Project 3

The aim of this project is to build a multiclass image classifier based on convolutional neural networks for the provided dataset ([**Lazebnik et al.**, **2006**]). 15 categories are present and already divided into trainset (1500 images, 100 per category) and test set (2985 images).

In the first place we build a CNN which act as a baseline (Section 1). Its performance are then compared with the results of some improved CNNs (Section 2) and other classifiers based on pretrained CNNs (Section 3).

1 Baseline

Problem statement

To begin we built a CNN which act as baseline following the given instructions.

Approach

First of all we defined the structurs needed to feed the network with our data.
Specifically we built a new Dataset class which, every time an image is accessed, loads the image itself in memory and applies a given transformation to it.
Then, we splitted the training dataset into train and validation and we construcetd two dataloaders which, at each epoch, take care of dividing data in batches and give them to the network. The same approach is also used for test data.

After having defined the needed structures we built a convolutional neural network following the requests, we trained it and finally tested it.

Implementation choices

In order to obtain images having size 64x64, we defined a transformation, consisting in an anisotropic rescaling, to apply to each image when loaded.

Then, to build the training and the validation dataloaders, we split the given dataset in two parts. The validation dataloader access to the 15% of the images, equally distributed among the classes, and load them all at once. The traning dataloader access the remaining 85% of the data, which are loaded in batches of size 32, as required.

After that we've built the network following the given architecture which is reported in Table 1. Since we've used cross entropy as loss function, and in python this loss already performs softmax, the corresponding layer is not present in the network. Hence the outputs of the network are scores and not probabilities.

As optimizer we've employed the stochastic gradient descent with learning rate 0.01, momentum 0.9, as set by default.

The weights of each layer have been initialized with a gaussian distribution with mean 0 and standard deviation 0.01. The bias of each layer has been set to 0.

Table 1: Structure of the baseline CNN

#	Type	Size
1	Image Input	64x64x1 images
2	Convolution	8 3x3 convolutions with stride 1
3	ReLu	
4	Max Pooling	2x2 max pooling with stride 2
5	Convolution	16 3x3 convolutions with stride 1
6	ReLu	
7	Max Pooling	2x2 max pooling with stride 2
8	Convolution	32 3x3 convolutions with stride 1
9	ReLu	
10	Fully Connected	15
11	Softmax	softmax
12	Classification Output	crossentropyex

By setting the parameters in this way, we were not able to reach a test accuracy of around 30%. Indeed our network performed as a random classifier, having a test accuracy of less then 7%. In particular the specified weights were too high for the last two layers using the default learning rate. To reach a statisfactory accuracy, using the required weight initialization, a smaller learning rate was necessary, so we set it to 0.001.

We employed, as a stopping criterion, the early stopping based on the validation loss which is a measure of the generalization error of our classifier. We've made this choice because, even if we may not reach the global minimum of the validation loss, we can train for fewer epochs stopping when the validation loss significantly increase.
In our specific implementation we evaluate the validation loss every 5 iterations. At each iteration we compare the validation loss with the best one obtained up to that moment. If the current loss is at least 1% higher then the saved one, we increase a counter. On the other hand, if the current loss is lower then the saved one we update it, save the parameters of the network and set the counter to zero. We stop the training when a certain number of epochs are performed or when the counter reach the value of the patience. In our case we have decided to perform at most 30 epochs and set the patience to 20.

Results

We performed the training as specified above and plot the evolution of the accuracy and of the loss on both training and validation data (Figure 1).

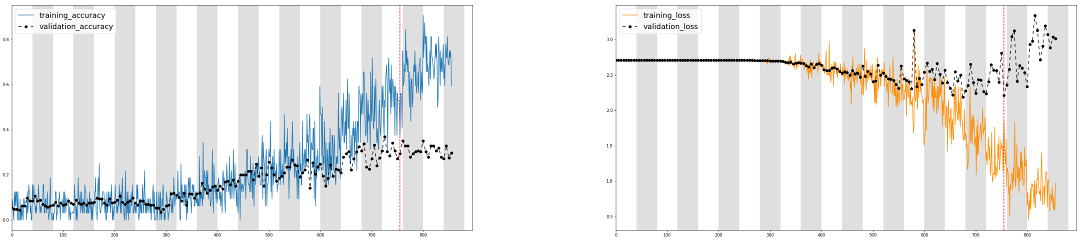


Figure 1: Training and validation accuracy and loss of the baseline

As we can see the training accuracy tends to increase linearly reaching the value of

0.6 before the stop of the training. Looking at the the validation accuracy we notice that, during the first epochs, it takes values similar to the ones of the training accuracy, but after 400 iterations it begins to be a bit lower then the training metric, stabilizing around values of 0.3.
In the same way we can see that the training loss descreases reaching a minimum of around 1.5. On the other hand the validation loss decreases very slowly and, after 14 epochs it begins to increase. Hence the training stopped.

We tested the network on the test set obtaining an accuracy of around 31%. But we've observed a lot of variability in this result: we get values in between 26-32%.

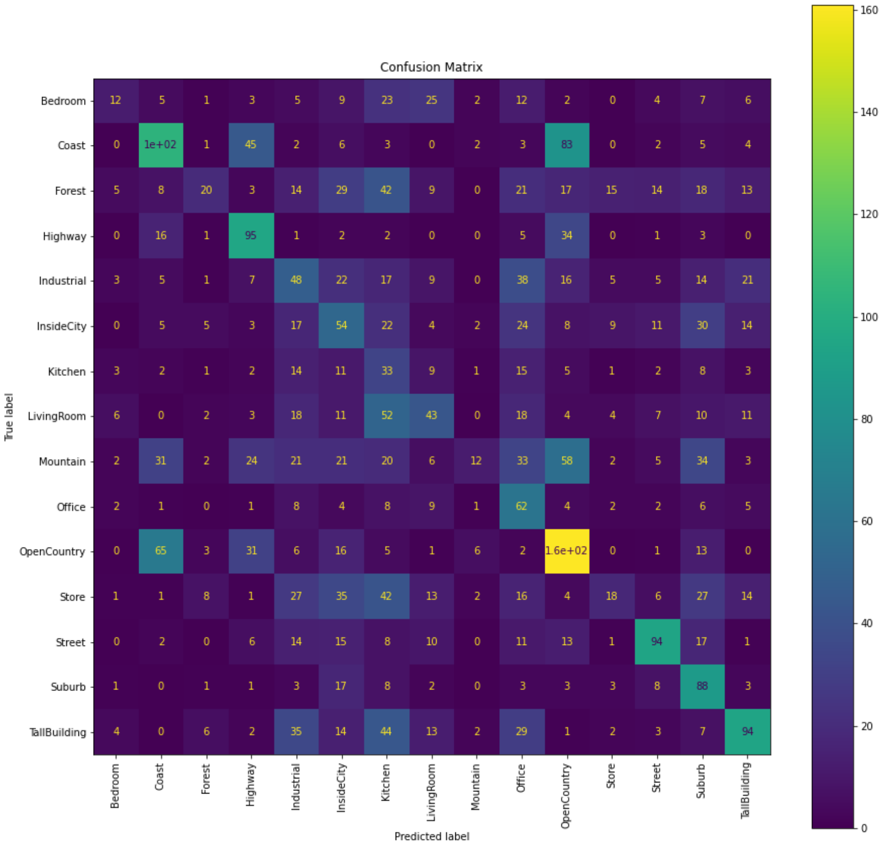


Figure 2: Confusion matrix of the baseline

Looking at the confusion matrix (Figure 2) we notice that there are some misclassifications.

2 CNN improvements

Problem statement

After having developed a CNN which acts as a baseline, we tried to improve the obtained results. In order to do so we applied some transformation to the training data, added some layers and changed some optimizaion parameters. After having found the optimal network we have also build an ensemble of networks.

2.1 Data augmentation

First of all we perform data augmentation. Since the training set is quite small and only contains 1500 images, applying some random transformations on the original data set allows to increase the diversity of the training set itself. In this specific case we applied a horizontal flip to each image in the training set only, while the validation and the test sets remained the same.

We tried two different approach. Firstly we defined a new transformation performing a random horizontal flip with probability 0.5 and we passed it to the trainloader. In this way the transformation is applied with probability 0.5 to each image every time it is accessed. Hence at each epoch the images are augmented but their number does not increase. Neverthelss we didn't obtain the expected results. Therefore we tried another approach which is to say we actually doubeled the training set.

We defined a new transformation always performing the horizontal flip. During the training we applied this transformation to each batch and concatenate the transformed batch to the original one. In this way the dimension of each batch double because half of it are the original images and the other half the flipped ones.

We perform the training keeping the options as for the baseline and we obtain the following results (Figure 3).

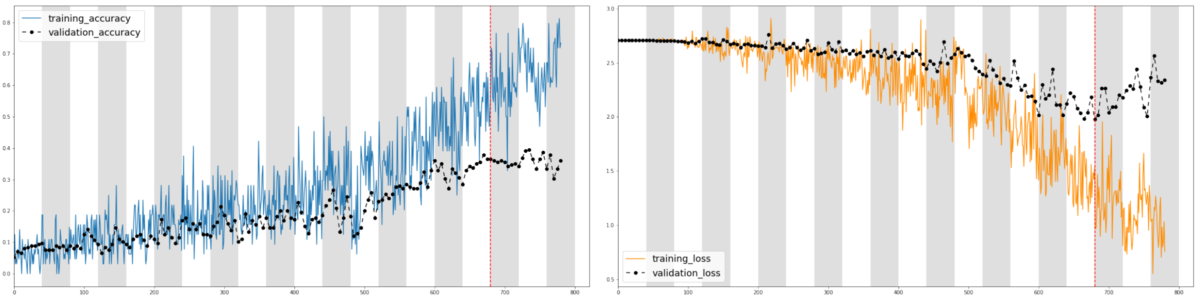


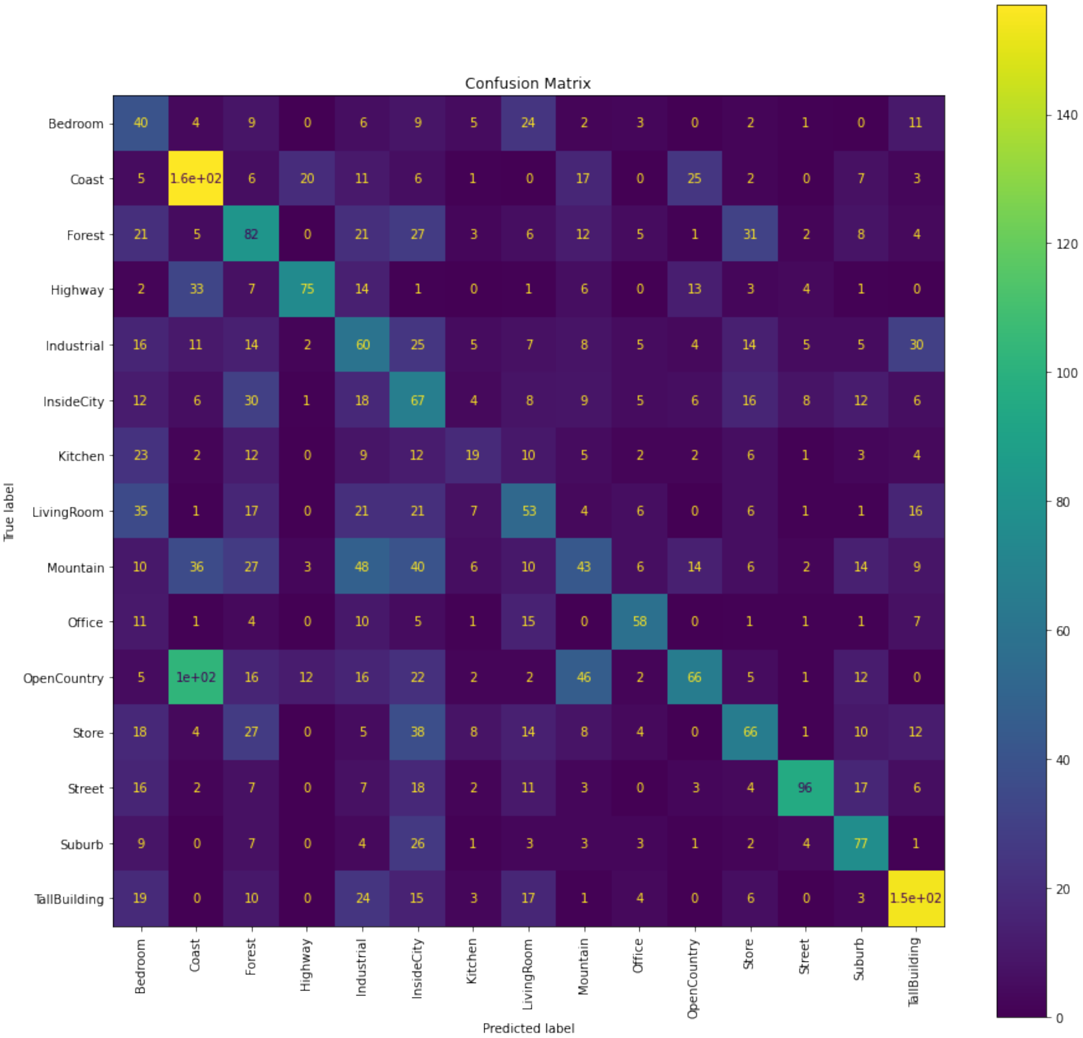
Figure 3: Training and validation accuracy and loss of the basline with data augmentation

We can see that the trend of the metrics is similar to the previous one. Neverthelss all of them seems to perform better. In fact both the training and the validation accuracy are slightly higher then in the baseline whereas the training and the validation loss are a bit lower.

We test our network on the test set and, as expected, we obtained better results.
The test accuracy of this network is around 37%, which is to say almost 7-8% better then the baseline.

Looking at the confusion matrix (Figure 4) we can see that there are still some misclassifiactions. Expecially the network classifies the 'Open County' as 'Coast' quite often. Nevertheless they are both natural environments, so the error is quite understandable, evan a human being could get them wrong.

After this first attempt we decided to perform more data augmentation. Hence we performed a random crop transformation of size $200x200$ to the images of the training set before the resize one. Then, during the training phase, in order to double the batches, we applied, not only the horizontal flip, but also a random rotation of 5 degrees.
We tried to implement those functions by ourself and we obtained a functioning version of each of them. Nevertheless our functions results to be quite inefficient so, in order to



	Bedroom	Coast	Forest	Highway	Industrial	InsideCity	Kitchen	LivingRoom	Mountain	Office	OpenCountry	Store	Street	Suburb	TallBuilding
Bedroom	40	4	9	0	6	9	5	24	2	3	0	2	1	0	11
Coast	5	1.6e+02	6	20	11	6	1	0	17	0	25	2	0	7	3
Forest	21	5	82	0	21	27	3	6	12	5	1	31	2	8	4
Highway	2	33	7	75	14	1	0	1	6	0	13	3	4	1	0
Industrial	16	11	14	2	60	25	5	7	8	5	4	14	5	5	30
InsideCity	12	6	30	1	18	67	4	8	9	5	6	16	8	12	6
Kitchen	23	2	12	0	9	12	19	10	5	2	2	6	1	3	4
LivingRoom	35	1	17	0	21	21	7	53	4	6	0	6	1	1	16
Mountain	10	36	27	3	48	40	6	10	43	6	14	6	2	14	9
Office	11	1	4	0	10	5	1	15	0	58	0	1	1	1	7
OpenCountry	5	1e+02	16	12	16	22	2	2	46	2	66	5	1	12	0
Store	18	4	27	0	5	38	8	14	8	4	0	66	1	10	12
Street	16	2	7	0	7	18	2	11	3	0	3	4	96	17	6
Suburb	9	0	7	0	4	26	1	3	3	3	1	2	4	77	1
TallBuilding	19	0	10	0	24	15	3	17	1	4	0	6	0	3	1.5e+02

Figure 4: Confusion matrix of the baseline with data augmentation

spend less time to train the network we decided to use the altready implemented functions of Python (torchvision package).

The training of the network resulted to be much slower then before. Indeed we increase the number of epochs, performing 50 of them, ad set the patience to 35. With these parameters we obtained good results.

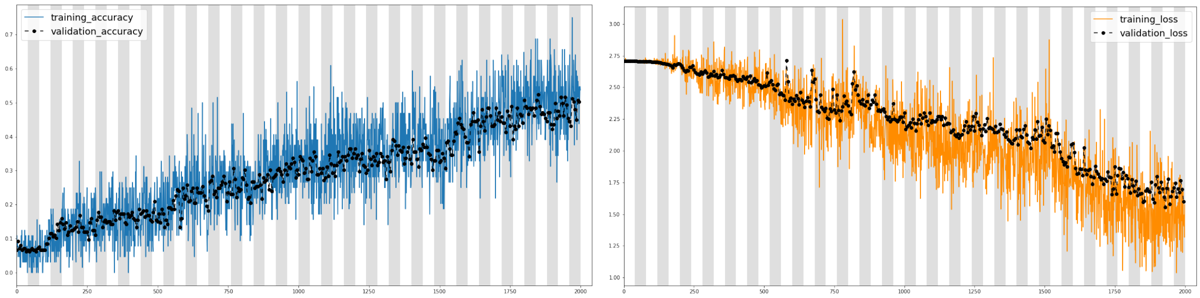
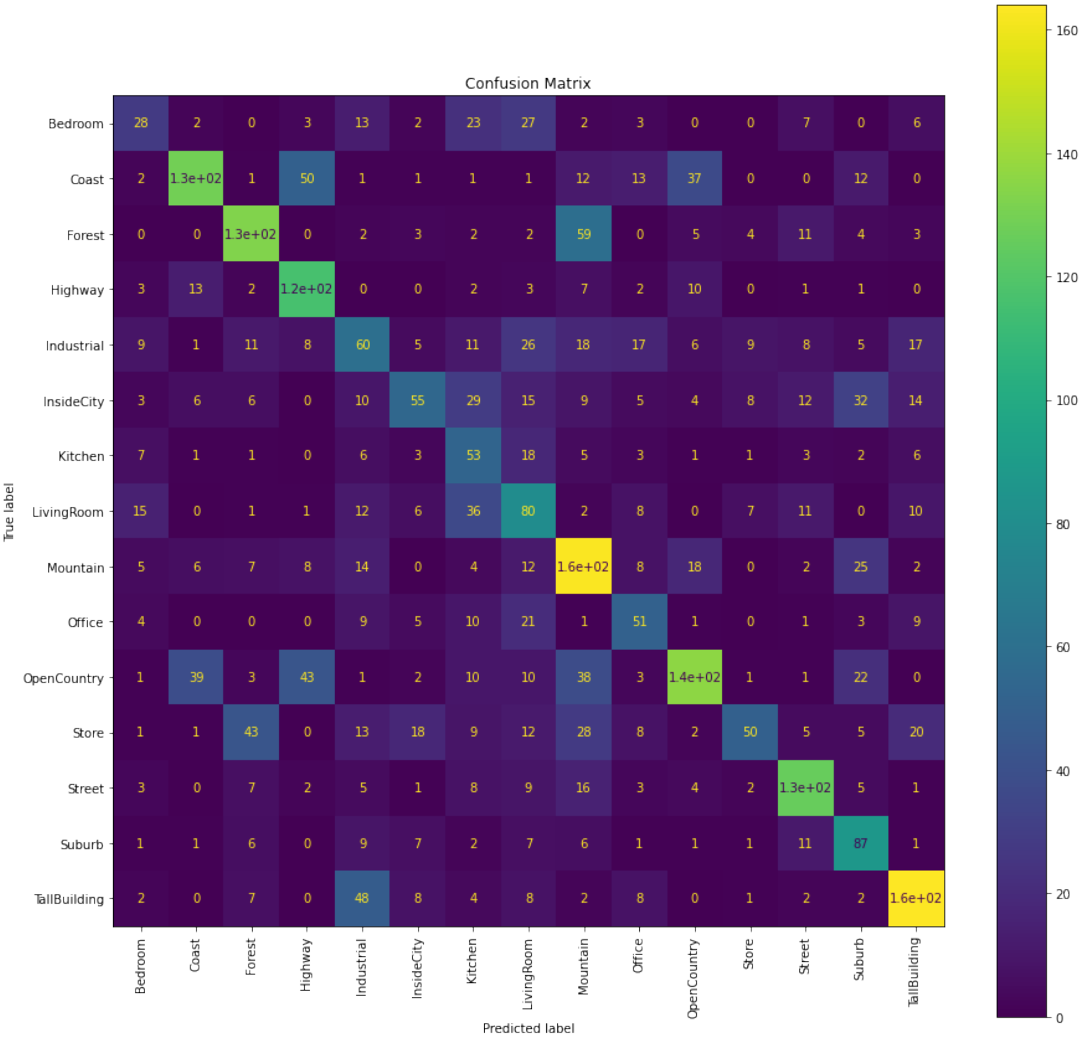


Figure 5: Training and validation accuracy and loss of the basline with data augmentation

As we can see in Figure 5 there seems not to be overfittin. Indeed the validation loss does not significantly increase whereas the validation accuracy tends to grow linearly. Looking at these plots the acheved results seems to be better then before.

We tested the network on the test data and the results confirmed our expectations. We obtained a test accuracy around 45%. Looking at the confusion matrix (Figure 6) we



	Bedroom	Coast	Forest	Highway	Industrial	InsideCity	Kitchen	LivingRoom	Mountain	Office	OpenCountry	Store	Street	Suburb	TallBuilding
Bedroom	28	2	0	3	13	2	23	27	2	3	0	0	7	0	6
Coast	2	1.3e+02	1	50	1	1	1	1	12	13	37	0	0	12	0
Forest	0	0	1.3e+02	0	2	3	2	2	59	0	5	4	11	4	3
Highway	3	13	2	1.2e+02	0	0	2	3	7	2	10	0	1	1	0
Industrial	9	1	11	8	60	5	11	26	18	17	6	9	8	5	17
InsideCity	3	6	6	0	10	55	29	15	9	5	4	8	12	32	14
Kitchen	7	1	1	0	6	3	53	18	5	3	1	1	3	2	6
LivingRoom	15	0	1	1	12	6	36	80	2	8	0	7	11	0	10
Mountain	5	6	7	8	14	0	4	12	1.6e+02	8	18	0	2	25	2
Office	4	0	0	0	9	5	10	21	1	51	1	0	1	3	9
OpenCountry	1	39	3	43	1	2	10	10	38	3	1.4e+02	1	1	22	0
Store	1	1	43	0	13	18	9	12	28	8	2	50	5	5	20
Street	3	0	7	2	5	1	8	9	16	3	4	2	1.3e+02	5	1
Suburb	1	1	6	0	9	7	2	7	6	1	1	1	11	87	1
TallBuilding	2	0	7	0	48	8	4	8	2	8	0	1	2	2	1.6e+02

Figure 6: Confusion matrix of the baseline with data augmentation

can see that the network performs significantly better on some classes, such as Mountain and Open Countri. Nevertheless there are still some misclassification problems.

2.2 Other improvements

Description of the approach

After having performed data augmentation we added other layers to the network and changed the optimization parameters in order to obtain better results.

First of all we added a Batch Normalization layer before each reLu layer. In this way the following transformation is applied to the output of the preceding layer:

$$y = \frac{x - E[x]}{\sqrt{VAR[x]}} * \gamma + \beta$$

The mean $E[x]$ and the standard-deviation $\sqrt{VAR[x]}$ are calculated over the mini-batches and their estimations are then employed during the evaluation phase to normalize the input images. The two parameters γ and β are vectors of size equal to the input size learned during the training. Thanks to the addition of this layer we avoid divergence of the optimizer due to large changes in the input distribution to each layer. It also reduce dependence on the initialization and acts as a regularizer.

We also add a dropout layer before the fully connected one. In this way, during the training phase, some of the outputs of the layer before are randomly set to zero with a given probability. This technique has proven to be effective for regularization and helps in avoiding overfitting.

Then we changed the size of the convolutional filters increasing their support, going from intput to output, to 3x3, 5x5 and 7x7. In order to obtain an output image of the same size of the input one, we added some replicate padding around the image itself. Specifically we set one padding in the first convolutional layer, two in the second and three in the third. In this way the image do not change size going through the network.

Finally we changed some of the optimization parameters.
As first thing we introduced weight normalization. Large values of the network weights can lead to unstable performances and can be an indicator of overfitting of the training data set. To avoid this problem we introduced a regularization term in the objective function which force the network to keep the weight small. In this way the objective function to minimize becomes

$$J(w) = \frac{1}{N}\sum_{i=1}^{n} L(f(x_i, w), y_i) + \lambda R(w)$$

In addition we tried to modify the value of the learning rate. Increasing its the value to 0.01 leads the network to not converge to the minimum because the steps are too big. On the other hand, setting it to smaller values, such as 0.0001, brings to very slow convergence. Hence, since we would like to have a bigger value of the learning rate at the beginning of the training in order to move closer to the minimum, and, after some iterations, a smaller value in order to reach the minimum itself, we introduced a learning rate scheduler. The scheduler decays the learning rate by a given factor every n epochs.
After that we modified the batch size setting it to 64 in order to speed up the computations and obtain a better estimation of the gradient.
As last imporovement we changed the used optimizer switching to Adam optimizer, which is faster than the Stochastic Gradient Descent but is still able to obtain the same performances.

Implementation choices

After having introduced the abouve mentioned additional layers, our improved network had the structure reported in the Table 2.

The objective function we aim to minimize was

$$J(w) = \frac{1}{N}\sum_{i=1}^{n} L(f(x_i, w), y_i) + \lambda R(w)$$

where $L(f(x_i, w), y_i)$ is the Cross-entropy loss function and the regularization term $R(w)$ is the L_2 penalty. We set the λ parameter to 0.02.
Concerning the oprimization parameters we set the initial learning rate to 0.001 and, through the scheduler, it was multiplied by a factor equal to 0.7 every 3 epochs. As already said we used Adam as optimizer.

Results

We trained the network using the early stopping and setting the patience to 10 evaluations. we ran the training and plot the evolution of the accuracy and of the loss (Figure 7).

Table 2: Structure of the final CNN

#	Type	Size
1	Image Input	64x64x1 images
2	Convolution	8 3x3 convolutions with stride 1 and padding replicate 1
3	Batch Normalization	8
4	ReLu	
5	Max Pooling	2x2 max pooling with stride 2
6	Convolution	16 5x5 convolutions with stride 1 and padding replcate 2
7	Batch Normalization	16
8	ReLu	
9	Max Pooling	2x2 max pooling with stride 2
10	Convolution	32 7x7 convolutions with stride 1 and padding replicate 3
11	Batch Normalization	32
9	ReLu	
10	Dropout	p = 0.25
11	Fully Connected	15
12	Softmax	softmax
13	Classification Output	crossentropyex

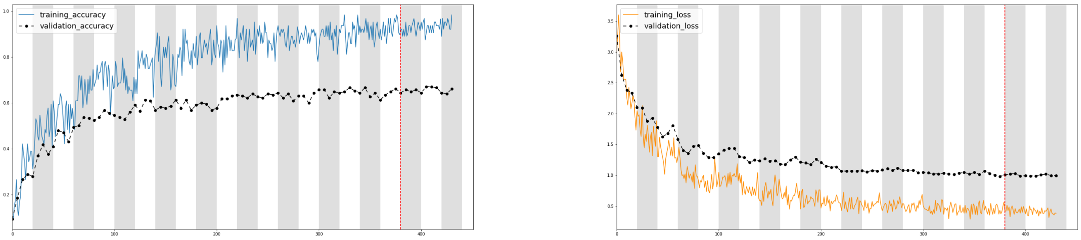


Figure 7: Training and validation accuracy and loss of the improved network

As we can see both the accuracyes and the losses respectivelly increase and decrease much faster then before. Indeed the training accuracy tends to one, while the validation accuracy is stabilized around the value 0.6. In a similar way the training loss tends to zero, while the vaidation loss is stabilized around 1. We also noticed that the training stopped because the maximum number of epochs was reached. Therefore we accomplished to reduce overfitting since the validation loss does not increase significalnty even after 30 epochs.

We test the trained network on the test set and we obtained satisfactory results. Indeed we get a test accuracy of around 65%.

The confusion matrix (Figure 8) confirms the good performances of the network. Even if we still notice some misclassifications expecially among Livingroom and Kitchen and few other classes we are quite satisfied with the obtained result.

Ensemble of networks

Our improved convolutional neural network performed good enough but its results were quite unstable varying between 62% and 67% of test accuracy. Hence, in order to reduce the dependence of the CNN on the initialization of the weights, and so reduce the variance of the predictions, we build an esemble of networks, which is to say we build and train

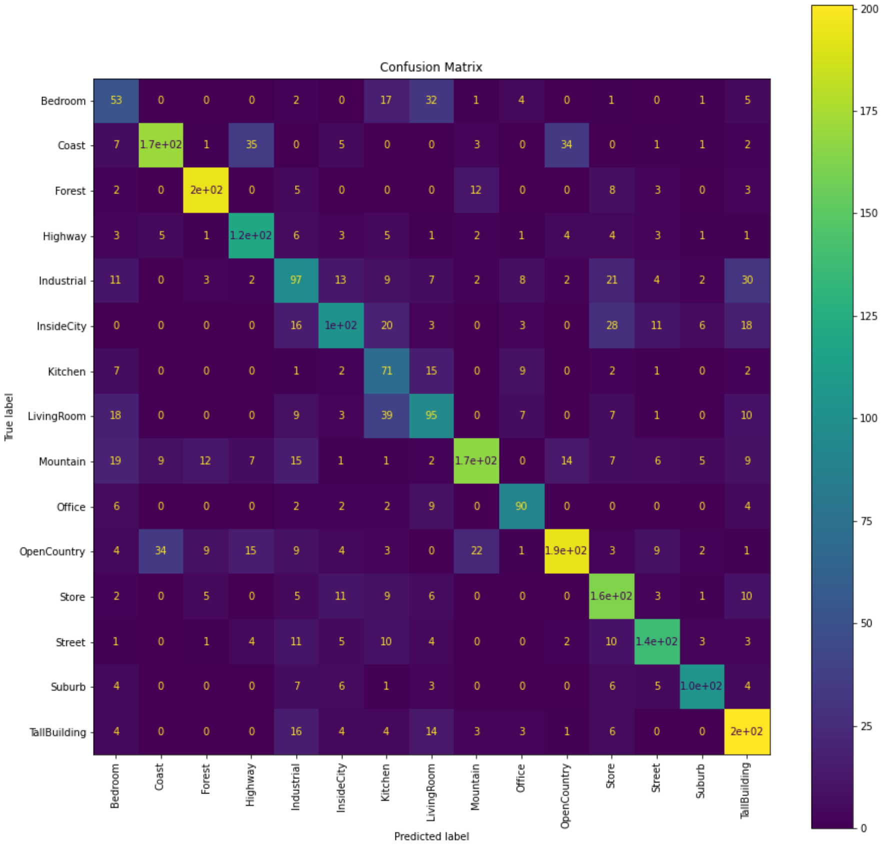


Figure 8: Confusion matrix of the improved network

several models and combine their predictions together.

We initialized five instances of neural network having the structure specified above and trained them independently. After the training phase we test each network on the test set. We apply each network to each image and make a softmax transformation in order to convert the output scores into probabilities. At this point we calculate the average probability of each class computing the arithmetic mean of the output probabilities of the networks. Finally we predict the class having the higher average probability.

Combining the results of independent networks slightly increase our perormances obtaining a of test accuracy of around 68%. Nevertheless the main achievements obtainied in this way was the reduced variability of the results.

3 Transfer learning

3.1 Fine tuning

Problem statement

After having bettered our own neural network we wanted to build a multiclass classifier for our classification problem fine tuning a pretrained convolutional neural network.

Approach

In order to perform fine tuning we took all layers of the chosen pretrained neural network except the last fully connected layer which we substituted with a new one having suitable output size. It was only needed to learn the weights of this last layer freezing all the others. Finally we evaluated the performance of the trained classifier on the test set using the same testing functions as in Section 1.

Implementation choices

We chose to use Alexnet, a convolutional neural network trained on the ImageNet dataset (1.2 million high-resolution images belonging to 1000 categories) proposed by Krizhevsky in 2012 [**4**].

In the Table 3 we can see the architecture of pretrained Alexnet available on pytorch:

Table 3: Structure of Alexnet

#	Type	Size
Features		
1	Image Input	224x224x3 images
2	Convolution	64 11x11 convolutions with stride 4
3	ReLu	
4	Max Pooling	3x3 max pooling with stride 2
5	Convolution	192 5x5 convolutions with stride 1
6	ReLu	
7	Max Pooling	3x3 max pooling with stride 2
8	Convolution	384 3x3 convolutions with stride 1
9	ReLu	
10	Convolution	256 3x3 convolutions with stride 1
11	ReLu	
12	Max Pooling	3x3 max pooling with stride 2
Avgpool		
1	Avg Pooling	output size 6X6
Classifier		
1	Dropout	probability 0.5
2	Fully connected	4096
3	ReLu	
4	Dropout	probability 0.5
5	Fully connected	4096
6	ReLu	
7	Fully connected	1000

It's composed by 5 convolutional layers and 3 fully connected layers. All these trainable layers except for the last fully connected one are followed by ReLu activation function. Moreover there are also 3 maxpooling layers and an adaptive average pooling layer, which calculates the right size of its kernel to obtain the specified output size.

In order to enter this network images need to be preprocessed in the same way the training images have been preprocessed. These are the required transformations:

- the network expects each image to be composed by 3 channels so I use PIL library tools to convert image to RGB (it simply replicates the image 3 times).

- each channel is normalized with vector of means $[0.485, 0.456, 0.406]$ and vector of standard deviations $[0.229, 0.224, 0.225]$.
- resize each image to $[224, 224, 3]$ performing anisotropic rescaling

In order to perform image preprocessing we used the dataloaders used in Section 1, we only needed to change the transformation to be applied when images are loaded.

We substituted the last fully connected layer having output size 1000 with one having output size 15, the number of our classes. In order to train only the last layer we need to pass to the optimizer only the parameters to be learned: weights and bias of this last layer. Then training and test phases are carried on as in Section 1, we still used cross-entropy loss and stochastic gradient descent with momuntum (learning rate 0.001 and momentum 0.9).
Also in this case we initialized the weights using a gaussian distribution with mean 0 and standard deviation 0.01.

Results

We used again early stopping based on validation loss to avoid overfitting and reduce the computational cost, setting the patience parameter to 20. Loss and accuracy on training and validation sets are reported in Figure 9.

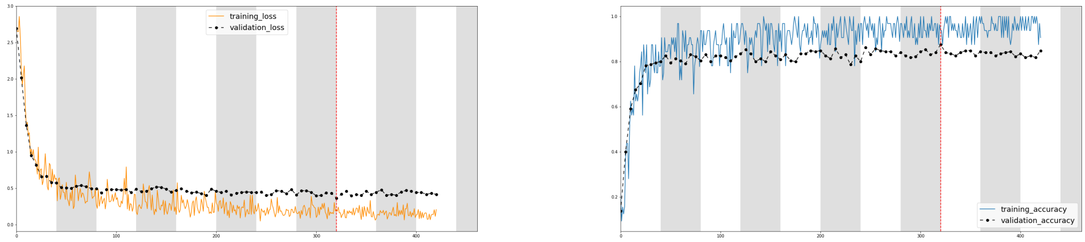


Figure 9: Training and validation accuracy and loss of the finetuned Alexnet

The training stopped after about 8-10 epochs where we get validation loss around 0.36 and validation accuracy around 0.87.
Looking at these plots we can see that, as expected, training loss goes to zero while validation loss reaches a plateau and it starts to slightly increase, so we stop the training. A similar consideration can be done for the accuracy: the training one goes to 1 while the validation one stops increasing after some epochs.

Since the majority of the layers are pretrained, they weights were already optimized to extract features from images and categorize them. Hence we just needed few iterations to tune the last layer's weights. Indeed the loss rapidly decreased and the accuracy rapidly increased during first epochs.

We evaluated the network on the test images and we got a test accuracy of about 0.86. In order to improve the test accuracy of our classifier we tried to perform data augmentation as described in section 2.1. We still obtain a test accuracy around 87%, so we don't get any significant improvement. In Figure 10 the confusion matrices for both cases are reported but they are almost identical.

The majority of the test images are correctly classified, but we still have some misclassifications. It's interesting to see that some images are misclassified in classes which are kind of similar, for example we have some misclassifications among house rooms (bedroom-living room-kitchen) and natural scenes (coast-forest-open country).

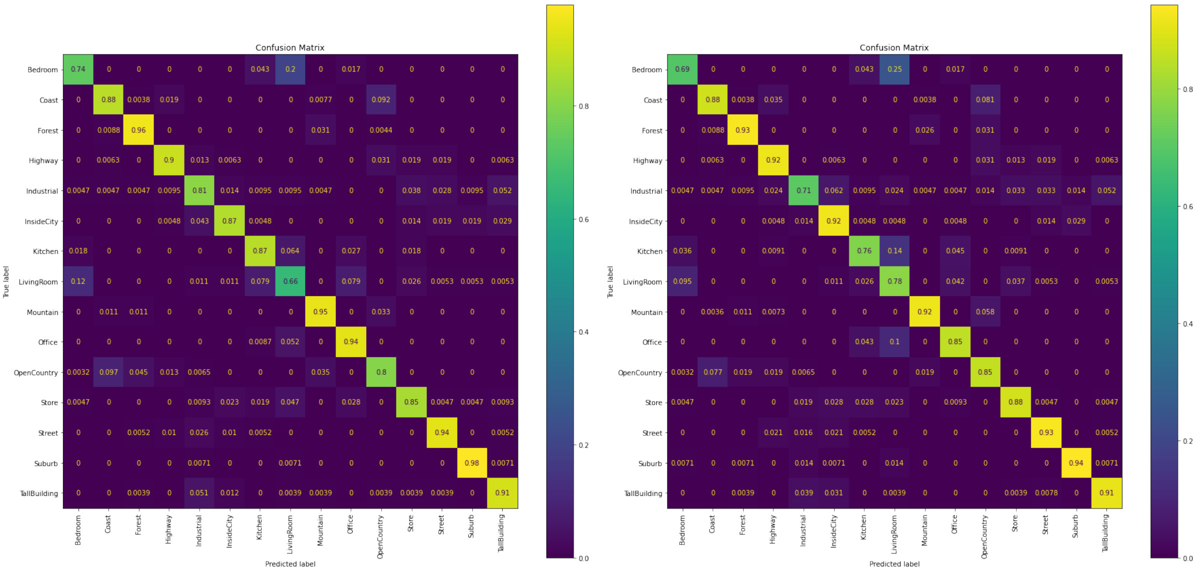


Figure 10: Confusion matrix of the finetuned Alexnet without and with data augmentation

3.2 CNN as feature extractor + SVM

Problem statement

To conclude we used a pretrained convolutional neural network to extract features from an image and then trained a multiclass classifier based on support vector machines.

Approach

The feature vector was extracted accessing the activations of an intermediate layer of the pretrained neural network. The multiclass classification problem was faced using pairwise linear svm with decision tree.

First a classifier for each pair of classes is learned, then each node of the decision tree will perform the classification of an image using one of the trained binary classifiers. The classification will then continue on the left or right subtree where the 'losing' class is excluded. We proceed in this way until we are left with only two possible classes, the predicted class is the output of this last classifier.

Implementation choices

We used again Alexnet to extract a vector of features from each image. In particular we used just the 'feature' layers of the net. the architecture is reported in Table 4.

Feature vectors of length 9216 are obtained and used to train a multiclass support vector machine classifier.

Firstly let's discuss the construction of the decision tree, then we will focus on how to train each binary classifiers.

Decision tree: Each node of the tree will contain the 2 classes to be compared, the function to perform the comparison and 2 pointers to other nodes which will be the left child and right child. It's a good idea to put in the higher part of the tree classes which are easily separable. In order to do so, when we train all the binary classifiers we will also store the number of support vectors, a measure of the generalization capability of the model.

In this way we can start building the tree from the root in a recursive way: we use a specific function that given the list of classes selects the most separable two, builds

Table 4: Structure of the layers of Alexnet used to extract feature vector

#	Type	Size
Features		
1	Image Input	224x224x3 images
2	Convolution	64 11x11 convolutions with stride 4
3	ReLu	
4	Max Pooling	3x3 max pooling with stride 2
5	Convolution	192 5x5 convolutions with stride 1
6	ReLu	
7	Max Pooling	3x3 max pooling with stride 2
8	Convolution	384 3x3 convolutions with stride 1
9	ReLu	
10	Convolution	256 3x3 convolutions with stride 1
11	ReLu	
12	Max Pooling	3x3 max pooling with stride 2

the node and sets as right and left children two nodes built in the same way but operating on the sublists in which one of the two selected classes is removed. The recursive calls terminate when we have to build a node from a list with just two classes. In this case the node will not have any children.

Once the tree is built we need a way to perform a binary classification on each node. In this way an image can go through the tree and be classified. In order to do this each node has an evaluate function which, given the image and the two classes, accesses the correspondent binary classifier and decides if send the image to the left child or to the right child. In the new node it will be evaluated again in the same way until a leaf is reached and we will get the predicted class.

We cannot process a batch of testing images all together because each image will follow a different path and go through 14 classifiers regardless the path. This causes the testing phase to significantly slow down.

The evaluation in each node corresponds to the decision function of a binary classifier. Hence now we need to discuss how to build each svm classifier and the correspondent decision function.

Pairwise svm: Since we need to manage images of just two classes we don't use the training dataloaders as in Section 1 but we load the whole dataset (relatively small), accessing then just the needed images for each classifier. In this way we can avoid to load many times the same images in memory.

The training of the svm classifier consist in solving the following quadratic programming problem (dual formulation):

$$\min_{\alpha} \quad \sum_{i=1}^{l} \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j k(x_i, x_j)$$

$$s.t. \quad \sum_{i=1}^{l} \alpha_i y_i = 0; \qquad 0 \leq \alpha_i \leq C, \forall i = 1, ..., l$$

Thanks to the library cvxopt, by setting the parameter C equal to 1 and deciding which kernel to use we are able to solve this optimization and to find the vector α.

In the first place we work with the linear kernel and we can identify the support vectors as the ones having *alpha* significantly different from 0 ($\geq 1e-6$). The threshold has been chosen empirically looking at the vector α and we use the linear kernel $k(x_i, x_j) =< x_, x_j >$.

For each optimization we store in a matrix the support vectors X^* and the correspondent labels y^* and α^*. Moreover we store the number of support vectors to build the decision tree as described above.

Now we can build the decision function for a new image x:

$$h(x) = \Theta(\sum_{SV} y_i \alpha_i k(x_i, x) + b)$$

where b can be computed from any unbounded support vector in the following way:

$$b = y_i - \sum_{SV} y_j \alpha_i k(x_j, x_i)$$

To avoid bounded support vectors having $\alpha_i \simeq C$ we first chose to use the support vector having the minimum value α_i. But in this way the decision function was not working for some pair of classes (maybe numerical problems because α_i very close to 0). We fixed the problem computing b_i for any support vector and then using the average value.

Now we are able to train all the pairwise svm, build the decision tree and evaluate the classifier on the test set.

Results

Using the classifier with the linear kernel gives us a test accuracy of about 87%.

The confusion matrix (Figure 11) is very similar to the one obtained finetuning Alexnet, also the classes which are misclassified the most are the same.

We've also tried to implement the SVM classifier using a different kernel. In particular we will present the results using a polynomial kernel of degree 2 and 3, and a gaussian kernel.

Polynomial kernel In this case we substitute the linear kernel in both training and decision functions with the polynomial one having the form:

$$k(x, x') =< x, x' >^d$$

We tried with $d = 2$ and $d = 3$. The obtained test accuracies are respectively 0.86% and 0.83%. In Figure 12 the confusion matrices are reported.

Gaussian kernel The used kernel has now the form:

$$k(x, x') = exp\{-\frac{||x - x'||^2}{c}\}$$

In this case we obtained a test accuracy of about 87%, the confusion matrix is reported in Figure 13.

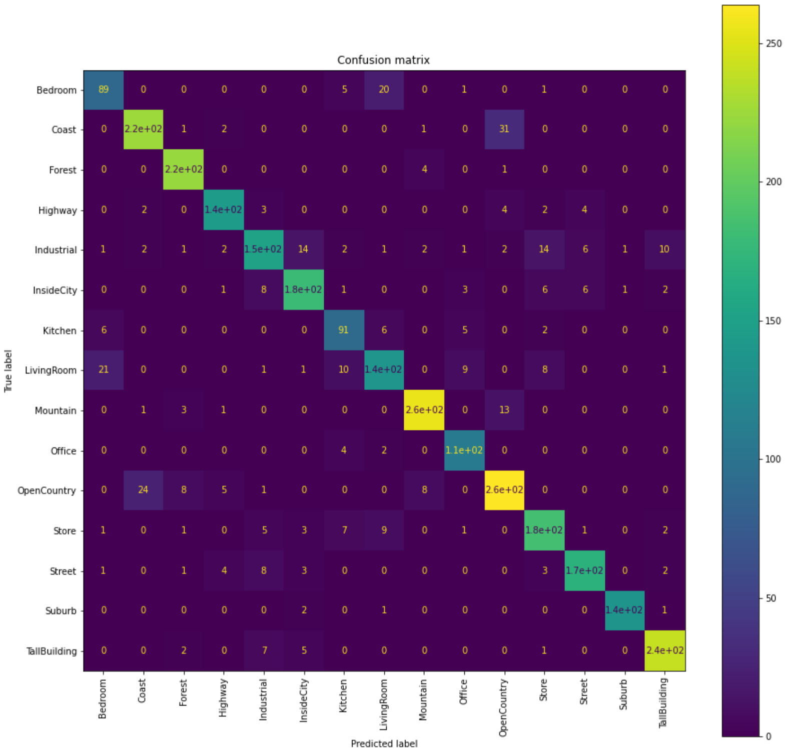


Figure 11: Confusion matrix of the multiclass linear SVM classifier

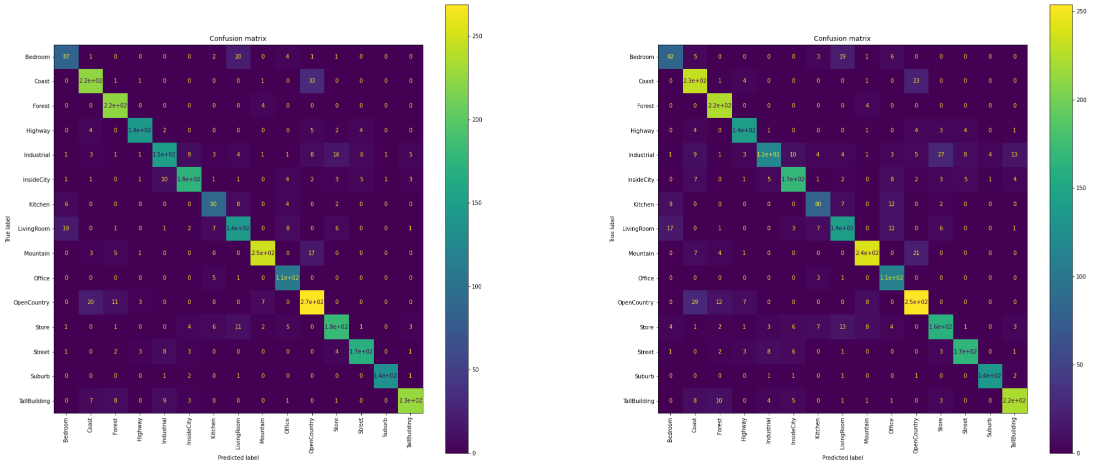
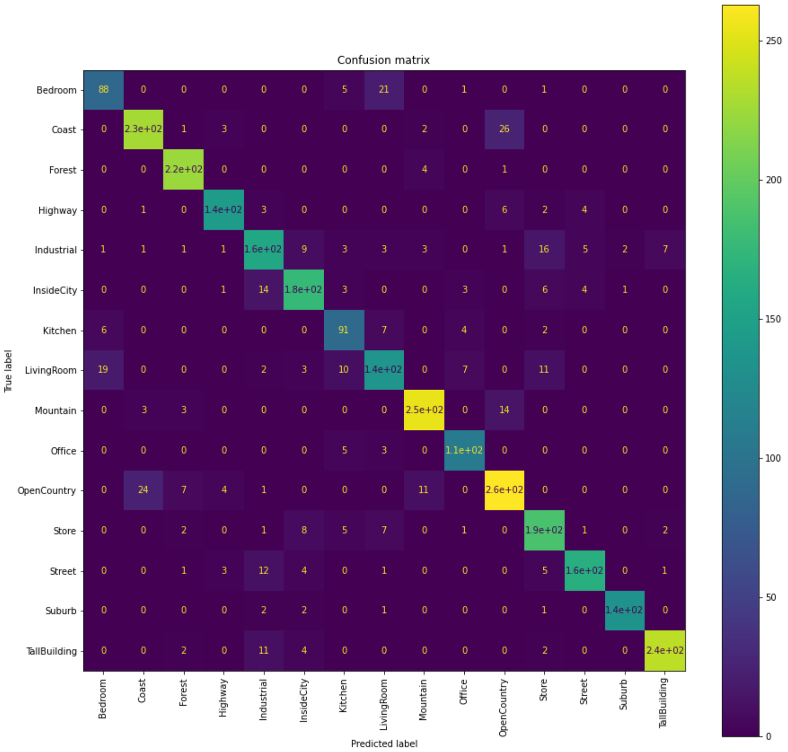


Figure 12: Confusion matrices using polynomial kernel of degree 2 and 3

The hardest part of using different kernels was to play with parameters to find configuration given better performance, and probably we haven't find yet the best values. Using polynomial kernel we get a lower accuracy with respect to the linear kernel one, which makes us think that we are overfitting. But also using a lower value for C (which should give a larger generalization capability) we get the same results.



	Bedroom	Coast	Forest	Highway	Industrial	InsideCity	Kitchen	LivingRoom	Mountain	Office	OpenCountry	Store	Street	Suburb	TallBuilding
Bedroom	88	0	0	0	0	0	5	21	0	1	0	1	0	0	0
Coast	0	2.3e+02	1	3	0	0	0	0	2	0	26	0	0	0	0
Forest	0	0	2.2e+02	0	0	0	0	0	4	0	1	0	0	0	0
Highway	0	1	0	1.4e+02	3	0	0	0	0	0	6	2	4	0	0
Industrial	1	1	1	1	1.6e+02	9	3	3	3	0	1	16	5	2	7
InsideCity	0	0	0	1	14	1.8e+02	3	0	0	3	0	6	4	1	0
Kitchen	6	0	0	0	0	0	91	7	0	4	0	2	0	0	0
LivingRoom	19	0	0	0	2	3	10	1.4e+02	0	7	0	11	0	0	0
Mountain	0	3	3	0	0	0	0	0	2.5e+02	0	14	0	0	0	0
Office	0	0	0	0	0	0	5	3	0	1.1e+02	0	0	0	0	0
OpenCountry	0	24	7	4	1	0	0	0	11	0	2.6e+02	0	0	0	0
Store	0	0	2	0	1	8	5	7	0	1	0	1.9e+02	1	0	2
Street	0	0	1	3	12	4	0	1	0	0	0	5	1.6e+02	0	1
Suburb	0	0	0	0	2	2	0	1	0	0	0	1	0	1.4e+02	0
TallBuilding	0	0	2	0	11	4	0	0	0	0	0	2	0	0	2.4e+02

Figure 13: Confusion matrix using gaussian kernel $c = 10^5$

Also using the gaussian kernel we are not able to improve the performance of the classifier. Regardless the choice of the kernel, the classes which are misclassified the most remain always the same, indicating that with none of these kernels we are able to properly separate them. Possible improvements for this approach is to tune the parameters of the kernel for each pairwise classifier, since up to now we've used the same values for all the binary classifiers.

Replicability

Our project has been developed in Python 3.6.9. In order to be able too replicate the results, we report the exact version of each package we used.

- numpy 1.19.5
- torch 1.7.0
- torchvision 0.8.1
- PIL 7.0.0
- sklearn 0.22.2
- matplotlib 3.2.2
- IPython 5.5.0
- cvxopt 1.2.5

Bibliography

[1] Lazebnik, S., Schmid, C., and Ponce, J. *Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories* In 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06), volume 2, pages 2169–2178.

[2] Ioffe, S. and Szegedy, C. *Batch normalization: Accelerating deep network training by reducing internal covariate shift* 2015, arXiv preprint arXiv:1502.03167

[3] *Going deeper with convolutions* Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). In The IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

[4] Krizhevsky, A. Sutskever, I., and Hinton, G E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097-1105