Computer vision and pattern recognition project

Presta Alberto

January 2020

Introduction

With this report I want to show how I carried out my project and I will try to describe in detail my choices both from a theoretical point of view and from an "implementation" point of view. The project is organized in the following Python scripts:

1. *Point_one*: First part of the project is implemented.
2. *Point_Two*: Second part of the project is implemented.
3. *Point_Three.A*: Part related to freezing the weights is implemented.
4. *Point_Three.B*: Feature extraction and linear SVM is implemented
5. *Utils*: Functions for data processing and plotting are implemented.

Problem Statements

The goal of the project is to build an image classifier between different scenes: in particular, we will deal with 15 different classes. Images of our available data-set are split between 1500-sized training set and a 2985-sized test set.

Overview of the workflow

1. In The first part I simply implemented, using keras,the shallow convolutional neural network requested by the exercise and I report the results in terms of accuracy, validation loss and confusion matrix. Obviously before training the cnn I had to preprocess my data in a way to make them "feasible" for my cnn (later on I will explay better how I dealt with data).
2. In the second part I tried to improve test accuracy of my cnn through several techiques, for example data augmentation, insertion of some dropout and batch normalization layer and the so called "esemble of network". My aim is to reach at least the 60% of the test accuracy.
3. In the last part I exploit a pre-trained cnn called VGG16 in two ways: first of all, I will freeze all weights except those of the last layer and I'll train only these one and then I will exploit feature from the last convolutional layer in order to apply a linear SVM.

First Part

Preprocessing of data

As I just said in the presentation, for this project we have a data set organized in the following way:

1. 1500 Train images from 15 different classes.
2. 2509 Test images from 15 different classes.

Obviously, train images are used to train my convolutional neural network while test images are used to validate and test it. In order to organize my data, I wrote two functions, explained below:

1. list_of_path(labels,path): It receives in input the list of labels (the 15 classes taken in consideration), and the path where to find images. It returns a permuted list of all the path for each images. This list is necessary for creating our image data-set and it represents the input of the next function
2. read_and_process_images(list_of_images): It receives the output of the previous function as input and it returns two numpy array: in the first one there are all the normalized images resized into (64,64) shape and in the second one there the respective labels.

I applied these two function both for training and testing data. Last thing to do is to perform a reshaping of the train and test data-set. In particular I add a dimension in order to specify the number of channel of each image (dealing with black and white images, it is equal to one). I do the reshape in this way:

```
train_data = train_data.reshape(train_data.shape[0],train_data.
    shape[1],train_data.shape[2],1)
test_data = test_data.reshape(test_data.shape[0],test_data.shape
    [1],test_data.shape[2],1)
```

Building the CNN

Initially I decide to follow the simple model requested by the project. I implemented it in python, in particular using keras. Here below there is the list of all the packages I used to build my first cnn, to plot some important features i.e. the accuracy or the confusion matrix.

```
import numpy as np
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense
import os
import cv2
from keras import initializers
from keras import optimizers
from keras.utils import to_categorical #to create dummy variable
from keras.layers import Conv2D,Flatten,Dropout, MaxPooling2D
from keras.callbacks import EarlyStopping
from keras.callbacks import History
from sklearn.metrics import confusion_matrix
```

Some important considerations about the model:

1. As Optimizer I decided to use sgd("stochastic gradient descent") with batch size equal to 32. Going ahead with the project it will turn out that the best choice is adam otpimizer.

2. To improve test accuracy I also added a nesterov momentum with coefficient equal to 0.90.

3. I divide the training set in two sub-groups: the first one is dedicated to the training and it represents the 85% of the total and the remaining 15% is dedicated to the validation.

4. I don't add any dropout layer, but to reduce overfitting I use the technique called earlyStopping with min_delta=0.0 and patience=10.

The following figure represents the structure of my first cnn:

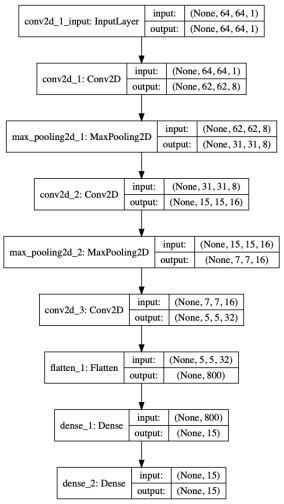

Figure 1: Graph which represents all the layer of the cnn, obtained with the function plot_model

Results

After training the cnn through this instance:

```
model.fit(train_data,train_labels_dummy,batch_size=32,epochs=100,
    validation_split=0.15,shuffle=True,callbacks=[earlyStopping,
    history])
```

The training stopped after 34 epochs and, in terms of test-loss and test accuracy, we reached the following results:

```
test-loss = 2.752298270917218
test-accuracy = 0.3815745413303375
```

Obviously this results may vary if I do another training of my cnn from scratch and this is due to the stochastic part of the model, but we can see that it is sufficient a shallow cnn without data augmentation or adam optimizer to achieve a quite good result.

We can take a look now to the confusion matrix, which is the following:

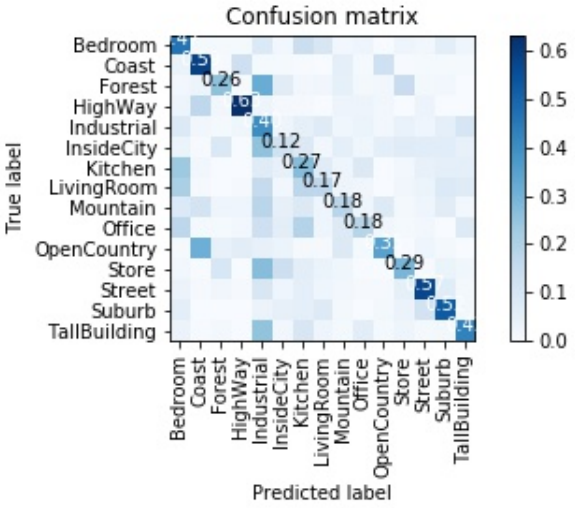

Figure 2: Confusion matrix related to the first cnn developed. In the diagonal there are the rate of the accuracy per each class

We can observe in the confusion matrix that the accuracy can vary from class to class; for example, class "suburb" and " store" have an accuracy of over 50% and it means that these two class are easily identifiable. Other classes, such as "Office" or "inside city" have a very low accuracy, so there are much more uncertainty on deciding the right class.

We can also take a look to the plot of these following values:

1. Validation loss.
2. Validation accuracy.
3. Loss.
4. Accuracy.

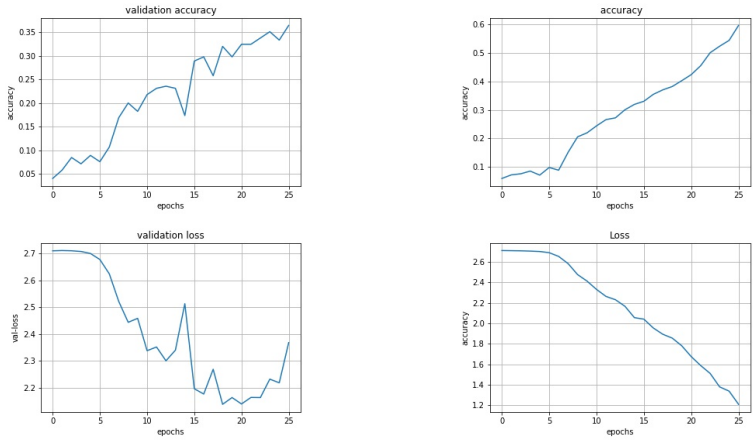

Figure 3: Here we have 4 different plots which represent how accuracy (first two figures) and loss (last two figures) vary with the epochs with validation and test data. We can observe that we have much more regularity in the variation on accuracy and loss with training data respect than validation loss and validation accuracy ; this is because the latter are evaluated in the validation data and so they are more unpredictable.

Second Part

So, until now I have reached a test accuracy of around 32-33%, which is not so bad as soon we used a very shallow cnn with few data...but it is possible to do better than this; in particular It is possible to develop some technique in order to increase test accuracy and make the cnn more precise.

Data augmentation

A first initial technique that I developed in my codes is so called "data augmentation". Using my initial training data (1500 images of 15 different classes) I augmented the number of available data with 2 simple transformations:

1. left-to-right reflections.
2. Cropping of images.

So from 1500 images we passed to a data-set of 4500 images. All the details of the implementation are described in the function "data_augmentation" in the script called "utils." Just with these simple modifications and using the same cnn developed in the first part, I reached a quite good implementation[1]:

```
test-loss = 2.326505197112884
test-accuracy = 0.46845337748527527
```

Dropping and batch normalization

Another two very common technique to improve results of a cnn are *Dropout*[2], which avoids the possibility of overfitting, and *batch normalization*, which controls the values of weights and bias avoiding the risk of gradient explosion. Moreover another thing to underline is the fact that I change the optimizer from *sdg* to *adam* with the default parameters. I add some layers of dropout and batch normalization on my previous cnn, obtaining the following structure:

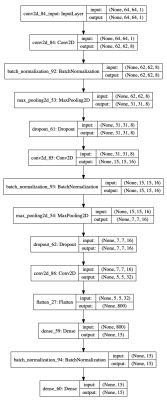

Figure 4: Same cnn as before, but with the adding of some dropout layers and batch normalization layers

With these modification, results begin to become much more better respect to the initial cnn:

```
test-loss = 1.3994566063186031
test-accuracy = 0.5670574903488159
```

Giving a look to the confusion matrix, we can observe that some classes are starting to become differentiable from the others, surpassing the 70% of the accuracy.

[1] I augmented also test data in order to have more data to evaluate my cnn.
[2] I used a dropout rate of 0.3.

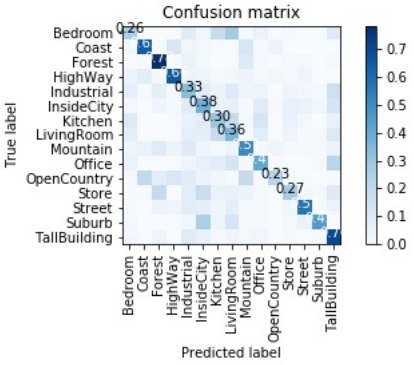

Figure 5: Confusion matrix related to the second cnn developed. In the diagonal there are the rate of the accuracy per each class.

Ensemble of Networks

Another very important technique to improve results is to train independently a several number of cnn of the same type (from 5 to 10 identical cnns) and then find a method to "merge" different outputs in a smart way to make the results better and much robust. I try two different ways to merge outputs:

1. *Geometric average*: being the output a discrete probability density, we can simply take the average.
2. *Voting*: Determine the class of a test images by voting, so the class which is chosen most of the time by the different cnns.

The results are pretty similar, indeed in both cases we exceed the threshold of 60% of test accuracy, as we can see below[3]:

```
\\ These results has been obtaining running my codes and taking
\\ 10 identical neural networks with both dropping and batch norm


test-accuracy-average = 0.6457844779452819
test-accuracy-voting = 0.6366275823562255
```

To achieve these results, I wrote some functions (listed below)[4]:

1. ensemble_of_network: This functions trains several independent and identical cnns (the type of the cnn is a parameter) and then calculates the mean of the probability density function calculated in the test class (detais are in the jupyter notebook script).
2. calculate_accuracy: It calculates the accuracy for the average ensemble of networks.
3. calculate_accuracy_with_voting: It calculates the accuracy for the voting ensemble of networks.

[3]In the github repository, there is also a folder containing all interesting pictures and there are also the path of loss, val-loss,accuracy and val-accuracy of different cnns

[4]All the details are in the jupyter notebook script.

Play with the number of layer

A final experiment before moving on to transfer learning was to play a bit with the number of layers in our convolutional neural network; I add some convolutional and dense layers, obtaining a deeper cnn. The structure is the following:

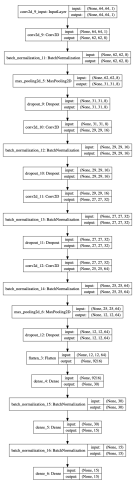

Figure 6: Structure of the augmented model

Unfortunately, this model was more difficult and slower to train, without much better results:

```
test-loss = 1.856033660365775
test-accuracy = 0.5686208605766296
```

Third Part

In the last part I will do something different; instead of building mi own cnn from scratch, I will exploit the features of a pre-trained cnn in two different ways:

1. Freeze the weights of all the layers but the last fully connected layer and fine-tune the weights of the last layer with the same data as before (not augmented).

2. Employ the pre-trained network as a feature extractor, accessing the activation of the last convolutional layer and train a multiclass linear SVM.

I decided to use a convolutional neural network called VGG16, which is included in keras package and it has got the following structure:

Layer		Feature Map	Size	Kernel Size	Stride	Activation
Input	Image	1	224 x 224 x 3	-	-	-
1	2 X Convolution	64	224 x 224 x 64	3x3	1	relu
	Max Pooling	64	112 x 112 x 64	3x3	2	relu
3	2 X Convolution	128	112 x 112 x 128	3x3	1	relu
	Max Pooling	128	56 x 56 x 128	3x3	2	relu
5	2 X Convolution	256	56 x 56 x 256	3x3	1	relu
	Max Pooling	256	28 x 28 x 256	3x3	2	relu
7	3 X Convolution	512	28 x 28 x 512	3x3	1	relu
	Max Pooling	512	14 x 14 x 512	3x3	2	relu
10	3 X Convolution	512	14 x 14 x 512	3x3	1	relu
	Max Pooling	512	7 x 7 x 512	3x3	2	relu
13	FC	-	25088	-	-	relu
14	FC	-	4096	-	-	relu
15	FC	-	4096	-	-	relu
Output	FC	-	1000	-	-	Softmax

Figure 7: Structure of VGG16

Freezing the weights

As I mentioned before, The first part is dedicated to train only the last fully connected component of the VGG16, while the remaining weights are freezed. Obviously, since VGG16 has been written for resolving a classification problem between 1000 different classes, I had to change the last classification layer with a layer conform to our problem.[5] For what concern the processing of data, I used the package *image* from *keras.preprocessing* in order to make data suitable for the VGG16.[6] Notice that, respect to the previous part, images now has a shape of 224x224. In particular, function preprocess_input is meant to adequate your image to the format the model requires.
I wrote a function, called *vgg16_model*, which initializes the right model and I compile it; As Optimizer I used *Adam* with default values and as loss function I decide to use *categorical_crossentropy*. As I mentioned before, only the last weights are trainable, but despite it is an expensive computation as soon there are a lot of weights; indeed we are in the following situation:

[5] I simply change the number of outputs from 1000 to 15.

[6] In particular, this cnn accepts only 3-channel images, so I have to make a copy of the channel of the images in our dataset.

```
Total params: 134,321,999
Trainable params: 61,455
Non-trainable params: 134,260,544
```

Results

I trained the weights of the last fully connected layer for 10 epochs and using a batch-size equal to 64 and I obtained very good results:

```
test-loss = 0.37291230101331796
test-accuracy = 0.8793969750404358
```

We can take a look also to the confusion matrix and we can see how the situation has improved a lot

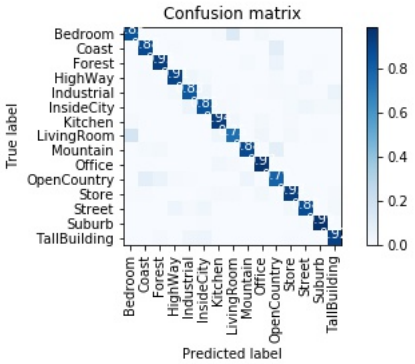

Figure 8: confusion matrix related to VGG16.

We can also take a look to the plots of loss, accuracy, validation loss and validation accuracy; we can see how the situation improves in the early ages, and then stabilizes in the successions.

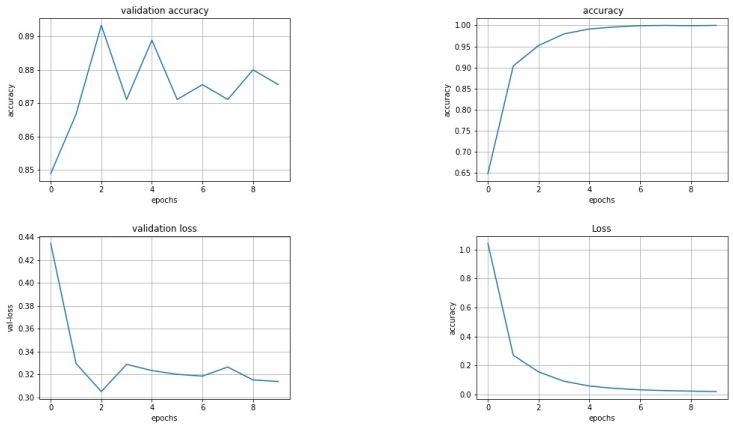

Figure 9: plots related to VGG16

Feature extraction

Last thing to try now is to exploit the cnn not for classifying directly classes of images, but to extract features to use after in another classification technique; in our case we will perform a linear SVM. In particular we will extract features obtained in the last convolutional layer of the VGG16; what kind of structure do these features have? for each image, we will obtain 512 features, each of them is a 7x7 grid of values. Obviously, to make these feature suitable for a SVM, we will flatten all the 512 features related to a specific image in a unique long vector! what we obtain is that for each image we associate a feature represented by a 25088-dimension vector. These feature extractor is performed by a function I wrote called *feature_extraction*, which is in the *Point_Three.B* script. Once we have obtained features from images, we simply use it in a SVM; this is done in the function called *predict_with_linear_SVM*, which returns the accuracy of technique based on SVM. Being a multiclass classification problem, it is necessary to decide what strategy one has to use for classification; I personally adopted a one-against-rest approach in which the winning class is the one that maximize the distance from the separating hyperplane.[7]. For what concern the obtained result, we can observe that the test accuracy overcomes 86 %, indeed:

```
test-accuracy = 0.8991624790619765
```

Specifically, we can see how the individual classes are classified in the following scheme:

	labels	True positive	True negative	False positive	False negative
0	Bedroom	91	2843	26	25
1	Coast	237	2695	30	23
2	Forest	220	2734	23	8
3	HighWay	151	2815	10	9
4	Industrial	169	2765	9	42
5	InsideCity	178	2761	16	30
6	Kitchen	101	2863	12	9
7	LivingRoom	147	2762	34	42
8	Mountain	256	2691	20	18
9	Office	113	2856	14	2
10	OpenCountry	254	2633	42	56
11	Store	202	2755	15	13
12	Street	180	2775	18	12
13	Suburb	140	2835	9	1
14	TallBuilding	245	2706	23	11

Figure 10: Dataframe representing, for each class, value of true positive rate, true negative rate, false positive rate and false negative rate.

[7] I choose this strategy for simplicity, even if it is not longer suitable if we deal with non linear kernels

I wrote also a function which create a matrix with is the equivalent of the confusion matrix, and I reach the following result:

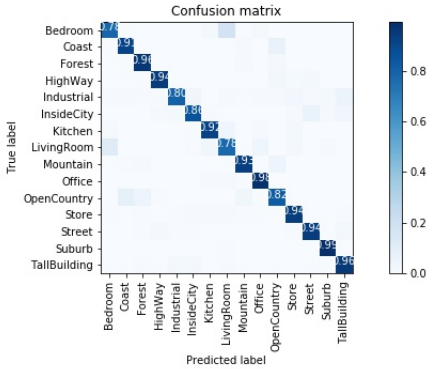

Figure 11: Confusion matrix related to the results obtained using linear SVM.

Conclusion

In this project, we will exploit all the most important feature of cnn for a multiclass classification problem and we have observed that techniques based on cnn are very powerful; A very shallow convolutional neural network was enough to obtain a quite good results and it has been possible improve these results with simple trick like batch normalization or data augmentation. In the end, we applied transfer learning, in particular we extract features obtained by the last convolutional layer of VGG16 and we exploited them in a SVM, obtaining a very high test accuracy.