CVPR- PROJECT 3

Milton Nicolás Plasencia Palacios

Lorenzo Bonin
Ionuţ Alexandru Pascariu

Course of AA 2020-2021

Tell me and I will forget.
Show me and I will remember.
Involve me and I will understand.
Chinese proverb

Simple things should be simple,
complex things should be possible.
Alan Kay

Contents

1 Introduction

This project is meant to explore the behaviour of a *convolutional neural network* as an image classifier on the dataset provided by Lazebnik et al [1]. Firstly, it was used a shallow approach in which we built the suggested architecture, which has been improved in the subsequent sections by applying different techniques such as *data augmentation, batch normalization*. Finally, *transfer learning* on a pre-trained network has been exploited.
The entire code was written in Matlab. The provided dataset contains two folders, named *test* and *train*, which are composed respectively by 1500 and 2985 images representing 15 different classes of environments.

2 Task 1

2.1 Proposed Solution/Procedure

In this first task, we tried to reach an accuracy of about 30% by implementing a Convolutional Neural Network (CNN) having the layout shown in table 1. The train folder was divided in two parts, in order to dispose of a set for the training and another one for the validation, while the test folder was used to compute the Confusion Matrix and the final accuracy that the model reached at the end of each step of the project. Each image from the entire dataset was resized since the *Image Input* layer of the net required a dimension 64x64. Then from the train folder it was created a training set, containing 85% of the data, whereas the remaining 15% was used as a validation set.

#	type	size
1	Image Input	64×64×1 images
2	Convolution	8 3×3 convolutions with stride 1
3	ReLU	
4	Max Pooling	2×2 max pooling with stride 2
5	Convolution	16 3×3 convolutions with stride 1
6	ReLU	
7	Max Pooling	2×2 max pooling with stride 2
8	Convolution	32 3×3 convolutions with stride 1
9	ReLU	
10	Fully Connected	15
11	Softmax	softmax
12	Classification Output	crossentropyex

Table 1: Layout of the CNN

2.2 First Train

For the former train it was needed to use all the default value of the *trainingOptions* apart from the value of the learning rate and mini batch size. As showed below the *InitialLearnRate* value is equal to 0.001. Instead, the *MiniBatchSize* value is equal to 32 as required. The learning rate value was chosen after several training session in order to achieve the required accuracy. Another request was to employ the *stochastic gradient descent with momentum* optimization algorithm, hence we decided to use the *sgdm* solver with default *momentum* equal to 0.90. According to the project assignment, the initial weights of the convolutional and of the fully connected layer had to be drawn from a Gaussian distribution with a mean of 0 and a standard deviation of 0.01, hence we set the weights of these layers by applying the *narrow-normal* function provided directly by Matlab. The Gaussian weights are one of the sources of randomness for which the results of the various simulation were slightly different. The initial bias of the different layers was set to zero. In this first simulation it hasn't been chosen a proper stopping criterion, so the train session stopped when it reached the training cycle composed by 30 epochs. In figure 1 it is shown how the validation Loss is increasing after iteration #400. This leads the training to overfit so it was decided to adopt the *early stopping* criterion as regularization technique, in order not to wait until the maximum number of iteration is reached.

```
options = trainingOptions('sgdm', ...
    'InitialLearnRate', 0.001, ...
    'ValidationData',imdsValidation, ...
    'Verbose',false, ...
    'MiniBatchSize',32, ...
    'ExecutionEnvironment','parallel',...
    'Plots','training-progress')
```

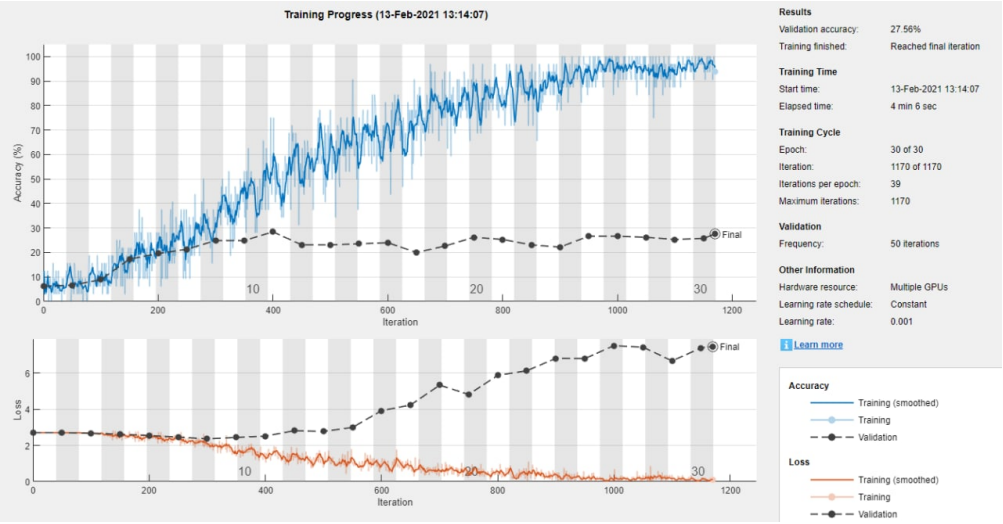

Figure 1: Train without stopping criterion

2.3 Second Train

In order to stop the training before reaching the maximum number of iteration it was decided to change the value of the *ValidationPatience* from the default one, which was infinite, to an integer value. The validation patience is the number of times that the loss on the validation set can be larger than or equal to the previously smallest loss before network training stops. It was also decided to leave the value of the *ValidationFrequency*, the number of iterations between evaluations of validation metrics, to its default value 50. Since it was noticed from figure 1 that after a certain number of iterations the Loss function was increasing it was decided to test several values for the validation patience parameter and a good one is 2. The final result of the train is shown in figure 2.

```
options = trainingOptions('sgdm', ...
    'InitialLearnRate', 0.001, ...
    'ValidationData',imdsValidation, ...
    'ValidationFrequency', 50, ...
    'ValidationPatience', 2,...
    'Verbose',false, ...
    'MiniBatchSize',32, ...
    'ExecutionEnvironment','parallel',...
    'Plots','training-progress')
```

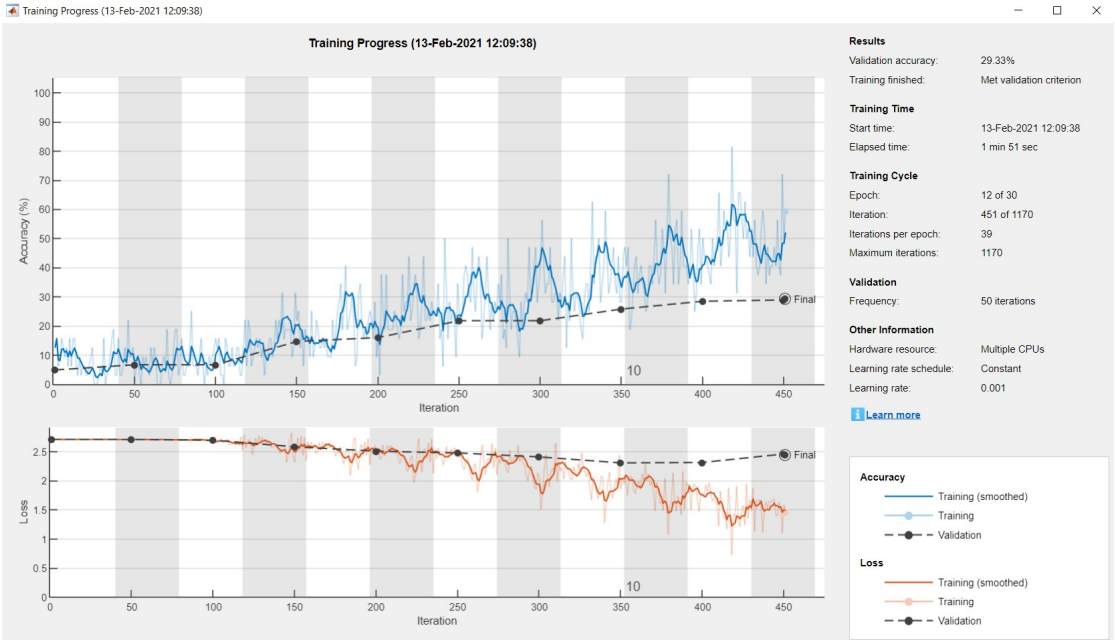

Figure 2: Train with ValidationPatience = 2

2.4 Final accuracy and Confusion Matrix

As regards the last part of this task, it was used the *test* folder to compute the accuracy and then the confusion matrix. In the first simulation, without the early stopping criterion, the accuracy obtained is 22,5%, while in the simulation in which the *ValidationPatience* was set it was obtained an accuracy of 28,74%. However, in subsequent simulations, we obtained values around 30%.
As regards the confusion matrix considering figure 3 it can be seen that the best classified classes, with an accuracy around 55%, are *Highway* and *Street*. Instead, the worst classified class is *LivingRoom*. It was to be expected to have several misclassifications as the accuracy obtained is very low.

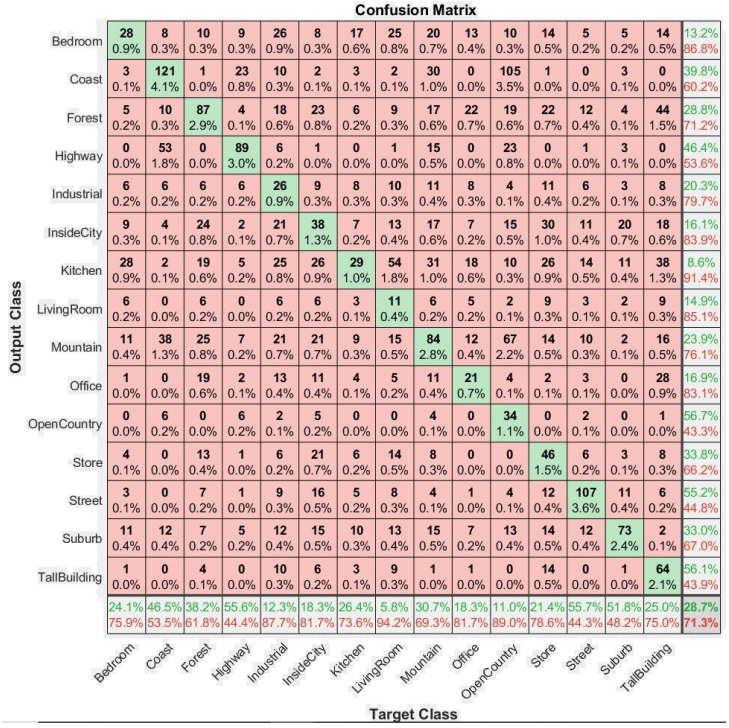

Figure 3: Confusion Matrix of the Train with ValidationPatience = 2

3 Task 2

We tried to improve the results from the first section by applying the following strategies:

- data augmentation
- addition of batch normalization layers
- changing the size of the convolutional filters

- switching to the Adam optimizer and changing the minibatch size
- addition of dropout layers
- ensemble of CNNs

3.1 Data augmentation

It was decided to augment the dataset by performing a random cropping and a left-right reflection, in order to "force" the model to be invariant to these transformations. It has to be specified that the data augmentation has been performed only on the "training subset" of the train folder (hence not on the validation and test sets). Even if Matlab is provided with a very useful object named augmentedImageDatastore, which allows to perform the data augmentation without actually saving new images to disk, it has been decided to do this procedure by hand, in order to add a further source of randomness in the size of the cropping window. In fact, this would not have been possible if we had used the functions provided by Matlab. Hence, the size of the original training set has been tripled, since two "transformed" copies (one for each transformation) have been added to it. Thanks to this improvement we managed to raise the overall accuracy to values which were very close to the 40%, as shown in figure 4.

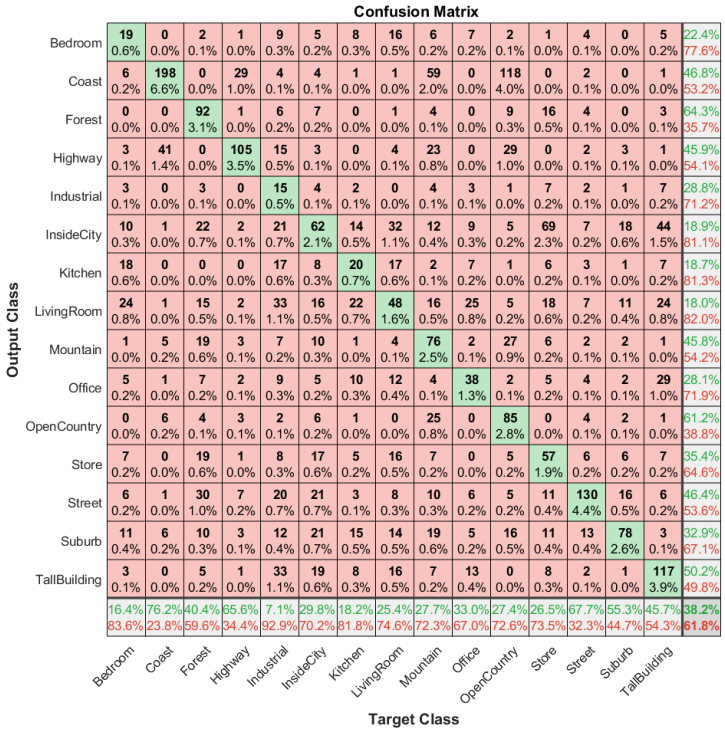

Figure 4: Confusion Matrix

3.2 Batch normalization layers

The second improvement that we performed was the addition of a batch normalization layer before each ReLu layer. This procedure usually allows to use bigger learning rates and, consequently, speeding up the training procedure, but, in our specific case, this was not convenient: values of the learning rate which were higher than the one used in the previous section always led to worse results in the accuracy, so it was decided not to modify it. The following network has been developed:

#	type	size
1	Image Input	64×64×1 images
2	Convolution	8 3×3 convolutions with stride 1
3	Batch Normalization	
4	ReLU	
5	Max Pooling	2×2 max pooling with stride 2
6	Convolution	16 3×3 convolutions with stride 1
7	Batch Normalization	
8	ReLU	
9	Max Pooling	2×2 max pooling with stride 2
10	Convolution	32 3×3 convolutions with stride 1
11	Batch Normalization	
12	ReLU	
13	Fully Connected	15
14	Softmax	softmax
15	Classification Output	crossentropyex

Table 2: Layout of the CNN with Batch Normalization

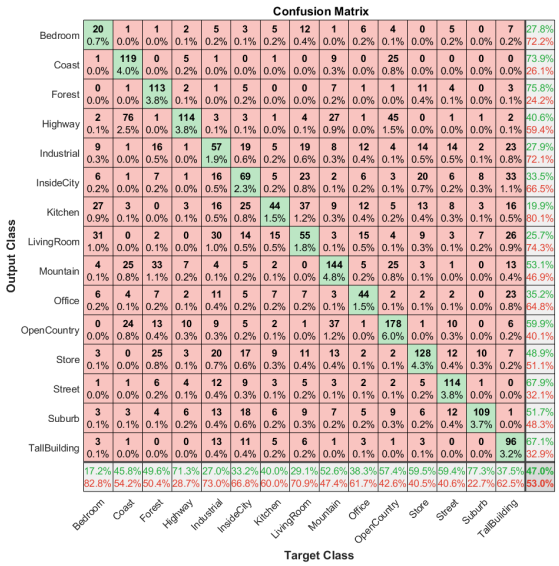

Figure 5: Confusion Matrix 3.2

This led to a huge increase in the accuracy, which reached values of about 47-48%, as shown in figure 5. Furthermore, it has to be mentioned the fact that we noticed a decrease in the variance of the results from the different simulations, which was probably due to the tendency of the batch normalization layers to reduce the dependence on the initialization.

3.3 Change the size of the filters and some parameters

After the addition of the batch normalization layers, it was decided to apply other changes. Firstly, we modified the size of the convolutional filters, by increasing the support of the filters moving from input to output, to 3x3, 5x5 and 7x7. This did not lead to a significant improvement, so it was tried to add some convolutional and fully connected layers, following different schemes, but without obtaining improvements as well. Hence, it was decided to remove these extra layers, in order to reduce the computation time.
Subsequently, the minibatch size was changed to 64 and it was decided to switch to the Adam optimizer. It is a faster extension of the stochastic gradient descent, which has proved to perform quite better than the sgdm in our specific scenario. These last two changes allowed us to obtain a larger accuracy on the test set, which reached values close to the 50%. One result is reported in figure 6.

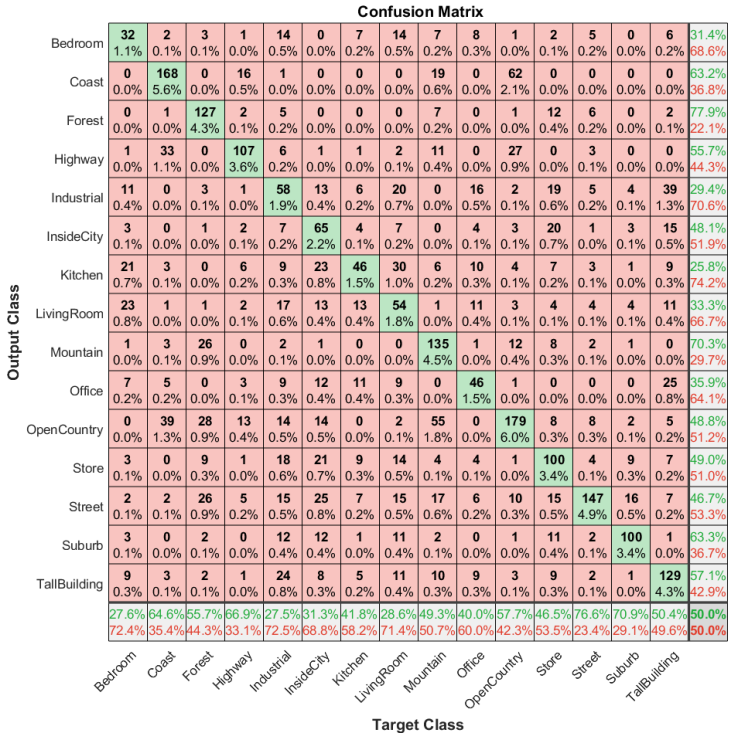

Figure 6: Confusion Matrix 3.3

3.4 Dropout layers

Although the results were already quite satisfying, we thought that they could be improved even more, especially acting on regularization. Hence, it was decided to add some dropout layers, changing the topology of the network once again:

#	type	size
1	Image Input	64×64×1 images
2	Convolution	8 3×3 convolutions with stride 1
3	Batch Normalization	
4	ReLU	
5	Max Pooling	2×2 max pooling with stride 2
6	Dropout	
7	Convolution	16 3×3 convolutions with stride 1
8	Batch Normalization	
9	ReLU	
10	Max Pooling	2×2 max pooling with stride 2
11	Dropout	
12	Convolution	32 3×3 convolutions with stride 1
13	Batch Normalization	
14	ReLU	
15	Dropout	
16	Fully Connected	15
17	Dropout	
18	Softmax	softmax
19	Classification Output	crossentropyex

Table 3: Layout of the CNN with Dropout

This helped to reduce overfitting, as it can be seen in figure 7, where the training error curve is constantly under the validation error one: The accuracy on the test set has further increased to the 52%,as shown in figure 8 providing us a good baseline for the last step of the task 2.

3.5 Ensemble of CNNs

The last technique that has been implemented in this section is the ensemble of networks, made up of 10 CNNs from the previous point, which have been trained independently. It was decided to assign the class by extracting the output probabilities of the different networks for that instance, averaging them over the number of networks and then taking the highest of the averaged values. We noticed both a huge increase in the accuracy, which reached the 60%, and way less variance in the results obtained from the different simulations.

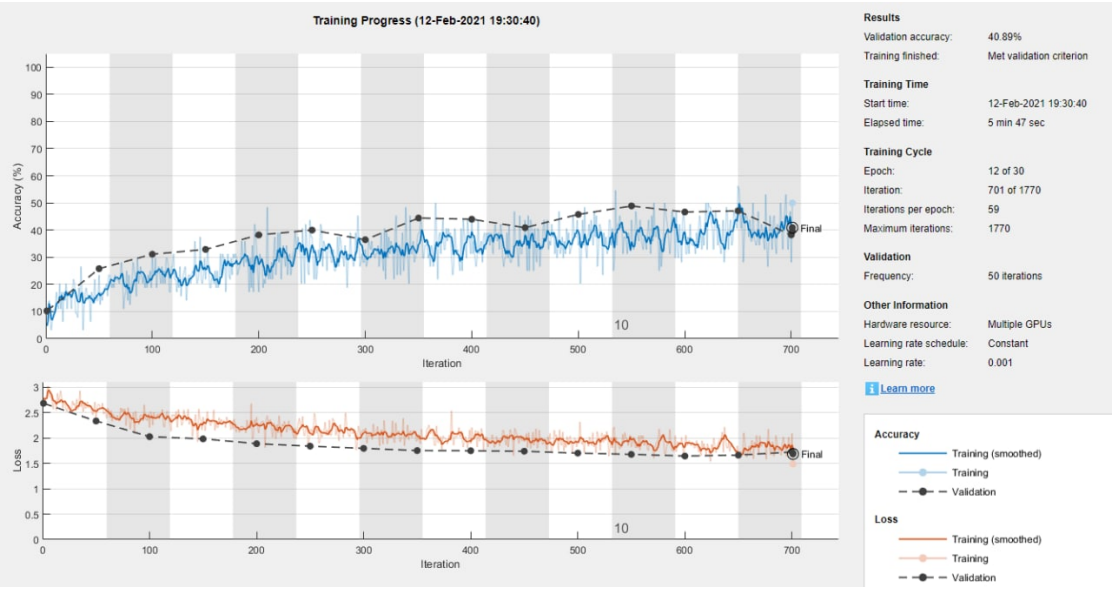

Figure 7: Train of CNN with Dropout layers

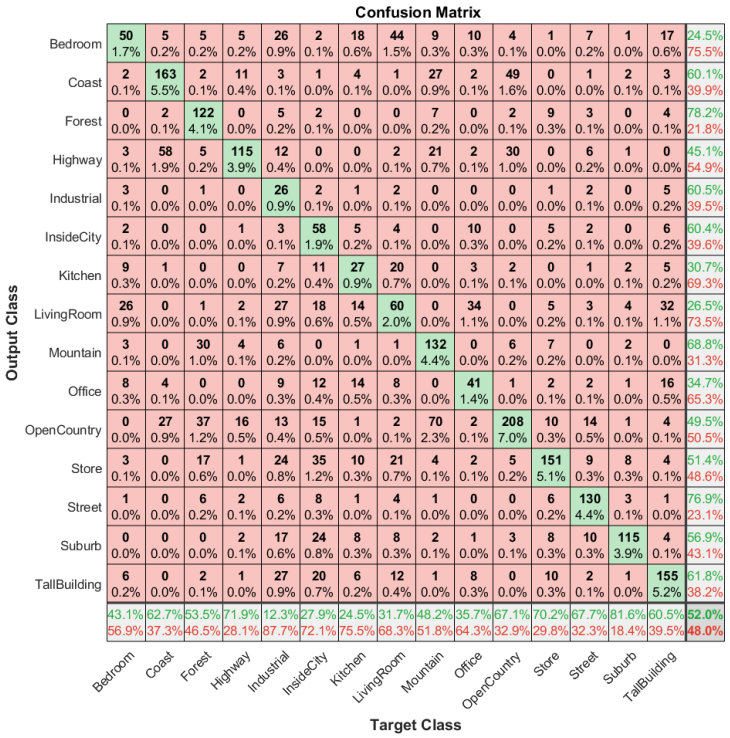

Figure 8: Confusion Matrix 3.4

4 Task 3

In the last task it was required to use *Transfer Learning* on a pre-trained network, so AlexNet was chosen in order to show the improvements brought by this technique. Its architecture is described in the following table:

#	type	size
1	Image Input	227×227×3 images
2	Convolution	96 11×11x3 convolutions with stride 4
3	ReLU	
4	Cross Channel Normalization	5 channels per element
5	Max Pooling	3×3 max pooling with stride 2
6	Grouped Convolution	2 groups of 128 5×5x48 convolutions with stride 1 and padding 2
7	ReLU	
8	Cross Channel Normalization	5 channels per element
9	Max Pooling	3×3 max pooling with stride 2
10	Convolution	364 3×3x256 convolutions with stride 1 and padding 1
11	ReLU	
12	Grouped Convolution	2 groups of 192 3×3x192 convolutions with stride 1 and padding 1
13	ReLU	
14	Grouped Convolution	2 groups of 128 3×3x192 convolutions with stride 1 and padding 1
15	ReLU	
16	Max Pooling	3×3 max pooling with stride 2
17	Fully Connected	4096 fully connected layer
18	ReLU	
19	Dropout	50% dropout
20	Fully Connected	4096 fully connected layer
21	ReLU	
22	Dropout	50% dropout
23	Fully Connected	1000 fully connected layer
24	Softmax	softmax
25	Classification Output	crossentropyex

Table 4: AlexNet Architecture

The network is composed by five convolutional layers, followed by max pooling layers, and by three fully connected layers in which the last one has one thousand neurons (because it was build to predict that number of classes). The chosen activation function was ReLU because it gave better results than sigmoid and tanh.
The purpose of using a pre-trained network is to improve the performance on the provided

dataset exploiting a network that has been already trained on a similar task and so it has learned the weights.

4.1 Fine-tuning last layer

Firstly, the size of the images has been changed in order to fulfill the requirement of the input layer, i.e. having an image of size $[227, 227, 3]$. For this task, we used the Matlab function called *repmat*. It was not considered the idea of changing the input layer in order to be able to use grayscale images because, in that case, the network should have been retrained and, hence, the advantage of transfer learning would have been lost. Then, as suggested, the weights of all the layers were frozen, but not the ones in the last layer. Moreover the last fully connected layer has been modified in order to be able to predict the fifteen classes of the provided dataset . Data augmentation was not employed during the first runs and the training options are showed in the following snippet:

```
options = trainingOptions('adam', ...
    'ValidationData',imdsValidation, ...
    'InitialLearnRate', 0.001, ...
    'MaxEpochs', 30, ...
    'ValidationPatience', 5,...
    'Shuffle', 'every-epoch', ...
    'Verbose',false, ...
    'MiniBatchSize',32, ...
    'ExecutionEnvironment','parallel',...
    'Plots','training-progress')
```

The improvement, with respect to the techniques that had been used previously, was impressive: the accuracy reached the 80%[1].
In order to analyze better the results and choose an appropriate patience value, it was decided to perform some tests using a high number of epochs (50) with a high patience (10 and 15). The options were basically the same as before, it was only changed the batch size (now increased to 64). There is no significant improvement in this test, the validation accuracy did not change after the 30-th epoch. Regarding the test set, the accuracy increased by 0.5-1%, reaching the 81%. Despite the improvement, this method requires a high computational effort, hence the *Max Epochs* was set to 30 and the Patience was set to 5. Therefore, it was decided to change other parameters in order to get a better performance. The changed ones were mainly: Batch Size, which was set to 32, 64 and 128, and Learning Rate which was set to 0.001 and 0.0001, using a higher learning rate when the batch size increased during the tests. In addition, *Weight Learn Rate Factor* and *Bias Learn Rate Factor* were modified in order to achieve a better performance: the first one was set to two and the latter to five, leading to an increase of 1% in the test accuracy. The biggest improvement derived from the data augmentation, by adding reflection in the pictures of the training set the test accuracy reached the 86%. In the following snippet there are presented the parameter used in this last training:

[1]This is the mean of five different runs

```
options = trainingOptions('adam', ...
    'ValidationData',imdsValidation, ...
    'InitialLearnRate', 0.001, ...
    'MaxEpochs', 30, ...
    'ValidationPatience', 5,...
    'Shuffle', 'every-epoch', ...
    'Verbose',false, ...
    'MiniBatchSize',64, ...
    'ExecutionEnvironment','parallel',...
    'Plots','training-progress')
```

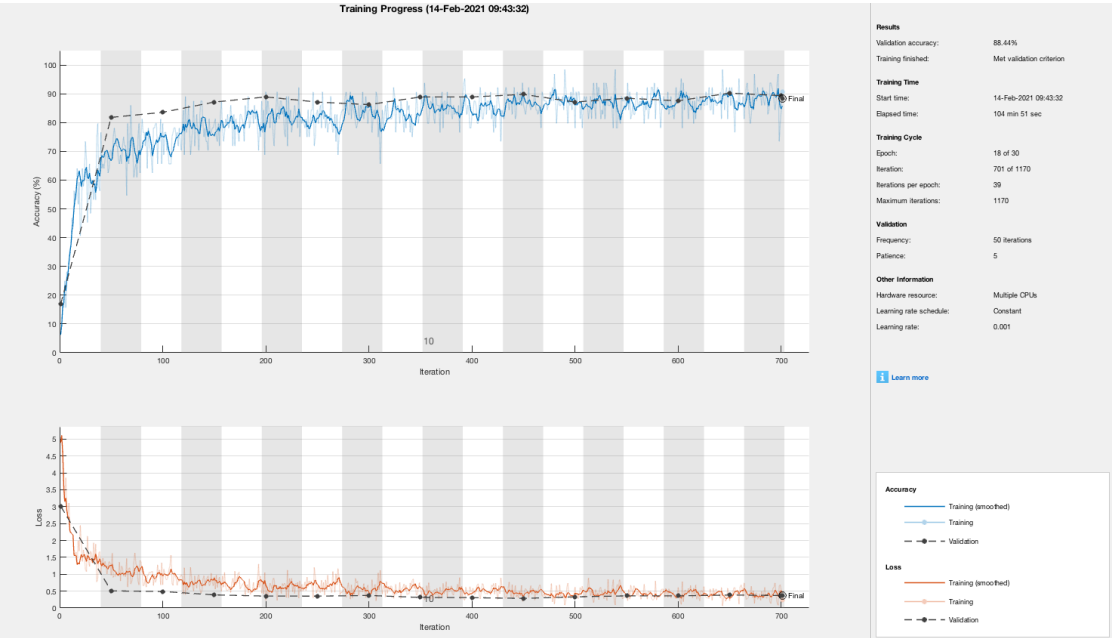

Figure 9: AlexNet with data augmentation

In this case *Living room* is the class that has been misclassified the most with 38% of wrong predictions. Instead, the class that had the best results was *Suburb* with an accuracy of 97.9%.
As suggested, random cropping was also exploited in the train set to get better results but it did not lead to significant improvements with respect to the results obtained using only reflections in the training set.

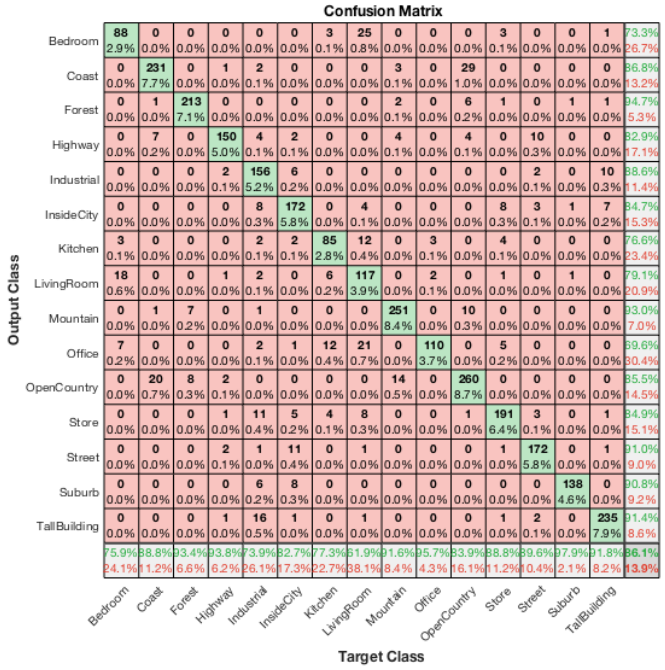

Figure 10: AlexNet: Confusion Matrix

4.2 Employ SVM

Another way to exploit transfer learning is to use the pre-trained network as a feature extractor. In this way a *Support Vector Machine* can be employed to classify the provided pictures by using features as input (the whole training dataset). In order to implement the multiclass SVM it was decided to use the *Error-Correcting Output Codes*. During the first training, we used, as suggested, the last convolutional layer of the network as a feature extractor, leading to results that were better than the ones obtained in *Task 2* but worse than the ones obtained with the first transfer learning approach. In fact, the test accuracy reached the 74% [1]. In order to improve these results, other layers were used as feature extractors, like one of the last fully connected layers. The best results were achieved using the first fully connected layer (Fc6), which has more expressive ability as explained in the article [2]. The final test accuracy is 87% [1].

[1] This is the mean of five different runs

In this case the best predicted class is *Forest* which has been correctly classified in the 96% of the pictures, while the worst predicted class is Living room which has been misclassified in the 25% of the pictures. An interesting thing to point out is that the other classes that have the worst accuracy, apart from *Industrial*, are *Bedroom* and *Kitchen*, which belong to the "House environment". This means that the features extracted from these rooms are similar.

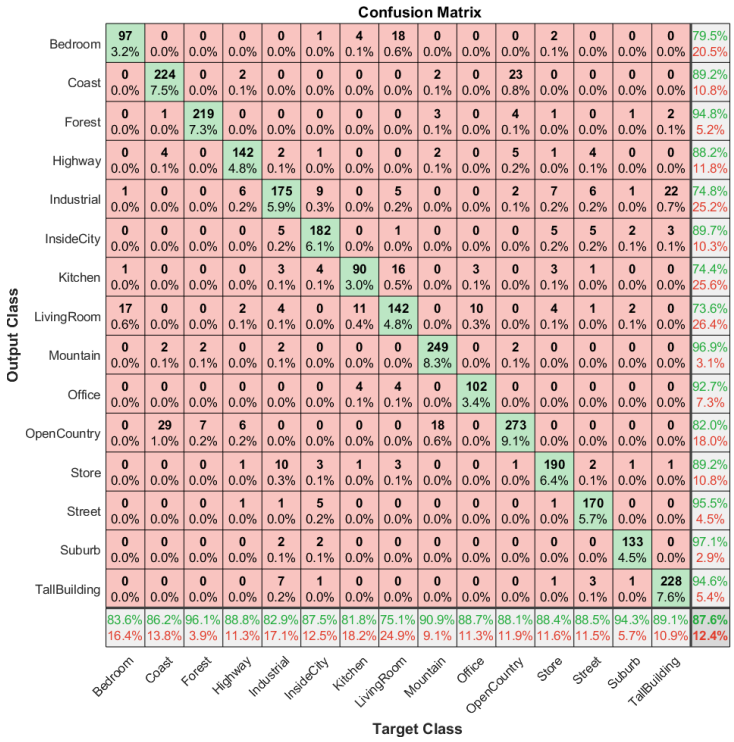

Figure 11: Confusion Matrix for SVM

To conclude this study it was also implemented a *multiclass SVM* with the methodology *One vs All* (file named "MultiSvmOnTransferLearning.m"). The implemented function can be found in the multisvm.m file, the code has been taken from *Multi Class SVM* library [3] that Matlab provide and has been changed in order to make it work with newer functions (svmtrain and *svmclassify* have been removed from 2019a). It was decided to use the *fitcsvm* algorithm. The features are extracted form the first fully connected as it was done before (Fc6).
The final prediction accuracy for the linear SVM is around 79%, so it was decided to run a non linear SVM by using *RBF* and Polynomial kernels (the latter with degree = 2, 3 and 4). The improvement was very low, in fact the test accuracy reached a maximum value of 80% by using *RBF* kernel. The obtained accuracy didn't reach the 85% because the parameters were tuned by hand and it wasn't find the best combination. This tuning can be done automatically with the function *OptimizeHyperparameters* provided by *fitcsvm* but the time for the training has increased a lot (more than ten hours) and the

computers that we used after a while started to work slowly. Figure 12 represents the confusion matrix of the multiclass SVM with RBF kernel:
Also in this case Living room is the class that has been misclassified the most with a prediction accuracy of 51% .

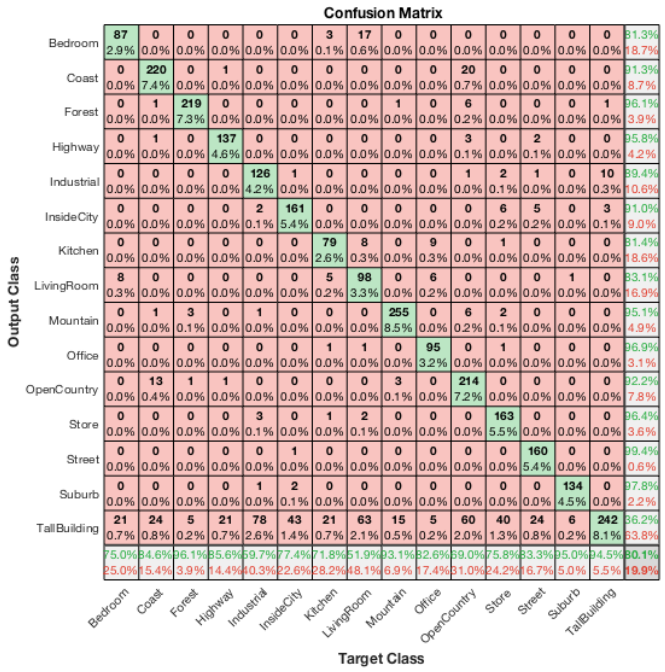

Figure 12: Confusion Matrix for non Linear SVM

5 Conclusions

This project proves once again how much powerful CNNs are in the context of the multiclassification problems. We managed to achieve quite good results even by using a network with a pretty simple structure, which has been optimized with some appropriate techniques. Transfer learning finally allowed us to reach very high values of accuracy and to dispose of a very effective tool for the classification of a specific kind of images.

References

[1] Jean Ponce Svetlana Lazebnik Cordelia Schmid. "Beyond bags of features: spatial pyramid matching for recognizing natural scene categories". In: *IEEE Conference on Computer Vision Pattern Recognition (CPRV '06)* 2 (June 2006), pp.2169–2178.

[2] L. Ding et al. "ALEXNET FEATURE EXTRACTION AND MULTI-KERNEL LEARNING FOR OBJECTORIENTED CLASSIFICATION". In: *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* XLII-3 (Apr. 2018), pp. 277–281. DOI: 10.5194/isprs-archives-XLII-3-277-2018.

[3] MathWorks. *Multi Class SVM*.