Report of homework 2: digit classification with PyTorch

Clara Eminente

January 19, 2021

1. Introduction

The aim of this assignment is to expand the code made available during the third laboratory session in order to successfully train a 2 hidden layers feed forward neural network over *60000* samples from the $MNIST$ dataset. The dataset consists of images of handwritten digits, each one along with its correct label.
The available code mainly consisted in a class, *Net*, which provided an initialization function and a **forward** function to evaluate the output of the network given an input. The network and the main ingredients of its training (e.g. backpropagation for weight update) are implemented using the *PyTorch* library.
A custom library that includes and expands the main class *Net* and some utility functions was implemented ($NN\_H2.py$). The main code (loading the dataset, training, random search of hyperparameters ecc) was implemented in the main file *"Eminente_H2.py"*. The file also includes some code developed to show the activations of some of the neurons and also investigate which type of image maximally activates them.

2. Implementation

The first task to implement was loading the *MNIST* dataset, which was done exploiting the *loadmat* function from *scipy*. We underline that the images come as a flattened vector of $28x28 = 784$ entries: after reshaping, in order to get the actual image of the digit, they also must be transposed. Since the images do not need to have the correct orientation to ensure the success of the training, **they have been transposed only when plotted**.
Initially the dataset is plit in training and test sets (80% and 20%): the latter will be employed only at the end of the training procedure.

2.1. Training

The main additions to the already provided code are early stopping and cross validation. They were both implemented from scratch without relying on libraries that interface *PyTorch* and *skleran* (e.g. *skorch*): this choice was made after having some problems when interfacing early stopping (performed via *skorch*) and cross validation (performed by *sklearn*). Their implementation is identical to the one used in the previous homework.
The basic training procedure starts by dividing the training set's indeces in lists ($k - folds$), which in turn will be used to select the portion of dataset that is used for validation. After feeding the images in batches to the network the loss is computed comparing the output of the network and the true label (ie. the digit) of the image. The loss is then backpropagated to update the weights. Training is early stopped after a certain number of epochs (*patience*) where the validation error has not decreased. The best configuration of the network is saved each time the validation error decreases so that it is possible to retrieve it in case of early stopping. The final validation error is computed as the average of the validation errors over the $k - folds$.
The loss function of the model is **cross entropy**.
Note that an additional output value can be returned by the $forward$ function. In general the network needs the values of the last 10 neurons to compute the loss. By setting $additional\_out = True$ the $argmax$ over the 10 neurons is computed, which corresponds to the most likely label of the image.
Mind, moreover, that the *PyTorch* implementation of the cross entropy loss automatically applies a softmax function to the input before computing the loss, so that there is no need to apply it to the last layer manually. It is important to recall that the output of the last layer has to be interpreted as the probability of the label of the image, so that the maximally activated neuron corresponds to the most likely label. In order to interpret the outputs as probabilities they must be bounded between 0 and 1 and normalized, hence the need of the

softmax.
We finish by noting that a feature that is believed to have sped up learning is the possibility to train the network using batches of data (batchsize will be one of the hyperparameters of the random search).

2.2. Random search of hyperparameters

Early stopping and cross validation were fundamental in order to successfully and efficiently be able to random search the best hyperparameters of the network. Table 1 shows the parameters that were kept fixed during the search. Mind that the number of input neurons and output neurons (*Ni* and *No* respectively) are fixed by the size of the images and the number of classification labels (784 and 10).

activation function	LeakyReLU
optimizer	Adam
patience	15
number of folds	3
number of epochs	150

Table 1: *Fixed parameters*

Table 2 shows the parameters that were randomly searched and their "domain".

parameter	**range**	**best**
Nh1	100-600	571
Nh2	100-Nh1	444
learning rate	$5 \cdot 10^{-4} - 5 \cdot 10^{-2}$	$9.7 \cdot 10^{-4}$
L2 weight	$5 \cdot 10^{-5} - 5 \cdot 10^{-3}$	$3.4 \cdot 10^{-4}$
batchsize	200,500,1000, 1500,2000	500
dropout fraction	0-0.5	0.32

Table 2: *Searched parameters*

All parameters were sampled with uniform probability in the range, except for the learning rate and the L2 weight, whose values were logaritmically spaced. Note that it appeared logical for *Nh2* (number of neurons in the second hidden layer) to be smaller than *Nh1* (number of neurons in the first hidden layer) as the input consists of 784 neurons and the output of only 10.
In table 2 is also reported the best set of hyperparametrs. It was obtained as the set with the smallest cross validation loss.
All the sets of hyperparameters where written on a *.csv* file *Models.csv* along with their cross validation loss.

3. Results

The final configuration of the network was obtained re-initializing the network with the best set of hyperparameters found and re-training it. The training set was again split in 80% training and 20% validation (mainly to be able to early stop the training). The test set was eventually used to check performances on unseen data.
The accuracy obtained (computed as the percentage of correctly labeled images) is **97.7%** on the test set (corresponding to a cross entropy loss of 0.074). The accuracy on the whole MNIST dataset is **99.1%** (cross entropy loss: 0.032).
At the end of the report it is possible to observe some images fed to the network and compare them with the output of the last layer (fig. 3).

3.1. Exploring neurons activation

The last task consisted in observing how the neurons behave for different values of the inputs. This was done both plotting their receptive fields and retrieving the input image that maximally activates a neuron through gradient ascent.

3.1.1. Receptive fields

For each layer the receptive field of neuron j is defined as follows:

$$RF(l = 1, n = j) = {W^{in,1}}_j$$

$$RF(l = 2, n = j) = W^{1,2}{W^{in,1}}_j$$

$$RF(l = out, n = j) = W^{2,out}W^{1,2}{W^{in,1}}_j$$

where W^{kh} is the matrix of the weights connecting layer k to layer h (k, h in $in, 1, 2, out$).
In fig. 1 the receptive fields of 9 of the 10 output neurons are shown. Fig. 4 in Appendix also shows the receptive fields of 4 random neurons of the first hidden layer.

Darker color corresponds to higher activation of the neuron.
We notice that it is not really possible to retrieve any clear information on the input image, even in the final layer. This is not, however, too surprising, expacially for the inner layer since a fully connected feed forward neural network does not really retain any clear spatial information on the input image (unlike a CNN, for example).

3.1.2. Gradient ascent

Using gradient ascent it is possible to generate an image that maximally activates a certain neuron.

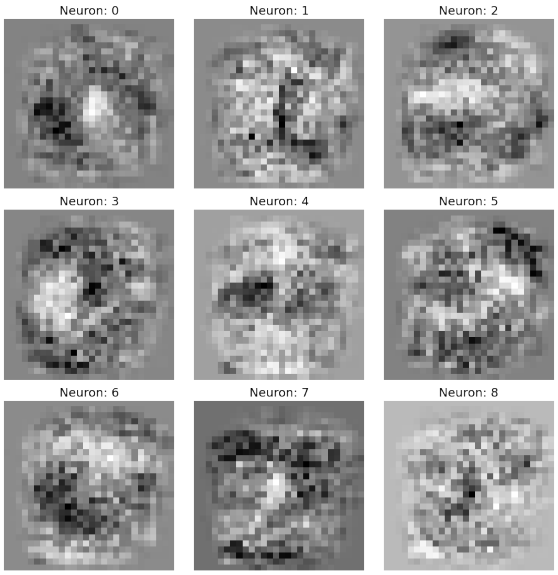

Figure 1: *Receptive fields of 9 out of 10 output neurons*

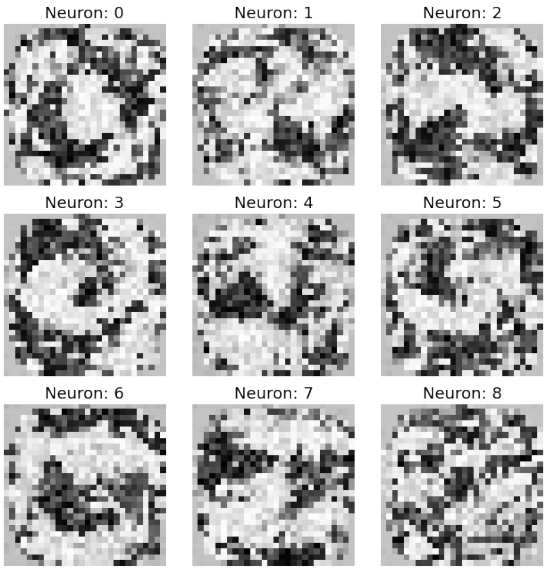

Figure 2: *Gradient ascent over output neurons activations*

In order to do so the following steps were implemented.
The function *ascent_img* starts from a random image (i.e. a 784-entries long *torch.Tensor*) and, using an optimizer (*Adam*, in this case), computes the activation of the neuron *neuron* in the layer *layer*. This value is then backpropagated in order to update the input image so that, at the end of *epochs* iterations, the resulting image approximates the one that maximally activates the selected neuron in the chosen layer. The function $from\_img\_to\_activation$ computes the activation of a selected layer given an input image.
The input images obtained backpropagating the activations of the neurons of the last layer are shown in figure 2: again, it is not possible to precisely identify the digits corresponding to each neuron, except maybe for the one corresponding to 0. However we can notice that this method provides way more definite input images compared to the one obtained computing the receptive fields.

Appendices

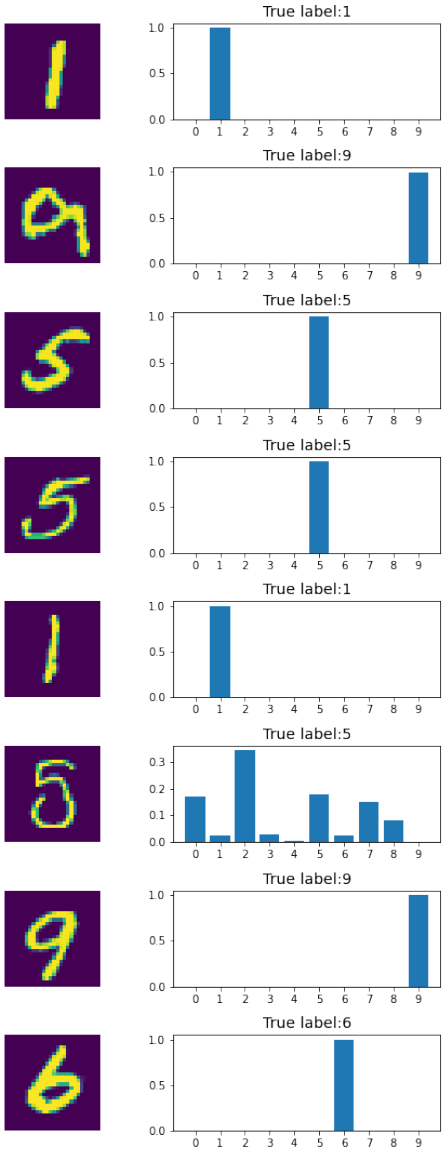

Figure 3: *Image in output vs softmax of the output layer*

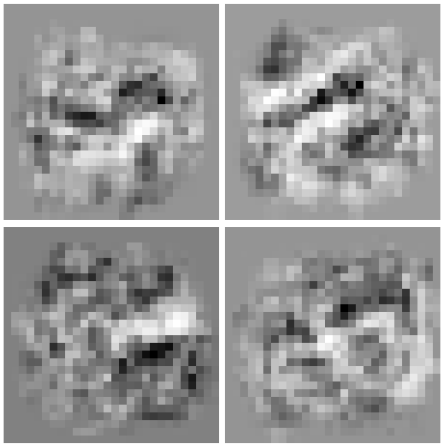

Figure 4: *Receptive fields of four neurons of the first hidden layer*