# HOMEWORK 1 NEURAL NETWORK AND DEEP LEARNING

Zambelli Francesco (2029014)

Academic Year 2021-2022

## 1 REGRESSION TASK

The goal of the regression task is to perform a regression on a synthetic dataset from a unknown function provided at the beginning of the work. The essential steps of this works are:

1. Data import and manipulation for the training
2. Definition of the network
3. Definition of a function for the training and testing of the net
4. Show of the results, loss trends, weights distribution and activations of the neurons
5. Hyperparameter optimization with Grid Search, Bayesian Search and Random Search, each with his own pros and cons, with KFold cross validation

The **regularization** methods used in this task are dropout layers in the network, customized weights initialization, weights decay (i.e. Tichknov regularization) in the optimizer, tuning of the size of the net in order not to crate a too complex model, proper splitting of train-validation sets. In addition in the hyperparameters optimization the KFold cross validation strategy is used.

### 1.1 Dataset

The dataset is composed by a training set of 100 points, extracted from specific sub intervals of an unknown function, and affected by some noise, and a test set composed by 100 points uniformly distributed within the domain of the function considered in the case, and affected by a weaker noise. The test set can be used to evaluate the goodness of the fit.

A **customized dataset class** is built in order to manipulate the data and prepare them for the training. Initially the **loadtxt** command from the *numpy* library is used to import the whole dataset, and each line, composed by the couple *(input, target)*, is divided by the loadtxt function itself. The __getitem__ function defined in the class allows to retrieve a particular sample of the dataset after it gets transformed by the transformations given in input. In this case, the only transformation applied is the conversion of each point in a couple of *float torch Tensors* of dimension 1.

In order to perform the network training 2 more steps are necessary: the first one is to split the training dataset given at the beginning of the work, into a **training** dataset and a **validation dataset**, in order to be able to test the results of the training over some samples which didn't participate to the training. In order to this the *randomsplit* function from sklearn is used. The second one is to build the **Dataloader** structure which allows to retrieve the mini batches of the training set for the training of the network. A function is then implemented in order to perform this 2 procedures. It has the goal to manipulate easily the structure of the Dataloaders by changing some parameters, as the *train-validation dataset proportions* and the*batch size*. The ratio between train and validation test is quite important in order to have a good evaluation of the goodness of the fit, because, due to the lack of samples, the problem of the **overfitting** is especially dangerous in this case.

### 1.2 Network

A class from *nn.Module* is defined in order to build the network for the regression task. It is composed by some fixed **Dense layers** at the begging and at the end, while the central part is composed by a **Sequential** module which can *vary in depth and number of units* according to some input parameters of the module. This helps to get an idea of how the depth of the network influences the performances. For the **regularization**, some **Dropout** layers are used, with probability to suppress a given units that can vary with the Module dropout

parameter. For the activations, the *Sigmoid* function is used by default, but in the hyperparameter optimization other function will be tested, as the *ReLU* and *Tanh*.

Another element that could affect the performance of the network is the **weights initialization**. Some function are defined at the beginning of the section which allow to initialize the weights of layers of a specific kind with values drawn from random distribution with parameters which can be chosen. Altough it's quite difficult to choose proper parameters of such distributions, which would require a search task by his-self, so in practice, during the work the default initialization of the layer are used.

The network for the regression is then initialized, moved to the GPU and finally its structure and output dimension are displayed thanks to the *summary* function from the *torchsummary* library. In order to obtain good performances, as it is possible to see in the following sections, the total number of parameters can be in the order of some thousands, so the network will be quite fast and easy to train as will be shown successively.

At the beginning the training of the net is performed in a **manual way**, so some functions for the training and the testing procedure for 1 epoch of the network are defined. The main difference between these 2 is that in the first one the *gradient* of the network is computed, and thanks to the **backpropagation** algorithm (*backward()* function), the weights are updated with respect to the loss computed between the output of the network and the target. In this case the *MeanSquaredError* function is used. In the second one, the backpropagation section is not present, because we only need to test the network performance, not to update it.

In order to perform the backpropagation it's necessary to define an optimizer. The initial choice is **Adam**, a good optimization algorithm which good performances are well known. It's characterized by two main parameters: the *learning rate*, which determines the speed of the training, and the *weight_ decay*, which performs a L2 regularization, i.e. into the computation of the loss, not only the error between the output and the label is considered, but also the squared sum of the weights values. In this way they are not allowed to grow too much, and also, the network is encouraged to use less neurons in order to compute the output. Also the **SGD** optimizer will be tested during the hyperparameter search phase, but it will lead to worse results.

Then a function for the whole training of the network is defined. The choice to write a single function arises from the fact that in this way the change of hyperparameters is much easier to handle in the hyperparameters optimization phase. After the training of the net, a function allows to plot the graph produced by the outputs of the network, both at its final state and at the iteration in which the validation loss reached the minimum (the *checkpoint* network in the code), and the training and loss trend with respect to the epochs (fig. 1).

After many trials it's possible to notice that *not always the state of the network which produces the lowest validation loss corresponds to the output that fits the test points the best*. This is probably due to the small amount of data, and the validation set, which consists in a few tens of points, could be not representative of the target function. This means also that the validation loss doesn't always tells everything about the goodness of the fit, so an evaluation "by eye" is often necessary.

The result of the regression task (with parameters properly chosen thanks to the hyperparameter optimization section that will be presented below), as it's possible to see, are quite good, as the network is usually able to detect both the slope which characterize the initial function.

By looking at the loss it's possible to see some steps, which, by analyzing the GIF, is possible to connect to the instants in which the network "understands" the presence of one of the two slopes. In fact usually in the early epoch there is the step for the slope with maximum in -2, the bigger one, while after some other thousands of epochs, another step occurs for the detection of the smaller slope with maximum in 2. In the particular case of this last run, only the first step is present, while the smaller slope is recognized gradually by the net (fig. 1.a).

## 1.3 Weights histograms, Activations and Receptive fields

The weights and biases values for some layers of the network are plotted as **histograms**, in order to get an idea about how they are distributed within the parameter space. An aspect that can be noticed is that the distribution of the weights over their possible values gets less sparse as we increase the layer depth (fig. 2).

Thanks to the *register_forward_ hook* function, it is possible to obtain the **activation** of the network for specific inputs. In the stem graphs some examples are shown, but to understand properly how the activation change with respect to the input, some GIF were generated. With these it's possible to see that in some region of the domain, not all the neurons are involved in the task, and keep almost the same activation value for all the points in the interval (fig. 3).

This fact can be highlighted by plotting the **receptive field** of each neuron(fig. 3), i.e. the value of it's activation given a certain value of the input for inputs distributed uniformly in the whole considered range. This is equivalent to see the region of the input space for which a specific neuron gets activated. A noticeable fact is that the activations of the last layer neurons display very small change in values, but, given the fact that their combination with the weights of the last layer applied to an activation function will produce the final output of the net, even these slight differences have to lead to the variation of the outcome in the whole interval given by the codomain of the target function.

## 1.4 Hyperparameters optimization

For the hyperparameters optimization task, different approaches were taken. The first one is the **Grid Search** with **SKORCH**: in order to perform a grid search over the hyperparameters of the model, the *Skorch* library is used. This allows to wrap in the *sklearn* environment a pytorch module. This allows to use all the functionalities of *sklearn*, as the *GridsearchCV* and *KFold* cross validation in an easy way. Skorch allows to insert all the parameters of the model as parameters of the *NeuralNet* functions (one for regression and one for classification). The the module if fitted with the dataset, which is automatically and randomically splitted into training and validation set, with a default ratio of 0.2. Now the GrisSearchCV function can be applied to the skorch model of the neural network. It allows to run the model over all the possible combination of the values of hyperparameters given in input, and compare the results obtained by evaluating a scoring function (in this case the MSE over the validation dataset), with the Cross validation procedure.

The second one is the **Bayesian Search** with **OPTUNA**: Another way to perform the hyperparameter optimization is by using the Optuna library, which implements a function that progressively chooses the different hyperparameters values within each corresponding range, by exploit a Bayesian research act to the minimization of the loss (or maximization of the score).

The following parameters were considered into the optimize the network: **Splitting ratio** between validation and training set, the dataloader **batch size**, **Number of units** of each linear layer of the network, **Depth** of the network, **Dropout probability** of the dropout layers, **Activation function**, **Weights initialization function**,**Learning rate** of the optimizer, **Weight decay** of the optimizer, i.e. the relevance of the $L_2$ regularization in the loss computation.

The results of the hyperparameters optimization (tab. 1, fig. 1.b) are quite unexpected, but plausible by reasoning about the structure of the problem. The ReLU activation function produce a *polyline*, fact which is not desirable as final result of the task, but the network, only by looking at the validation loss, doesn't make any distinction over the smoothness of the output, so even this kind of trend is good as long as it produces good results. Then the other parameters, as the learning rate and the weight decay, change accordingly. In particular the learning rate can be smaller with respect to the case in which a Sigmoid or Tanh activation are used, while the weight decay value is higher. Even the number of units and the depth of the network increase is we are satisfied with a polyline as output, because, by enlarging the number of units, even the vertex of the polyline increase, so the generalization capabilities increase.

A problem arises. The shortage of data has as a consequence the difficulty in evaluating the goodness of the regression results. A good final score of a trial could be due to a lucky split of training-validation set. In order to reduce the plausibility of such a scenario, it's good practice to use the **KFold cross validation** procedure, i.e. the dataset is splitted in *cv* folds and *cv-1* of them are used to perform the training, and the last one to evaluate the score as a validation dataset. Then all the permutations are performed and at the end the final score given by the function is the mean of the accuracy for each of the *cv* iterations.In this case the number of folds is chosen equal to 5, so the training-validation ratio will be 0.8 (fig. 1.c).

But another problem arises: a given combination of hyperparameters that returns a good score could be very distant with respect to more than one parameter from another good combination, which could lead even to better results. E.g the values of the learning rate that produce good results strongly depend on the choice of the activation function, or also, the weight_decay parameter relevance in the problem is correlated to the size of the architecture, i.e. the number of weights. The OPTUNA algorithm tends to focus its search around the good combinations it finds, so it's very that it manages to "jump" to a different and distant one, that would require to guess a completely new combination of the parameters based on the previous results. To avoid this problem also a **RANDOM SEARCH** algorithm is implemented, in which the hyperparameter combination of a certain trial doesn't depend on the combination of the past ones (fig. 1.d), and each trial is evaluated with the KFold cross-validation procedure.

The hyperparameters used in the first example of the regression net training were chosen after observing many attempt's results and by evaluating the goodness of the fit also by looking by eye the output with respect to the samples. We call this procedure *"by eye"*.

## 1.5 Results

In order to compere the different methods it's possible to both look at the final output of the net (fig. 1) or by looking at the loss over the test set produced by each network (tab. 1). From what we can see, all the results have good generalization proprieties, so the task can be considered solved. The best one is reached by the "by eye" method, but it has a little throwback: by looking by eye, our brain already knows the shape of the function, so it's kind like using the test set also for the hyperparameter evaluation. The other methods does not have such problem, so the best result can be considered the one reached with the Bayesian search with KFold cross validation

# 2 CLASSIFICATION TASK

The goal of this task is to perform a classification over the FashionMNIST dataset, a dataset composed by a training set of 6000 samples and a test set of 1000 in which every sample is composed by an image of a piece of cloth $28 \times 28$ with only 1 color channel, and its corresponding target label (10 possibilities).

The main steps of the task are:

1. Dataset download and transformation (In the first part no image transformation are applied)
2. Definition of the linear network and training function
3. Show of the results, weights plots and activations
4. Switching to the dataset with transformed images and results show
5. Hyperparameters search with Optuna
6. Definition of the Convolutional network, training and results show
7. Hyperparameters search with Optuna

The regularization strategies used in this case are Dropout layers, Batch Normalization layers, customized weights initialization functions, weight decay in the optimizer, input transformation. In addition in the hyperparameters optimization the KFold cross validation strategy is used.

## 2.1 Dataset

Thanks to the *torchvision.dataset.FashionMNIST* command, both the downloading of the data, the organization into a Dataset structure and the application of some transformations to the samples are applied.

Both for the training and test datasets we transform the samples into tensors and translate them from the range [0,1] (in which the values of each bit are rational numbers) to [0,255] (in which they become integers). This last procedure results in better performances for the future training of the net.

Two different composition of transformations are defined for the training dataset. In the *composed_ transform_ clean*, there are only the transformation used for the test set, while in the *composed_ transform* also some image transformation, as the *Vertical flip* and the *Random Rotation*, which allow the network to become more robust in the classification task (fig. 4).

For the first part of the work the "clean" dataset will be used, then, after showing some interesting results, the dataset with the image transformation will be adopted in order to make the task more complete.

After the transformations, the training dataset is splitted into 2 subsets, one larger as training set and one smaller as validation one. The ratio is 0.9. In order to do so, it's possible to use the *random split* command from the torch library.

## 2.2 Linear Network

As first case a classifier composed only by Linear layers, activations and Dropouts, is studied. The depth and general shape of the net is kept fixed, and the only parameter of the module is the dropout probability.

The network is then initialized and trained manually, with a function in which both the updating of the network weights and the check of the performance over the validation set are performed.

Finally, the train and validation loss are plotted in a graph, and next to it, a heatmap shows how the classification of the samples in the test set correspond to the true targets. This procedure is performed with the **confusion_matrix** command from sklearn.metrics, which return a matrix matrix $C$ such that $C_{i,j}$ is equal to the number of observations known to be in group $i$ and predicted to be in group $j$. Then each row of the matrix is normalized over the number of true labels in order to return the relative frequencies for each label. In addition to this, the **accuracy** over the test set is computed. By looking at the confusion matrix and the accuracy value (0.881), it is possible to see that even with a very simple model, the classification performance of the network is pretty good (fig. 5).

### 2.2.1 Weights visualization and Activations

Given the fact that the input samples are $28 \times 28$ images, also the number of **weights** of each unit of the first linear layer would be vectors 28*28=784 long. So by reshaping them in the image format, it is possible to plot them as images that **encode some important features** of the inputs. In fact some of this weight plots display the shape, although quite noisy, of some kind of cloth, meaning that probably that specific unit is specialized in recognizing that specific kind of sample. And there are also more than one unit which display similar shapes,

and this means that there is some kind of redundancy in the net, which makes it more robust. Some other images display instead some more generic features, as shades or fill-empty zones (fig. 6).

The **activation** of the neurons of some linear layers can be visualized with the same procedure of the regression task. In some cases it is possible to see that input belonging to the same class produce similar activation patterns in the first layer. And also similar objects sometimes show similar patterns, as in the case of T-shirt/top vs Shirt (fig. 7).

#### 2.2.2 Images transformation

Now the network is trained with the dataset in which *RandomVertical* and *RandomRotation* transformation are applied. The dataloaders are redefined and all the main commands for training and results showing of the network are called.

The network gets retrained with this new dataset and the performances are evaluated by analyzing the confusion matrix and the accuracy value. As it is possible to see, the performances are less good than in the case in which the structure of the images wasn't changed (accuracy 0.814). In fact, the network cannot anymore create weight patterns similar to the original images and compare them with the original sample, because the rotation and flipping of them makes this procedure useless. So the network has to identify local and simpler patterns, similarly to the way a convolutional layer operates. This reasoning can be verified by looking at weights images (fig. 6).

#### 2.2.3 Skorch

Training the model with Skorch in the classification task results in a great increase of the computational speed. That's because the sklearn wrapper only allows to get as input torch Tensors, and not dataloaders. This is possible in this case because the dataset is not too large and can fit in the RAM memory, but for larger dataset this wouldn't be recommended. But, after realizing the advantages and disadvantages of this methods, Skorch training results much faster than the manual one in our case, and for this reason in use in the hyperparameters selection sections. Skorch also allows to use the *cross_val_score* of sklearn on the pytorch model of the network. It computes the accuracy of the validation set by applying the **KFold cross validation** procedure.

#### 2.2.4 Hyperparameters optimization

For the reasons explained before, in the hyperparameters optimization section Skorch and *cross_val_score* are used in order to have a fast and robust training and evaluation of the network for each trial of hyperparameters combinations. Only the Optuna method is used, because it leads to better results in less time, as the research get closer and closer to the combinations that produce better performances at each iteration.

From the results (tab. 2) it's possible to see that the ReLU activation, with suitable values of learning rate and weight decay, leads to better results. Another important factor is the optimizer: Adam works better in almost all the cases, but it's difficult to state its superiority in the task, because SGD could work as well but in different ranges of the learning rate, which are hard to explore.

### 2.3 Convolutional Network

After the analysis of the Linear Network for classification, the performances of a classifier composed also by **Convolutional layers** are explored. This kind of layers in principle are very useful tools when dealing with images, as in our case. Another type of layers added in the module, is the **BatchNorm2D**, which perform a normalization over the weights of the network with the goal to speed up the training, perform some kind of regularization and avoid gradient problems as exploding or vanishing gradient.

The network is trained as the one before, on both the transformed image's dataset and the clear one, and its performances are encapsulated in the confusion matrices and accuracy values. In general it has better results than the linear one, in both the cases, with accuracy respectively equal to 0.843 and 0.893. This is because the convolutional layers are able deal in a better way with **local patterns**, which in general is the most efficient method to analyze images (fig. 8).

All the following results were taken by considering the network trained on the transformed dataset.

#### 2.3.1 Weights and Activations

The weight plots of a convolutional layer are at first sight less interesting than the one of a linear layer, because their purpose is just to recognize local patterns of the image, as shades, angles, edges, and for the small kernel dimension used for the small dimension images of the dataset, it's even hard to recognize such patterns (fig. 9).

Much more interesting is to look at the activation of a convolutional layer. In this case we consider the first layer of the network, and the result is a series of images, each of a smaller dimension than the original one,

due to the padding and the kernel dimension, which shows how the image is filtered by each of the pattern recognition that compose the convolutional layer (fig. 10). For the second convolutional layer the activation images result less clear, but works in the same way of the first case. The difference here is that they are not applied on the original image, but on the "level of activation" of each neuron of the previous layer with respect to the corresponding pattern represented by the kernel weights.

### 2.3.2 Hyperparameters optimization

In this final case only the Optuna optimization is performed, because is computationally less expensive and, for a fixed number of iterations, it leads to better results than the GridSearch. From the results (tab. 2) it's possible to see that the dropout contribution is important, and values around 0.1 usually lead to better performance with respect to the case in which the dropout probability is 0. The best activation is the ReLU, a likely result for a convolutional network used for a classification task.

## 2.4 Results

The convolutional network performs better both in the case of transformed and not transformed images, as predictable for its higher affinity with the problem structure. On the other hand the linear network is more understandable in its way to operate because it's easy to see which kind of feature it searches into the images. All the accuracy values reached by evaluating the test set are summed up in tab 3. Both the hyperparameter search procedures leads to slightly worse results with respect to the case in which the parameters were chosen a priori. That could be probably due to the fact that the Optuna algorithm, as said before, sometimes gets stuck in local minima and it's hard for him to escape. This said, the results are generally quite good.

| | By eye | Bayes Search | Bayes Search KFold | Random Search |
|---|---|---|---|---|
| **Train-Validation ratio** | 0.9 | 0.6 | - | - |
| **Batch size** | 20 | 42 | - | - |
| **Number of units** | 10 | 15 | 12 | 29 |
| **Depth** | 3 | 2 | 2 | 2 |
| **Dropout** | 0 | 0.053 | $8.2 \cdot 10^{-4}$ | 0.13 |
| **Activation** | Sigmoid | ReLU | Tanh | Sigmoid |
| **Weights init.** | Default | Default | Default | *Normal* |
| **Learning rate** | 0.01 | $6.2 \cdot 10^{-3}$ | $7.5 \cdot 10^{-3}$ | $6.3 \cdot 10^{-3}$ |
| **Weight decay** | $10^{-5}$ | $2.3 \cdot 10^{-5}$ | $1.4 \cdot 10^{-5}$ | $3.0 \cdot 10^{-6}$ |
| **Optimizer** | Adam | Adam | SGD | Adam |
| **TEST LOSS** | 0.0327 | 0.1496 | 0.0864 | 0.1635 |

Table 1: Result for the hyperparameters optimization for regression task

| | **Dropout** | **Activation** | **Learning rate** | **Weight decay** | **Optimizer** |
|---|---|---|---|---|---|
| LINEAR | 0.17 | ReLU | $6.9 \cdot 10^{-5}$ | $9.7 \cdot 10^{-5}$ | Adam |
| CONVOLUTIONAL | 0.09 | ReLU | $5.8 \cdot 10^{-4}$ | $3.4 \cdot 10^{-5}$ | Adam |

Table 2: Result of the hyperparameters optimization, for both the linear and the convolutional classification networks.

| **LINEAR** | | | |
|---|---|---|---|
| | By hand; No transf. | By hand; Transf. | Hyp. opt.; Transf. |
| Accuracy | 0.881 | 0.814 | 0.792 |
| **CONVOLUTIONAL** | | | |
| | By hand; No transf. | By hand; Transf. | Hyp. opt.; Transf. |
| Accuracy | 0.8929 | 0.843 | 0.821 |

Table 3: Accuracy values for the different classification network architecture in the cases of training dataset with transformed and not transformed images.

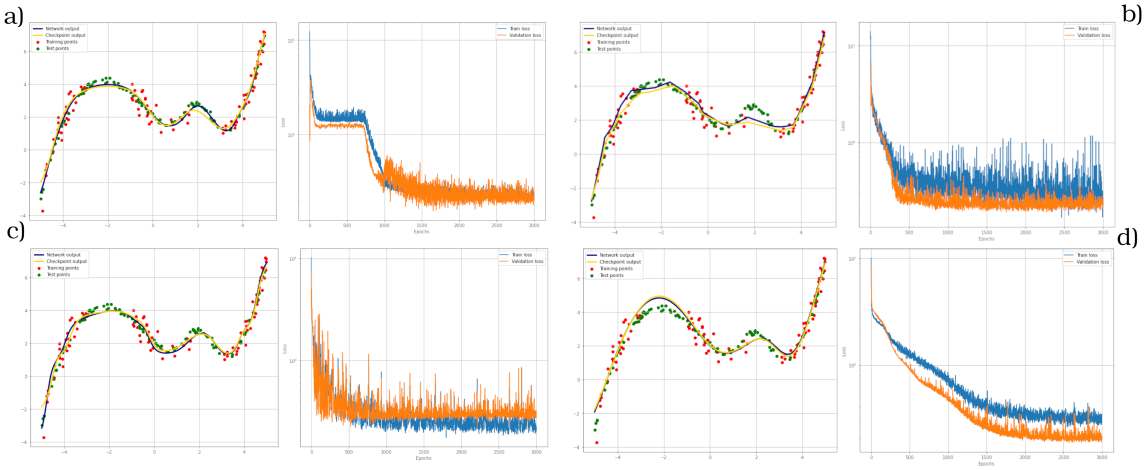


Figure 1: Each image is composed by 2 plots: on the left the output of the net. With the blue line the output of the net at the final epoch state, in yellow the output of the state in which the minimum value of the validation loss is reached. On the right the loss trend, blue for the training, orange for the validation. a) Case of hyperparameters tuned "by eye". b) Results of the Bayesian hyperparameters search. c) Results of the Bayesian hyperparameters search with KFold validation procedure. d) Results of the Random hyperparameters search.

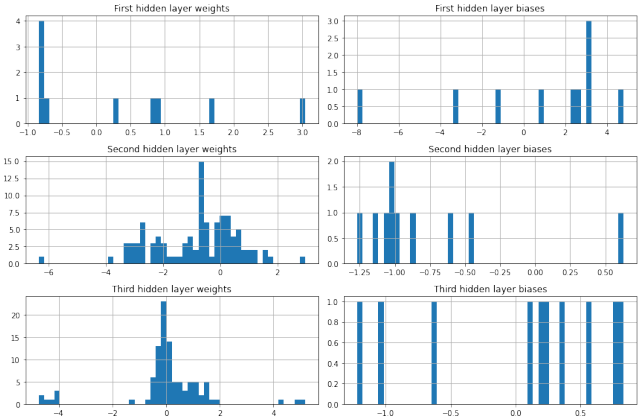


Figure 2: Weights and biases distribution for different layers of the network.

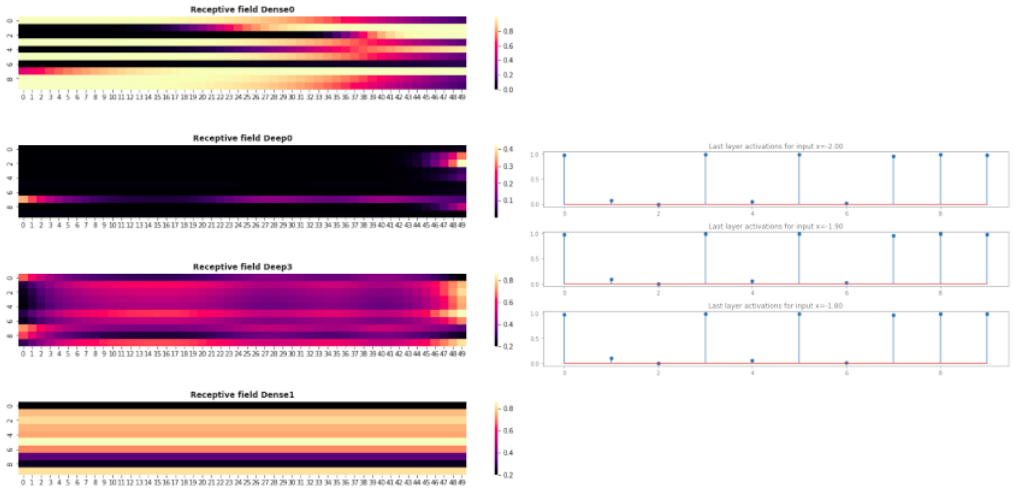


Figure 3: On the left: receptive fields for each neuron of each layer of the net with respect to possible input. On the right: activations of the neurons of a specific layer.

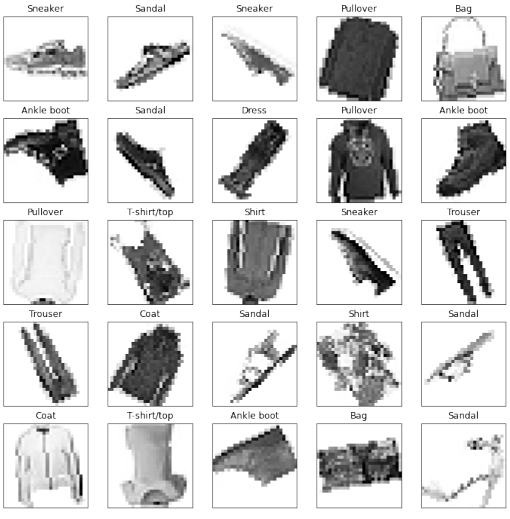


Figure 4: Some images taken from the training FashionMNIST dataset with some transformations applied on.

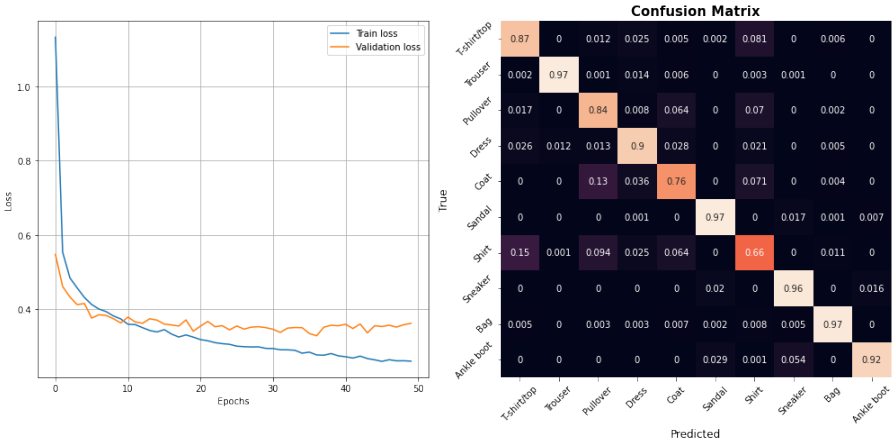


Figure 5: On the left: the training (blue) and validation (orange) loss trend with respect to the epochs. On the right: confusion matrix built with the prediction of the Linear classification network trained with images not transformed. The accuracy is 0.881

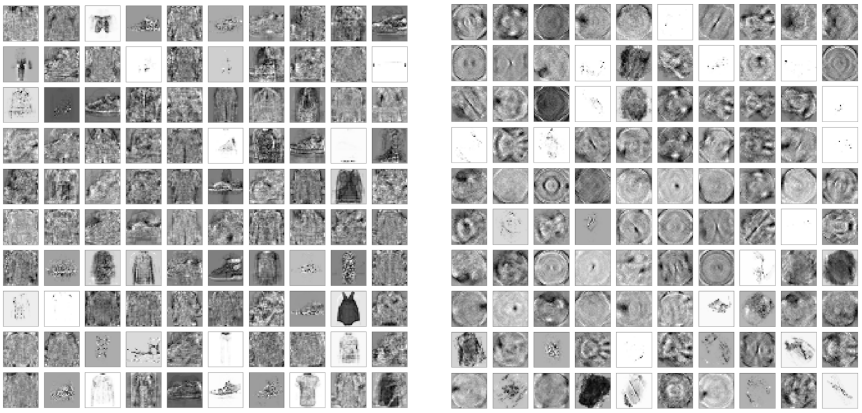

Figure 6: On the left: weights of the first convolutional layer in the case of training dataset with no image transformation. On the right: weights of the first convolutional layer in the case of training dataset with image transformation.

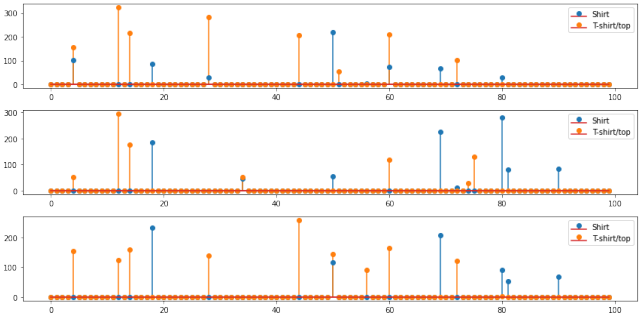


Figure 7: Comparison between the activations of the neurons of the first layer of the linear net for some samples labelled as shirts (blue) ans some other corresponding to T-shirts (orange)

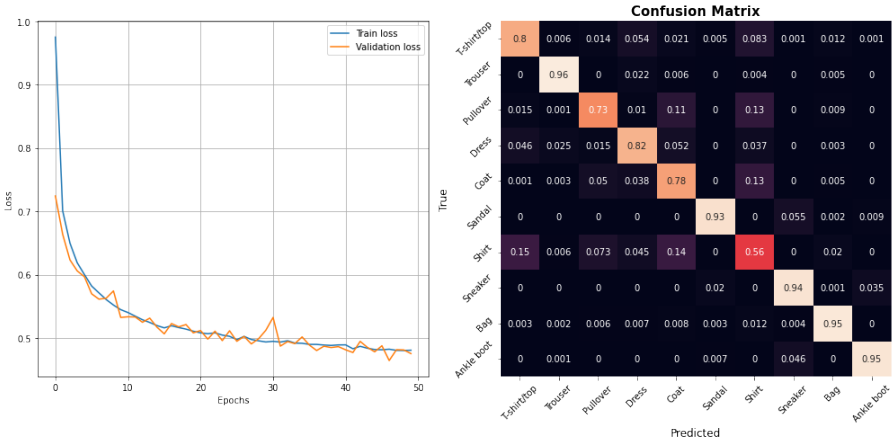


Figure 8: On the left: training and validation loss trends. On the right: confusion matrix for the output of the Convolutional classification network trained with the transformed images. The accuracy is 0.843

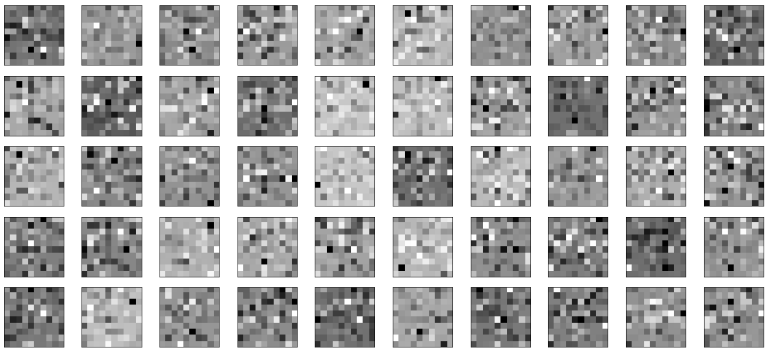

Figure 9: Weights of the first convolutional layer of the convolutional classification network.

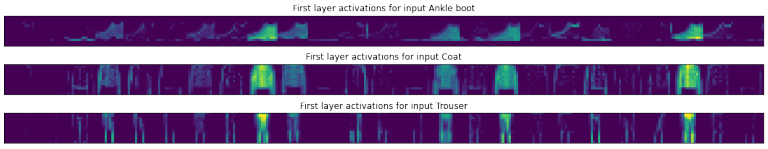


Figure 10: Activations of the first convolutional layer for three different types of samples.