Fully Connected Feed-Forward Neural Network in PyTorch on MNIST Dataset

Matteo Guida[†]

[†] *Physics of Data M.D. - Dipartimento di Fisica e Astronomia G. Galilei, Università degli Studi di Padova, Via Marzolo 8, I-35131 Padova, Italy.*

January 10, 2021

1 Introduction

In the present work a fully connected neural network and the search for the best model class among the ones defined by different configurations of selected hyperparameters is implemented in the machine learning framework PyTorch. The model is trained in order to tackle multi-classification problem over the MNIST, a dataset of 60,000 small square 28×28 pixel grayscale images of handwritten single digits between 0 and 9. In the first part an overview of the implementation of the program and choices made is given. In the second part some results related to the best found hypothesis are presented.

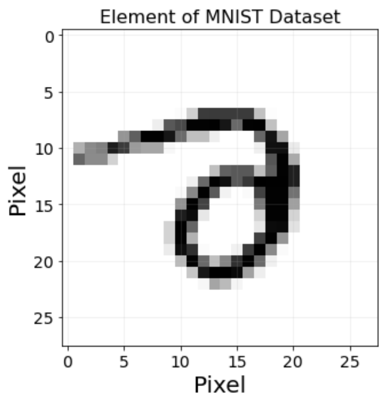

Figure 1: Example of element of MNIST dataset with true label 6

2 Program description

The assignment is solved through one python script FFNN_Pytorch.py and a jupyter notebook EX2_Matteo_Guida.ipynb. In the script FFNN_Pytorch.py the class Network is defined with the initialization and other 5 methods plus a function not contained inside the class (get_list_rand_search).

```
class NN_model(nn.Module):
def __init__(self, params_dict):
def set_network(self,params_dict):
def forward(self, x, additional_out=False):
def accuracy(self, x_test,y_test):
def trainin(self,data_train, label_train,
    data_val, label_val, params_dict,
    early_stopping=True, return_log=False,
    verbose=False,save_model=False):
def Kfold_cross_val(self, params_dict, x_data,
     label,k_fold=5):

def get_list_rand_search(params_dict):
```

Following an up-down approach to describe them, firstly, in the jupyter EX2_MatteoGuida.ipynb. the dataset is divided into 90% training set and 10% test set and it is transformed in the adequate torch.Tensor objects. As the laptop, on which the work is performed, has a dedicated GPU, that device is selected. The tensors and the network are allocated there in order to exploit the GPU during the training greatly speeding up the execution time. The features of the network are quite similar to the ones implemented in the assignment 1, but there are some important differences which are summarised in the following.

Thanks to the package torch.optim we use Adam and Adamax **optimizers** to update the weights during the training. Without dwelling, one of the key difference with gradient descent is that the learning rates for each parameter in the network is individually adapted from estimation of first and second moments of the gradients. The values of the optimizer parameters are kept to the default values set by the methods, we only fix the weight_decay parameter in order to incorporate an **L2 penalty**.

Another important addition is the use of **dropout**, specifically during the training all the neurons are set to zero, multiplying its output value by zero, with a *certain probability* removing it from the network and for this reason also all their incoming and outgoing links. Too high values of p can lead to underfitting because it reduce too much the complexity of the model.

There is no more an activation function for the output layer because the used **loss function** nn.CrossEntropyLoss(), adequate to classification problem with N classes, already does the log_softmax of the last layer neurons' outputs.

Among the possible improvements of the current implementation for sure could be the possibility to include an an arbitrary number of hidden layers and the use of mini-batch gradient descent. The last method is necessary when you work with very large datasets and networks, for the preset task we can efford to have the highest accuracy and stability in the estimate of all the gradients.

Differently from the previous assignment a **random search** for the best hyperparameters is used. The computational cost of grid-search in fact exponentially grows $\mathcal{O}(n^m)$, where m are the hyperparameters and n the values which it can assume. When there are several hyperparameters that do not strongly affect the final performance, random search converges much faster to good values of the hyperparameters and allows to *explore a larger set of values*. In this approach is necessary to define a marginal distribution for each hyperparameter from which at each step the value of the hyperparameter is extracted. In table 1 the values of the hyperparameter used for random search is reported.

# Neurons	[50,600]
Activation	[nn.LeakyReLU(),nn.ReLU(),nn.Hardswish()]
Optimizer	[torch.optim.Adam, torch.optim.Adamax],
Dropout	[0.3,0.5],
# Combinations	[200]
# Epochs	2000

Table 1: Values of the hyperparameters used for random search, # neurons and dropout have the intervals of the distributions, for activation and optimizer there are the possible values among some option, while the other parameters are fixed in the training procedure.

To fix the number of neurons in the hidden layers we assumed that an architecture for which a constant compression rate is present, i.e. the ratio between the number of neurons between the considered layer and the previous one, makes *theoretically* sense. If we call w the compression rate, considering $N_{h_0} = 784$ and $N_{h_3} = 10$ we have that $N_{h_n} = w^n h_0$. At this point the number of neurons are extracted from a Gaussian distribution **priors**:

1. $N_{h1} \sim \text{gauss}\left(\mu = \sqrt[3]{\frac{N_{h_3}}{N_{h_0}}} N_{h_0},\, \sigma = \sqrt{\frac{N_{h_3}}{N_{h_0}}} N_{h_0}\right)$
2. $N_{h2} \sim \text{gauss}\left(\mu = \sqrt[3]{\left(\frac{N_{h_3}}{N_{h_0}}\right)^2} N_{h_0},\, \sigma = \frac{N_{h_3}}{N_{h_0}} N_{h_0}\right)$

As can be notice, $\mu_{N_{h_n}} \overset{!}{=} w^n h_0$ and $\sigma_{N_{h_n}} \overset{!}{=} w^{n/2} h_{N_{h_n}}$ in this way the first layer has a larger variance than the second layer and with higher probability a larger interval of values can be explored fixed the number of extractions. Two low-key adjustments are considered, firstly the distributions does not return integer values, so the outputs are round to the nearest integer, secondly there is an extremely small probability that outputs are negative numbers, in that case the extractions are repeated.

After having clarified those aspect, going ahead in code flowing, by the method get_list_rand_search a list of lists with all the values of the hyperparameters for a number of combinations model classes is created. After that, in a for loop for each iteration a new dictionary corresponding to a single model class is created and passed to NN_model. A standard **k-fold cross validation** is used to perform model selection with k = 5 and the models and results are saved in the file train_results.data, making a rankings based on the average validation accuracy, estimated thanks to the function accuracy in each of the 5 iterations in k-fold procedure.

2.1 Results

In table 2 the best model class among the ones extracted through random search is shown.

# Neurons	[784,249,36,10]
Activation	ReLU
Optimizer	Adam
Dropout	0.35
Learning rate	$5.4\ 10^{-3}$
Penalty	$1.2\ 10^{-4}$
Epochs	2000

Table 2: Hyperparameters for the final model.

In figure 2 the learning curve for the selected hypothesis class is presented, please notice that the early stopping condition is always applied. We retrain it over the entire training set and use the accuracy on test set to evaluate the performance.

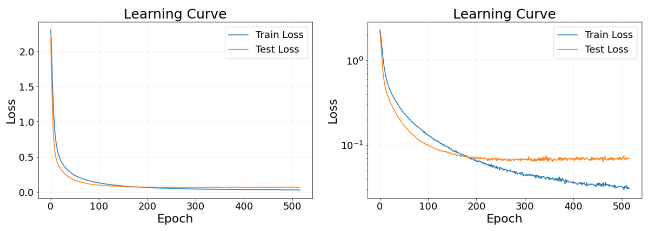

Figure 2: Learning curve for the selected hypothesis.

The final accuracy on test set is:

$$\boxed{\text{ACCURACY} = 97.98\,\%}$$

2.2 Visualization Features Encoded by Neurons

The scope of this section is to try to have an insight on how and what information is encoded in the neurons of the trained selected hypothesis. The following study is restricted to the neurons of the final layer, basically for simplicity and because it's known that each of them is somehow related to each digit. First of all we extract the matrices related to the weights, excluding the biases, and we multiply them to obtain the **receptive fields**. We obtain as a result a matrix with dim = 10 x 784 and resizing each column of dimension 784 in a 28 x 28 matrix (as the starting images) the ten plots shown in figure 3 are obtained.

Even making an effort to recognise some patterns related to the 10 digits, without the titles of the single plots one could not established who is referred to whom. A possible key to understand the unsatisfactory results is the fact that biases and activation functions are not taken into account.

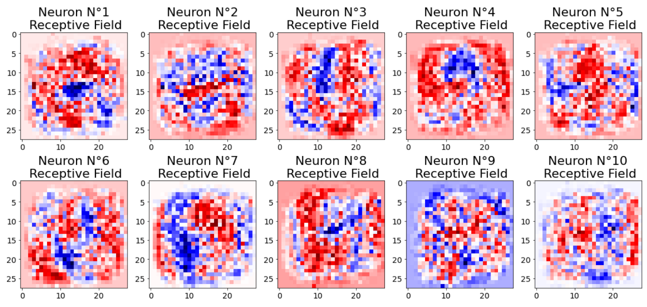

Figure 3: Receptive field for the neurons in the last layer.

We give a more advanced method a go in order to try to visualize something meaningful. The **gradient ascent over the image pixels** aims to synthesize an image, whose pixels initially are set randomly with values $\in [0, 1)$, so that it maximizes the activation of a given neuron, in our case the ones in the last layer. W.r.t the training procedure the weights are fixed and the gradient of the output of a neuron w.r.t. each pixel of the starting image is computed and it is used to update the pixel values. The final result is shown in figure 4 and are obtained with 3000 epochs and learning rate 0.01.

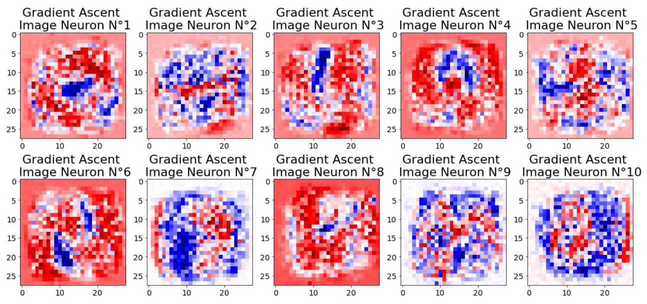

Figure 4: Synthesized images which maximize the output of the neurons in the last layer.

The results does not seem to lead to positive advances, a remarkable fact is that the obtained images are significantly consistent with the ones obtained with the previous method.