# Supervised Deep Learning

Tommaso Faorlin (2021857)

Università degli Studi di Padova
tommaso.faorlin@studenti.unipd.it

## I. Introduction

In this report we solve the first homework of the course in Neural Networks and Deep Learning. The topic discussed is supervised deep learning: we are going to design and optimize some deep neural networks in order to perform a *regression* and a *classification* task within the `Pytorch` framework.

## II. Regression

### I. Dataset

We would like to approximate, with a Fully Connected Network (FCN), a function $f(x)$, having only some noisy outputs $\hat{y} = f(x) + \eta$ of it (represented in Fig. 1). The training set consists of 100 samples, stored in a labeled fashion $(x_i, y_i) \in \mathbb{R}^2 \; \forall \, i = 1, \dots, N$. This dataset is incomplete: some points are missing in the intervals $[-3, -1]$ and $[2, 3]$: the challenge consists on training a model able to recover the behavior of the original unknown function also on the missing domain pieces. In a supervised learning framework, $y_i$ plays the role of the label when we train a model $f : x \to y$. Another 100 points are contained in the test dataset, and we will use them as a final benchmark for the trained models.

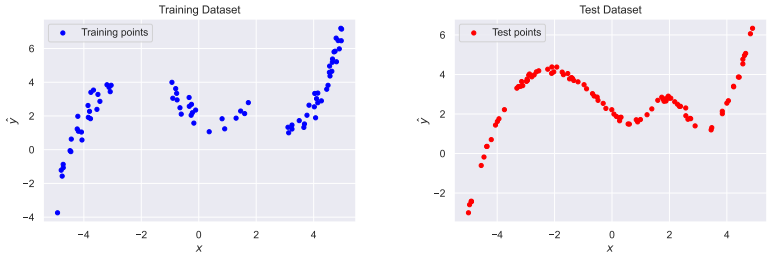


**Figure 1:** *(**Left**) Training points, 100 in total. (**Right**) Test points, 100 in total.*

### II. Simple model

The train dataset is small and we divide it in 80% effective training set and 20% validation set. In this case, a simple model is enough to solve the task, so we choose to build a first FCN architecture with two hidden layers with 16 neurons each. We run the code different times and we try by hands to optimize the hyper-parameters.

The learning is performed iteratively, minimizing in 500 epochs the `nn.L1Loss()` (Mean Absolute Error MAE $x$ and $y$) with a backpropagation algorithm, the `Adam` optimizer and a fixed learning rate $lr = 0.005$ (chosen by hands after few iterations of the algorithm, more sophisticated methods will follow). The results for the losses and for the final function are shown in Fig. 2. As we can observe, this very simple model is able to approximate the behavior of the function, except in the second hole of the domain. We also note how training the model on different random permutations of the training and validation set leads to completely different results, signaling a low generalization ability of the model.

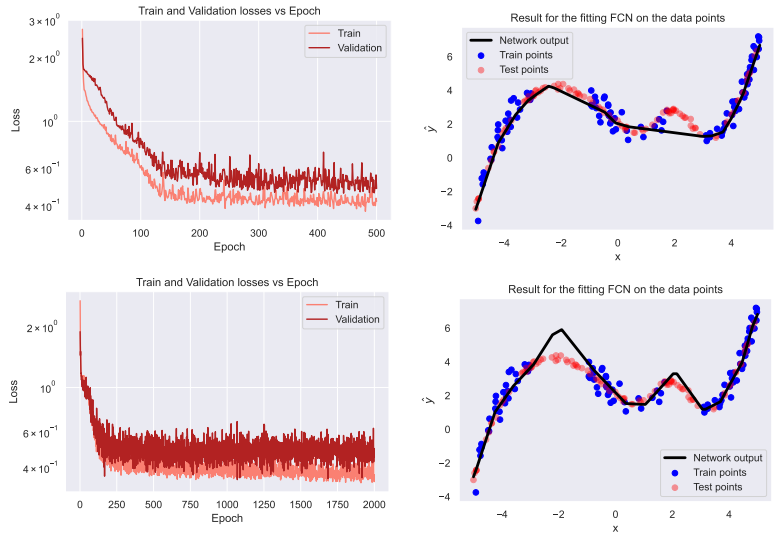


**Figure 2:** *(**1**) Train and validation losses for the simple model described in Section II. (**2**) Forward pass for the simple model: we recover the behavior on the first missing domain, but the model draws a straight line between the middle and the end of the domain, not following the real behavior of the data. We must observe how the model misbehave although some points of the 'second ascent' are present in the train dataset. (**3**) Train and validation losses on the simple model, but after 2000 epochs. With the help of (**4**) we notice how this model overfits the data: the train loss keeps decreasing while the validation loss slightly starts to increase. To avoid this we will implement early stopping and increase regularizations.*

## III. Optimization of an advanced model

With respect to the simple model, we define an additional fully-connected layer and we add a small $L2$ regularization factor, i.e. a term added to the loss function and proportional to the sum of the weights squared, that forces them to be smaller (a constraint in the minimization) and so avoid overfitting (very probable with this simple model). Another regularization technique was considered but not used in the end: the dropout of probability $p$, where a percentage $p$ of neurons are switched off during the training procedure. We did not use it in the end because from previous runs we noticed that in the best model, the value of $p$ was always almost 0.

Since the number of data points is not too high, we perform a Random Search (RS) on 400 models with the custom function `random_GS`, to find the best architecture and hyper-parameters of the network. For each model, we sample a set of parameters with the function `random_choice` from the following dictionary `grid` and we build the model and the optimizer:

```
grid = {'net':    {'Ninp': [1], 'Nout': [1], 'Nh1': [4, 8, 16], 'Nh2': [4, 8, 16], 'Nh3': [4, 8, 16],
                   'act_func':   [nn.ReLU(), nn.LeakyReLU()]
                   },
        'learning':{'n_models':   [400], 'num_epochs': [300], 'folds': [5], 'device': [device],
                   'batch_size': [5,7,9,10],
                   'opt':        ['SGD', 'Adam', 'RMSprop'],
                   'lr':         loguniform.rvs(1e-4, 1e-1, size=100),
                   'reg':        loguniform.rvs(1e-5, 1e-1, size=100)}}
```

We also perform a K-Fold cross-validation with $k = 5$, in order to enhance the generalization capabilities of the model, which are limited by the smallness of the dataset. With this method, the training set is divided in $k$ parts of the same dimension. The model is trained for 300 epochs on each of the remaining $k - 1$ sets and then evaluated on the set that was put apart.

### IV. Results

The best model, result of the random grid search with a loss of 0.58, is the following:

| Nh1 | Nh2 | Nh3 | act_func | batch_size | opt | lr | reg |
|---|---|---|---|---|---|---|---|
| 16 | 16 | 16 | ReLU() | 10 | Adam | 0.01201 | 0.00044 |

**Table 1:** *Best model after random grid search (`L1Loss()` in validation of* 0.58*).*

We then train a model, on 500 epochs, built on the parameters reported in this table, and we take it as final reference. We do not perform the cross-validation with the final model, as it is optimized in this way before, but we split the training set in 80% effective training set, and 20% validation (as with the first simple model). In the end, we obtain a very good result shown in Fig 3, and we must point out that in some unlucky runs of the model, i.e. when there are too few points in $(1.5, 2]$, we still do not recover perfectly the behavior for the second hump. Therefore, generalization capabilities of the model are better than the previous case but not completely perfect.

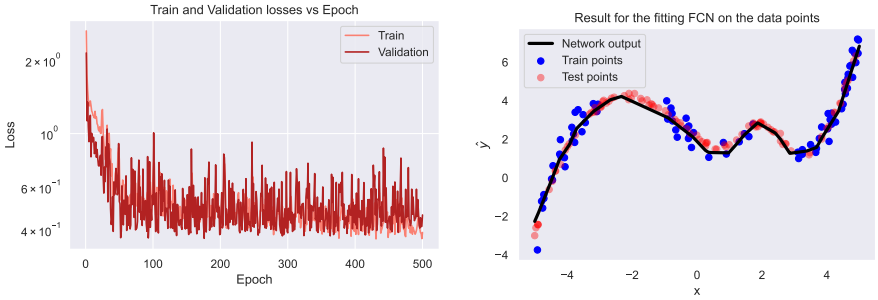


**Figure 3:** ***(Left)*** *Train and validation losses for the advanced model. (Right) Forward pass for the advanced model.*

We have been able to train a model capable of tracing the behavior of an unknown function, given a incomplete dataset of its point in the $\mathbb{R}^2$ plane. We also report in Fig. 4 the activation profiles of each hidden layer, and in Fig. 5 their weights for a different run of the optimization.

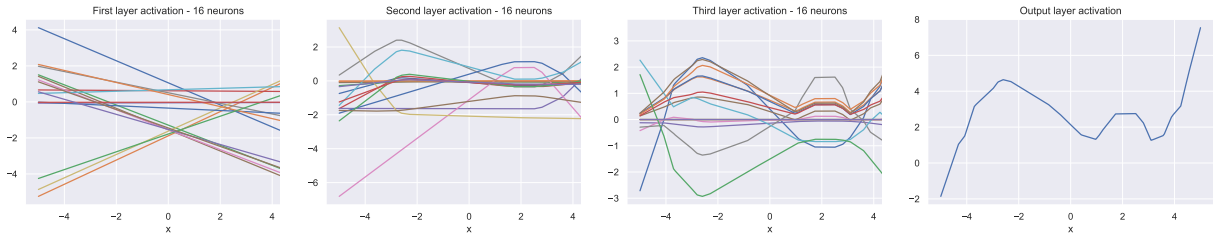


**Figure 4:** *Activation profiles for each layer of the FCN network. The complexity increases with the depth of the layer under examination. The first layer is specialized in the distinciton between negative and positive value, on the third layer we can already observe that some neurons are already giving the shape of the final function while some are still deactivated.*

## III. Classification

### I. Dataset

We perform a classification task on the well-known Fashion MNIST dataset. The latter contains a total of 70000 images of $28 \times 28$ black and white clothes, and we should build a model able to address each image to its correct class among the 10 available. We avoid to perform a cross-validation since the dataset is large,

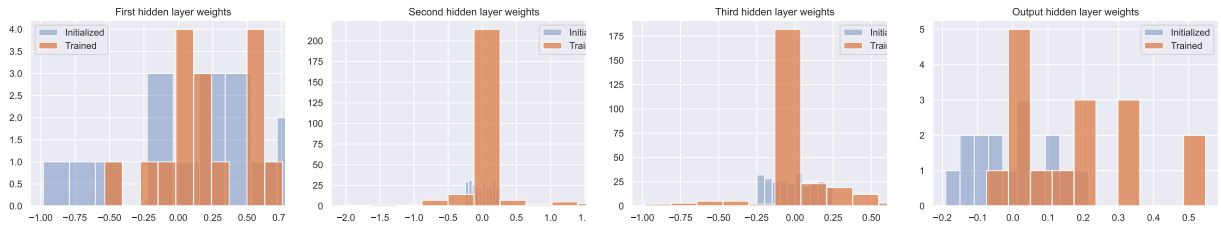


**Figure 5:** *Weights visualization for each layer of the FCN network. At the beginning, they are uniformly distributed.*

we instead split the training set, consisting of 60000 samples, in 80% for effective training and 20% for validation. We keep the test set only for a final generalization benchmark of the model, and we plot here some samples:


**Figure 6:** *Some samples coming from the FashionMNIST dataset.*

Furthermore, we perform data augmentation on the dataset: the data we already have is transformed in order to increase the diversity and help our machine learning models generalize better and reduce overfitting. In particular, we apply the following transformations to the original dataset and we report some examples in Fig.7:

- `Normalize()`. This is not properly considered as data augmentation but it is used to transform the dataset to its normalized version (mean zero and unitary standard deviation);
- `RandomCrop()`. The objects of interest we want our models to learn are not always wholly visible in the image or the same scale in our training data. So, this creates a random subset of an original image, helping the model in generalization;
- `RandomErasing()`. Combined with the previous method, this one selects a region of the image and fills it with random pixels.

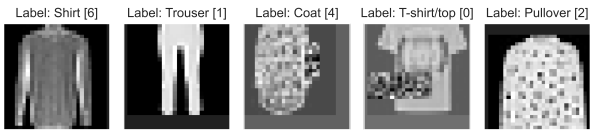


**Figure 7:** *Some samples coming from the FashionMNIST dataset after data augmentation. In some cases the transformations totally modify the starting image making it unrecognizable to the human eye, as happens in the third case with the coat.*

## II. Method

Unlike for the regression task, here we use `Skorch`: we define the model in `model.py` with the class `ConvolutionalNet` and instantiate a `NeuralNetClassifier` object. The model in use has the architecture represented in Fig. 8.

We perform a `RandomizedSearchCV` over 100 models for 10 epochs each. Hyper-parameters are searched within the following dictionary:

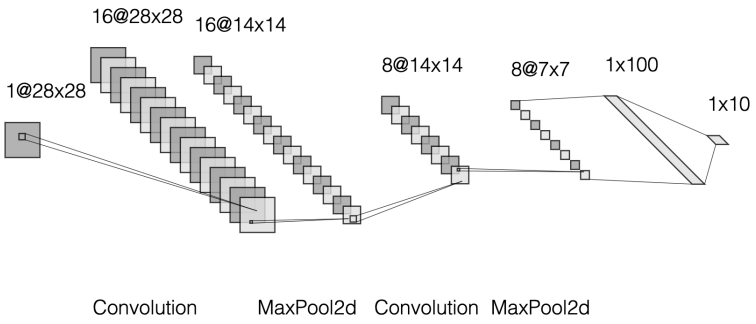


**Figure 8:** *The chosen architecture. Both the convolutional layers have* `stride` $= 1$ *and* `padding` $= 2$*. This particular setup has been chosen balancing the time required for its optimization and the final result in terms of validation accuracy. Credits to* `http://alexlenail.me/NN-SVG/LeNet.html` *for the drawing tool.*

```
params = {"module__prob_drop"          : (loguniform.rvs(.05, .65,  size = 10)),
          "module__prob_drop2"         : (loguniform.rvs(.05, .65,  size = 10)),
          "module__act_func"           : [nn.ReLU(), nn.LeakyReLU()],
          "batch_size"                 : [64,128],
          "lr"                         : loguniform.rvs(1e-4, 1e-2, size = 20),
          'optimizer'                  : [optim.Adam],
          'optimizer__weight_decay'    : loguniform.rvs(1e-5, 1e-2, size = 20) }
```

## III. Results

The best model, built on the following parameters, achieves an accuracy of 90.54% in validation and 90.04% on test.

| prob_drop | prob_drop2 | act_func | batch_size | opt | lr | reg |
|---|---|---|---|---|---|---|
| 0.1019 | 0.0536 | ReLU() | 64 | Adam | 0.0002 | 0.0004 |

**Table 2:** *Parameters of the best model after random grid search.*

The latter is trained again on a larger number of epochs (30) with early stopping (patience 10 on validation loss). The learning curves are presented in the following figure:

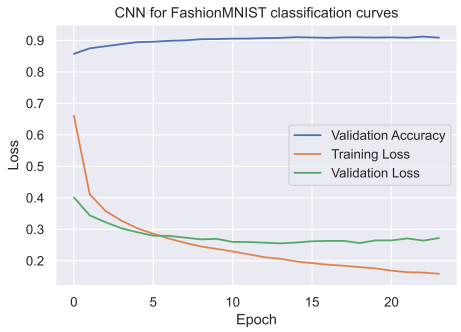


**Figure 9:** *Train and validation losses for the final model. 30 epochs, early stopped at 23.*

The 8, $5 \times 5$ images shown in Fig. 10 are the convolutional filters of the first hidden layer (also called weights). The one for the second layer are not reported since they are very similar, also in dimensions,

to the one already present. Those are important for the network to learn, but it is always very difficult to understand how it is able to do so. Instead, it is more intuitive to look at the activation profiles of the network, that is, how the initial image is transformed in the deep layers of the network. They are reported in 11 and we can understand how the first layer concentrates more on the edges, while it is already complex to understand what happens after the second convolution.

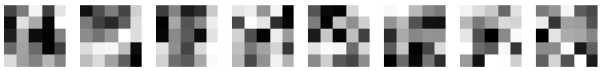

**Figure 10:** *Filters of the first convolutional layer.*

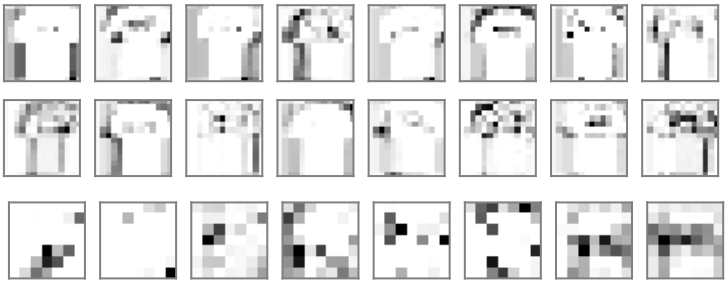

**Figure 11:** *Activation profiles of the network. (**First two rows**) After first convolutional, activation and max-pooling layers. (**Last row**) After second convolutional, activation and max-pooling layers (just a subset of the total activation profiles is reported).*

Finally, we report the confusion matrix for the final model on the test set. As we can infer, shirts are the most difficult objects to assign a label to: this may be related to their similarity to other classes. The total misclassified samples are 996 and in the notebook are reported also some examples of them.

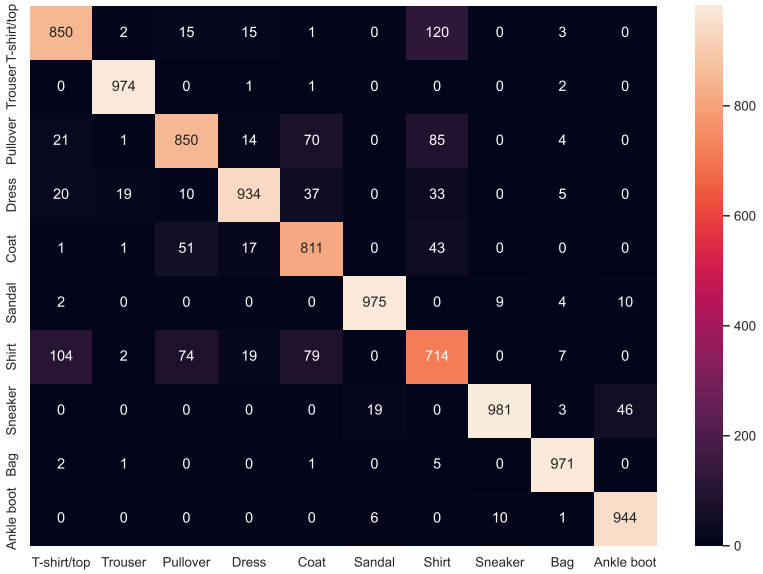


**Figure 12:** *The confusion matrix on the test set for the final CNN model. Predicted label on the x axis, true label on the y axis.*