Lab 3 - Handwritten Digit Classification

Vidaich Francesco

1 Introduction

The goal of this assignment is to build a neural network to solve a multi-classification problem over the MNIST dataset, which is composed by 60000 handwritten digits, stored in grayscale images of 28x28 pixels, and by their corresponding labels (examples in Figure 1). For this exercise we will use Pytorch, a very powerful framework for deep learning, to develop and train our networks. In order to fine the best value of the hyper-parameters, a Random Search method was implemented. This procedure was built on the `sklearn` Cross Validation method, so it was necessary to introduce an interface to use PyTorch networks with `sklearn` methods: this was done by the `skorch` library. We will also discuss some methods for visualizing features encoded in the network.

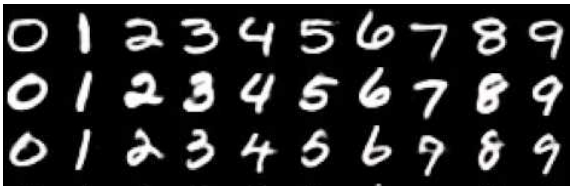

Figure 1: *Samples from MNIST dataset*

2 Neural Network class

To define the structure of a network in PyTorch, one has only to define its class, which needs at least two methods: `def __init__()` and `def forward()`. Every developed network takes in input a 1dim tensor of length 784 (i.e. the flattened 28x28 image) and returns 10 values corresponding to the predicted probabilities of each possible label. They were always composed by two hidden layers of variable size and the activation function applied on their outputs was always the `nn.LeakyReLU`. The output layer should contain also a `nn.Softmax` activation function, but it does not since the used loss function is `nn.CrossEntropyLoss()`, which already does the `log_softmax` of the network output. After each hidden layers we also added a dropout layer to improve regularization (its probability is passed as an argument).

3 PyTorch Training procedure

In this section we will see how a generic network is trained in the PyTorch framework.

After splitting the dataset in Train and Test sets (which contain respectively 85% and 15% of the original one), the first thing to do is initializing the network by creating an instance of the described class. Before the training it is also necessary to define the optimizer, an object containing the algorithm that updates directly the NN weights using the values of the gradients stored in their corresponding member of each weight (in most cases the adopted one was `optim.Adam`).
The training procedure is done by few lines of code:

```
net.train()
optimizer.zero_grad()
net_output = net(X_train)
loss = loss_fn(net_output,y_train)
loss.backward()
optimizer.step()
```

After each iteration the network is evaluated considering the loss value that obtains with the test set. This step has to be done after setting the network evaluation mode with `net.eval()` in order to disable dropout layers.
After completing the training procedure, the final performance is measured with the network accuracy over the Test set.

4 Random Search

4.1 The skorch library

To implement the sklearn Random Search to search the best values for the parameters of torch networks, we used the skorch library. Its method NeuralNet() can be seen like a wrapper around the torch network that has a sklearn interface.

4.2 Parameters' distributions

Before implementing the Random search, the distributions over each parameter has to be defined. Since wanted to build custom distributions over some parameters, there have been some problems passing them as arguments to the RandomizedSearchCV() method (that normally samples the parameters' values from them autonomously), so we opted to build the random search procedure in pure python that calls during each cycle the cross_val_score() method from sklearn to execute the cross validation. In this case, the combinations of parameter are generated by the method gen_values(n) of a class called Param_generator(). The method takes as argument the number of combinations to generate, sampling the values in the following way:

```
Nh1_vals = randint(Nh_min, Nh_max, n_combins)
Nh2_vals = randint(Nh_min, Nh1_vals)
optims = choice(optim_funcs, n_combins, p_optims)
drop_vals = uniform(drop_min, drop_max, n_combins)
lr_vals = 10**(uniform(lr_min, lr_max, n_combins))
L2_vals = 10**(uniform(L2_min, L2_max, n_combins))
```

Let's do some considerations: the networks take in input 784 values and its output is made of only 10 values, since this means that it has to be able to encode the input image, it makes sense to choose a decreasing number of neurons at each consecutive layer. We also wanted to test different optimization algorithms and using p_optims we can set the probability of each of them. For some parameters (like lr and L2_decay), we wanted to have a uniform distribution between the different orders of magnitude.
All the min/max values are defined during the initialization of a class instance: In our case the chosen values were Nh_lims=[50,600], drop_lims=[0,0.5], lr_lims = [-4,-1], L2_lims=[-6,-2], optims=['optim.Adam', 'optim.RMSprop'], p_optims=[0.8,0.2].

4.3 Random Search Procedure

After generating 150 combinations of parameters, we want to test them to see which are the best ones. So, for each one, we generate a skorch.NeuralNet object by passing to it the PyTorch NN class, the optimizer method, the arguments needed to initialize them, the loss function to minimize during the training (NN.CrossEntropyLoss) and the max number of epochs. Moreover, it is also possible to pass some optional arguments in the form of callbacks. The only one used in this work was EarlyStopping(patience=15), which stops the training if no improvement is observed during the last 15 epochs.
With this skorch object, we can use the cross_val_score method from sklearn to evaluate each network on the training portion of the MNIST dataset. Since the machine that performed the training provides 4 cores, the cross validation was done by splitting the dataset in 4 folds. By setting the flag n_jobs=-1 of the method, each available core was used to perform independently each of the 4 trainings, reducing the computation time of the entire cross validation to just one simple training.
After evaluating each of the 150 networks, the obtained scores (final values of the test loss function) and their corresponding set of parameters were sorted to build a ranking of the best tested configurations and then stored in the files *sorted_scores.dat* and *sorted_params.dat*.

4.4 Final Training

At the end of the Random Search, we considered the 15 network configurations that placed on top of the ranking and re-trained them over the whole training set (proceeding in the way described in Section 3) for 300 epochs. Completed the training of each network, we test them on the test set (that was never used during the Random Search and on this training, so it is completely new) by measuring their accuracy and comparing it to the best one measured until that moment. If the new accuracy is higher, save it as the new target accuracy and the trained network becomes the winning one. When all the 15 networks have been trained and tested, store the final winning network in *winning_net.dat*.
The winning network in our case had Nh1=532 and Nh2=195; it was trained using a dropout probability of 0.477, theoptim.Adam optimizer with lr=9.51e-06 and L2_decay=1.53e-06. It achieved an accuracy of 97.93% on the test set and 99.34% on the whole MNIST dataset.

5 Visualization of Neurons

5.1 Receptive Fields

The receptive field of a neuron is the visualization of the effect that the input image has on the activation value of the neuron. Since we are working with feedforward NNs, each pixel of the image can have an influence on any neuron of the network. The receptive field of each neuron in the first hidden layer is simply given by the weights that connect it to the input layer. For the following layers, the receptive fields are obtained computing the matrix multiplication between all previous dense layers and the weights that connect the considered neuron to the layer preceding it. Here we report the 25 receptive fields of the hidden layers and 9 of the output layer (neurons corresponding to 0-8 labels) of the winning network.

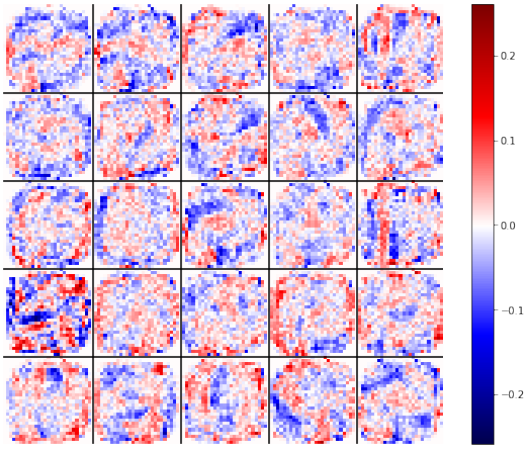

Figure 2: *Receptive fields of the first 25 neurons of the first hidden layer*

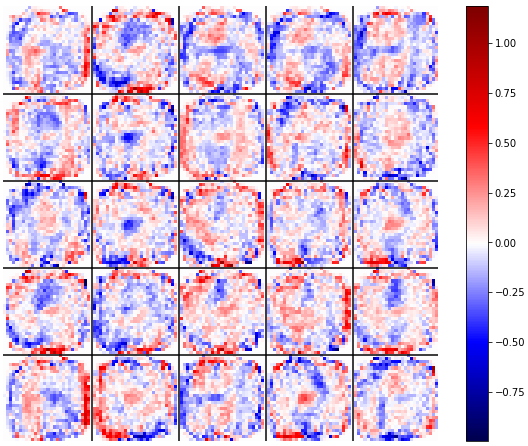

Figure 3: *Receptive fields of the first 25 neurons of the second hidden layer*

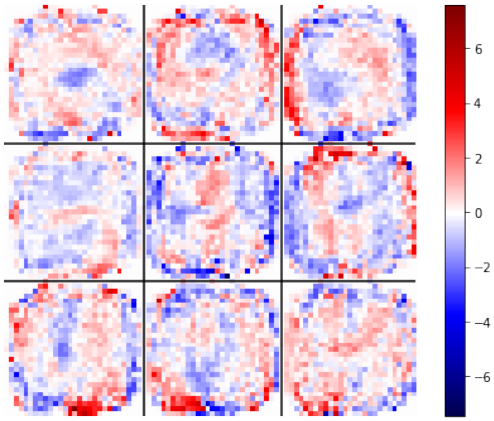

Figure 4: *Receptive fields of the first 9 neurons of the output layer*

The receptive fields of the output layer are particularly interesting, for example in the first cell (corresponding to the label '0') it is possible to see the expected the red circle with the blue center.

5.2 Gradient Ascent over neuron's activation

Now we will perform a gradient ascent procedure to find which input images will maximize the output of the first 25 neurons of the hidden layers and the first 9 of the output layer. This was done by generating the input image with a `torch.Tensor`, computing the activation value of a neuron of the winning network and then calling its `backprop()` method. This computes the derivatives of the output neuron wrt each input pixel. We perform the update of the image using the gradient ascent: $X = X + \lambda \Delta X$ (where X is the 28x28 input image). By repeating this for many iterations, the obtained image becomes almost equal to the the receptive field of the considered neuron, which makes sense since inputs and weights should have the same sign in order to maximize the neuron's output. Since the values of the input image has to be contained to the interval [0,1], the real maximum of the neuron activation is obtained setting all its negative values to 0 and its positive values to 1. In this way, the images that maximize the neurons' output are presented in Figure 5, 6 and 7.

Using feedforward networks those images do not add any relevant information compared to the receptive fields, but using a CNN or a network not composed of dense layers (i.e. where the neurons do not depend on all the neurons of the previous layers) these could provide some interesting insights into the network.

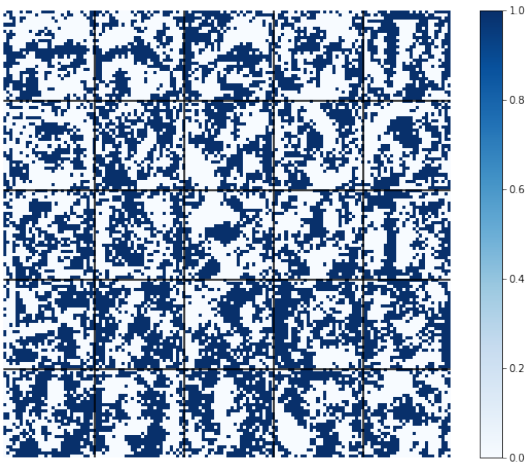

Figure 5: *Input images that maximize the activation of first 25 neurons of the first hidden layer*

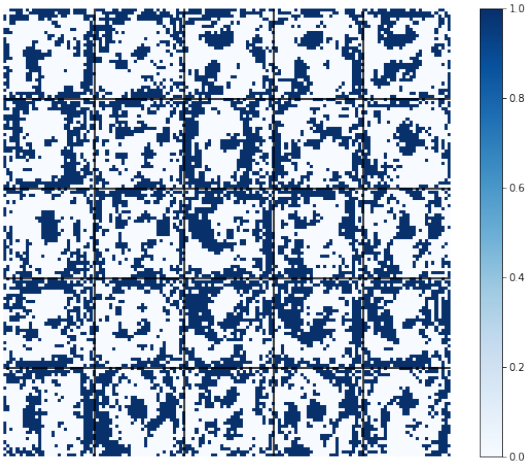

Figure 6: *Input images that maximize the activation of first 25 neurons of the second hidden layer*

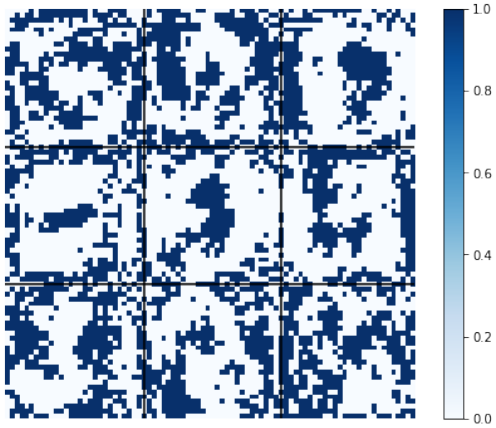

Figure 7: *Input images that maximize the activation of first 9 neurons of the output layer*