# Predicting MNIST characters with Feed-Forward Neural Network

Nicola Dainese

## I. INTRODUCTION

In this work I present my implementation of a Neural Network (NN) in Pytorch and a framework to tune the hyper-parameters (HPs) of the network with a random search based on some prior probability distributions. I then show how to use two consecutive random searches to find the best model and discuss its performance. Finally I present some techniques that can help in visualizing what each neuron is learning.

## II. NEURAL NETWORK IMPLEMENTATION

The architecture of the network is the one of a standard feed-forward network, the only notable upgrades are:

1. There is the possibility of setting a custom number of hidden layers and neurons. This is done by passing a list called $h\_size$ which length is the number of hidden layers plus the input layer and the element $h\_size[i]$ contains the number of neurons in the $i-th$ layer. The code is the following:

```
self.hidden = nn.ModuleList()
for k in range(len(h_sizes)-1):
    self.hidden.append(nn.Linear(\
        h_sizes[k], h_sizes[k+1]))
    self.hidden.append(nn.Dropout(dropout))
```

2. There is the possibility of using the dropout technique, that is implemented as shown in the code above.

3. The output does not have an activation function because this is already done by the CrossEntropyLoss. An identical alternative would have been to use the softmax activation in the last layer and the nll_loss as a loss function.

Also I tried different optimizers, namely: Adam, AdamW, Adamax and Adagrad.

### A. Training and evaluation procedure

The workflow for the training is the following:

1. Receive training and validation (or test) sets as numpy arrays.

2. Convert them into tensors, then create a TensorDataset, a SubsetRandomSampler and finally a DataLoader. The sampler is used to sampling with permutations batches from the dataset without using two times the same sample in an epoch. The data loader instead uses a parameter to select the size of the batches and it is responsible for the parallelization of the training using $num\_workers$ CPU cores.

3. During each epoch of training I feed all the batches of the training set to the network using the forward method of the Net class, then compute the loss, call its back-propagation to compute the gradient for all the parameters of the network and finally call the step method of the optimizer to update the parameters according to the gradient.

4. After one epoch of training I disable the dropout and compute both the mean loss and the accuracy on the validation set, just for monitoring purposes.

5. I repeat this procedure for a certain number of epochs, specified by the internal parameter $n\_epochs$ of the model and I return some logging variables and the trained model according to some Boolean parameters that can be passed as inputs to the training function.

## III. RANDOM SEARCH WITH PRIOR DISTRIBUTIONS

In this experience I moved from a grid search to a more sophisticated approach that I call random search with prior distributions.

The most critical aspect of a grid search is that the number of combinations scales exponentially with the parameters that we want to test. Given the fact that we all have limited computational power, this forces us to restrict the possible values of each variable and ultimately making some arbitrary choices about those possible values. These choices are strongly biased by our hypothesis on the problem. The critical aspect here is that those hypothesis remain completely untested and the result will typically present the survivor bias (e.g. we increase the importance of the factors that we control and forget about what could be more important factors that we kept fixed during the grid search).

A simple solution could be to define a greater ensemble of all possible combinations of parameters that we want to try and then select only $n$ of those combinations. This would mean to test less systematically the dependence of the performance on a given parameter, but trying out more parameter values with the same computational effort. The question here is: how to define this greater ensemble so that the combinations sampled represent as well as possible what we would like to sample? My idea is to sample each parameter from a distribution defined a priori (a.k.a. prior distribution) that encodes the bias (in the sense of hypothetical knowledge) that we have towards the values that are more promising and less promising.

For example making some trials I noticed that few hidden layers work at least as well as many hidden layers, hence, since we like simplicity, we would like to use most of the times few layers, but sometimes to try also deeper architectures. In this perspective it is much less biased to choose a decreasing exponential distribution (on integers) for the number of hidden layers than to choose once and for all to keep the number of hidden layers fixed to 2.

### A. Prior distribution for architecture depth and number of neurons

In this subsection I discuss more in depth the most elaborated prior distribution that I defined, that is used to define the number of hidden layers and their number of neurons.

Basically the idea is to define a first function that samples the number of hidden layers so that, assuming a constant compression rate C (e.g. the ratio between neurons in one layer and the previous one), given the input dimension of the network it holds the right output dimension for the last layer. Then we would like have a second function that, given the actual number of hidden layers, it samples the number of neurons of each hidden layer so that on average we have the right compression rate.

Going a bit deeper on the argument we have that, since the number of neurons depends on the depth of the layer and on the compression rate, if we fix C, we can obtain what on average would be the ideal depth of architecture for that compression rate.

If we have a constant compression rate, it holds:

$$N(n) = N_0 \cdot C^{-n}$$

where

- $N(n)$ is the number of neurons in layer $n$;
- $n = 0$ for input layer;
- $n = n_h$ for the number of hidden layers;
- $n = n_h + 1$ for the output layer.

We can compute the number of hidden layers required for a given C to compress the input layer in the output layer imposing $N(n_h + 1) = N_{OUT}$ and $N(0) = N_{IN}$ and using

$$N(n_h + 1) = N(0) \cdot C^{-(n_h+1)}$$

We find

$$\overline{n_h} = \frac{1}{logC} log\left(\frac{N_{IN}}{N_{OUT}}\right) - 1$$

that is the average number of hidden layers that we *must* get by sampling from a generic distribution if we want to keep the compression rate *constant*.

In my specific case I choose to sample from an exponential distribution $Ae^{-\alpha n}$ defined for $n > N_{min}$, e.g. $N_{min} = 2$ if we require at least 2 hidden layers.

To find its exponent we must impose

$$\sum_{n=1}^{\infty} Ane^{-\alpha n} = \overline{n_h} \longrightarrow \frac{Ae^{\alpha}}{(e^{\alpha} - 1)^2} = \overline{n_h}$$

with the normalization factor computed as

$$\sum_{n=1}^{\infty} Ae^{-\alpha n} = 1 \longrightarrow A = e^{\alpha} - 1$$

Finally we get $\alpha = log\left(\frac{\overline{n_h}}{\overline{n_h} - 1}\right)$.

Concretely what I do for sampling is:

1. Fix $C$ (e.g. 3);
2. Compute $\alpha$;
3. Sample $n_{layers}$ from $Ae^{-\alpha n}$;
4. Recompute the effective $C$ inverting the formula for $\overline{n_h}$, this time using $n_{layers}$;
5. Sample for each hidden layer the number of neurons with a binomial distribution with number of trials equal to the number of neurons on the previous layer and with success rate $1/C$.

## IV. MODEL SELECTION AND TRAINING

To select the best model I used two consecutive random searches, tuning the parameters of the second search by hand after having analyzed the results of the first one. Each random search used 54000 out of 60000 samples of the dataset and used a 3 fold cross validation.

In figure FIG. 2 (appendix) it is possible to see the score of the different models as a function of the learning rate, optimizer and number of hidden layers. This led me to choose Adamax as optimizer, since it was the most stable across all those HPs, then to increase a bit the compression rate C in order to limit even more the times when deeper architectures were selected and finally to restrict the learning rate range between $2 \cdot 10^{-3}$ and $2 \cdot 10^{-2}$.

The second random search was treated as definitive, hence I choose as best model according to a sort of worst-case accuracy, using as metric the accuracy minus two times its standard deviation in the 3 folds of the cross validation. For the sake of completeness I report in the table below all the HPs of my best model.

Finally I re-trained the final model on the whole 54000 samples of the training and validation set, and tested its accuracy on the test set, obtaining a final accuracy of 98.23%.

| Hyper-parameters | |
|---|---|
| HP | Value |
| Number of neurons | [784, 278, 94, 24, 10] |
| Activation function | ReLU |
| Dropout | 0.088 |
| Number of epochs | 15 |
| Optimizer | Adamax |
| Learning rate | 0.004 |
| Type of penalty | L2 |
| Penalty weight | 0.0014 |

## V. PERCEPTIVE FIELD AND EXCITATORY STIMULI

In this section I report some techniques that I used to visualize what information each neuron learns to encode and react to. Each of the following plots is realized starting from the weights and biases of my best architecture after the final training. Moreover, since we know what each neuron of the final layer should encode (i.e. some sort of sensitivity to a particular digit), I focused only on visualising them.

First of all I computed the perceptive field of each of the final neurons, that is obtained simply by the matrix product of all the weight matrices

$$PF = W_{34} \cdot W_{23} \cdot W_{12} \cdot W_{10}$$

The result is shown in figure FIG. 3 (appendix) and it is not clear at all which neuron encodes which digit. This is probably due to the fact that we completely neglected both the bias and the activation function, thus the next step of my visualization was to take into account both those factors and try to show which parts of an input are responsible for the excitation or the inhibition of each final neuron.

To do that, I basically recreated the flow of the feedforward propagation without compressing the dimension of the input. First, I observed that the moment when we lose the input dimension is when we compute the summation of the inputs before applying the activation function. If we want to keep intact the input shape, for the first hidden layer we can just multiply each pixel for the respective weight and avoid to do the summation. This is a weighted view of the input, or can be thought as an activation map. Once we have the activation maps of the first hidden layer we can propagate those maps to the following layers in this way:

1. For each neuron of the first hidden layer, we check if the sum of the activation map elements plus the bias is greater than zero.

2. If that is the case, we propagate the map to all the subsequent neurons weighting it with the corresponding weights. Instead, if that is not the case, we don't propagate that map to any neuron.

3. We repeat this procedure for each pair of layers until we obtain the activation maps for the output layer.

Notice that this way of propagating the activation map depends on the input and is completely consistent with the ReLU activation function. The results that I obtain with this method are summarised in FIG. 4, where I show the activation map of each neuron (columns) for an example of each digit (rows). Even if it is not perfect, this map makes a lot more sense than the perceptive field representation, because it highlights in red the features of the input recognised as matching with the digit that the neuron is encoding, and in blue the ones that do not match. For instance we can see, as expected, that the diagonal is in general more reddish than the off-diagonal terms and also that the "false positive" activations resemble the shape of the encoded digit (for instance look at the activation of the neuron encoding the 7 to the digit 9).

Finally I tried to see if something interesting could be obtained by optimizing a random input image so to maximize the activation of a neuron. Again I focused just on the last layer's neurons and what I obtained basically was a copy of the perceptive field with a different scaling in values. I think that this technique is really interesting, but the network considered is too simple to obtain interesting results.

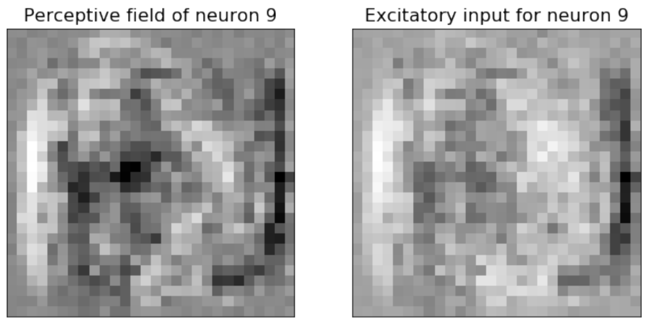

FIG. 1: Perceptive field of the neuron encoding the digit 9 and its excitatory stimulus obtained by gradient ascending on the activation value of neuron 9.

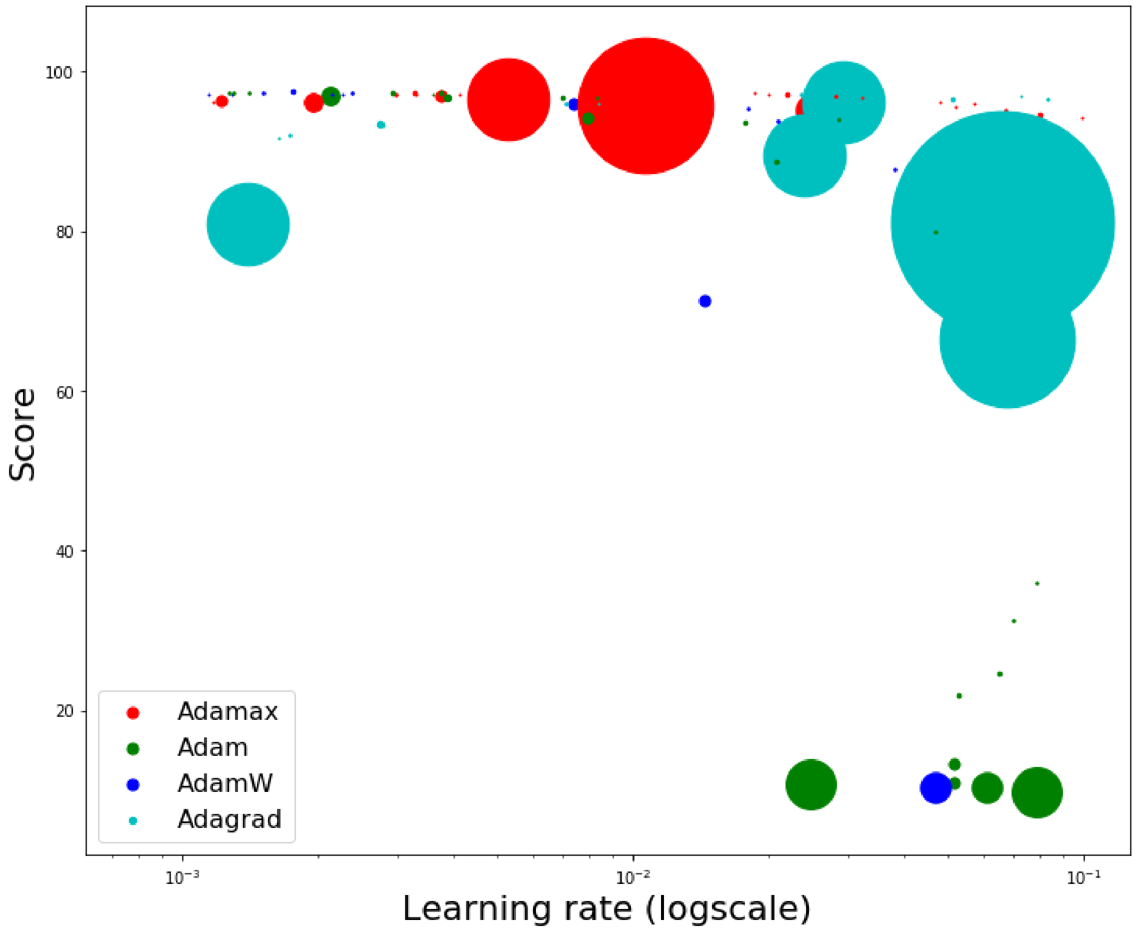

FIG. 2: Scatter plot of the results of the first grid search. On the x axis there is the learning rate in logarithmic scale and in the y axis the mean accuracy on the 3 folds of the cross validation. The color of the points represents which optimizer has been used, whereas the size of the points is exponential with the number of hidden layers (ranging from 2 to 11). We can understand that a good portion of the models with bad performances has a combination of high learning rate and high number of number of layers. Adamax confirms itself as the most stable optimizer and it seems that the number of hidden layers can be shrunken even more, considering mainly networks with 2 or 3 hidden layers. Finally I will restrict the learning rate between $10^{-3}$ and $10^{-2}$.

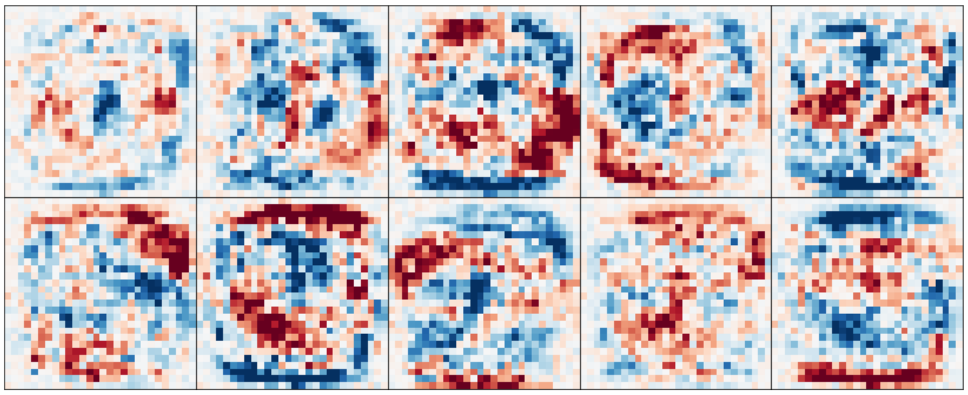

FIG. 3: Perceptive field of the 10 neurons of output, where the positive terms are in red, the negative ones in blue. No clear representation emerges.

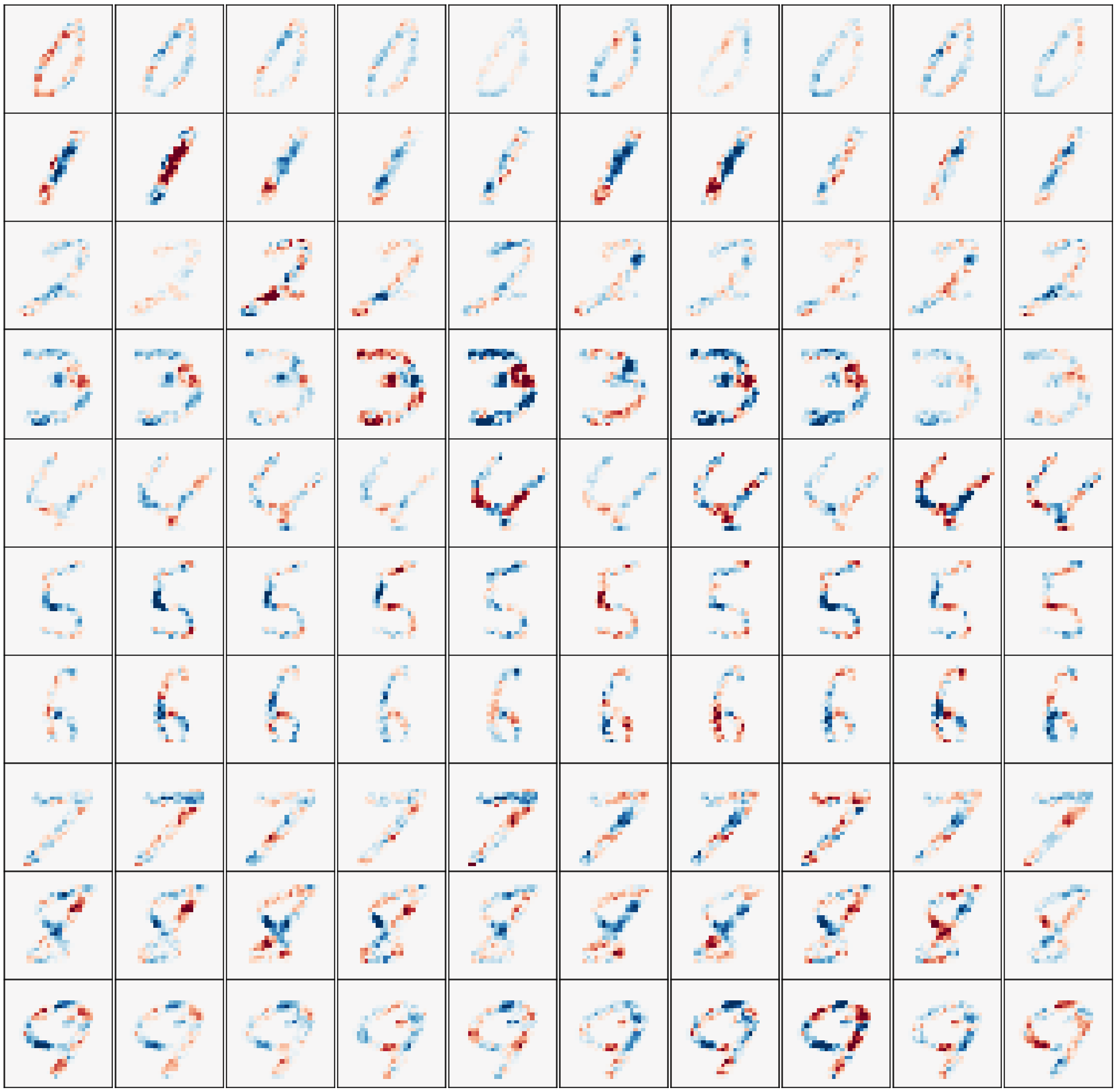

FIG. 4: Activation map of the perceptive field of the 10 output neurons for the 10 digits. Every column represents the activation map of a single neuron to different digits, each row represents a single digit.