Report Homework Object Classification- Lorenzo Mattia 1793272

The aim of this homework was to use a neural network to classify images belonging to eight different classes. My eight classes were: Blueberries, Flavored Water, Jelly Beans Bag, Knives, Mop heads & Sponges, colored paper bag, pasta sides, teacup.

For all my trials I've used Google Colab because of the possibility of running the code on a remote computer with a GPU. In fact, on my pc I've a not good GPU and running the code on it would have taken too much time.

Giving a brief view to the images of different classes I've immediately noticed that data were a bit noisy, in fact, there were lots of outliers and errors.

My first step has been randomly splitting the images in train and test subfolders for each class. For doing this I've used a python script written from scratch:

```python
import shutil, random, os

#names of images folders
folderList = ['Blueberries', 'Flavored_Water', 'Jelly_Beans_Bag',
              'Knives', 'Mop_heads_&_Sponges', 'colored_paper_bag',
              'pasta_sides', 'teacup']
for folder in folderList:
    #create the directories
    os.mkdir('path/to/testFolder' +folder)
    .mkdir('path/to/trainFolder' +folder)

    dirpath = 'path/to/imageFolder' + folder
    trainDirectory = 'path/to/trainFolder' +folder
    testDirectory = 'path/to/testFolder' +folder

    #move the 70% of the images to the train folder
    list = os.listdir(dirpath)
    number = int(0.7 * len(list))
    filenames = random.sample(list, number)
    for fname in filenames:
        srcpath = os.path.join(dirpath, fname)
        shutil.move(srcpath, trainDirectory)

    #move the remaining 30% to test folder
    remaining = os.listdir(dirpath)
    for fname in remaining:
        shutil.move(os.path.join(dirpath, fname), testDirectory)
```

I've decided to split the images assigning about 70% of images to the train set and the remaining 30% to the test one. In particular, the train set was composed by 5797 images and the test one of 2488. I will use the test set also as a validation set to monitor the trend of the learning in real time.

To solve this classification problem, I've tried two different approaches and below I will show, comment and compare the best results obtained by them.

The first method I've tried has been a complete training made on a CNN made by myself, that I will call OwnCNN. For this I took inspiration from AlexNet and the exercise on it published on Classroom.

Instead, my second approach has been to use a pre trained model, in particular *VGG16* trained on the *Imagenet* dataset, and to use a small non convolutional network, *TranferNet*, to complete the training adapting the pretrained model to my dataset. This was to implement the Transfer Learning approach and even here I've used an exercise uploaded on Classroom to help myself. I will call this trial TransferCNN.

OwnCNN- First trial

Pre-processing

First of all, I want to talk about the preprocessing I have made on images for this network.

I have used the *ImageDataGenerator* class provided by the *Keras* library for doing data augmentation. During my trials I have noticed that an aggressive data augmentation reduced a lot the performance of this particular network and for this reason I've decided to try with a very poor data augmentation.

```
train_datagen = ImageDataGenerator(
    rescale = 1. / 255,\
    zoom_range=0.1,\
    rotation_range=10,\
)
```

Then I've also resized the images, bringing them to a target size of 150x150.

For what concerns the test set, I have just rescaled and resized (150x150) the images belonging to it.

Network structure

The very first net I've build was realized as follows:

```
model = Sequential()
#first Convolutional
model.add(Conv2D(filters = 32, input_shape = input_shape, kernel_size = (5,5), strides=(1,1), padding ='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size = (3,3), strides = (2,2), padding = 'same'))
model.add(BatchNormalization())

#second Convolutional
model.add(Conv2D(filters = 128, input_shape = input_shape, kernel_size = (5,5), strides=(1,1), padding ='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size = (5,5), strides = (3,3), padding = 'same'))
model.add(BatchNormalization())

model.add(Flatten())
shape = (input_shape[0]*input_shape[1]*input_shape[2],)

# D1 Dense Layer
model.add(Dense(1024, input_shape=shape, kernel_regularizer=regularizers.l2(regl2)))
model.add(Activation('relu'))
# Dropout
model.add(Dropout(0.4))
# Batch Normalisation
model.add(BatchNormalization())

# Output Layer
model.add(Dense(num_classes))
model.add(Activation('softmax'))
```

I want to comment the choices I have made:

- For both the *convolutional layers* I decided to use a 1x1 *stride*, in fact I wanted not to lose any information.
- For the same reason, not losing information even next to the bounds, I used the "same" *padding*
- I've also used *pooling layers* to reduce images size and then I flatten them with a *flatten layer*. A problem with this, in this particular network configuration has been that because of the shape of the input of the flatten layer was not like 1x1xn, its output was a huge number of trainable parameters. And I easily noticed that this only confused the net giving it the chance of learning some non-useful and distracting patterns.
- The two *dense layers* are fundamental because they practically implement classification, for this reason the last one has as number of connections the number of the classes of this problem.
- I introduced various *batch normalization* layers and a *dropout* ones because I learn that they help to prevent overfitting and to improve performances.

Details

The optimizer I chose for this trial was *Stochastic Gradient Descent* and a *learning rate* of 0.01.

The loss function chosen was the *categorical cross-entropy*, which is suggested by the *keras* documentation for problems with more than two classes.

I chose a *batch size* of 128, in fact considering the quite big number of train images I thought that it would be an appropriate number.

To not waste time, I have introduced the *early Stopping callback*, for the fit method, with a *patience* value of 3, meaning that after 3 epochs with no improvement the system will stop the training.

In addition to this, to exploit at its best the training I put the parameter *restore best weights* to True in the *early Stopping callback*, to take as final parameters at the end of the training the ones which gave the best results, and not the last ones.

```
early_stop = callbacks.EarlyStopping(monitor='val_accuracy',patience=3, restore_best_weights = True)
```

As said before, this was only my first trial and its training had an accuracy on validation set of 0.68. I will not comment more this result because I will give more space to the best one obtained with this network.

OwnCNN- Best trial

Preprocessing

Even in this trial I decided to have the same “poor” preprocessing done before.

Network structure

One of the biggest issues of the previous network was the huge number of parameters that it generates. For solving this problem, I have modified the *pooling layers* changing some parameters of the one that were already present in the previous version (increasing the size and the *stride*). In addition to this I have added a new Convolutional Layer (with also *activation function, pooling layer* and *batch normalization*), to obtain for the flatten a 1x1xn shaped input (instead of something like a x b x n) having in this way as output of the *flatten layer n* trainable parameters instead of a*b*n.

Moreover, still to reduce the number of trainable parameters obtained as the output of the entire network I reduced the number of connections of the dense layer from 1024 to 512.

```
model = Sequential()
#first Convolutional
model.add(Conv2D(filters = 32, input_shape = input_shape, kernel_size = (5,5), strides=(1,1), padding ='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size = (5,5), strides = (3,3), padding = 'same'))
model.add(BatchNormalization())

#second Convolutional
model.add(Conv2D(filters = 128, input_shape = input_shape, kernel_size = (5,5), strides=(1,1), padding ='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size = (6,6), strides = (4,4), padding = 'same'))
model.add(BatchNormalization())

#third Convolutional
model.add(Conv2D(filters = 128, input_shape = input_shape, kernel_size = (5,5), strides=(1,1), padding ='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size = (10,10), strides = (10,10), padding = 'same'))
model.add(BatchNormalization())

model.add(Flatten())
shape = (input_shape[0]*input_shape[1]*input_shape[2],)

# D1 Dense Layer
model.add(Dense(512, input_shape=shape, kernel_regularizer=regularizers.l2(regl2)))
model.add(Activation('relu'))
# Dropout
model.add(Dropout(0.6))
# Batch Normalisation
model.add(BatchNormalization())

# Output Layer
model.add(Dense(num_classes))
model.add(Activation('softmax'))
```

Details

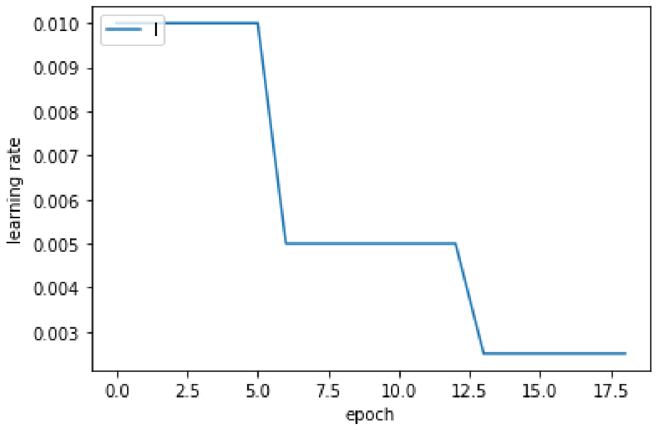
The optimizer chosen in this trial was *Adam*, because I learnt it was the one that in general guarantees the best results.

With respect to the previous trial, some features remained unchanged, in particular I kept untouched: the initial *learning rate*, the *batch size*, the *loss function*. But also, the *early stopping callback* with the *restore best weights* parameter still at True because I found it very useful.

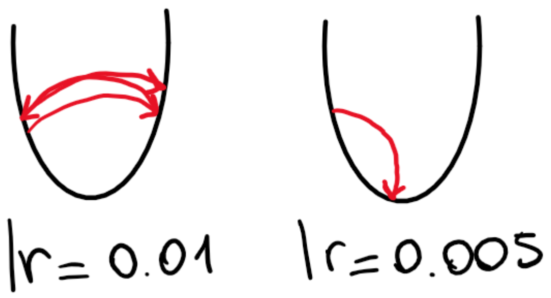
Since I was worried by the fact that the initial learning rate value was too high for the last part of the training in which is required a major precision, and reducing it from the beginning did not gave good results, I have introduced a new callback in the *fit* method, called *learning _rate*, that allows to define a function for manipulating "by hand" the learning rate value. After some trials I noticed that the best solution was to use a function that drops the value of the *learning rate* of a fixed drop quantity every tot epoch. The best combination I found have been to drop the *learning rate* of about the half every seven epochs.

```
import math
def lr_scheduler(epoch):
    initial_lrate = lr
    drop = 0.5
    epochs_drop = 7.0
    lrate = initial_lrate * math.pow(drop,
            math.floor((1+epoch)/epochs_drop))
    return lrate
```

Below a graphical representation of the learning rate trend obtained through this function:



A particular aspect of this feature that I have noticed is the following: whatever value *v* I chose for the number of epochs after which dropping the *learning rate*, every *v* epoch the network performed a marked improvement. I think that this was because before the drop the network got stacked in a "ping pong" over a peak and reducing the learning rate allowed it to get closer to this peak instead of still doing "ping pong". To sustain this thesis, I will show the graphical trend of the model accuracy.

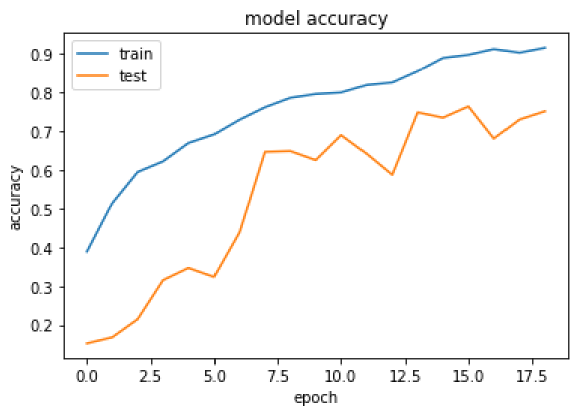


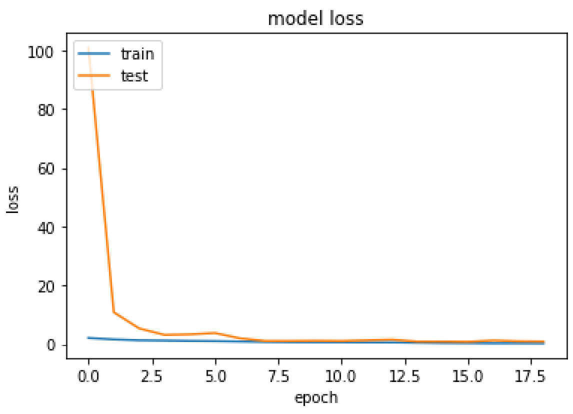
Results

Before going deeper, to summarize the results obtained:

- The final accuracy obtained on the validation set was of 0.7637 in just 19 epochs, for a total computation time of 627 seconds.
- The value obtained for the accuracy on the training set reached a value of 0.91, highlighting that the network was suffering of overfitting, but I was not able to solve this issue. Even the trials with an aggressive data augmentation, as already said, were not sufficient to solve it, and instead decreased the performances.

The first important thing to see are the graphs representing the trend of model accuracy and loss.





It is possible to see how the model accuracy graph supports the theory aforementioned about the learning rate. In fact, in correspondence of the seventh and the fourteenth epoch (I set the drop of the learning rate every seven epochs), is possible to see a peak in the learning trend.

I will show now a table representing the precision, recall and F1-score of each class to have a quick overview of the general performance of the network on each class:

```
                   precision    recall  f1-score   support

       Blueberries     0.833     0.725     0.775       331
    Flavored_Water     0.614     0.794     0.693       325
    Jelly_Beans_Bag     0.772     0.557     0.647       201
            Knives     0.892     0.828     0.859       379
Mop_heads_&_Sponges     0.883     0.744     0.808       285
  colored_paper_bag     0.632     0.805     0.708       344
       pasta_sides     0.898     0.736     0.809       299
            teacup     0.742     0.824     0.781       324
```

As highlighted by the table the class that has been classified worst is Jelly Beans Bag, in particular its recall, and I think two are the main causes to this result:

- This was the class with the fewest number of samples in the whole dataset
- A spread presence of noisy data in this class, like the following one

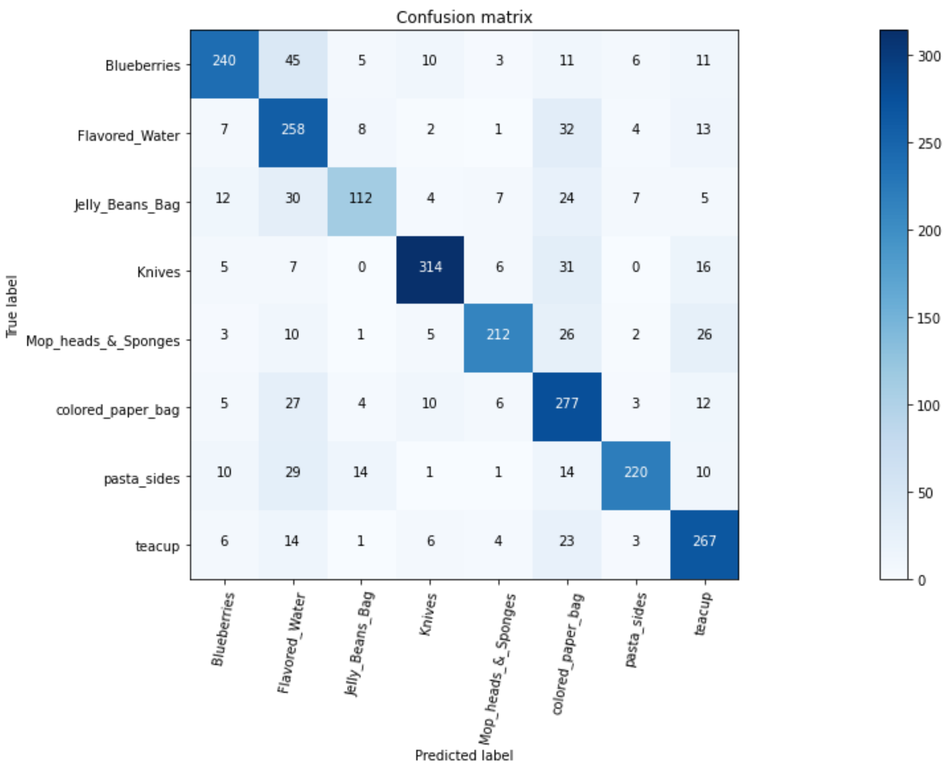
As I said before noisy data is a problem that afflicts all the classes, but maybe this one suffers it particularly.

It is important also to highlight the bad performance of the network, in terms of precision, for Flavored Water and colored paper bag classes. (other details on this later)

In average the results obtained by the various classes are the followings:

```
              precision    recall  f1-score   support

    accuracy                          0.764      2488
   macro avg      0.783     0.752     0.760      2488
```

Additional interesting information are given in the confusion matrix below, which shows the most common errors in class classification:



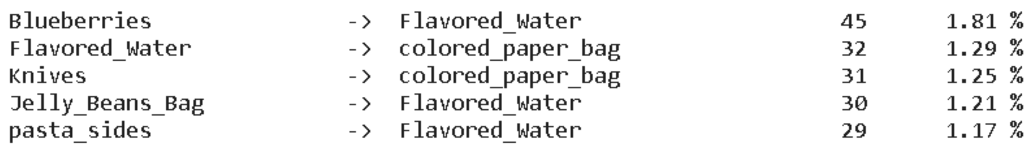
This table highlights some other limits of the network, that were also visible from the general table shown before.

- Flavored Water and colored paper bag are the worst classes in terms of precision; thus, the network often classify samples of other classes as one of these two

There are more the two hundred samples wrongly predicted as one of these two classes.
The confusion between the Blueberries and the Flavored Water classes, which is the most frequent, is due to the fact that lots of images of the second class contains blueberries because that was the flavor of the bottle represented.

- From the table is again possible to see also the low recall of the Jelly Beans Bag class, which samples are very often predicted as other classes. More than any other class, for a total amount of 89 wrongly predicted samples.

Below the five most common errors made by the system:



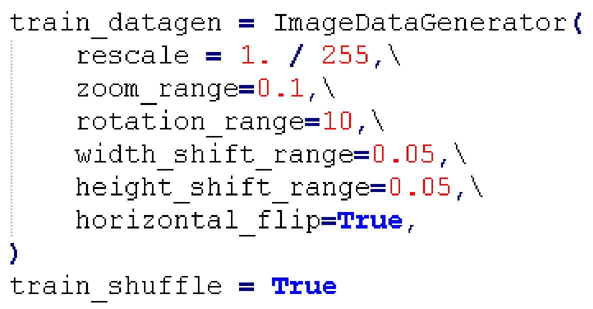
```
Blueberries              -> Flavored_Water          45      1.81 %
Flavored_Water           -> colored_paper_bag       32      1.29 %
Knives                   -> colored_paper_bag       31      1.25 %
Jelly_Beans_Bag          -> Flavored_Water          30      1.21 %
pasta_sides              -> Flavored_Water          29      1.17 %
```

TransferCNN – Best Trial

For this approach I will not split the report into a first and a best trial, because the results obtained, since the first trial, were all similar.

Preprocessing

In this case the network experienced better performances with a more complete *data augmentation*. Even here I have used the *ImageDataGenerator* class applying the following transformation to the images.



```
train_datagen = ImageDataGenerator(
    rescale = 1. / 255,\
    zoom_range=0.1,\
    rotation_range=10,\
    width_shift_range=0.05,\
    height_shift_range=0.05,\
    horizontal_flip=True,
)
train_shuffle = True
```

In particular I've used very small values for shift transformations, because I noticed, printing some random images after having applied the transformations, that, with major values for shift range, some samples had the subject of the image cut, losing important information.
Then I have resized the images with a target size of 224x224, I have chosen this particular size, because I read that *VGG16*, the pretrained model I have used in this network, was trained on 224x224 images, and so I thought it would be perfect to give it images of the same size.
As before, test set images have been only rescaled and resized.

Network Structure

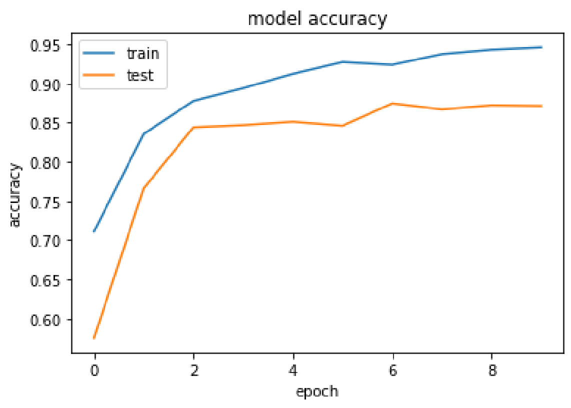
As already said, the TransferCNN (the network which is used after the pre-trained model) is not composed by convolutional layers, because it is supported by *VGG16*, for this it has only *Dense layers* to implement the real classification, and *Dropout* and *BatchNormalization* ones to improve performances and reduce overfitting.
In addition to the training of the TransferCNN, it is possible to train again one or more layers of the *VGG16* model to suite it to the current dataset. In particular, I chosed to train again the last layer, the output one.
In the code below is mentioned a variable called *output_extractor* which contains the output extracted from the last layer of the pretrained model.

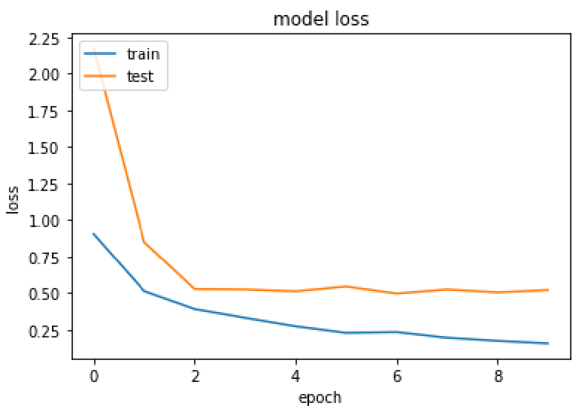
```
# flatten
flatten = Flatten()(output_extractor)
flatten_norm = BatchNormalization()(flatten)

# First Dense layer
dense1 = Dropout(0.5)(flatten_norm)
dense1 = Dense(200, activation='relu')(dense1)
dense1 = BatchNormalization()(dense1)

# Second Dense layer
dense2 = Dropout(0.5)(dense1)
dense2 = Dense(100, activation='relu')(dense2)
dense2 = BatchNormalization()(dense2)

# Output layer
dense3 = BatchNormalization()(dense2)
dense3 = Dense(num_classes, activation='softmax')(dense3)
```

Details

Considering that in this approach the most of the work was already done there are not so many things to explain in this section.
In this case I did not experienced a real improvement of performances using the learning rate callback and so I decided to not use it.
Instead, I still used the *early stopping one*.
The optimizer used has been *Adam* and the *batch size* was again 128.

Results





First of all the accuracy obtained one the validation set has reached the 0.8746 while the accuracy on the train one has reached about the 0.95.
Above are shown the graphs of the model accuracy and loss. In the first it is possible to see how the training curve of the train and the test set were more or less parallel for the first 2-3 epochs, while after these first epochs, the train accuracy has started increasing faster with respect to the test one. The same trend is observable also in the model loss graph. An interesting aspect is that the network was able to reach a high value of accuracy in a very small number of epochs, increasing it of a small quantity in all the rest of the training.

I will now show, as already done for the other approach, the table containing general information about all the classes.

```
                     precision    recall  f1-score   support

         Blueberries     0.864     0.828     0.846       331
      Flavored_Water     0.843     0.862     0.852       325
     Jelly_Beans_Bag     0.760     0.866     0.809       201
              Knives     0.934     0.963     0.948       379
  Mop_heads_&_Sponges     0.980     0.856     0.914       285
    colored_paper_bag     0.900     0.866     0.883       344
         pasta_sides     0.807     0.826     0.817       299
              teacup     0.883     0.907     0.895       324
```

It is interesting to notice that, again, the worst classified class has been Jelly Beans Bag, even if with less distance with respect to the other classes this time. In particular, the precision of this class has been even decreased by this network, while the recall had a marked improvement (about the 30%).
The other classes have experienced an improvement, but not evident like the one obtained by the Jelly Beans Bag class.
Below the general results obtained in average by the network, highlighting the quite good results obtained.

```
              precision    recall  f1-score   support

    accuracy                         0.875      2488
   macro avg      0.871     0.872     0.870      2488
weighted avg      0.877     0.875     0.875      2488
```

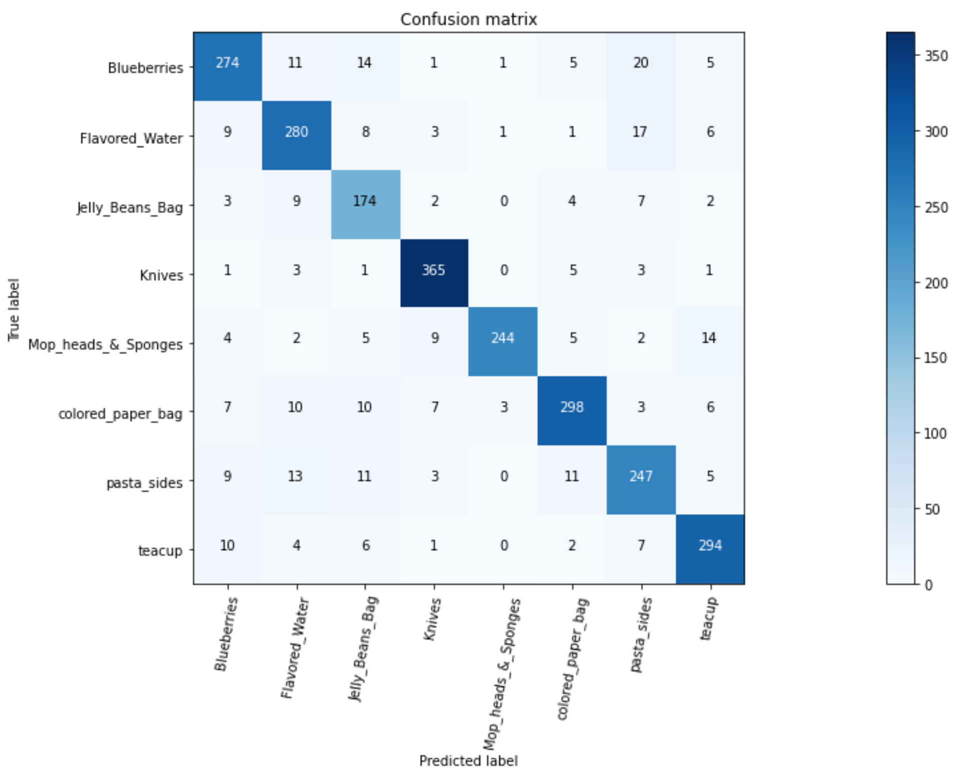
The following tables show the classification erros made by the network.
As it is possible to see, with respect with the previous network, they are different.
In particular:

- The most common error of the "OwnCNN" was Blueberries as Flavored Water, with a percentage of 1.81. This error, with this network, is strongly reduced to just 0.44%

- Instead, the most common error of this network, that is Blueberries as pasta sides, with a percentage of error of 0.8, in the previous network was mistaken in just 0.24% of samples.

Five most common errors:

```
Blueberries               ->  pasta_sides            20      0.80 %
Flavored_Water            ->  pasta_sides            17      0.68 %
Blueberries               ->  Jelly_Beans_Bag        14      0.56 %
Mop_heads_&_Sponges       ->  teacup                 14      0.56 %
pasta_sides               ->  Flavored_Water         13      0.52 %
```



Conclusions

As it was reasonable and predictable, the network supported by a pretrained model has given better results. It is also important to consider that in my first approach, the "OwnCNN", the network built was very small, in particular with respect to the other one.