Homework 1:
Supervised Deep Learning

Michele Guadagnini - ID 1230663

March 24, 2022

Abstract

In this first homework we are asked to solve two exercises using Supervised Deep Learning algorithms. The first exercise requires to use a neural network to approximate an unknown function; the second to learn a classifier over images of the FashionMNIST dataset. Both tasks will be solved inside a Jupyter Notebook.

1 Introduction

The homework is divided into two parts, both requiring to use Supervised Learning methods to solve a particular task.

The first task consists in a function approximation by mean of a neural network. The network has been implemented as a *PyTorch* module and, in order to choose a good set of hyper-parameters, a random search was run by exploiting the *skorch* package in order to wrap the model inside the *sklearn* framework. Due to the small size of the dataset, both hyper-parameters search and actual training was done using cross validation.

The second task requires to use a network to classify images of the FashionMNIST dataset. A Convolutional Neural Network (CNN) has been implemented, again as a *PyTorch* module. This time the use of *PyTorch-Lightning* framework has been explored and the hyper-parameters has been tuned by mean of the *optuna* package. The dataset has been split into train and validation set. Also, during hyper-parameters search only a subset of the available samples has been used.

2 Regression task

The function $f : \mathbb{R} \to \mathbb{R}$ must be learned from a noisy dataset with only 100 samples. Points are defined in range $[-5, 5]$ and presents two regions where points are missing. A plot of the training points can be seen in Figure 1. In Subsection 2.1 the chosen architecture and strategies will be explained, while Subsection 2.2 will contain the obtained results.

2.1 Method

The network has been implemented as a fully connected neural network with 2 hidden layers. For simple regression tasks, also a network with only one hidden layer could be sufficient to fit the data, but our dataset is noisy and also presents some holes in its range of definition, hence a 2 hidden layers model has been adopted. This ensure us to have enough complexity to grasp the function, while eventual over-fitting will be mitigated by the use of regularization techniques, like dropout or $L2$ penalty. Indeed, dropout layers and batch normalization can be optionally placed between actual layers by passing respectively the dropout probabilities or a boolean flag to the model initialization. Also the activation function and the initialization scheme of the layers weights can be passed to the network at initialization.
Hyper-parameters search has been done with the function *RandomizedSearchCV* from the *sklearn* package, wrapping up the network model with *skorch*. A Random Search strategy should perform better than a Grid Search one in case of a large number of hyper-parameters to optimize, especially if some hyper-parameters are more important than others. The hyper-parameters space has been defined with lists of possible values or with

Name	*Value*	**Name**	*Value*
learning rate	0.00186	activation	ReLU
Nh1	288	Batch norm.	TRUE
Nh2	288	optimizer	Adam
Pdrop1	0.05	L2 penalty	0.001
Pdrop2	0.125	epochs	200

Table 1: Best set of hyper-parameters obtained from randomized search.

Model	**Test loss**
Random search model	0.2574
CV net_0	0.1303
CV net_1	0.2249
CV net_2	0.1223
CV net_3	0.1767
CV net_4	0.1413
CV average	0.1284

Table 2: Performance over test dataset for all the trained models.

a continuous distribution (using *scipy.stats*) for some continuous variables. The ranges of hyper-parameters values has been chosen after some simple experiments. The parameters that have been optimized include:

- **number of units** in hidden layers (*Nh1* and *Nh2*) to be chosen from $[160, 192, 224, 256, 288]$ (to reduce the searching space we added the constraint that both layers must have the same number of units)
- **activation** function (chosen between *ReLU* and *Tanh*)
- **dropout** probabilities after first and second hidden layers (*Pdrop1* and *Pdrop2*) in range $[0, 0.25]$
- the use of **batch normalization** before the second and the output layer (boolean value)
- **optimizer** (chosen between *Adam* and *SGD* with a momentum of 0.9)
- (initial) **learning rate** of the optimizer from a log-uniform distribution in range $[10^{-5}, 10^{-2}]$
- **L2 penalty** coefficient from a log-uniform list of values in range $[10^{-3}, 0.5]$ (note that also a 0 value has been added to the list)
- number of **epochs** between $[100, 150, 200]$

Since the dataset is very small, it has been used in a 4-folds cross validation setup during random search and batch size has been set to be equal to the number of training points (75 during the random search due to 4-folds splits). The adopted loss function was the *Mean Squared Error* (MSE).
On the random search function the flag *refit* has been set to *True*; this way, after the last iteration, the best performing configuration is used to fit a model using the whole dataset, obtaining a predictor.

While the strategy above showed to acceptably solve the task, a different approach that should make better use of the dataset has been explored: train 5 models in 5-folds cross validation setup and use ensemble averaging to obtain the approximated function value. Each model adopted the hyper-parameters set resulted from the random search above, except for the number of epochs; indeed here *EarlyStopping* and *Checkpoint* callbacks has been used.

2.2 Results

The number of hyper-parameters configurations tested is 600, requiring a total number of fits of 2400. A dual-core laptop CPU took around 40 minutes to complete the task. The set of hyper-parameters resulting from random search is reported in Table 1.
Table 2 reports the performances over the test dataset of the output predictor of the randomized search, the networks trained in cross validation and the ensemble average model. It can be noticed that the test loss of the average predictions is smaller than the average of the single test losses; this confirms the chosen approach, although the difference is not so high. Indeed, even if sometimes there is a single net with a better test loss than the the average, it must be said that test points are not sampled uniformly in the range [-5,5], so an error in a region with more test points counts more than errors in less represented regions.

The plot in Figure 1 presents the train dataset and Figure 2 reports a comparison of the function approximations obtained from random search best model and the average model. From Figure 2 it is possible to notice that the average model function seems smoother than the random search best model. Looking at the two regions in the train dataset where points were missing it can be noticed that the first one (in range [-3,-1]) can be

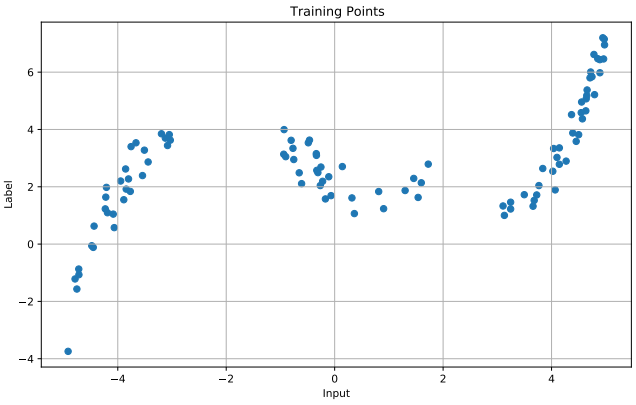

Figure 1: Plot of training points.

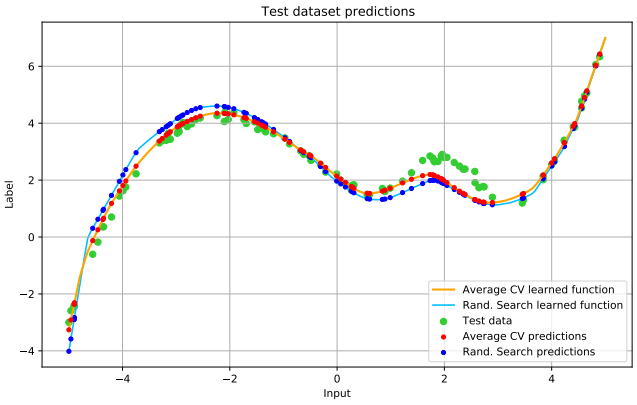

Figure 2: Performance comparison over test dataset.

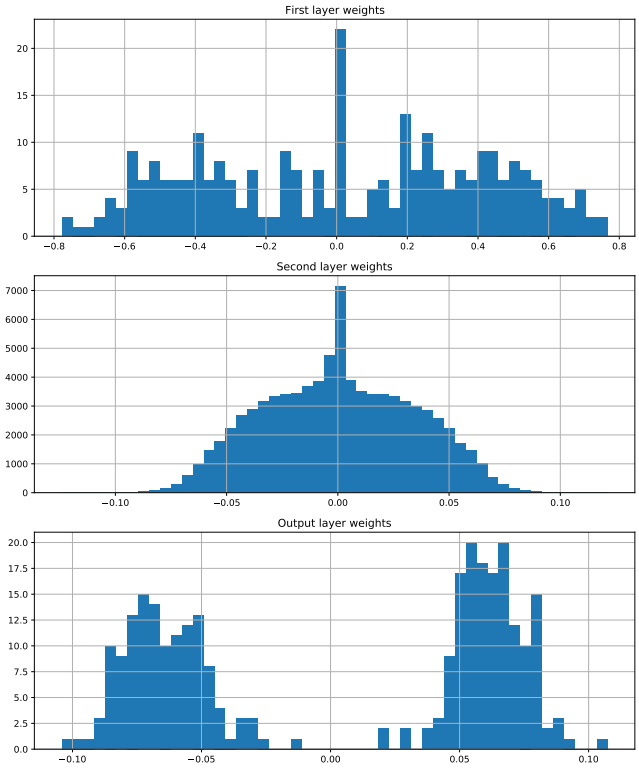

Figure 3: Weights distributions of the network layers after training.

acceptably approximated by both the obtained predictor, while the second one (in range [2,3]) is more difficult to learn. Note that in both cases the average model is performing better, probably due to weaker overfitting thanks to cross validation and average between models predictions and also to the use of checkpoint callback.

Figure 3 reports the plot of weights distribution on each layer of the random search best model after training. The very narrow peak at 0 for hidden layers is probably an effect of the *ReLU* activation function (we saw that this peak disappears if we replace *ReLU* with *Tanh*). The effect of L2 penalty is clearly visible in the second hidden layer, as frequencies increase while approaching 0. Interestingly, the output layer presents a bimodal distribution. Regarding the activation pattern of hidden layers (Figure 4) it can be noticed that second hidden layer has usually stronger activations. Also we can see that when 0.10 is passed as input, the activations of the first layer are close to 0, while the ones of the second layer are still high, but sparser with respect to activation pattern from different input values.

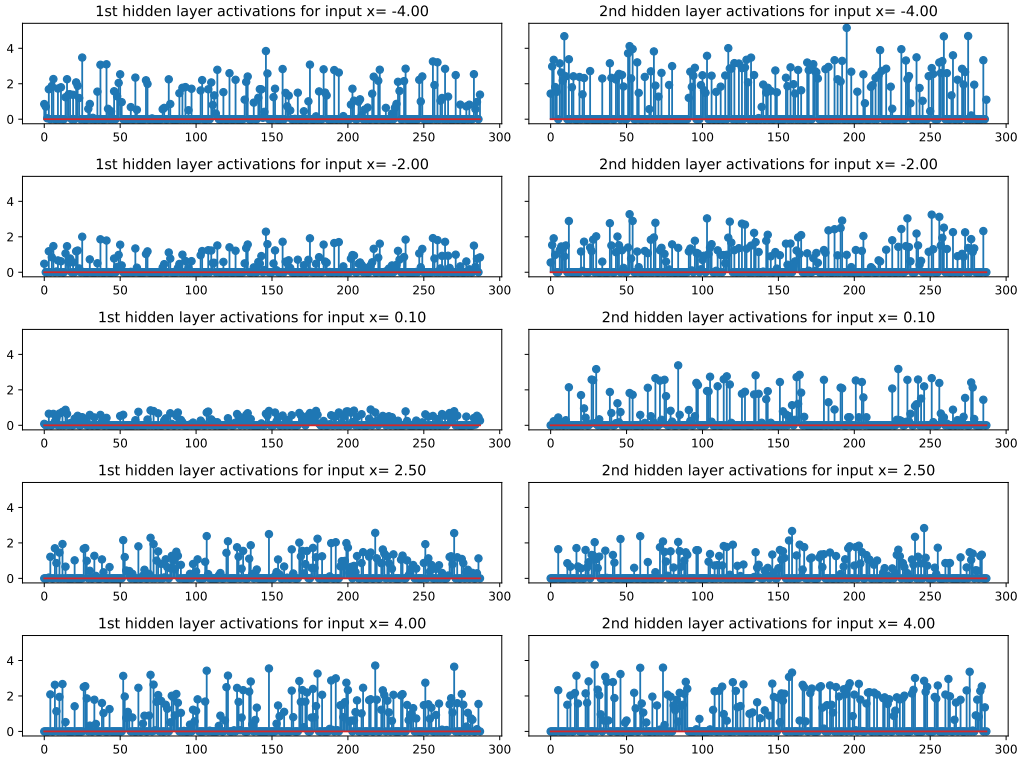

Figure 4: Activations of network units of first and second layer for inputs in $[-4, -2, 0.1, 2.5, 4]$.

3 Classification task

The classification task consists in learning a classifier over the FashionMNIST dataset, which contains images of size 28×28 divided into 10 classes, each class representing a different category of Zalando products. Training set contains 60000 images and test set 10000. The dataset can be easily downloaded exploiting the package *torchvision*. Subsection 3.1 explains the developed architecture and strategies, while Subsection 3.2 will contain the produced results.

3.1 Method

To solve this task, the *PyTorch-Lightning* framework has been chosen. First thing done was defining the data-module class that will be used to split dataset into train and validation sets, apply transforms and to create dataloaders. We decided to test also the use of some random transformations to use as data augmentation. In particular two transforms have been implemented, one that add gaussian noise and one that add a rectangular occlusion with a certain probability to the image. Also random flipping (both vertical and horizontal) from *torchvision* package was used.

The network model built to solve the problem was implemented as a series of convolutional layers followed by a fully-connected part. The number of convolutional layers can be chosen freely, as well as their parameters like kernel size, number of feature maps, stride and padding. The fully-connected part is made of 2 hidden layers, whose number of units can be set at initialization. After each convolutional or linear layer batch normalization can be optionally added using a boolean flag. Model parameters are passed to the initialization function as a dictionary.

The model was wrapped inside a *PyTorch-Lightning* module, where the methods needed for the training step, the validation step and the setup of the optimizer are defined. Loss function was set to *CrossEntropy*. Network and optimizer hyper-parameters are passed to the initialization function of the *lightning* module; this way it is possible to easily test different configurations and select the best one. Indeed, the hyper-parameters were optimized using *optuna* package by creating a *study* object that runs the *objective* function for a certain number of trials (30 in this work). In each trial a different configuration of hyper-parameters was sampled from the hyper-parameters space, starting from selecting one architecture settings among a list of 12 configurations. Each item in the list is a dictionary containing the number of convolutional layers (one of $[2, 3, 4]$), list of

Name	*Value*	**Name**	*Value*
# of conv. layers	3	Nl1	1024
kernel sizes	[7, 3, 3]	Nl2	320
strides	[2, 2, 1]	Batch norm	TRUE
paddings	[2, 2, 1]	optimizer	Adam
output feature maps	[96,64,32]	Batch size	256
learning rate	0.000566		
L2 penalty	0.000169		

Table 3: Set of used hyper-parameters. Left column: parameters optimized with *optuna*; right column: fixed parameters.

Random transf.	*Test Loss*	*Test Accuracy*
FALSE	0.287	0.906
TRUE	0.259	0.911

Table 4: Test loss and accuracy for both the trained models without (*FALSE*) and with (*TRUE*) random transformations applied to input images.

kernel sizes, list of strides and list of padding values. Then the number of feature maps for each layer was sampled from $[32, 48, 64, 96, 128]$ and finally learning rate and L2 penalty were sampled from two log-uniform distribution in range $[10^{-4}, 10^{-2}]$ and $[10^{-5}, 10^{-1}]$, respectively. Other hyper-parameters (number of units *Nl1* and *Nl2* in fully-connected layers, optimizer, batch size, batch normalization) have not been optimized due to time constraints, but set to fixed values which we expect to work sufficiently well. Sampling is not completely random, as *optuna* can use results of previous trials to select promising configurations using a technique called *Tree-structured Parzen Estimator* (TPE). To save time during hyper-parameters optimization, the dataset was reduced to only 8192 samples, 1024 of these used for validation. Also, a pruning and an early stopping callbacks were passed to the PyTorch-Lightning *trainer* object inside the objective function.

Once found the best set of hyper-parameters based on the minimum validation loss reached by each trial, the network was trained on the full dataset (50000 for training, 10000 for validation). Early stopping and checkpoint callbacks were used, together with a custom callback created to track the losses during training. The training was run twice, first without any random transformations to input images and secondly with the random transformations described above in sequence; probability of each transform was set to 0.05 in order to have a probability of at least one transform applied equal to 0.185. For both the trained models test loss, test accuracy and confusion matrix were calculated.

Attempting to inspect what the network have actually learned, some plots of the learned filters have been produced. Also, the output feature maps of each convolutional layer for an image randomly selected from test dataset have been visualized.

Finally, we tried to maximize the activation of some filters of each convolutional layer in order to understand what the network is looking for at different levels. The same has been done also with neurons of the output layer. For this purpose the class *MaximizeActivation* was implemented. It uses a hook to retrieve activation from a particular filter or unit and then uses *Adam* optimizer to tune the pixels of the randomly initialized input image. To improve the quality of the produced images, constraints like gaussian blurring, clipping or L2 penalty have been added.

3.2 Results

The final list of used hyper-parameters is reported in Table 3. The hyper-parameters search was not exhaustive, as some of the parameters have not been optimized and also allowed ranges for some of the optimized ones could have been extended. However, the used models have been able to acceptably solve the task, as confirmed from the results reported in Table 4. It can be noticed that the random transformations are reducing overfitting and increasing the generalization capability of the model, although the difference in performance is not very important.

In Figure 5 we report the plot of train and validation losses during training and the obtained confusion matrix for the model without random transforms applied. In Figure 6 an example of the filters learned by the network is reported. It is hard to assign to them a clear meaning, and also the corresponding feature maps plotted in Figure 7 seems to be noisy, suggesting that the convolutional filters are not well trained. The reason for this could be that the fully-connected layers have too much complexity and overfit too quickly the output feature maps. Maybe reducing the number of neurons or removing batch normalization from the fully-connected classifier could improve the learned filters.

Finally, the images generated by maximizing some filters activation seems to represent some basic patterns

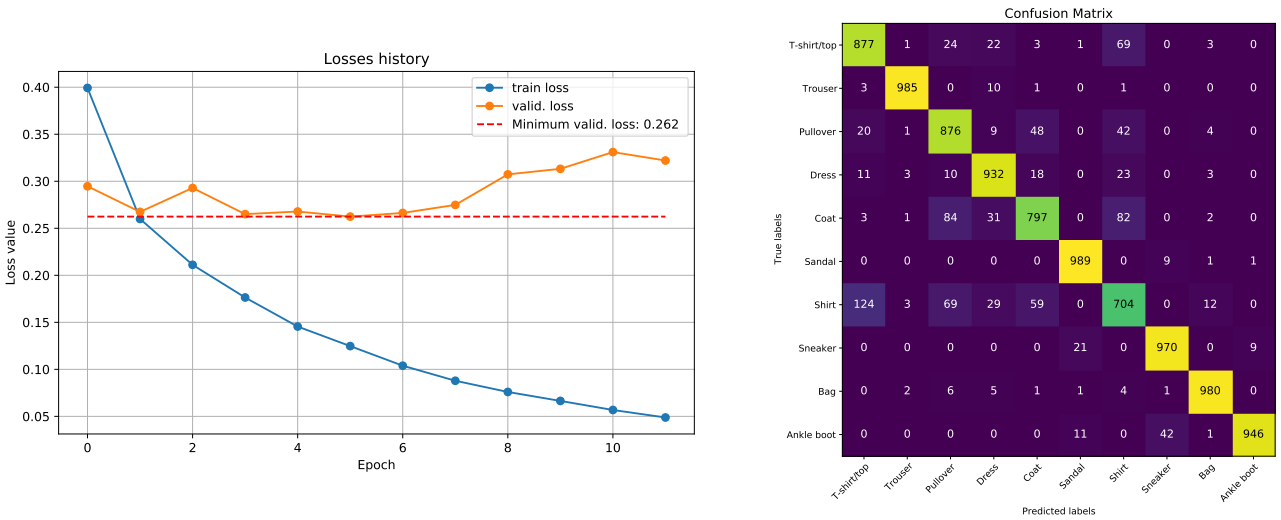

Figure 5: Results of the model *without* random transformations. Left: plot of the losses history; Right: confusion matrix.

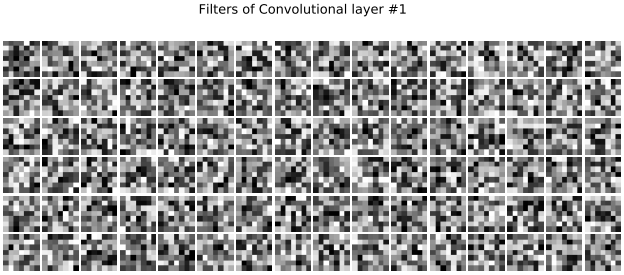

Figure 6: Examples of learned filters for the first convolutional layer.

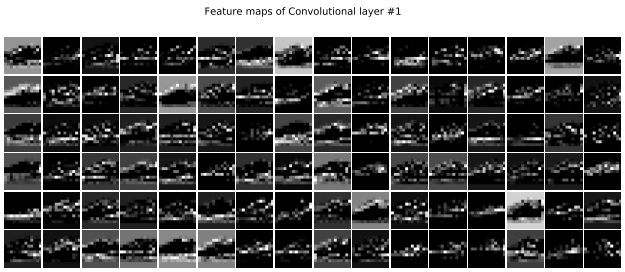

Figure 7: Output feature maps of the first convolutional layer for a random sample image.

that become more complex for inner layers. An example of generated images for second layer filters is reported in Figure 8. However, it must be said that, to obtain such visualizations, constraints like clipping and rescaling have been applied.

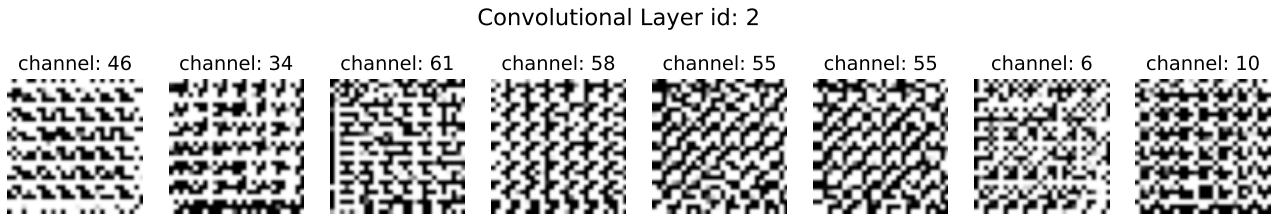

Figure 8: Images produced maximizing activation of randomly chosen filters of second convolutional layer.

4 Conclusions

In this work two Supervised Learning models have been used to solve a function approximation and an image classification. Models hyper-parameters have been tuned with two different tools: a *random search* from *sklearn* has been used for regression task, while a more complex tool (*optuna*) have been used for the classification task. Optuna resulted to be an effective and flexible tool, while the random search resulted more rigid, probably because the *sklearn* framework was developed with different models and applications in mind. Anyway, in both cases a good set of hyper-parameters was found and used to successfully solve the problems.

Appendix

In this appendix we report additional plots and images produced during the classification task.

Loss history and confusion matrix for model trained with random transformations

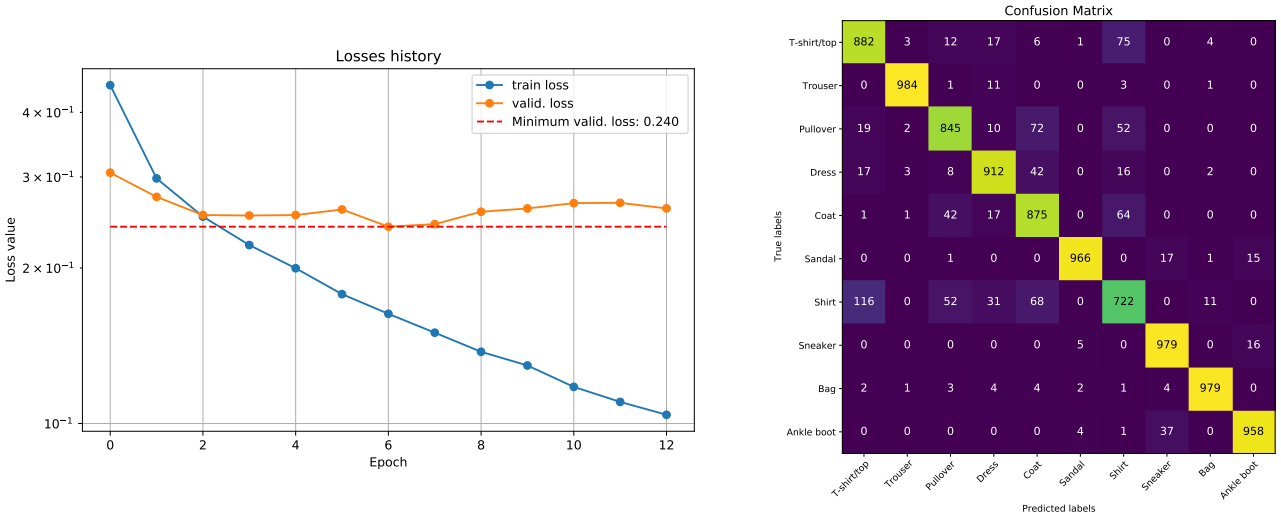

Figure 9: Results of the model *with* random transformations. Left: plot of the losses history; Right: confusion matrix.

Learned filters and feature maps

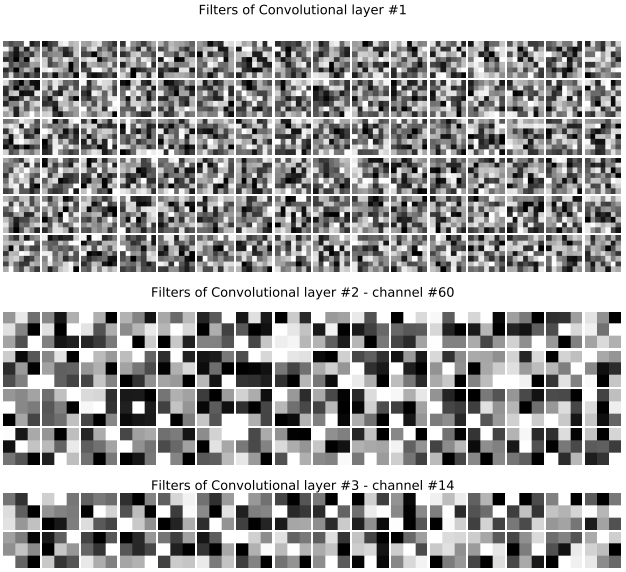

Figure 10: Examples of learned filters for all convolutional layers.

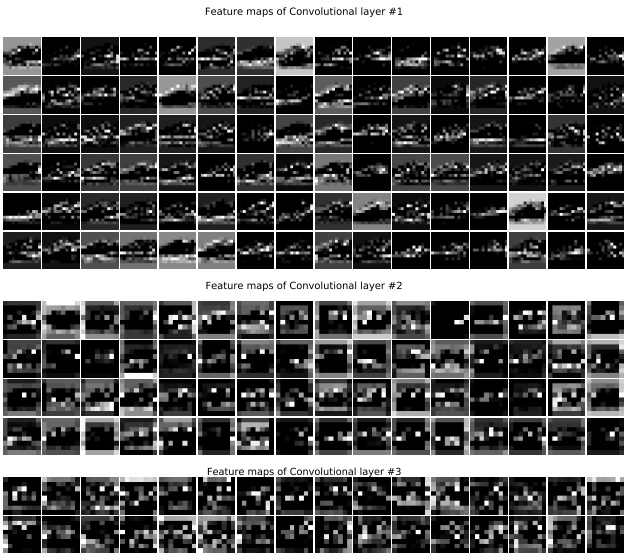

Figure 11: Output feature maps of all convolutional layers for a random sample image.

Maximize filters activation

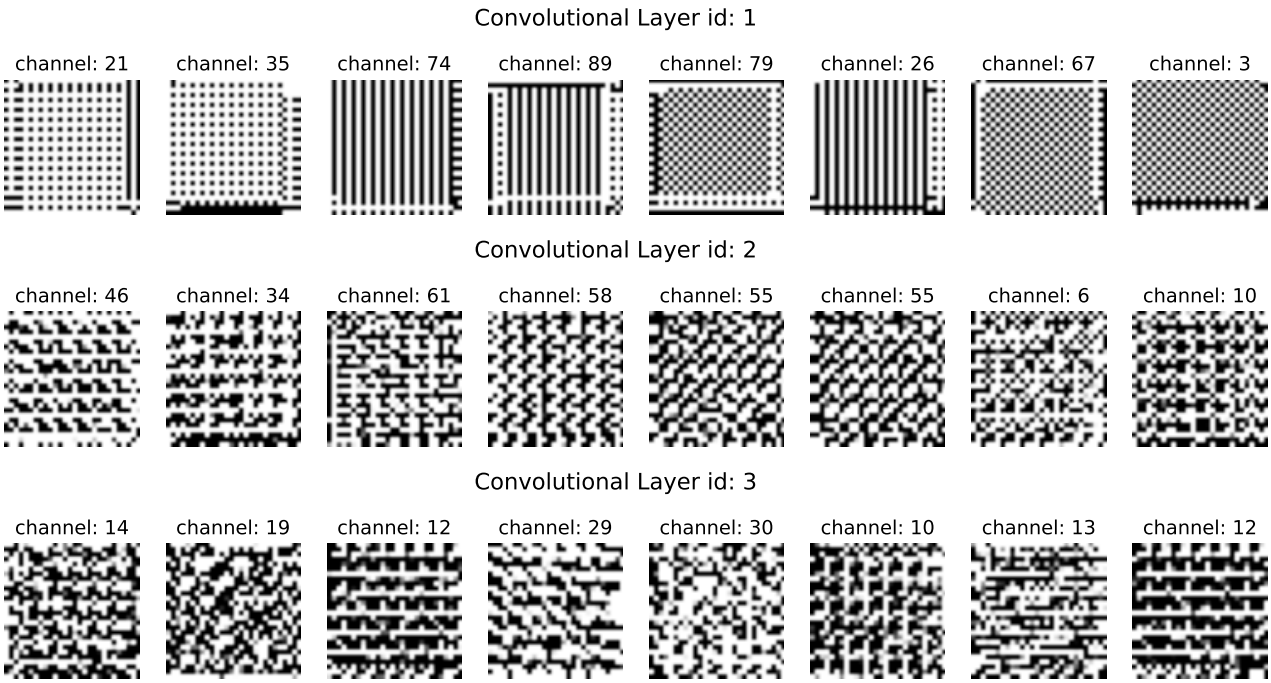

Figure 12: Images produced maximizing activation of randomly chosen filters of all the convolutional layers.

Maximize output neurons activation

The images generated by maximizing the activation of an output neuron are not very similar to dataset images of the same class; they present some of the features of the class they belong, but they are far from being a good quality sample, even if the network is highly confident on which label to assign to them. Also, to obtain these images strong constraints have been applied to the optimization (very high L2 penalty, gaussian blurring, clipping)

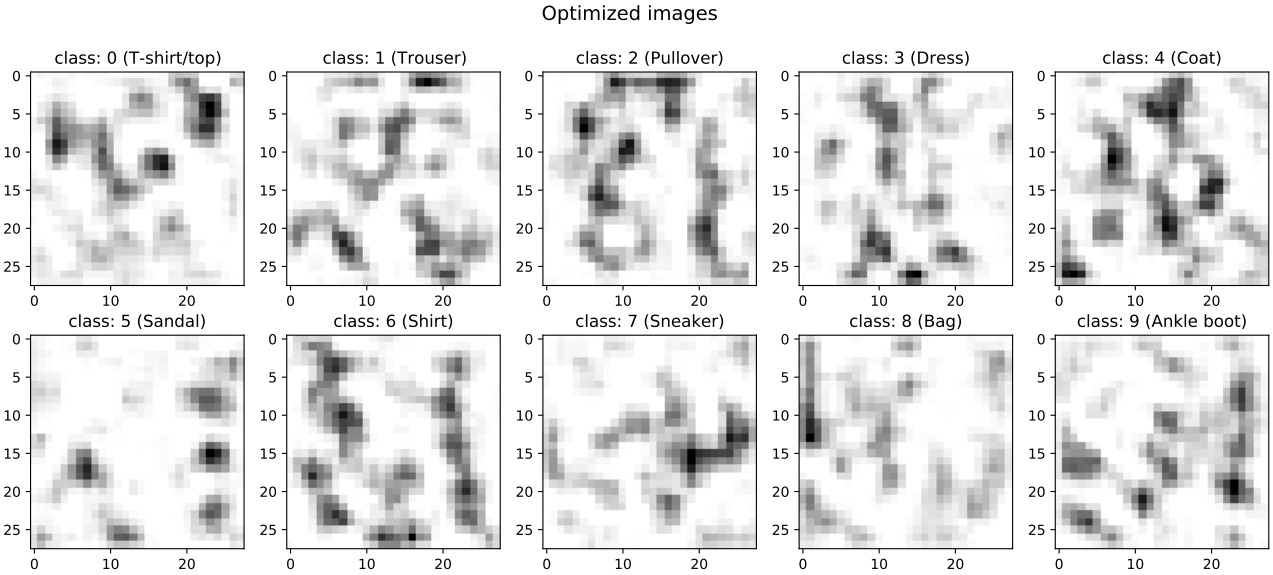

Figure 13: Images produced maximizing activation of output neurons.

Maximize filters activation of a pretrained AlexNet

In order to ensure that the implemented class *MaximizeActivation* is working properly we tested it on a pretrained network (**AlexNet**) from the torchvision package.

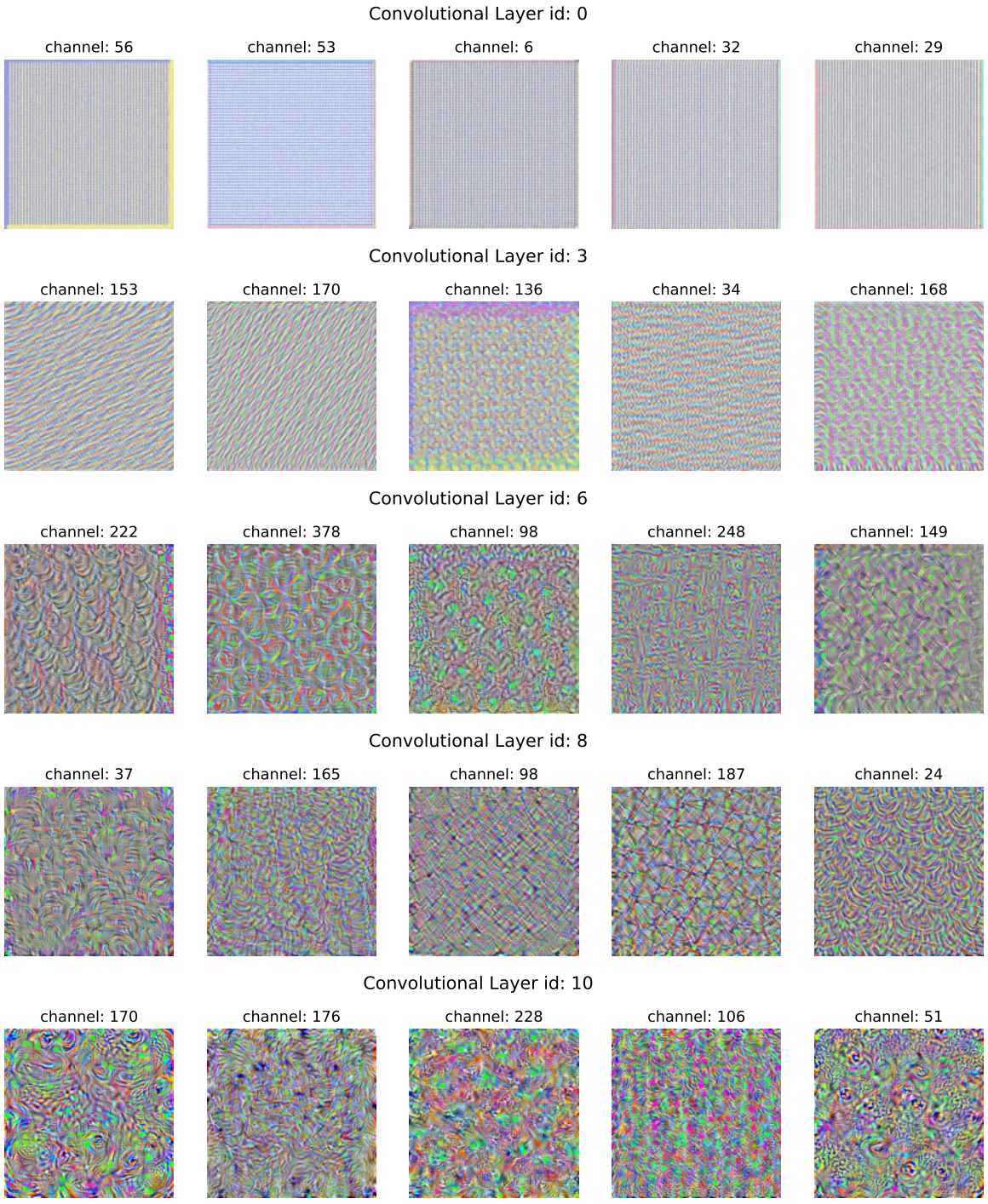

Figure 14: Images produced maximizing activation of randomly chosen filters of all the convolutional layers of a pretrained AlexNet.