Università degli Studi di Padova

Dipartimento di Fisica e Astronomia “Galileo Galilei”

Corso di Laurea in Physics of Data

Exercise 2 Deep Learning and Neural Networks

Author

Marco Agnolon

Abstract

The aim of this exercise was to implement a neural network that is able to classify the MNIST dataset.

Anno accademico 2019-20120

Data prepocessing

It was given a subset of it that is composed of 60000 images and the correspondent labels. The images represents handwritten digits. These data are contained inside a MatLab file.
The dimension of the image is a vector of size 784 that represents a square image 28x28. Figure 1 shows some of these digits with their correspondent labels. As one may see the digits are rotated and also mirrored but this should be insignificant for the learning task. Maybe it has to be considered when analyzing the receptive fields of the neurons.

First one has to import the data and split them in train and test set. This is done using a scikit function called *traintestsplit*. It was chosen as dimension of the test set 1/6 of the all dataset.

The neural network

The hyper-parameters

The network created has two hidden layers network. The dimension of the two hidden layer will be two hyper-parameters of the model and they are called respectively Nh1 and Nh2. The input layer has 784 neurons, one for each component of the input vectors. The output layer has 10 neurons as the number of the possible classes to detect. There were made attempt to introduce dropout layers but the results found with them were poorer with respect to the ones without them and so they were discarded.
Finally the a softmax function is applied to the last layer. This function returns a list of probabilities that a certain input belongs to the class represented by that output neuron. Hence an argmax is used to get the class that maximize this probability and so the proper output of the network. Then the result of the argmax is compared with the true label of that input using a Cross-entropy loss that is the typical one for multi-classification problems.

Before training the best network one has to set the hyper-parameters that work better:

- Number of neurons of the first layer (Nh1)
- Number of neurons of the second layer (Nh2)
- Number of epochs
- Learning rate

Instead as activation function was set the ReLU.
Let us investigate how to attack each hyper-parameter setting.

DataLoader

It is desirable to train the network using batches of data and not the all the dataset. In fact, in this case a mini-batch stochastic gradient descend leads significant improvements with respect to the standard gradient descend. This batches speeds up very much the computation and also the accuracy of the model.
Therefore it is important to load the data using the Pytorch function DataLoader that can split the data in batches of the desired dimension. Obviously when the cross validation is taken into account the DataLoader will split the appropriate subset of data instead of splitting the all dataset.

Random search 5-fold cross validation

For what concerns the choice of the activation function and the number of neurons, in the first attempt it was implemented a Grid search but, since it was supposed that the choice of the activation function would not be so determinant in the final result, it was preferred to use a random search, in order to span as widely as possible the plane where the other two hyper-parameters reside.

So it is decided that the range where the number of neurons of the two hidden layers must be draw out. It was noticed that the neural network works better if the first hidden layer has twice the number of the second hidden layer. Therefore parameters to test are sampled from two uniform distributions with averages one twice of the other (so in mean the first number of neurons will be twice the second one):

- Nh1: $\mathbb{U}(100,500)$
- Nh1: $\mathbb{U}(50,250)$

So at every step the program samples randomly two number from the two distributions. Then it does all the computation with these hyper-parameters fixed.
The scope of the random search is to find the best set of hyper-parameters between the tried ones.

In order to do this in a robust way it is implemented a 5-fold cross validation, it means that the train set is divided in five equal folds and the network with fixed hyper-parameters is trained over four over the five folds per time. The remaining fold is used as a validation set.
At every epochs the weights are update minimizing the error over each point of the four folds chosen for training and then the network makes the prediction over the fifth fold (the validation one), finally the error of these predictions is calculated. This procedure is iterated till the early stopping (that will be described later) is satisfied. Once the stopping condition is reached, the minimum error is retrieved from the list containing all the errors on the validation set (so this will be the minimum value of the trend of the validation error, it is the best results on that validation set). Also the number of epochs used to reach this minimum is saved. Now all this stuff is repeated exchanging the folds that will be the training set and the validation set, for the five times requested.

Finally five minimum validation errors and five max number of epochs are obtained, one of them for each validation set. The five errors are averaged, instead between the five number of epochs the maximum one is selected.
Thus, since this work is done for all the chosen set of hyper-parameters the average error is the fundamental result of all this computation, in fact is the number that allows to compare them. The best set is the one that has the lower average error among the five folds.

Early stopping

Now it is important to analyze the number of epochs of training for each training set. In fact if this number is too high the network will overfit the data, because, since a neural network can, in principle, implement any functions, if it is left training for too long it will adapt its weight in order to fit perfectly the training sample and as a consequence very badly the test ones.
The best thing to do, instead of choosing a arbitrary maximum number of epochs, is to implement an algorithm that is able to stop the network from training if it realizes that the net is starting to overfit the data.
The selected algorithm (Lutz Prechelt, *Early Stopping | but when?*, Fakultat fur Informatik; Universitat Karlsruhe D-76128 Karlsruhe; Germany, 1998) is based on the generalization error, i.e. the validation error. It stops the algorithm when the validation error at the last epoch is α times greater then the minimum error found till that epoch. In formulas:

$$GL_\alpha(t) = \frac{E(t)}{E_{min}} - 1 > \alpha$$

The parameter α represents the minimum percentage of the minimum error that is not allow to exceed. It is chosen accordingly to the necessity, the greater the value, the greater the goodness of the found minimum but the stop condition will arise later.

ADAMS

If one wants to use different activation functions the choice of the correct learning rate could be very tricky. Therefore, instead of using a fixed learning rate, or a decaying one it was used an algorithm that can change adaptively. ADAMS is a good solution and it is implemented inside Pytorch.

Results

Many random search were performed here the results of the best one of those are shown.
The network was trained over 10 different sets of parameters: Figure 2 shows the results for the random sets selected.

The best results in this random search was obtained using:

- Number of neurons of the first layer: Nh1=430
- Number of neurons of the second layer: Nh2=114

Then, the network with these parameters was trained over all the training point without leaving anyone as validation set. The training was stopped using the maximum number of epochs, between the five training on the 5-folds, saved before.
Figure 3 shows the trend of the training and the test error. It is clear from this figure that the two errors behave as expected: the train error tends to decrease and reach zero while the test error decreases till it reaches a plateau.
One may argue looking at the picture that there was possibility for the second one to further decrease (because the trend of the orange line seems to preserve a slightly decreasing behavior) but it is known that the test set cannot be use to improve the performances when training the network and so its error trend cannot be considered in order to stop the training.

So once the epochs had reached the max value chosen before the training was stopped and tested on the test set.
The accuracy found is: 0.9814

Discussion

Implementation and results

There were tried two different ways to implement the model. The first one consists of implementing everything manually using Pytorch, so all the "for" cycles used for the Random seach, the batch optimization and the cross validation. The second one using a specific Pytorch wrapper named Skorch that allows to using the modules defined with Pytorch inside a sklearn environment; this allows to use the routines implemented inside the sklearn library to perform random search and cross validation. Both the approaches are shown in the Python notebook.

They show very significant differences in performances and speeds. The network trained with the manual algorithm outperforms the Skorch one in both the comparison. Indeed it reaches better accuracies in a less number of iterations and also the training is two time faster. These differences might be due to many reasons: the skorch environment has to take a Pytorch model and make it comprehensible for sklearn and this may requires additional time at each iteration; the early stopping implemented in skorch is worse than the manual one, because it will stop the training if for a previously set number of epochs (patience) there were not improvements in the validation error. The manual algorithm, instead, uses an improved version of it as described in its own section. Thus, it was decided to use the manually implemented algorithm.

The network obtains a very high accuracy in the test set, some examples where the it fails are shown in Figure 4. It is clear from these pictures that the network was deceived by some digits that look very similar to what the network said.

Visualization

It could be interesting to visualize what some neurons of the network learned. This could be done for each hidden or output neurons by multiplying all the weights that reach it. The shape of the images will be 28x28 as the input ones.

Figure 5 and 6 show this for a neuron of the first and respectively second hidden layer. It were chosen to get the patterns for the first the middle and the last neurons of the considered layer. From these representations it is not clear what each neuron is learning. The patterns that these neurons are trying to extrapolate seems to be random. A sort of improvement might be seen between the first and the second layer however it is possible to recognize it only after having considered the patterns of the output layers.

Indeed, for what concerns the last layer with only ten neurons the situation is different. Figure 7 shows that each neuron is able to get a pattern that is similar to the correspondent digit. They are shown in ascending order, the yellow squares are trying to represent the number of that class.

So in light of these, now it could be easier to see that the second hidden layer is more similar to the last one and it is trying to capture some particularities common among similar the digits. On the other hand,

the first hidden layer is trying to guess some common features among all the digits and so it seems very different from the output layer.

Appendix

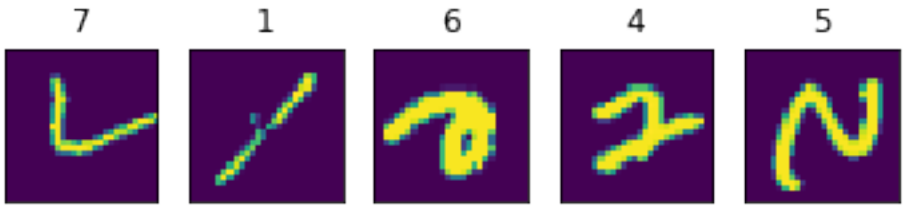

Figure 1: Some elements of the dataset with their labels

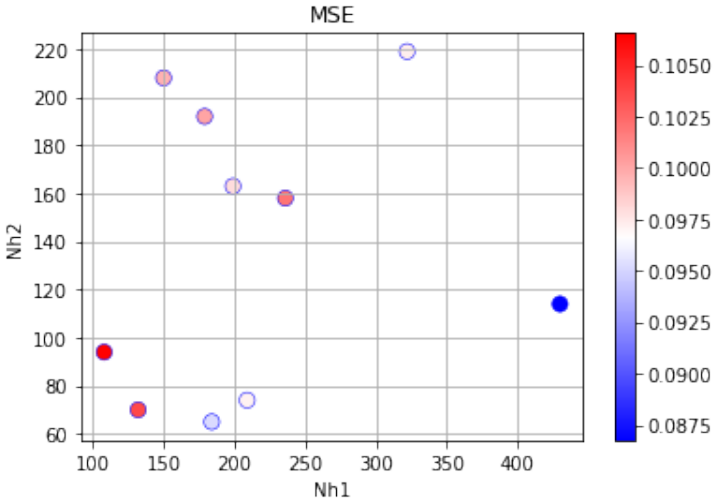

Figure 2: Each point is a couple of numbers of hidden neurons and its color represents the average error over five validation sets

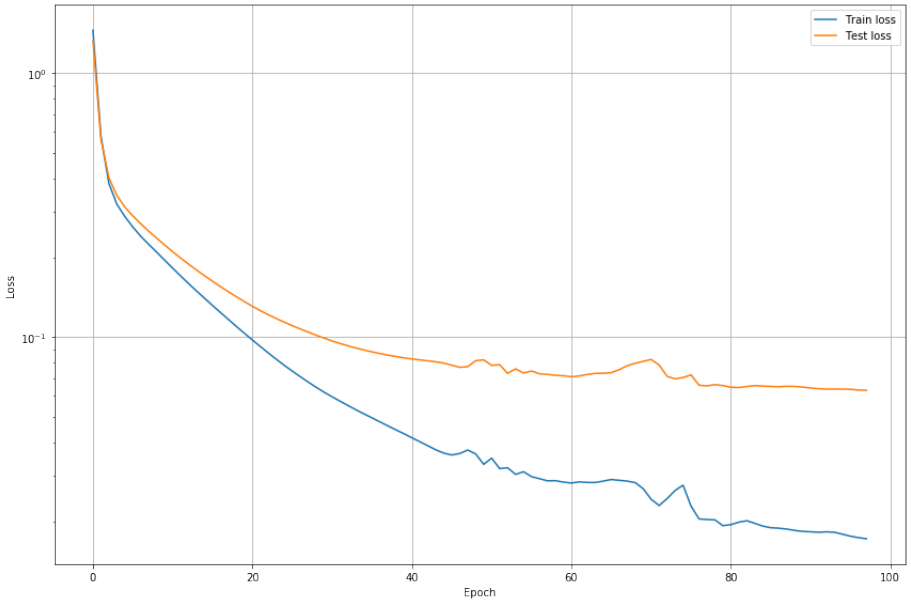

Figure 3: Trends of train and test error for the best trained model

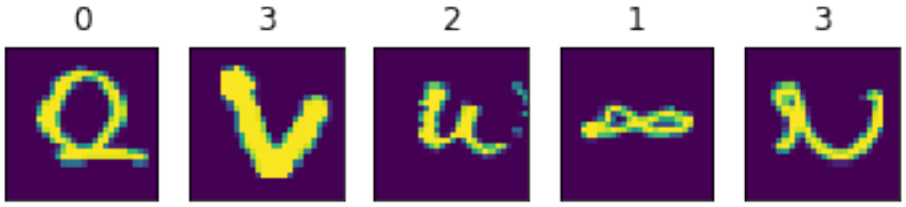

Figure 4: Some elements of the dataset with their wrong associated labels

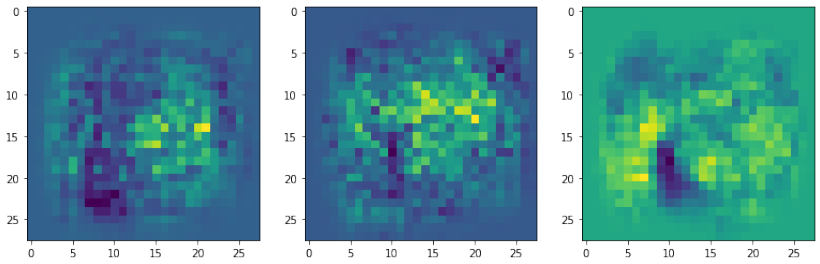

Figure 5: Representations of first hidden layer neurons patterns

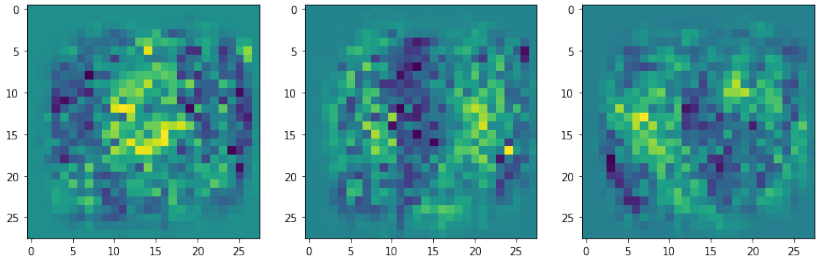

Figure 6: Representations of second hidden layer neurons patterns

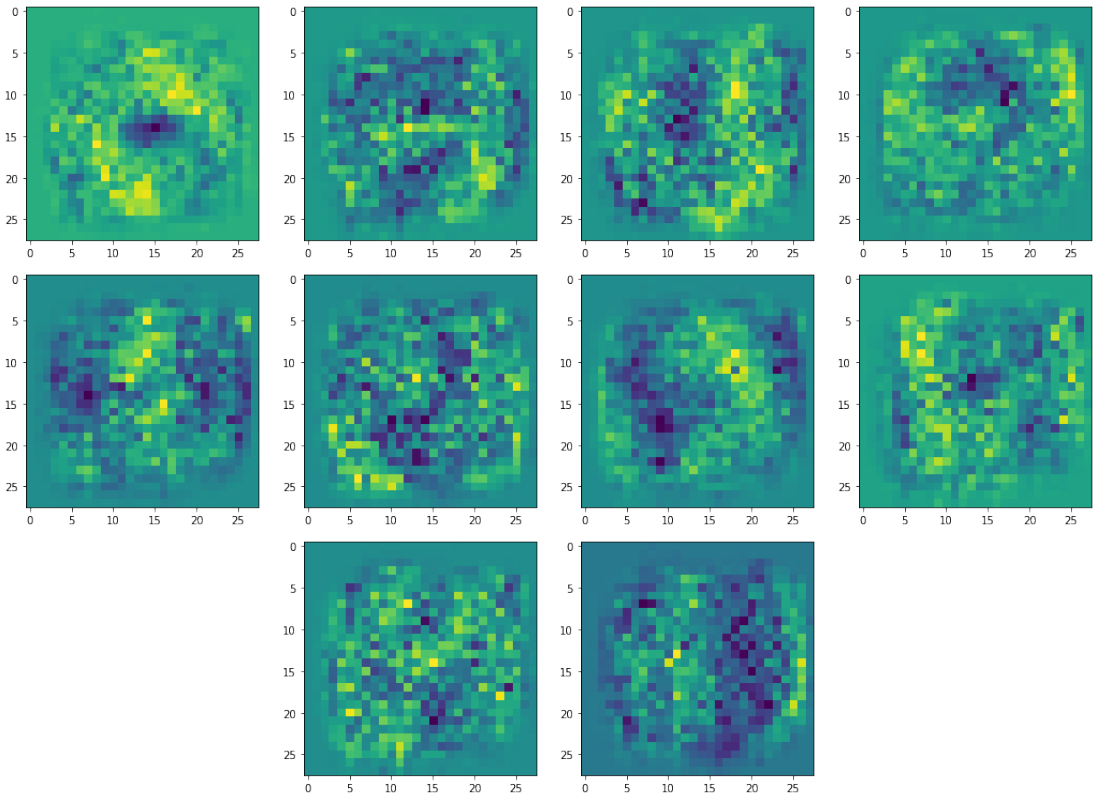

Figure 7: Representations of ten output neurons patterns