Computer Vision

Maria Grazia Berni

Implementation of an image classifier based on convolutional neural networks

First Point

In this project I implemented an image classifier using the Python programming language and the Pytorch framework. The image classifier is based on a neural network with the following structure:

#	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

In Pytorch there's not a way to explicitly constrain the input size, but the Fully Connected layer, which has size 32*12*12, is such that the network fits only on image of size 1*64*64. In my implementation of the network structure the Softmax is missing because in Pytorch, when using the Cross Entropy Loss, this is automatically implemented and it would be a mistake to explicitly implement it. In the network I implemented a function to initialize the weights of the layer from a Gaussian distribution with mean 0 and standard deviation 0.01 and the biases to 0. In the module "utility.py" I implemented the class My _Dataset, which inherits from Dataset, and it is meant to feed the network with the proper format of data and to apply some transformation to the data, like resizing and normalizing it. After splitting the data set in train and validation data, I trained the network for 40 epochs. To do this I tried different learning rates, and batch sizes of 32. In Pytorch, using the optimizer SGD, the default value of the momentum is zero, and this configuration is reach without putting any value for the momentum.

Here are the plots of the training accuracy and of the loss related to this model, using a learning rate of 0.01:

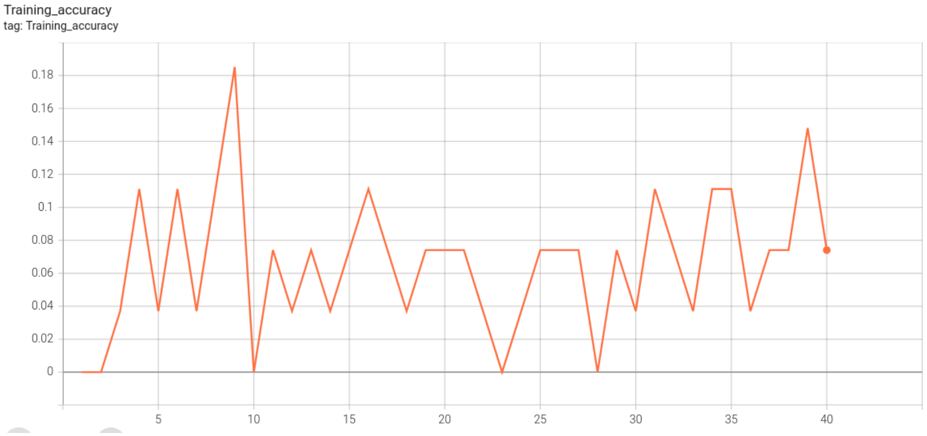


Figure: Running Accuracy during the 40 epochs of training

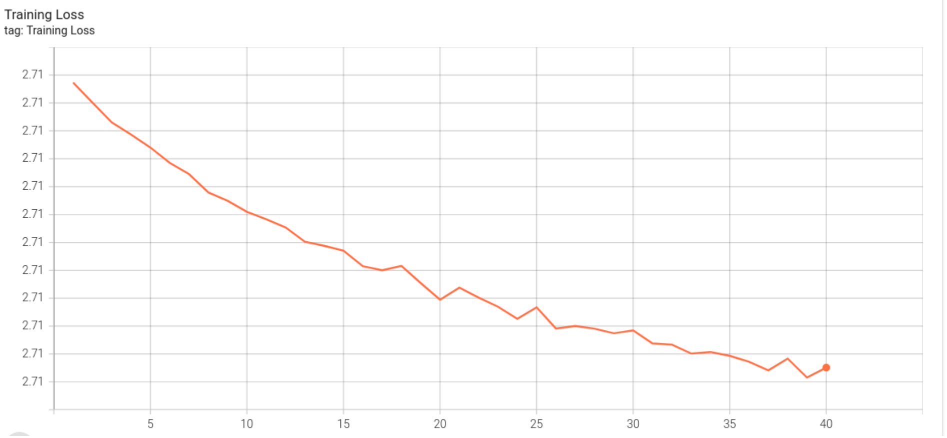
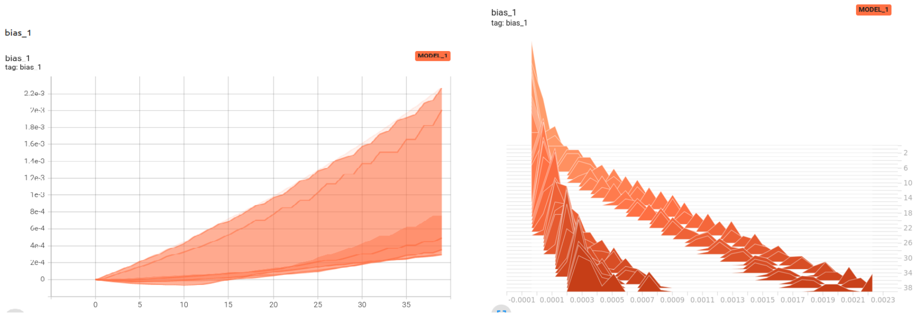


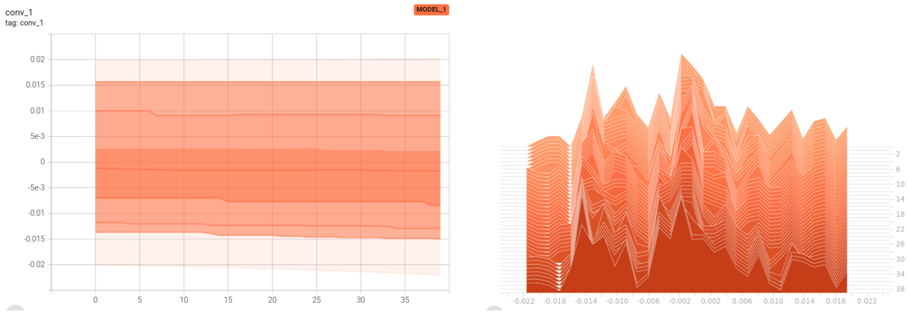
Figure: Loss during the 40 epochs of training

The trend of the accuracy shows that the model is not improving during the training, the loss slightly decreases but to this is not associated any improvement in the accuracy. The same result is obtained for greater values of the learning rate, and even worse when the learning rate is decreased. Analyzing the values predicted by the network, I noticed that it tends to predict the only one value for all the data in the batch with which is dealing, and this value is the most frequent of the batch. When predicting without training, so in the $model.eval()$ setting of Pytorch, the network predicts the value that was most frequent in the last batch of the training. I observed the same behavior even using a weighted cross entropy loss. Analyzing the weights of the network layers I found out that this happens because during the training only the biases change, while all the other weights remain practically constant, as can be seen from the plots below:

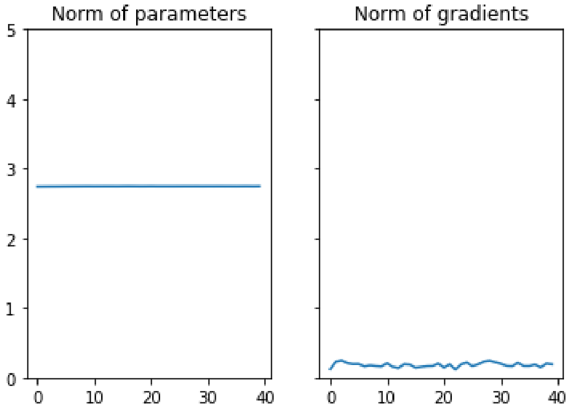
Distribution of biases of the first convolutional layer during training



Distribution of weights of the first convolutional layer during training

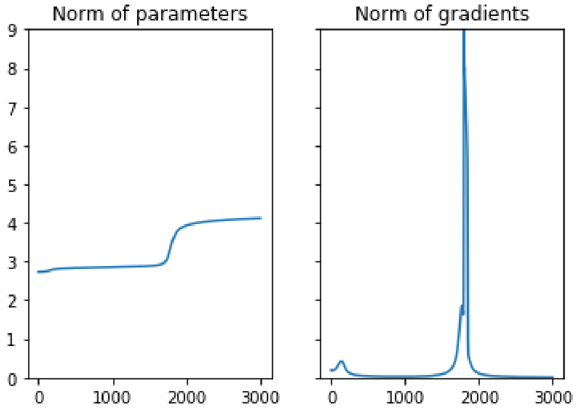


Norm of weights and of the gradients during training



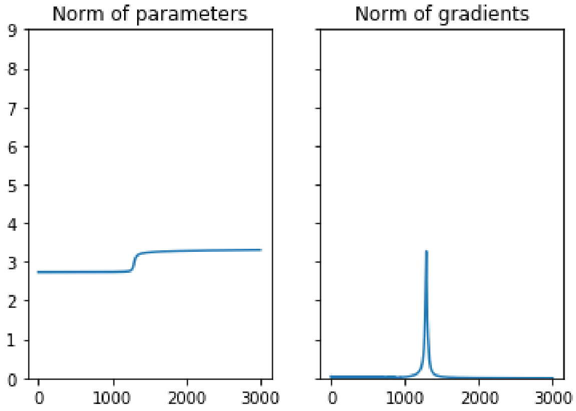
So I evaluated if the network was capable to overfit a single batch (always of size 32). I observed that, considering the small size of the input, only after a huge number of epochs the loss of the model starts decreasing, and this happens when the weight starts to change. Also if the batch to be overfitted is not balanced it may happen that the weights do not change and the net behaves as described above. Furthermore, if the weights finally change there is gradient explosion:

Norm of weights and of the gradients during the overfitting of a single batch



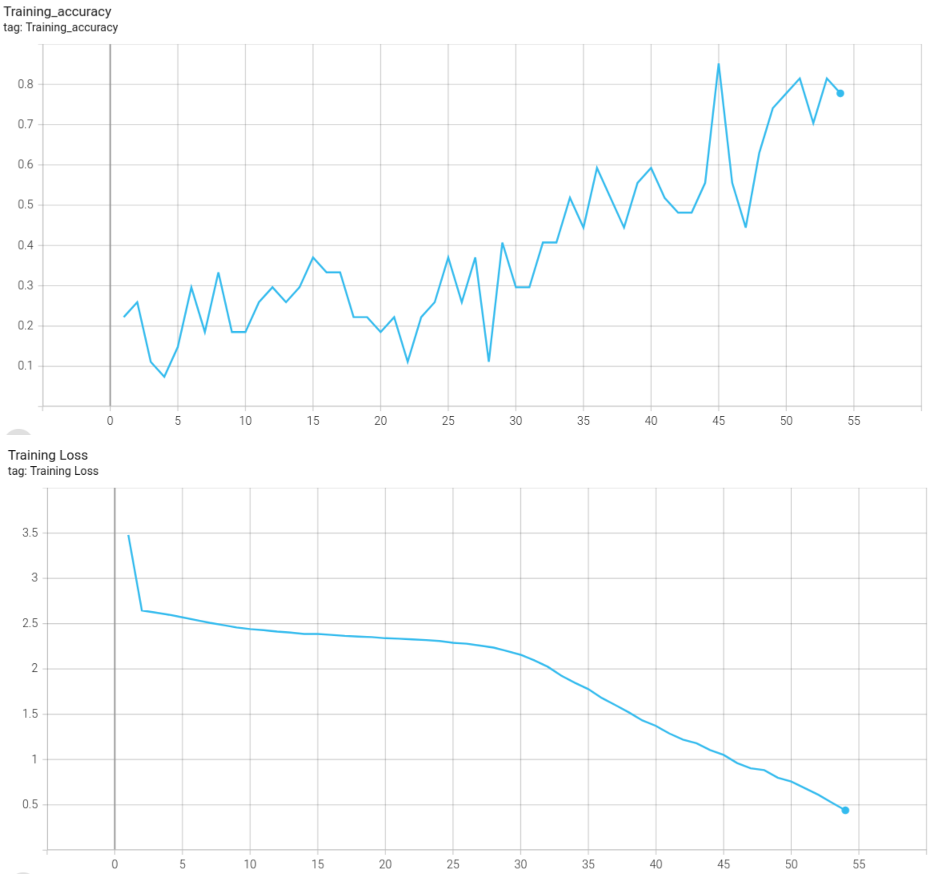
I have not encountered this behavior using the Adam optimizer or, with the classical SGD, using an initial weight distribution with a larger standard deviation, for instance 0.1. To overcome this problem I made the network overfit a single batch of one instance for class. Almost the same result is obtained overfitting a single batch using weighted cross entropy loss.

Norm of weights and of the gradients during the overfitting of a single balanced batch

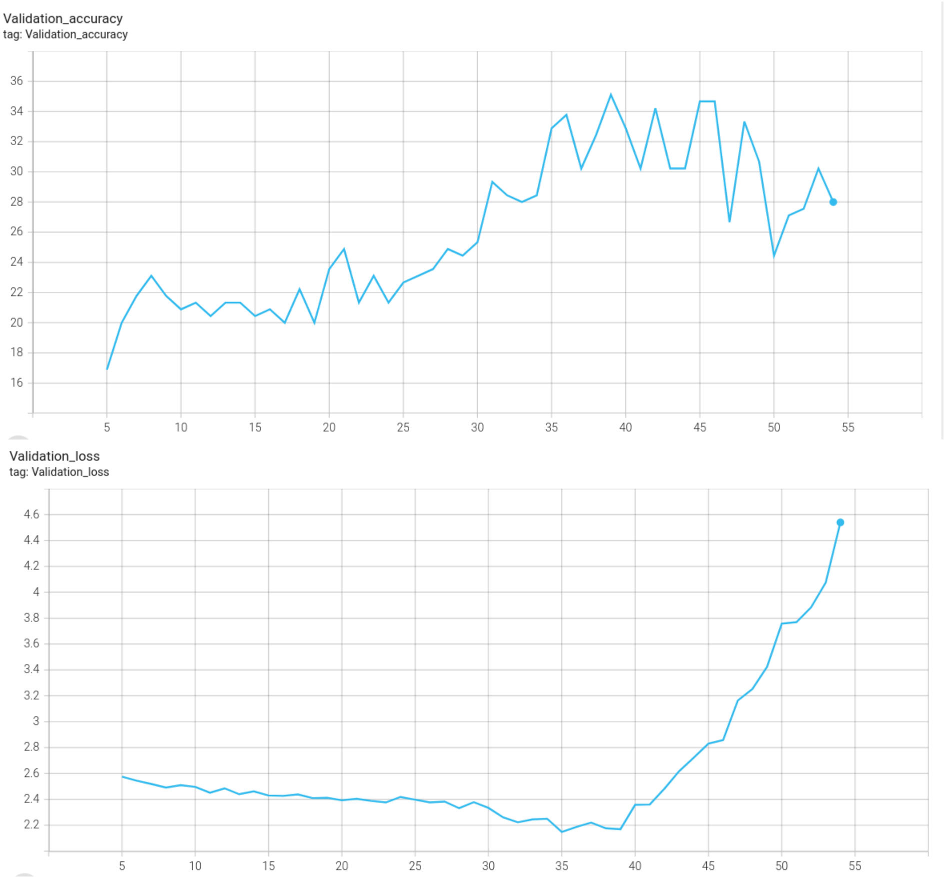


This trick can be considered as a kind of second initialization of the weights. Then I trained the model on the whole train data set, just using the simple Cross Entropy Loss and therefore without applying any type of balancing on the train data. In order to avoid overfitting, I used early stopping to train the model, writing a proper function (the method is not implemented in the Pytorch framework). The model is first trained for an initial low number of epochs, 5 in this case, and then after each epoch of training it is tested on a validation dataset, if the validation accuracy increases this is repeated for another epoch and so on, if the validation does not increase and it does not increase for a fixed number of attempts, which is the patience of the model (15 in this case), the training stops and the function returns the number of epochs that match the best validation accuracy. During the validation procedure I used a batch size of 64, this does not have any impact on the results of the model and the only "crucial" thing is to find a value that fits well on the gpu (in the case this is present). In my implementation the model is also saved in a file, in order to avoid training it again.

Training accuracy and loss of the model

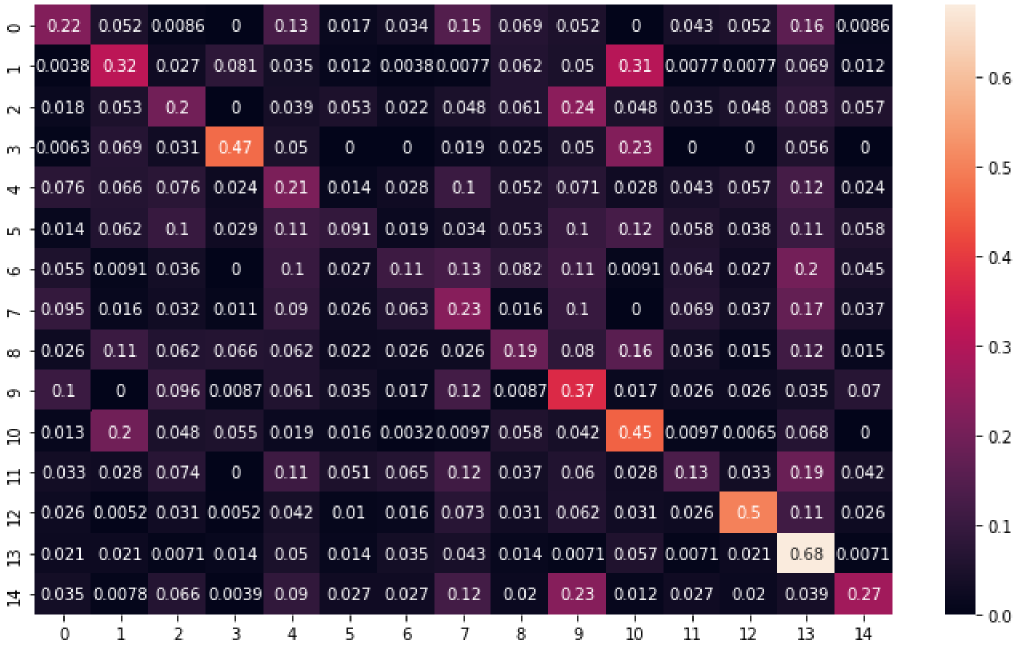


Accuracy and loss of the validation dataset during the training



The model reaches the highest value of the validation accuracy after 39 epochs of training. The training lasts 54 epochs because after reaching this value there are 15 attempts to improve it. After 39 epochs the loss on the validation set starts to increase, while the validation accuracy fluctuates a bit and slightly descends, so, after this point the model starts to overfit the train dataset, and this is clear form the plot of the training accuracy, which is always increasing and the loss is going down to zero. The model saved with the early stopping technique is then tested on the test dataset and it reaches an accuracy of $29,28\%$.

Confusion Matrix computed on the test set



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	0.22	0.052	0.0086	0	0.13	0.017	0.034	0.15	0.069	0.052	0	0.043	0.052	0.16	0.0086
1	0.0038	0.32	0.027	0.081	0.035	0.012	0.0038	0.0077	0.062	0.05	0.31	0.0077	0.0077	0.069	0.012
2	0.018	0.053	0.2	0	0.039	0.053	0.022	0.048	0.061	0.24	0.048	0.035	0.048	0.083	0.057
3	0.0063	0.069	0.031	0.47	0.05	0	0	0.019	0.025	0.05	0.23	0	0	0.056	0
4	0.076	0.066	0.076	0.024	0.21	0.014	0.028	0.1	0.052	0.071	0.028	0.043	0.057	0.12	0.024
5	0.014	0.062	0.1	0.029	0.11	0.091	0.019	0.034	0.053	0.1	0.12	0.058	0.038	0.11	0.058
6	0.055	0.0091	0.036	0	0.1	0.027	0.11	0.13	0.082	0.11	0.0091	0.064	0.027	0.2	0.045
7	0.095	0.016	0.032	0.011	0.09	0.026	0.063	0.23	0.016	0.1	0	0.069	0.037	0.17	0.037
8	0.026	0.11	0.062	0.066	0.062	0.022	0.026	0.026	0.19	0.08	0.16	0.036	0.015	0.12	0.015
9	0.1	0	0.096	0.0087	0.061	0.035	0.017	0.12	0.0087	0.37	0.017	0.026	0.026	0.035	0.07
10	0.013	0.2	0.048	0.055	0.019	0.016	0.0032	0.0097	0.058	0.042	0.45	0.0097	0.0065	0.068	0
11	0.033	0.028	0.074	0	0.11	0.051	0.065	0.12	0.037	0.06	0.028	0.13	0.033	0.19	0.042
12	0.026	0.0052	0.031	0.0052	0.042	0.01	0.016	0.073	0.031	0.062	0.031	0.026	0.5	0.11	0.026
13	0.021	0.021	0.0071	0.014	0.05	0.014	0.035	0.043	0.014	0.0071	0.057	0.0071	0.021	0.68	0.0071
14	0.035	0.0078	0.066	0.0039	0.09	0.027	0.027	0.12	0.02	0.23	0.012	0.027	0.02	0.039	0.27

The diagonal elements of the confusion matrix are the accuracy per each class. This vary a lot from class to class reaching the 68% for the class 13, which is the Suburb, while only $9,1\%$ for class 5, InsideCity, which is one of the classes that the model confuses with all the others.

Second Point

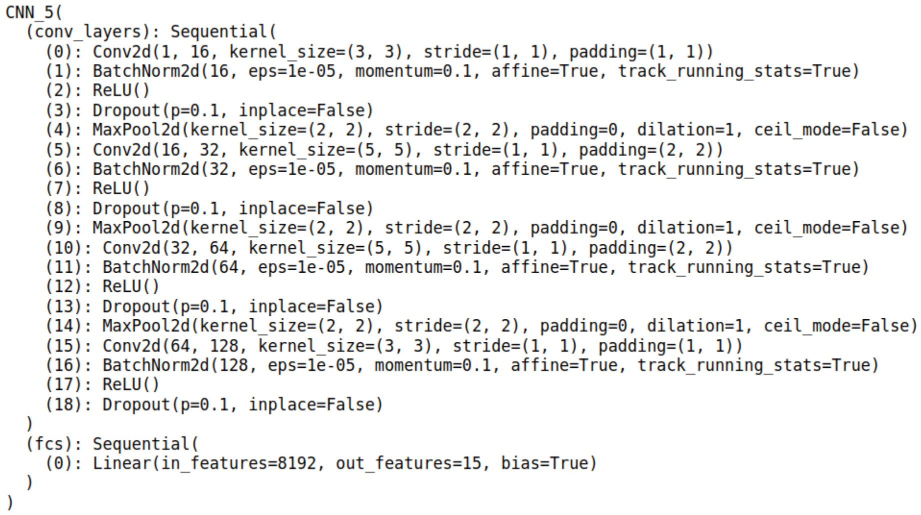
First of all I switched to the Adam optimizer, which is able to quickly overfit a single batch without special precautions. Then I restored the initial weights of the model in order to erase the effects of this first overfitting procedure and trained the model using early stopping and a learning rate of 0.001. So the only change in the optimizer led the model to a test accuracy of 36.28% Playing with the batch sizes for the model training, I found that the best value is 32 in fact many data-scientists consider this as the perfect dimension to provide the right amount of stochasticity. Then I augmented the training set using the class Augmented_Dataset in the module utility.py to implement the dataset. Only the training dataset is augmented in this way:

- The training set is initially doubled cropping all the images in the train-set
- The training set is tripled horizontally flipping all the original images in the train-set

- All the original images of the train set are consequently fisrt horizontally flipped and then cropped. This because I always crop in the same way.

After training the model using the augmented dataset, always with early stopping, it reached a test accuracy of 47.94%. Adding batch norm before the reLU layers the test accuracy increase to the 57.45%. Increasing the sizes of the kernels and the number of channels, having in the first convolutional layer a kernel of size $(3, 3)$ and 16 channels, in the second convolutional layer a kernel of size $(5, 5)$ and 32 channels and in the last convolutional layer 64 channels with kernel size $(7, 7)$, all having padding 1 the model reached a test accuracy of 61.87%. Doubling the number of channels another time the test-accuracy decreases. Adding dropout the test accuracy slighly increases only when using a probability of "detaching" the parts of the net equal to 0.1. If early stopping was important before in the training of the model, it is even more important when using dropout.

Finally I used a model with 4 convolutional layers with number of channels respectively $16, 32, 64, 128$, kernel sizes $(3, 3), (5, 5), (5, 5), (3, 3)$ and setting the padding such that the size of the data is preserved during the action of the kernel, using dropout, batch-normalization and the augmented trainset the model reached the test accuracy of 65.13%.

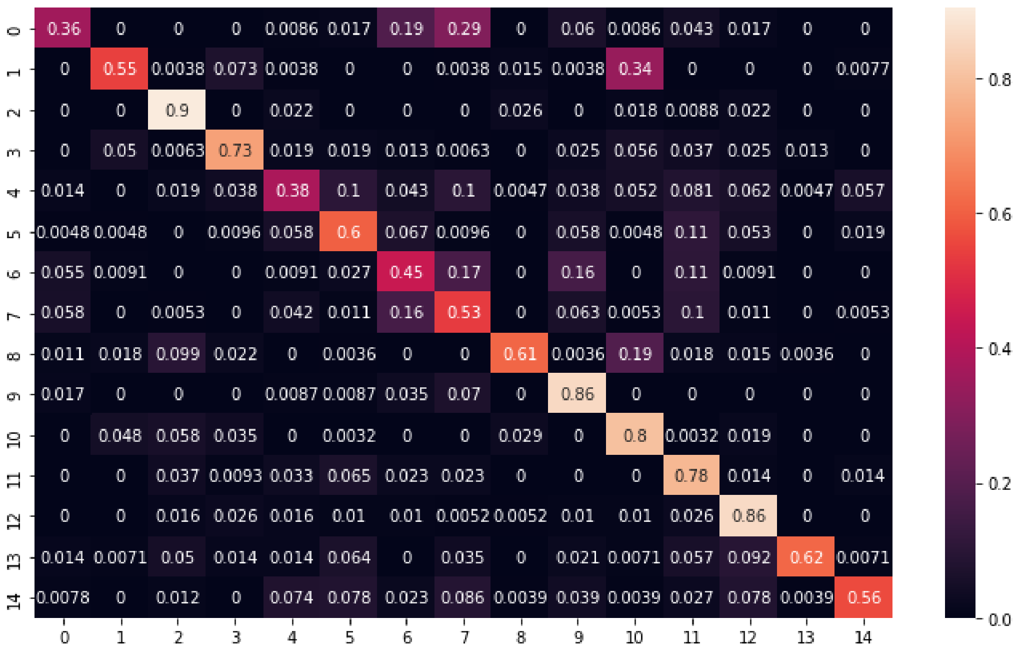


```
CNN_5(
  (conv_layers): Sequential(
    (0): Conv2d(1, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): Dropout(p=0.1, inplace=False)
    (4): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(16, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (6): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): ReLU()
    (8): Dropout(p=0.1, inplace=False)
    (9): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (11): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (12): ReLU()
    (13): Dropout(p=0.1, inplace=False)
    (14): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0, dilation=1, ceil_mode=False)
    (15): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (16): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (17): ReLU()
    (18): Dropout(p=0.1, inplace=False)
  )
  (fcs): Sequential(
    (0): Linear(in_features=8192, out_features=15, bias=True)
  )
)
```

Figure: Structure of the model with higher accuracy

Adding another layer the test accuracy remarkably decreases, in fact the gradient back-propagate more difficult in the layers. Here the confusion matrix computed on the test data using the best performing model found:

Confusion Matrix computed on the test set

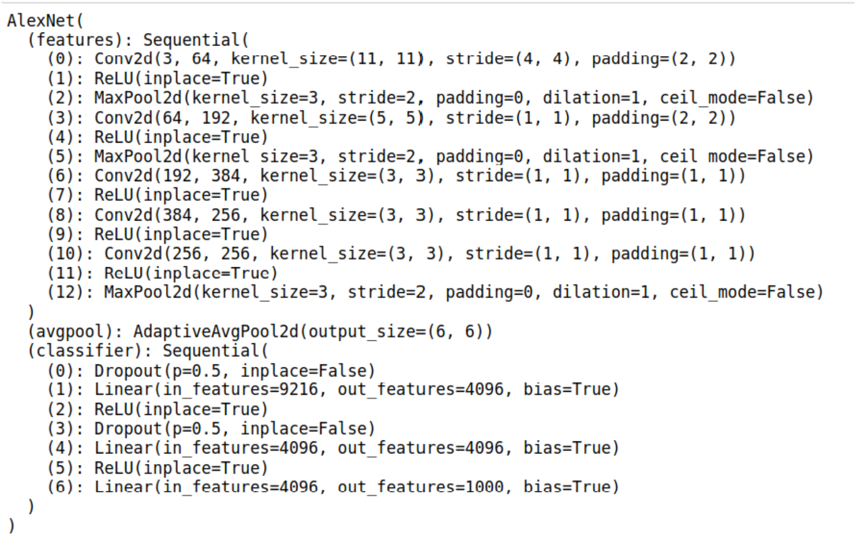


	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	0.36	0	0	0	0.0086	0.017	0.19	0.29	0	0.06	0.0086	0.043	0.017	0	0
1	0	0.55	0.0038	0.073	0.0038	0	0	0.0038	0.015	0.0038	0.34	0	0	0	0.0077
2	0	0	0.9	0	0.022	0	0	0	0.026	0	0.018	0.0088	0.022	0	0
3	0	0.05	0.0063	0.73	0.019	0.019	0.013	0.0063	0	0.025	0.056	0.037	0.025	0.013	0
4	0.014	0	0.019	0.038	0.38	0.1	0.043	0.1	0.0047	0.038	0.052	0.081	0.062	0.0047	0.057
5	0.0048	0.0048	0	0.0096	0.058	0.6	0.067	0.0096	0	0.058	0.0048	0.11	0.053	0	0.019
6	0.055	0.0091	0	0	0.0091	0.027	0.45	0.17	0	0.16	0	0.11	0.0091	0	0
7	0.058	0	0.0053	0	0.042	0.011	0.16	0.53	0	0.063	0.0053	0.1	0.011	0	0.0053
8	0.011	0.018	0.099	0.022	0	0.0036	0	0	0.61	0.0036	0.19	0.018	0.015	0.0036	0
9	0.017	0	0	0	0.0087	0.0087	0.035	0.07	0	0.86	0	0	0	0	0
10	0	0.048	0.058	0.035	0	0.0032	0	0	0.029	0	0.8	0.0032	0.019	0	0
11	0	0	0.037	0.0093	0.033	0.065	0.023	0.023	0	0	0	0.78	0.014	0	0.014
12	0	0	0.016	0.026	0.016	0.01	0.01	0.0052	0.0052	0.01	0.01	0.026	0.86	0	0
13	0.014	0.0071	0.05	0.014	0.014	0.064	0	0.035	0	0.021	0.0071	0.057	0.092	0.62	0.0071
14	0.0078	0	0.012	0	0.074	0.078	0.023	0.086	0.0039	0.039	0.0039	0.027	0.078	0.0039	0.56

After that I used an ensemble of five models, all of the type shown in the figure above, all trained independently using early- stopping and using the arithmetic average of the outputs to assign the class. Using a value for the dropout probability equal to $p = 0.1$,the same as the model before, the model reached a test accuracy of 54.44%, against the 65.12% of the model alone. Using dropout probability of 0.2, the test accuracy has increased to the value of 63.41%. In general, when using ensemble of networks, it is important to choose components with low bias and high variance, so increasing the probability of dropout respect this condition. When using dropout with probability 0.4 the accuracy of the model decreases to the value of 54.43.

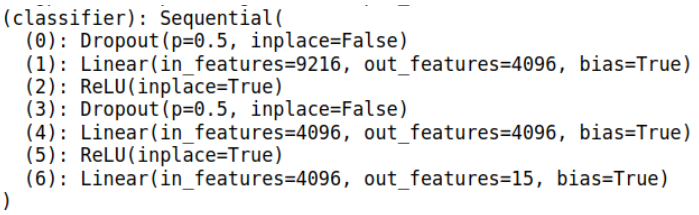
Third Point

To apply the method of the transfer learning I used the pretrained network AlexNet, which has the following structure:



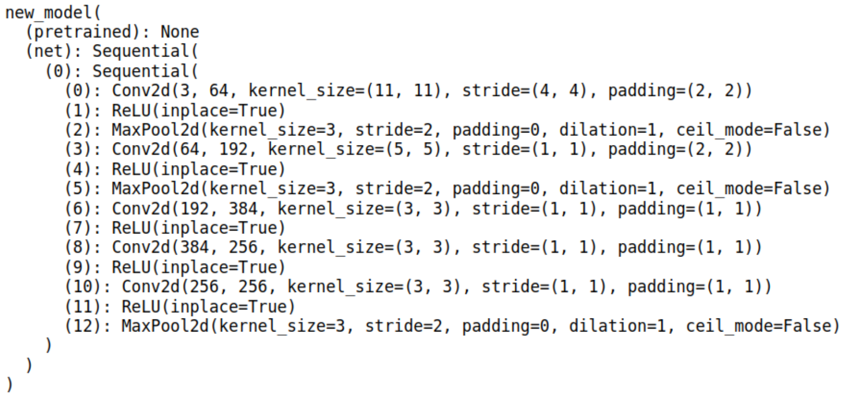
```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel size=3, stride=2, padding=0, dilation=1, ceil mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

This pretrained model expects images of shape $(3, H, W)$, where H and W should be at least 224, so I resized the image and repeat it three times in order to fill the 3 channels. I freezed the weights of the networks setting the parameter "require_grad" of all the weights of the network to False and then replacing the last layer of the classifier:



```
(classifier): Sequential(
  (0): Dropout(p=0.5, inplace=False)
  (1): Linear(in_features=9216, out_features=4096, bias=True)
  (2): ReLU(inplace=True)
  (3): Dropout(p=0.5, inplace=False)
  (4): Linear(in_features=4096, out_features=4096, bias=True)
  (5): ReLU(inplace=True)
  (6): Linear(in_features=4096, out_features=15, bias=True)
)
```

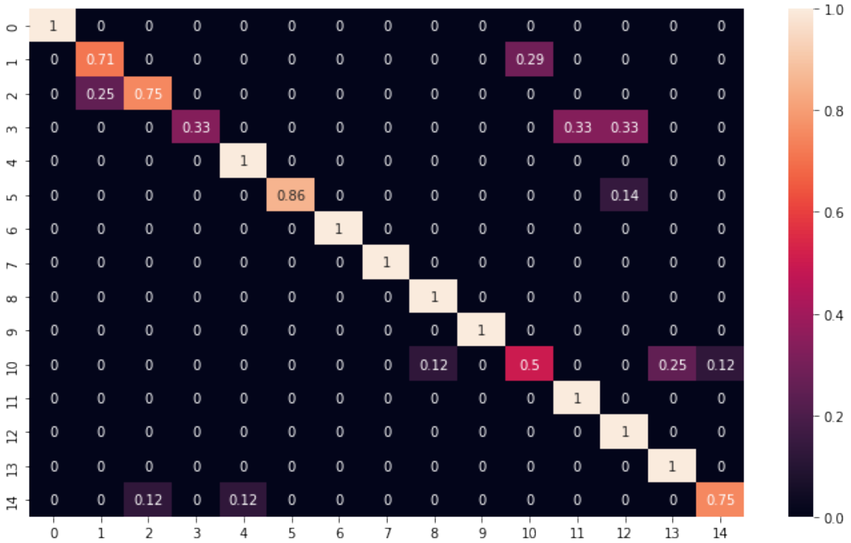
So I trained the network with early stopping, using the Adam optimizer with learning rate 0.001 and after 17 epochs the model reached the validation accuracy of 100% and the test accuracy of 84, 4%. Another possibility is to use the network as a feature extractor, for instance to feed another classificator, SVM in this case. I used AlexNet for this task too, extracting the features from the last Convolutional Layer, just before the averaging pool layer. This is the resulting network structure:



```
new_model(
  (pretrained): None
  (net): Sequential(
    (0): Sequential(
      (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
      (1): ReLU(inplace=True)
      (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
      (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
      (4): ReLU(inplace=True)
      (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
      (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (7): ReLU(inplace=True)
      (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (9): ReLU(inplace=True)
      (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (11): ReLU(inplace=True)
      (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    )
  )
)
```

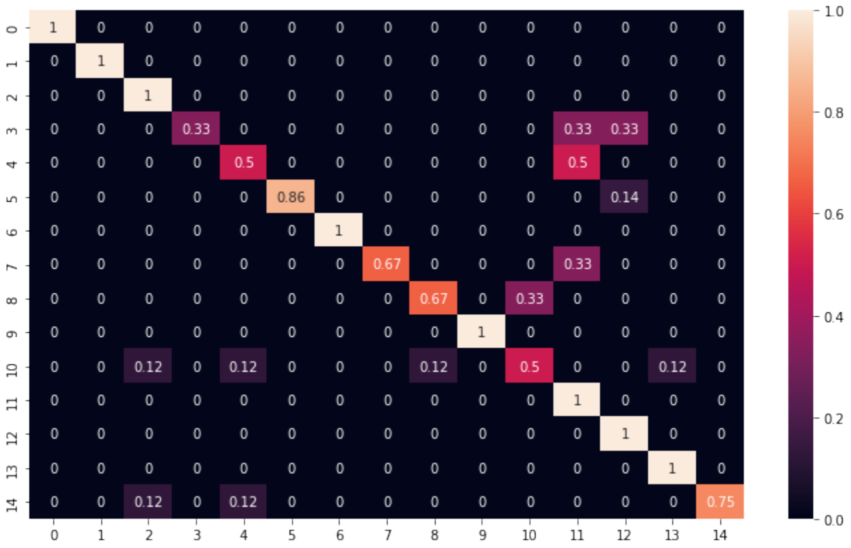
The output of this last layer consists of 256 features, each in the form of a $6 * 6$ grid. To make this output suitable for a SVM, the function $feature\_extraction$ flatten it to a one dimension vector with 9216 entries. I fit the linear SVM using the One-vs-All method and the model reached a train accuracy of 99% and test accuracy of 81%, while using a radial basis function as kernel and setting the hyperparameter C to 0.5 it reached the train accuracy of 97% and a test accuracy of 81%.

Confusion Matrix computed on the test set of the lineae SVM



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0.71	0	0	0	0	0	0	0	0	0.29	0	0	0	0
2	0	0.25	0.75	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0.33	0	0	0	0	0	0	0	0.33	0.33	0	0
4	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0.86	0	0	0	0	0	0	0.14	0	0
6	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0.12	0	0.5	0	0	0.25	0.12
11	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
14	0	0	0.12	0	0.12	0	0	0	0	0	0	0	0	0	0.75

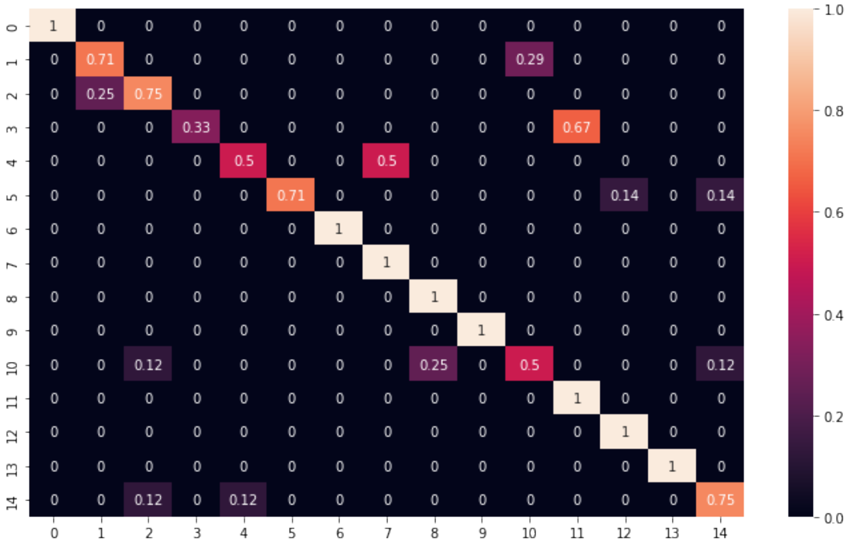
Confusion Matrix computed on the test set of the non-lineae SVM



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0.33	0	0	0	0	0	0	0	0.33	0.33	0	0
4	0	0	0	0	0.5	0	0	0	0	0	0	0.5	0	0	0
5	0	0	0	0	0	0.86	0	0	0	0	0	0	0.14	0	0
6	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0.67	0	0	0	0.33	0	0	0
8	0	0	0	0	0	0	0	0	0.67	0	0.33	0	0	0	0
9	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
10	0	0	0.12	0	0.12	0	0	0	0.12	0	0.5	0	0	0.12	0
11	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
14	0	0	0.12	0	0.12	0	0	0	0	0	0	0	0	0	0.75

For the implementation of Error Correcting Output Code for classification, I used a random binary matrix., called code-matrix, of size $(15, 20)$, where the first dimension has to match the number of classes and the second dimension is the number of different binary classifiers to train. I decided to implement linear-classifiers. Ideally, all the rows of the code-matrix should be at the same Hamming-distance one from the other. The labels of the train and test data must be transformed in order to fulfill the codification of the columns corresponding to each classifier. These 20 binary classifiers were trained and saved in a file. During the predictions the output of each of this binary classifier is considered, and it forms a word-code. The winning class is the one with the smaller Hamming-distance, as codified in the corresponding row of the code matrix, from this word-code. This last model reached a train accuracy of 99% and a test accuracy of 78%.

Confusion Matrix computed on the test set using Error Correcting Output Code



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0.71	0	0	0	0	0	0	0	0	0.29	0	0	0	0
2	0	0.25	0.75	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0.33	0	0	0	0	0	0	0	0.67	0	0	0
4	0	0	0	0	0.5	0	0	0.5	0	0	0	0	0	0	0
5	0	0	0	0	0	0.71	0	0	0	0	0	0	0.14	0	0.14
6	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
10	0	0	0.12	0	0	0	0	0	0.25	0	0.5	0	0	0	0.12
11	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
14	0	0	0.12	0	0.12	0	0	0	0	0	0	0	0	0	0.75