is equal to 1). We can interpret the softmax values for a given image as relative measurements of how likely it is that the image falls into each target class.

Implementation

In the current and the next section, portions of the code will be commented using (1) screenshots and (2) some graphs/images produced by Tensorboard; a suite of visualization tools created to make easier to understand Tensorflow programs.
First of all, we have to build the network itself. In the homework have been used:

- *Convolutional Layer 1*, that applies 32 5x5 filters on the original images (so, it extracts 32 features) using a ReLU activation function. Note that, before that the original images of the training set are manipulated, their shape has been modified; reducing their dimension by a 10 factor [1] and transform them in grayscale images to avoid a huge computational power required [2];
- *Pooling Layer 1*, that performs max pooling with a 2x2 filter and stride of 1 (so, basically, it halves the dimension of the image in input);
- *Convolutional Layer 2*, that applies 64 5x5 filters on the previous output (extracting 64 features) using again a ReLU activation function.
- *Pooling Layer 2*, same as previous pooling layer (now the orignal image is reduced by $\frac{1}{4}$);
- *Fully Connected Layer 1*, that perform classification on the features extracted by convolution/pooling layers; using 1,024 neurons, with *dropout* regularization rate of 0.5 (probability of 0.5 that any given element will be dropped during training) to avoid overfitting;
- *Fully Connected Layer 2*. This is the final layer of the network that will return the values for our prediction. It's a dense layer with 24 neurons, one for each digit target class.

After that the structure of the network has been defined, we have to introduce a *loss function*, a function that measures how closely the model's predictions match the target classes. For multiclass classification problems, cross entropy is tipically used as loss metric. In the same pass, the softmax activation function is applied on the output of the last dense layer.
The last thing to do is to introduce an *optimizer* for the loss function. There exist several optimizer offered in the Tensorflow library such as the *Gradient Descent Optimizer* [3]. For this case, it's used the *Adam Optimizer* [4], that is more sophisticated.
The following image shows a visual structure of the CNN (figure 3).

[1] the original images were 800x240 pixel, now they are 80x24.
[2] The PC used for the training has an Intel Core Duo and no dedicated GPU.
[3] Gradient descent is a first-order iterative optimization algorithm for finding the minimum of a function. To find a local minimum of a function using gradient descent, one takes steps proportional to the negative of the gradient (or of the approximate gradient) of the function at the current point.
[4] Adam is an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments. The method is straightforward to implement, is computationally efficient, has little memory requirements, is invariant to diagonal rescaling of the gradients, and is well suited for problems that are large in terms of data and/or parameters.

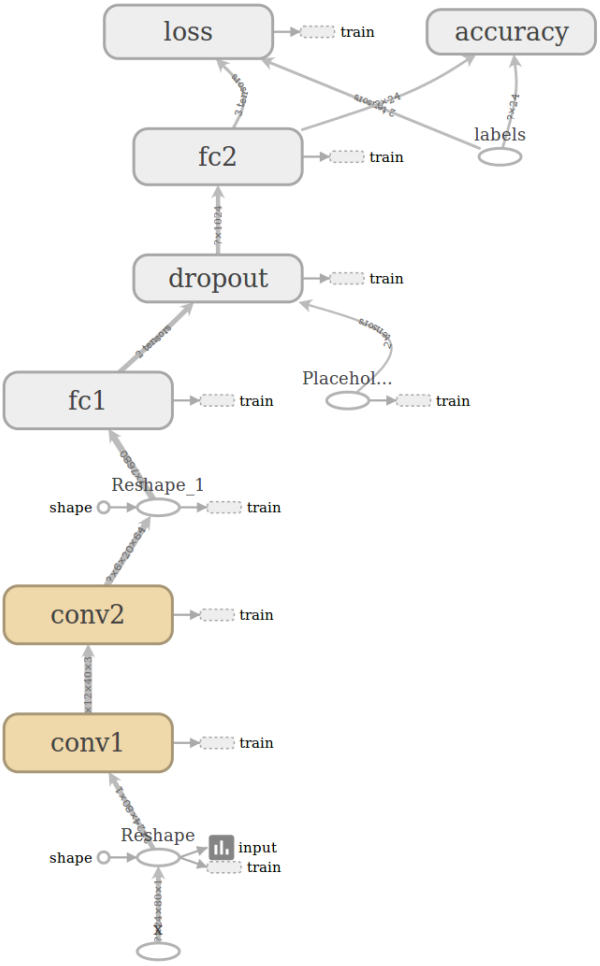

Figure 3: Visualization of the CNN's layers through Tensorboard

building CNN in code

```
# Functions for setting up the convolutional and max-pooling layers
def conv2d(x,W):
    return tf.nn.conv2d(x, W, strides=[1,1,1,1], padding='SAME')
def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1,2,2,1], strides=[1,2,2,1], padding='SAME')

#Network building
x = tf.placeholder(tf.float32, shape=[None, 24, 80, 1])
t = tf.placeholder(tf.float32, shape=[None, 24])
W = tf.Variable(tf.zeros([24*80,24]))
b = tf.Variable(tf.zeros([24]))

'''
From here we start defining the CNN:
- first convolutional layer
- second convolutional layer
-
'''
# Reshape data to original shape
x_image = tf.reshape(x, [-1, 24, 80, 1])
tf.summary.image('input_image', x_image)

# Define shape of the first layer parameters
#(Filter tensor: `[height, width, in_channels, out_channels]`)
with tf.name_scope('conv1'):
     W_conv1 = weight_variable([5, 5, 1, 32])
     b_conv1 = bias_variable([32])

     # Define first convolutional layer
     #ReLU then sets all negative values in the matrix in output to the conv2d to zero
     #while all other values are kept constant.
     h_conv1 = tf.nn.relu(conv2d(x_image,W_conv1)+b_conv1)
     h_pool1 = max_pool_2x2(h_conv1)

# Define shape of the second layer parameters
with tf.name_scope('conv2'):
     W_conv2 = weight_variable([5, 5, 32, 64])
     b_conv2 = bias_variable([64])

     # Define second convolutional layer
     h_conv2 = tf.nn.relu(conv2d(h_pool1,W_conv2)+b_conv2)
     h_pool2 = max_pool_2x2(h_conv2)

'''
Define first fully connected layer ($1024$ units)
NOTE: 6x20 is the size of the pooled image at the end of the second
convolutional layer
'''
with tf.name_scope('fc1'):
     W_fc1 = weight_variable([6*20*64, 1024])
     b_fc1 = bias_variable([1024])

     h_pool2_flat = tf.reshape(h_pool2, [-1, 6*20*64])
     h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

# Apply dropout at fc1
keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
```

```
# Define second fully connected layer
with tf.name_scope('fc2'):
     W_fc2 = weight_variable([1024,24])
     b_fc2 = bias_variable([24])

y_conv = tf.matmul(h_fc1_drop,W_fc2) + b_fc2

# Define loss function based on softmax
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=t, logits=y_conv))

# Use ADAM Optimizer to minimize the loss
optimizer = tf.train.AdamOptimizer(1e-4)
#optimizer = tf.train.GradientDescentOptimizer(0.01)
train_step = optimizer.minimize(loss)
```

Once all the parameters and the structure of the CNN are defined, we can finally start the training process. Note that, before the learning, both training set and testing set are preprocessed; collecting the data in some useful data structures. In particular, from each set are produced two lists:

- *filename_list*; that contains the path of each images in the dataset;
- *label_list*; that contains the label of each images in the dataset.

A label is a vector of 24 elements (representing the 24 classes that we are considering), where 23 elements are zeros and only one is 1. For example, if the i-th element of *filename_list* is the path of an image representing a Vaporetto, then the same i-th element of *label_list* will have the cell of the array corrispondent to the Vaporetto class equal to 1.
For the training set, the relationship between labels and images is trivially collected using the name of the folder containing the images. For the testing set, instead, the relationship is given by *ground_truth.txt* file.

preprocessing in the code

```
#function for the training set
def images_label(path):
    labels_name = iterate_dir(path) #Taking boat names to build labels
    filename_list = list()
    label_list = list()
    class_id = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
    index = -1
    for class_name in labels_name:
        current_id = class_id[:]
        index += 1
        current_id[index] += 1
        img_list = os.listdir(path + '/' + class_name)
        for img in img_list:
            filename_list.append(path + '/' + class_name + '/' + img)
            label_list.append(current_id)
    return filename_list, label_list
```

```
#function for the testing set
def parsing_groundtruth(path):
     boat_label = dict()
     labels_name = iterate_dir('sc5')
     i = 0
     vec = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
     for e in labels_name:
          v = vec[:]
          v[i] = 1
          boat_label[e] = v
          i = i+1
     test_images = []
     test_labels = []
     file = open(path+'/ground_truth.txt', 'r')
     lines = file.readlines()
     for l in lines:
          l = l.replace(":", "")
          l = l.replace(" ", "").strip()
          element = l.split(';')
          if((element[1] != 'SnapshotBarcaParziale') and
             (element[1] != 'SnapshotBarcaMultipla')):
               if(element[1] == "SnapshotAcqua"):
                    test_images.append(path + '/' + element[0])
                    test_labels.append(boat_label['Water'])
               elif(element[1] == 'Mototopocorto'):
                    test_images.append(path + '/' + element[0])
                    test_labels.append(boat_label['Mototopo'])
               else:
                    test_images.append(path + '/' + element[0])
                    test_labels.append(boat_label[element[1]])
     return test_images, test_labels
```

Results

For this homework has been used a PC with an Intel Core Duo 2GHz and no dedicated GPU, so there were some tweaks to allow a proper training without requiring a lot of computational resources. First of all, the training - and the consequent evaluation - runned using 1000 iterations and batches of 100 images in each iteration. Each image in input was downscaled from 800x240 RGB to 80x24 grayscale image.

Thanks to Tensorboard, in each training run we have access to a lot of interesting graphs. Some of them are samples of how good the model is after the training phase - like the graphs of the accuracy and the loss calculated on the 100 batches per iteration (figure 4) [5] -, others instead try to give some insight on how the CNN worked in the training phase. In particular, the following figure (5) describes how the images change, starting from the downscaled 80x24 grayscale images and ending up to images produced by the last convolutional and pooling layer (which features are put as input in the fully connected layers).

[5] Note that, after each run of training/evaluation we have access to two kind of accuracy: (1) the one calculated in the training phase by Tensorflow on the same batches in which the CNN is actually learning - so it can be seen like an index of how the model is effectively learning - and of course (2) the accuracy that is calculated on the test set.

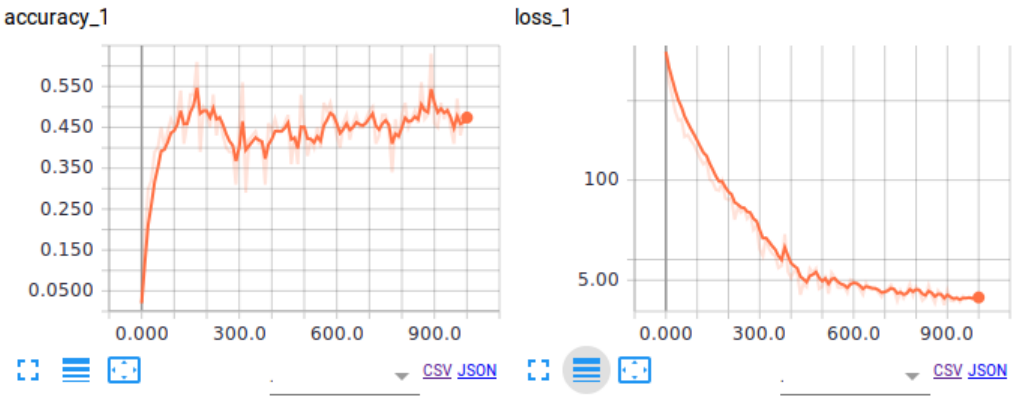

Figure 4: Accuracy and loss graphs based on the batches and calculated through Tensorboard, using 1000 iterations with Adam Optimizer (epsilon=1e-4). Multiclass classification (on all classes)

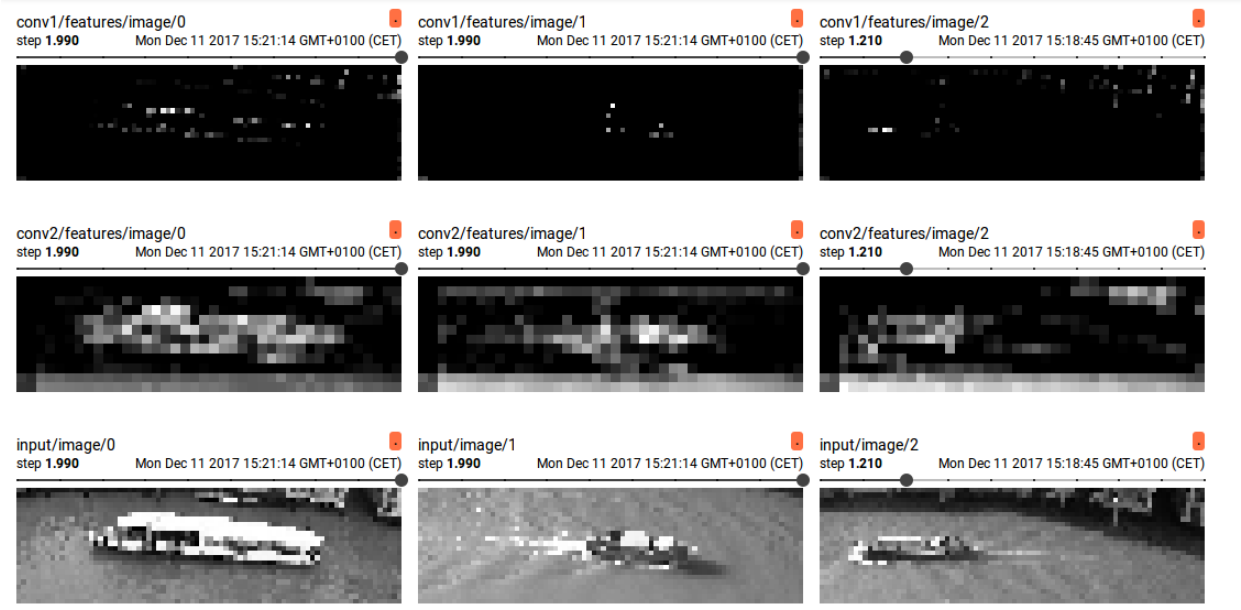

Figure 5: Evolution of the input images through the CNN layers

The comparison was done by modifying some parameters and then evaluating the accuracy. Some relevant parameters in our scenario were:

- number of considered classes (all the 24 classes or different subsets);
- number of considered images' channels (1 channel for grayscale images, 3 for RGB images);
- dynamic learning rate epsilon of the Adam optimizer;
- number of convolutional and pooling layer.

Some others parameters like (1) the number of cosidered batches in each iteration, (2) the number of iterations in the training phase and (3) other optimizers (like the Gradient Descent) are not considered in this section because the results were not so different from the one obtained modifying the relevant ones.

First of all let's consider the following training runs on all the classes, using only 1 channel and different dynamic learning rate of the Adam Optimizer:

- $epsilon = 1e^{-4}$, 700 steps. Accuracy based on training batches: 0.55. Accuracy based on test set: 0.52;
- $epsilon = 1e^{-1}$, 700 steps. Accuracy based on training batches: 0.198. Accuracy based on test set: 0.193;
- $epsilon = 1e^{-7}$, 700 steps. Accuracy based on training batches: 0.027. Accuracy based on test set: 0.015.

The first run is clearly the best, reaching a moderate accuracy on the multiclass classification problem. The other two runs, instead, are awful. The second run (with $1e^{-1}$) doesn't perform well because the greater dynamic learning rate doesn't allow to the optimizer to converge. The third run, instead, hypothetically speaking could achieve a better performance (because it has a smaller learning rate) but the number of steps are not enough to achieve the point of minimum loss. For these evalutions, graphs produced on the accuracy on training batches are visible in the figure 6.

The following evalution was done by removing some classes:

- 24 classes (all), 700 steps. Accuracy based on training batches: 0.54. Accuracy based on test set: 0.52;
- 14 classes (removing the classes with <15 samples in the training set), 700 steps. Accuracy based on training batches: 0.56. Accuracy based on test set: 0.55;
- 13 classes (same as before but removing also the Water class), 700 steps. Accuracy based on training batches: 0.48. Accuracy based on test set: 0.4.

Not surprisingly, removing the classes with few samples (slightly) benefits the training. The interesting part lies in the third run: removing the Water class, we lose a lot of accuracy because that particular class helps the model to learn the false positive cases. So, it should be considered.

The next comparison involved input images with 1 channel (grayscale) and 3 channels (RGB):

- 1 channel (all classes), 700 steps. Accuracy based on training batches: 0.55. Accuracy based on test set: 0.52;
- 3 channels (all classes), 1000 steps. Accuracy based on training batches: 0.54. Accuracy based on test set: 0.47.

In this case, surprisingly, we get that the accuracy is better on 1 channel w.r.t. 3 channels. This is the only case in which I personally thought that the accuracy would have been better on the other case. Maybe too many channels creates some kind of noise that reduce the performance of the model.

Finally, the last comparison was between a CNN with 2 convolutional layers and 2 pooling layers and a CNN with 3 layers each:

- 2 conv + 2 pool, 700 steps. Accuracy based on training batches: 0.55. Accuracy based on test set: 0.52;
- 3 conv + 3 pool, 700 steps. Accuracy based on training batches: 0.56. Accuracy based on test set: 0.54.

The boost obtained by adding a new double layer is very minimal, but still recognizable. The output of the third pooling layer is a very small image ($\frac{1}{8}$ of the input image).

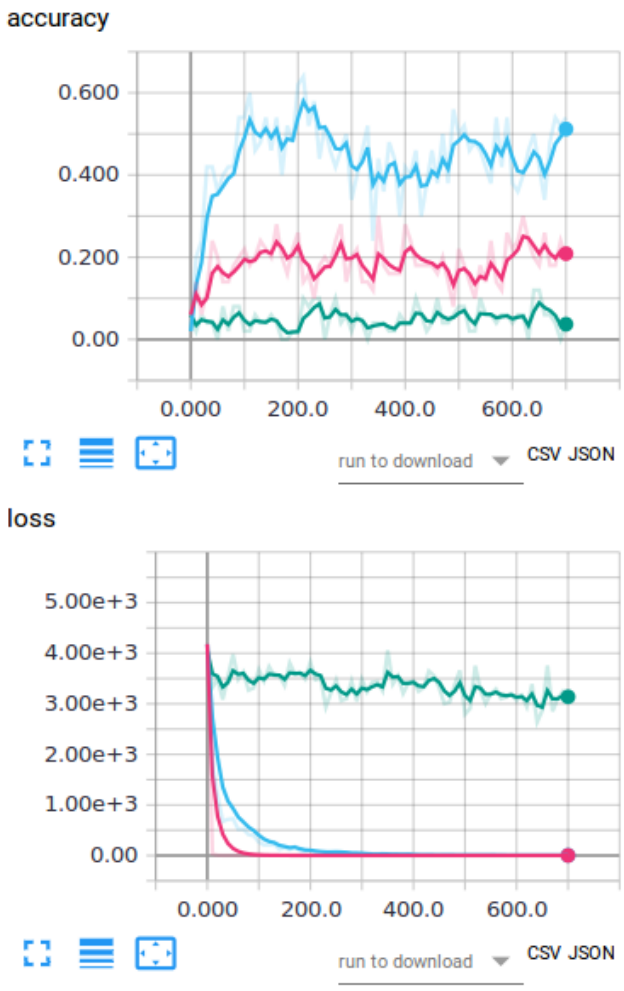

Figure 6: Accuracy and loss (based on the batches) registered during different learning sessions on 700 steps on all the classes (24). In particular, they differ from the parameter epsilon of the Adam Optimizer: blue trend has epsilon equal to 1e-4, pink one has 1e-1 and green one 1e-7

References

[1] *The MarCDT datasest.* 2013. [http://www.dis.uniroma1.it/ labrococo/MAR/classification.html]

[2] *Tensorflow.* [https://www.tensorflow.org]

[3] *Convolutional Neural Network.* [Wikipedia]