Homework 2: feed-forward networks using Pytorch

Abstract

In this homework, the focus is on the creation of a feed-forward network using Pytorch and related tools. The dataset is a variation of MNIST; the network will be trained and tested on it, and feature visualization will be performed. A check on the results is also performed.

Code development

The first step in the assignment is the loading of the dataset `MNIST.mat`. This is a MATLAB file, which requires a specific loading procedure. In my case, the choice fell on `scipy.io` utility `loadmat`. The resulting dictionary is then used to extract all the meaningful data, that is, input images and output labels. Pytorch offers a great deal of tools for data handling: among them, `Dataset`s and `Dataloader`s are the most common. Despite their functionalities, after some trials I decided to resort to the skorch library, which interfaces Pytorch with scikitlearn. This is particularly convenient when having to perform cross validation and random search, as they are not natively implemented in Pytorch, while scikitlearn has fully-fledged interfaces which also allow parallel processing. Once the approach was chosen, I split the dataset into a train and a test set: the latter, in particular, was set to be 1/10 of the whole dataset. As the dataset was balanced (i.e., all classes were composed of roughly the same number of elements), a simple random shuffle was used to perform the split.

A feed forward network was developed by overloading Pytorch's `Module` class; in addition to three linear layers, dropouts and batch normalizations were also added to reduce overfitting.

The network's first and last layers are constrained by the input and output dimensions; the hidden layers sizes and the dropout factor were chosen using scikitlearn's `RandomizedSearchCV` API, by interfacing it to the Pytorch model through skorch wrappers. This also allowed the deployment of an early stopping mechanism.

The parameters over which to perform the random search are set as distributions (from the `scipy.stats` module); in this case, all distributions were uniform. Of note is the choice for the hidden layers: if h_i is the size of hidden layer i, I set $h_1 \sim \mathcal{U}(a, b)$ and $h_2 \sim \mathcal{U}\left(\frac{a}{2}, \frac{b}{2}\right)$, which resulted on hidden layer 1 having, on average, double the neurons of layer 2. From my experience, this structure seem to achieve good results, as it progressively compresses information.

The random search has a built in method for performing k-fold cross validation; as a consequence, the model returned by the skorch utilities is already cross-validated, and ready to be trained on the whole train dataset; the values of patience in the early stopper and the maximum number of epochs, however, were increased as to allow a longer and possibly better training. The Pytorch model itself is then simply retrieved from the skorch wrapper, saved and loaded.

The score is calculated by computing the `argmax` of the size-10 softmaxed tensor that the network outputs, comparing it with the true labels and averaging the errors.

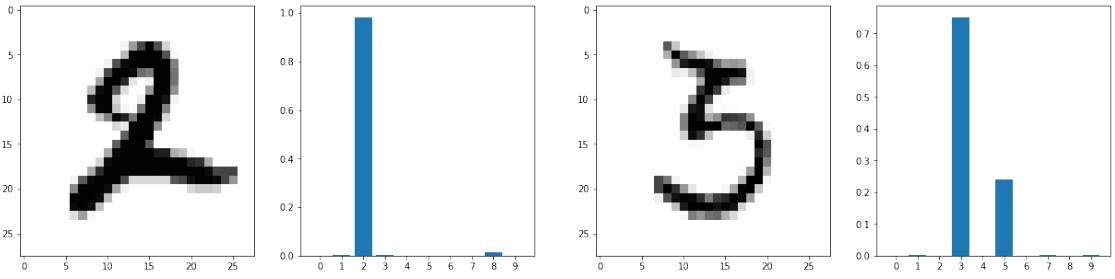
Once training was over, the receptive fields of neurons in layer i were computed, by 'propagating' the weights from the first layer to the i-th.

Results

The chosen parameters are:

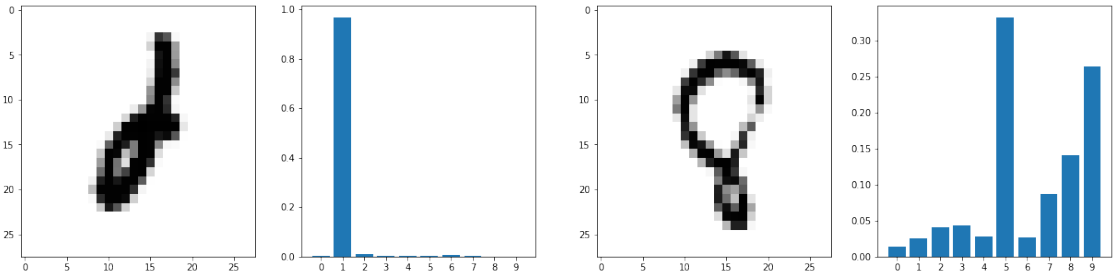
- Layers: [784,360,187,10]
- Dropout factor: 0.5782

The loss function was the `NLLLoss` (coupled with the softmaxed output to provide probabilities), and Adam was used as an optimizer. The number of epochs was set to 200, with a patience window of 10. The network achieved a mean classification accuracy of 0.9634 over the test set that was extracted from the MNIST file. In 2, examples of correct and wrong classifications are shown, together with the related probabilities. While most predictions are correct, with probabilities being largely skewed towards the correct answer, there are cases in which the prediction is not as clear-cut, but still correct. There are however also clear errors, where the network completely misses. Finally, the most interesting case is that of a significantly spread probability landscape: while in the previous examples our intuition may justify the errors and uncertainties (subfigure c hardly seems a 2), in this case the true answer seems clear enough. However, the network is not able to clearly identify the number.



(a) True label: 2

(b) True label: 3

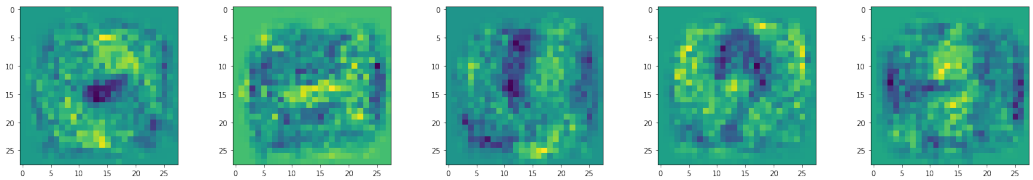


(c) True label: 2

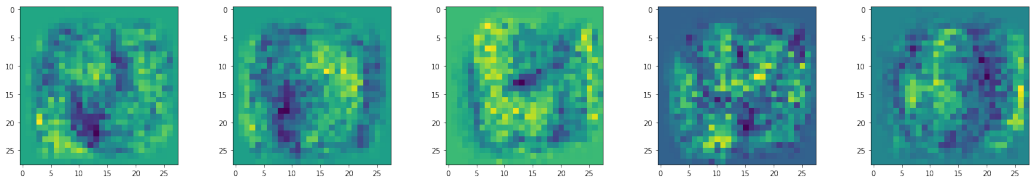
(d) True label: 8

Figure 1: Examples of the probabilities returned by the network, compared with the true results.

A possible justification may come by visualizing the receptive fields of the neurons in the last layer[1], which are responsible for the end probabilities.



(a) Neuron 0 (b) Neuron 1 (c) Neuron 2 (d) Neuron 3 (e) Neuron 4



(f) Neuron 5 (g) Neuron 6 (h) Neuron 7 (i) Neuron 8 (j) Neuron 9

Figure 2: Receptive fields for the neurons in the last layer.

Despite a lack of a clear visual correlation between the neuron's receptive field and the corresponding number, the fields have a set of common characteristics. Only a central, somehow rounded area is effectively changed during the training: this is consistent with the fact that the numbers are written in the center, and the corners are left untouched. In addition to that, most of the receptive fields show large, contiguous sections of high/low activation. Subfigure d, in contrast, shows a thin and light number, which is not consistent with any of the receptive fields. This may explain why it activates the last layer so much.

Comments

The model, given its simplicity, shows a rather impressive performance: more than 95% of the images are correctly classified, while overfitting is avoided by the conjunction of a train-validation split (done by skorch) and dropout-batch normalization layers. One possible improvement is an antagonist-based training, where modified images are passed as input, in order to boost the network's generalization properties.

A separate ending comment regards the failure of the activation maximization procedure; despite many trials (and even more failures), I did not manage to make it work.

Addendum: activation maximization

I tried the activation maximization technique in order to visualize the images which correspond to a spike in the activation of a particular neuron. To do so, I tried using Pytorch's `autograd` functionalities. Given a tensor and a scalar function which depends on it, a gradient can be computed automatically by `autograd`. In this case, the function to maximize is the activation of a neuron, and the tensor is the input image. By using Pytorch's *hooks*, one can access the layer outputs of a neural network: each time

[1] While previous layers' receptive fields were produced, they were not as interesting.

the `forward` method is invoked, hooks containing the values of the outputs can be registered. By selecting one of the values from these hooks, one can access the activation of a particular neuron; then, being a scalar function, it can be easily treated as a loss, and backpropagation can be automatically performed. An optimizer, set to work on the input image, can then perform gradient descent on each pixel, resulting in the wanted output. If a whole layer was to be examined, computing the average of the activations of the layer would produce a scalar function.

However, this approach did not work in my case; despite requiring the computation of the gradient for the input image, `autograd` did not seem to treat it as a *leaf variable*, and so the computed gradient was simply `None`.