REPORT HOMEWORK 2

The homework is to solve an image classification problem by using two different approaches with neural networks.

First, I needed to choose a dataset. I decided to choose a balanced dataset (about 400 images for class) with 10 classes. It is about architectural heritage elements. Precisely, there are 4128 images in total.

In both the two approaches, I decided to organize my work in the following way:

- Preprocessing
- Neural network
- Train
- Evaluation

APPROACH 1

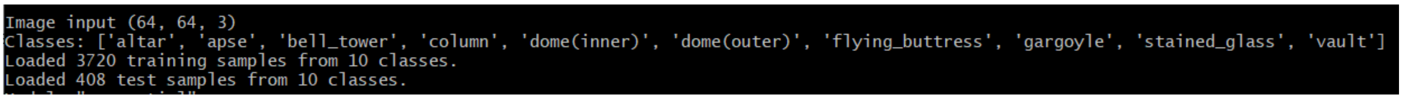
In the first approach I did some preprocessing on train data and test data.

What I did is to initialize two elements (train_datagen and test_datagen) that apply some random modifications to the images of the dataset. For example, I rescale the images (normalize information between 0 and 1), apply a zoom with a value between 0 and 0.2 and flip the image in horizontal.

It means that, before to go in input to the convolutional neural network, the images are modified with these random modifications. In this way, I reduce the possibility to have the problem of overfitting since the images that go in input to the cnn are not exactly the same which are in the dataset.

Another thing I did in the preprocessing is to specify that I want to use 10% of the dataset for testing and the 90% for training. furthermore, the size of the target is (64, 64) (since the shape of the images in the dataset is (64, 64)) and the batch size is 64. This means that there will be 59 batches that contains all the images for training (the whole upper part of 3720 / 64 is 59).

In fact, in my case I have 3720 images for training and 408 for testing, with the images in input that have a shape of (64, 64, 3)

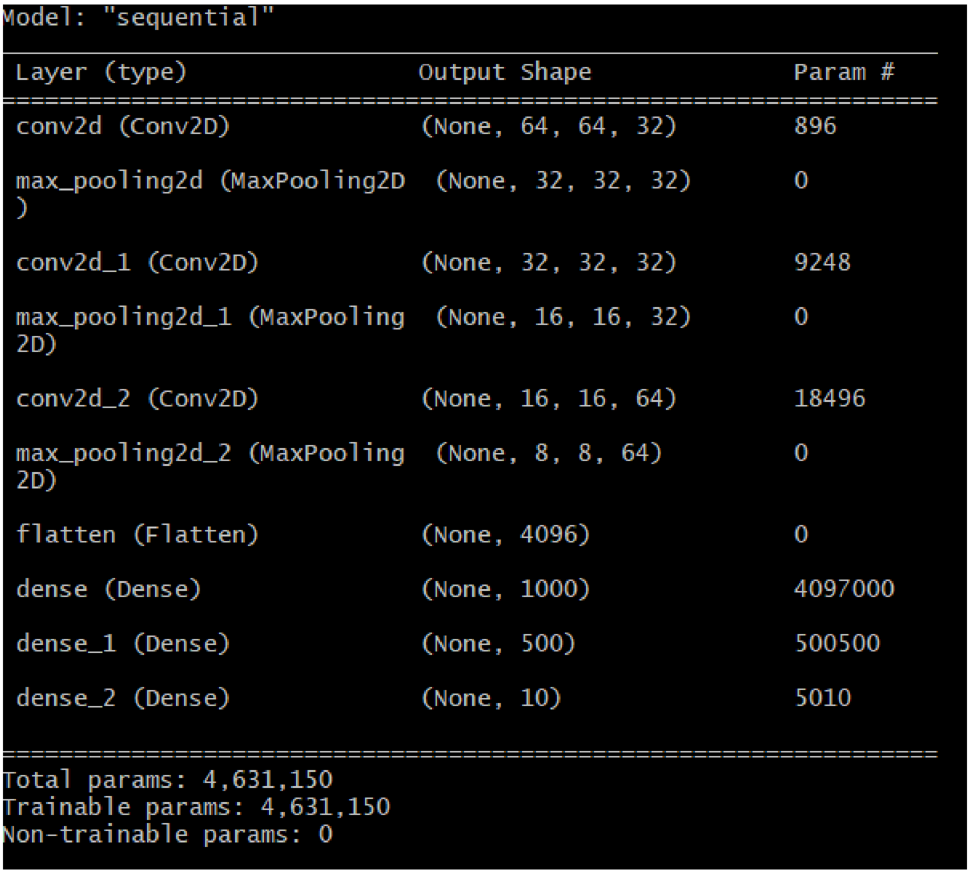


```
Image input (64, 64, 3)
Classes: ['altar', 'apse', 'bell_tower', 'column', 'dome(inner)', 'dome(outer)', 'flying_buttress', 'gargoyle', 'stained_glass', 'vault']
Loaded 3720 training samples from 10 classes.
Loaded 408 test samples from 10 classes.
```

After the preprocessing, I started with the composition of the convolutional neural network.

In this case, it has 3 convolutional layers, 3 maxpooling layers and 3 dense (fully connected) layers.

The following is the summary of the network



```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 64, 64, 32)        896

 max_pooling2d (MaxPooling2D  (None, 32, 32, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 32, 32, 32)        9248

 max_pooling2d_1 (MaxPooling  (None, 16, 16, 32)       0
 2D)

 conv2d_2 (Conv2D)           (None, 16, 16, 64)        18496

 max_pooling2d_2 (MaxPooling  (None, 8, 8, 64)         0
 2D)

 flatten (Flatten)           (None, 4096)              0

 dense (Dense)               (None, 1000)              4097000

 dense_1 (Dense)             (None, 500)               500500

 dense_2 (Dense)             (None, 10)                5010

=================================================================
Total params: 4,631,150
Trainable params: 4,631,150
Non-trainable params: 0
```

As input of each convolution layer, there are the number of kernels, the kernel size, the strides and the activation function.

We know that the kernel is used to “scan” all the input in order to produce an output.

In the convolutional layers of my neural network, the kernel has size (3, 3) and the strides is (1, 1). It means that the kernel will “scan” the input by shifting of one unit in the two directions.

The activation function is the relu. We know that relu(x) = max(0, x) and that an activation function decides if a neuron should be activated or not. In other words, it decides if the input of a neuron is important for the network.

Then there are the maxpooling layers. These ones downsamples the input along its dimensions by taking the maximum value over an input window for each channel of the input. The window is shifted along each dimension. In the cnn, the pool size is (2, 2). It means that the maximum value will be taken over a 2x2 window. The strides is (1, 1). This means that the window moves one unit in the two directions.

Before to give the input to the dense layers, I have to flatten it. So, there is a flatten layer between the convolutional layers and the dense ones. This flatten layer modifies the shape of the input from multiple dimensions to one dimension.

Then there are the dense layers that correspond to the fully connected layers.

The first one has 1000 units and the regularizer l2 applied to the kernel. The second one has 500 units and the last one has 10 (which is the number of classes) units with the softmax as activation function.

We know that the softmax function takes as input a vector of k elements and returns k probabilities proportional to the exponentials of the inputs.

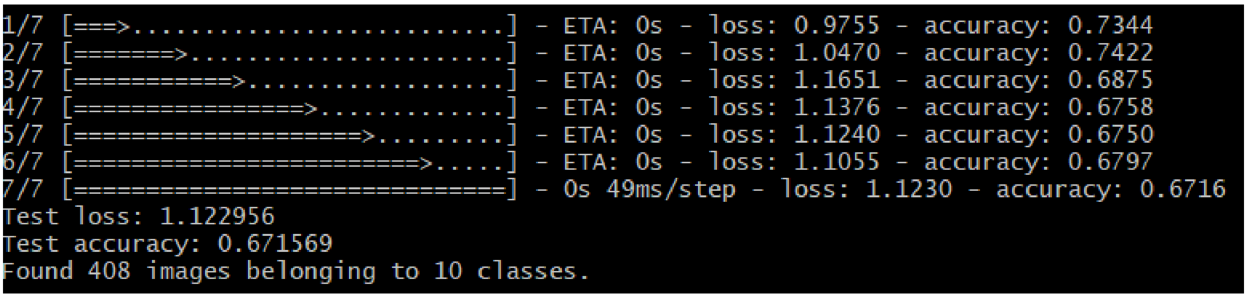
All the above is the description of the convolutional neural network.

Now I compiled it with the Adam optimizer (the optimizer is the process that will execute the actual training), the loss function that is the categorical cross entropy and the accuracy.

After that, I started with the train. I trained the model with the train_generator (the element that generate images from the dataset but with some random modifications as specified in the preprocessing) and the number of epochs that is 20. I know that, in order to improve the performances, I can increase the number of epochs.

Finally, I evaluated the model. To do the evaluation of the network, I used the accuracy and the loss both on train and test data, the precision, the recall and the f1-score.

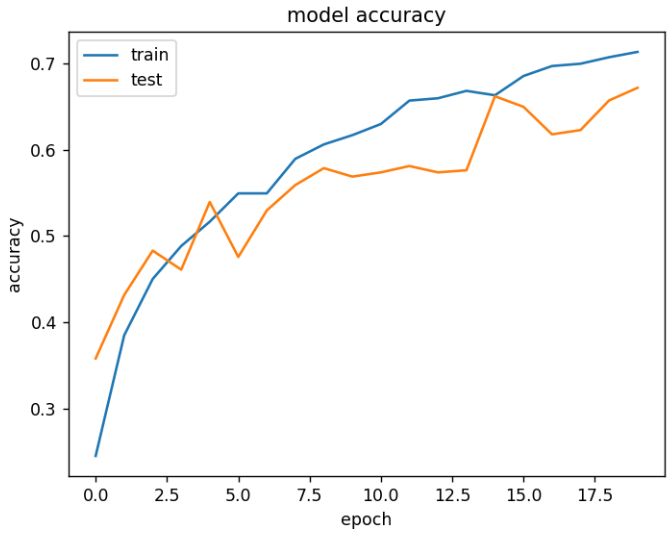
About the accuracy and the loss on the test data, the results are the following ones



```
1/7 [===>..........................] - ETA: 0s - loss: 0.9755 - accuracy: 0.7344
2/7 [=======>......................] - ETA: 0s - loss: 1.0470 - accuracy: 0.7422
3/7 [===========>..................] - ETA: 0s - loss: 1.1651 - accuracy: 0.6875
4/7 [================>.............] - ETA: 0s - loss: 1.1376 - accuracy: 0.6758
5/7 [====================>.........] - ETA: 0s - loss: 1.1240 - accuracy: 0.6750
6/7 [========================>.....] - ETA: 0s - loss: 1.1055 - accuracy: 0.6797
7/7 [==============================] - 0s 49ms/step - loss: 1.1230 - accuracy: 0.6716
Test loss: 1.122956
Test accuracy: 0.671569
Found 408 images belonging to 10 classes.
```

So, there is a loss of 1.1 and an accuracy of 67%.

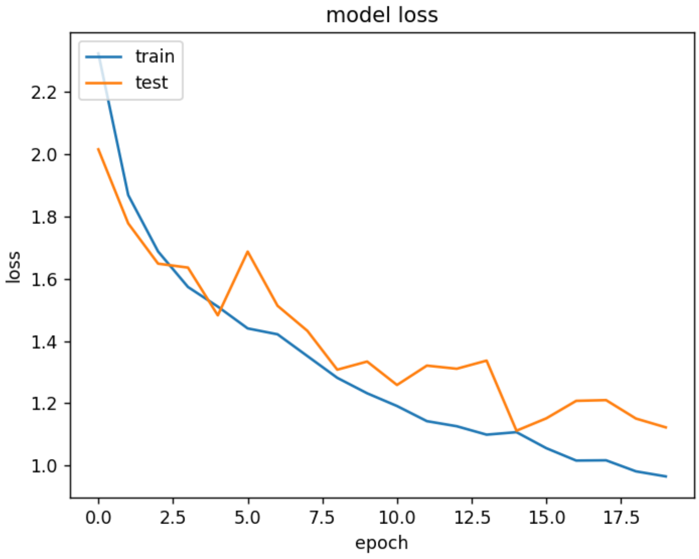
The following one is the representation of the accuracy trend both on train data and test data



We can notice that the accuracy always increases during the train phase, while it decreases in some time intervals but it is generally crescent also during the test phase. This behaviour is usual during the test phase.

It is important that the accuracy always increases since it means that the model is learning with the passing of the epochs.

The following one is the representation of the loss trend both on train data and test data



We can notice that, in this case, the loss always decreases during the train phase, while it increases in some time intervals but it is generally decrescent also during the test phase. This behaviour is usual during the test phase.

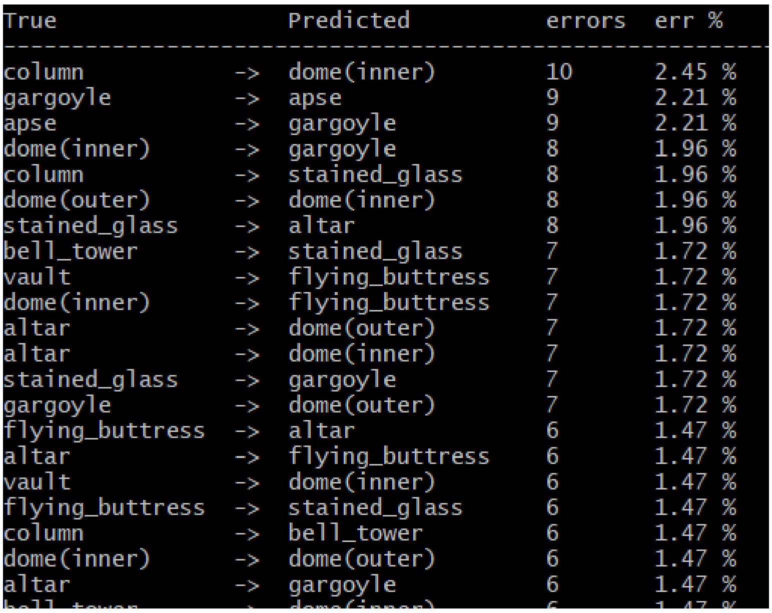
It is important that the loss always decreases since it means that the model is learning with the passing of the epochs.

Other metrics used to evaluate the network are precision, recall and f1-score.

I have 408 samples for the support and about 40 samples for each class. We know that the most important metric is the f1-score since it is a weighted average between precision and recall. The closer to 1 it is, the better is.

In my case I obtained a wighted avg f1-score of 0.59, which is quite good. It means that more than half of the support samples have been well predicted.

At the end, I print also the most common mistakes. Here I show only some of them



```
True                 Predicted          errors  err %
---------------------------------------------------------
column           ->  dome(inner)        10      2.45 %
gargoyle         ->  apse               9       2.21 %
apse             ->  gargoyle           9       2.21 %
dome(inner)      ->  gargoyle           8       1.96 %
column           ->  stained_glass      8       1.96 %
dome(outer)      ->  dome(inner)        8       1.96 %
stained_glass    ->  altar              8       1.96 %
bell_tower       ->  stained_glass      7       1.72 %
vault            ->  flying_buttress    7       1.72 %
dome(inner)      ->  flying_buttress    7       1.72 %
altar            ->  dome(outer)        7       1.72 %
altar            ->  dome(inner)        7       1.72 %
stained_glass    ->  gargoyle           7       1.72 %
gargoyle         ->  dome(outer)        7       1.72 %
flying_buttress  ->  altar              6       1.47 %
altar            ->  flying_buttress    6       1.47 %
vault            ->  dome(inner)        6       1.47 %
flying_buttress  ->  stained_glass      6       1.47 %
column           ->  bell_tower         6       1.47 %
dome(inner)      ->  dome(outer)        6       1.47 %
altar            ->  gargoyle           6       1.47 %
bell_tower       ->  dome(inner)        6       1.47 %
```

To finish this first approach, let’s try different values for one of the most important hyperparameters of a neural network: the learning rate.

With a learning rate of 0.0001, I notice that the convergence of the validation loss is very slowly and with a learning rate of 0.001 it is more rapidly, but the final validation loss value is higher than the previous one. Therefore, it means that the optimal value for the learning rate is between 0.0001 and 0.001. hence, if I try with 0.00012, then the convergence of the validation loss in quite rapid and its value is close to the minimum.

To summarize:

- Learning rate 0.0001: the convergence of the validation loss is slowly (1.1988 at epoch 9) but its final value is good (0.9474)
- Learning rate 0.001: the convergence of the validation loss is more rapid (1.1591 at epoch 9), but its final value is higher than the previous one (1.2874)
- Learning rate of 0.00012: the convergence of the validation loss is rapid and its final value is close to the minimum I measured (0.9387)

APPROACH 2

In the second approach I changed the architecture of the network, the optimizer, the regularizer, some hyperparameters and something of preprocessing.

Starting from the preprocessing, I set the batch size to 32. It means that there will be 104 batches that contains all the images for training (the whole upper part of 3306 / 32 is 104).

In fact, I also changed the percentage of test data from 10% to 20%. The train data are the 80% of the dataset. So, there are 3306 images for training and 822 images for testing.

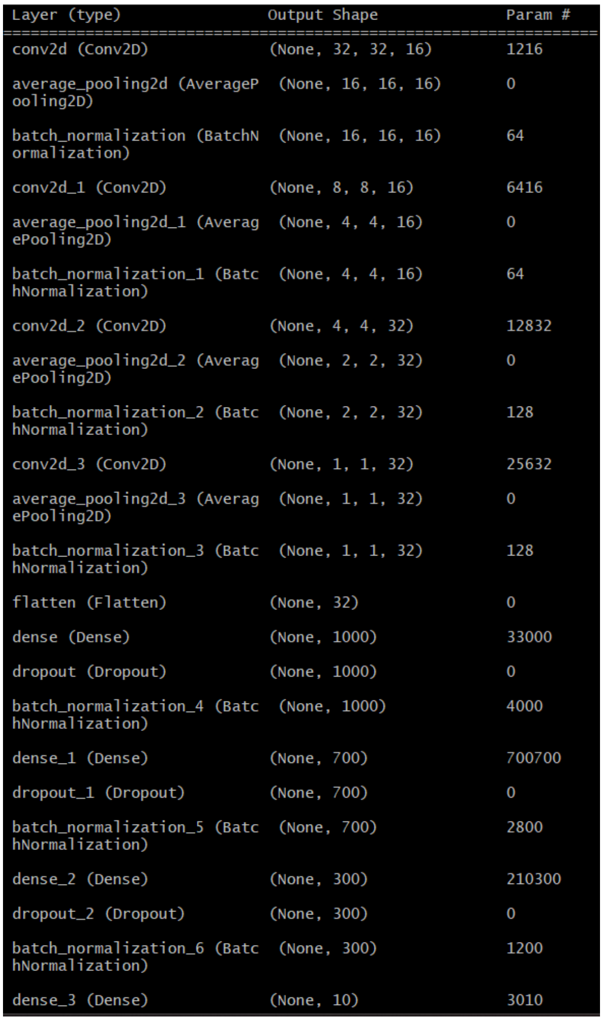
Since the batch size is smaller than the previous approach, I expect that the loss function decreases more rapidly. I will see if it happens during the evaluation.

Also in this case, we have the test_datagen and the train_datagen that put some random modifications on the images in input to the network.

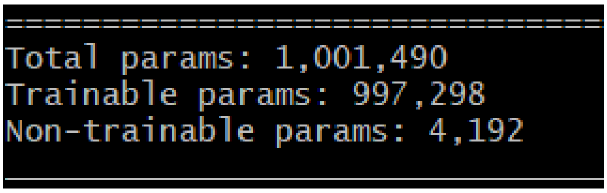
Then there is again the convolutional neural network.

In this case, it has 4 convolutional layers, 4 average pooling layers and 4 dense (fully connected) layers. There is also a batch normalization after each convolutional and dense layer and a dropout after each dense layer (except for the last one).

The following is the summary of the network



```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 32, 32, 16)        1216

average_pooling2d (AverageP  (None, 16, 16, 16)        0
ooling2D)

batch_normalization (BatchN  (None, 16, 16, 16)        64
ormalization)

conv2d_1 (Conv2D)            (None, 8, 8, 16)          6416

average_pooling2d_1 (Averag  (None, 4, 4, 16)          0
ePooling2D)

batch_normalization_1 (Batc  (None, 4, 4, 16)          64
hNormalization)

conv2d_2 (Conv2D)            (None, 4, 4, 32)          12832

average_pooling2d_2 (Averag  (None, 2, 2, 32)          0
ePooling2D)

batch_normalization_2 (Batc  (None, 2, 2, 32)          128
hNormalization)

conv2d_3 (Conv2D)            (None, 1, 1, 32)          25632

average_pooling2d_3 (Averag  (None, 1, 1, 32)          0
ePooling2D)

batch_normalization_3 (Batc  (None, 1, 1, 32)          128
hNormalization)

flatten (Flatten)            (None, 32)                0

dense (Dense)                (None, 1000)              33000

dropout (Dropout)            (None, 1000)              0

batch_normalization_4 (Batc  (None, 1000)              4000
hNormalization)

dense_1 (Dense)              (None, 700)               700700

dropout_1 (Dropout)          (None, 700)               0

batch_normalization_5 (Batc  (None, 700)               2800
hNormalization)

dense_2 (Dense)              (None, 300)               210300

dropout_2 (Dropout)          (None, 300)               0

batch_normalization_6 (Batc  (None, 300)               1200
hNormalization)

dense_3 (Dense)              (None, 10)                3010
```



```
================================
Total params: 1,001,490
Trainable params: 997,298
Non-trainable params: 4,192

________________________________
```

By increasing the number of layers of the network, the accuracy on the test set can decrease. In fact, with more layers, the network can slightly overfits on the train data. For this reason, the performances of the model are worse than the performances of the previous approach.

The batch normalization layers apply a transformation which maintains the output main close to 0 and the output standard deviation close to 1.

The dropout layers randomly set input units to 0 during the training time. This should help to prevent overfitting.

Batch normalization and dropout layers are a kind of regularization.

About the optimizer, this time I used SGD (stochastic gradient descent). This optimizer calculates the gradient by using a randomly selected instance from the training data.

Instead about the regularizer of the dense layer, in this case I put a l1 regularizer. We have to remember that regularizers apply penalties on layer parameters during the optimization.

Now let’s talk about the hyperparameters.

The first one is the kernel size that in the previous approach was 3x3, now is 5x5.

In general, a larger kernel makes the computational time faster (this is because the training of approach 2 is faster than the training of approach 1) and leads in a loss of details. Actually, this last thing matters when the images in input to the network are very large (for example 1000x1000) but, since the images of my dataset are 64x64, this isn’t an important problem. The advantage of using a larger kernel is that it helps us to prevent overfitting (since it takes less details from the images).

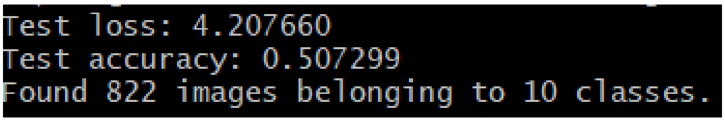
So, since there are more layers in this network that can lead me to overfit, I used the dropout layers and a larger kernel which can help me to reduce overfitting.

I also changed the strides (that now is increased to (2, 2). This allows me to have a lesser memory needed for the output since the output volume is smaller) and the activation function to the tanh.

So, considering this, we expect that:

- The performances decrease with respect to the first approach (since the training set is smaller and the number of layers of the network is grater and so there can be overfitting)
- The loss function decreases more rapidly than the first approach (since the batch size is smaller)

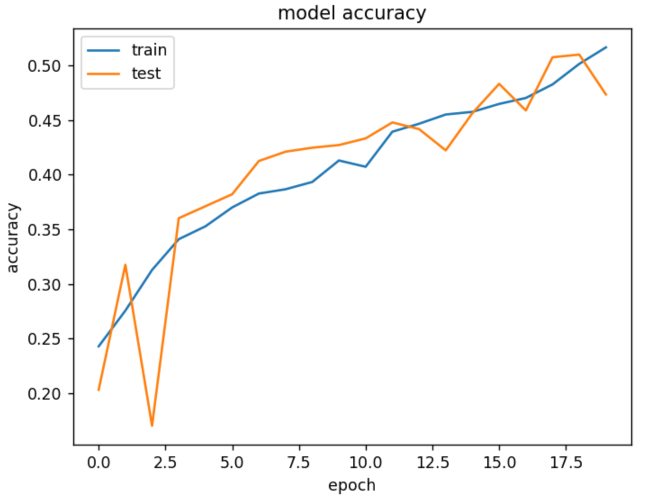
In fact, the results about accuracy and loss on test data are the following ones

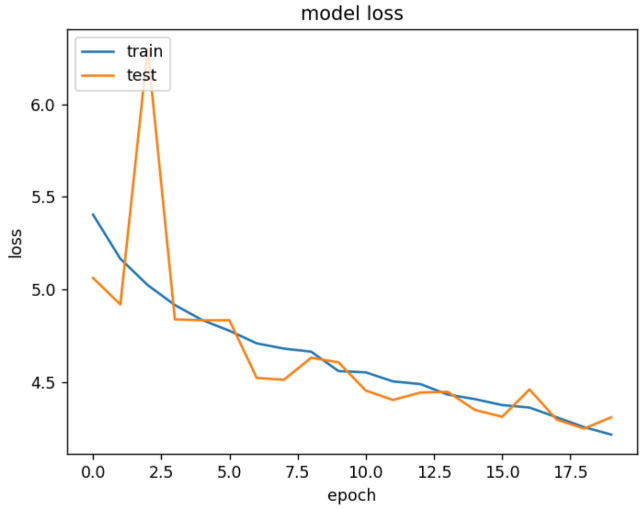


```
Test loss: 4.207660
Test accuracy: 0.507299
Found 822 images belonging to 10 classes.
```

Int approach 1 the accuracy was about 67%, here it is 50% (it decreases for the reasons I mentioned).

These are the representations of the accuracy and loss trends on train and test data





Also with this approach, the accuracy trend is crescent while the loss trend is decrescent. It means that the model is learning epoch after epoch.

Other metrics used to evaluate the network are precision, recall and f1-score.

I have 822 samples for the support and about 80 samples for each class. We know that the most important metric is the f1-score since it is a weighted average between precision and recall. The closer to 1 it is, the better is.

In my case I obtained a wighted avg f1-score of 0.48, which isn’t very good. It means that less than half of the support samples have been well predicted.