EE-559: Deep Learning

Mini-project 1

May 2018

By Claudio Loureiro, Simon Canales and Cyril van Schreven

I- Set-up

- The data

The training data is composed of 316 samples, each having 26 channels and 50 values.
The testing data is composed of 316 samples, each having 26 channels and 50 values.

-Validation method

We set up an environment to validate and tune the neural networks and methods that we are experimenting on.

The provided dataset is already split in training data and testing data. The testing data is not considered until the final test to avoid any meta-overfitting. Validation is performed only on the training data. Since the accuracy is relatively high with this dataset, we are going to use a cross-validation. With this validation method we will also be able to measure the variance of a given neural network.
We must cope with the fact that the dataset is relatively small: 316 samples. We can't afford to diminish the training data too much for the training process, neither can we diminish the validation set too much to get meaningful results. As a compromise we chose a 5-fold cross-validation. The variance can still be high on 5 separate runs, and it is influenced by the randomness of the split. To diminish this effect the 5-fold cross-validation is ran 2 times. Thus, we perform a 2x5-fold cross-validation. This will be the case for all our validations.

-Predetermined aspects

Some aspect of our model will not be tested. Instead we rely on prior research and theoretical considerations.

Loss: The basic loss for regression methods is the mean-squarred-error. This is not appropriate for classification. In addition, it brings the unwanted effect of penalizing predictions that are 'too' correct.
We will be using the cross-entropy loss. From a probabilistic approach, it searches to maximize the likelihood of the prediction of the input being in the correct class. With the Bayes theorem, we can instead work with the probability of the class given the input and the prior probability. For easier computation, the logarithm of the likelihood is used. Since we want to work with a loss, we use the negative of the log-likelihood, which we will minimize as usual. The output of this loss function is one-hot encoding. For our binary problem we thus have two output.

Activation function: For the activation function we use the rectified linear unit (ReLu). ReLu is non-linear. This is a necessary condition for an activation function in a neural network, else the whole network could be summarized as one affine mapping. Methods such as the hyperbolic tangent are thus dismissed. With ReLu the network will ‘accept’ correctly predicted inputs and stop considering them. Wrongly predicted inputs still contribute proportionally to the error. A possible improvement is the leaky ReLu which still slightly takes into account the positively classified inputs. For this work, leaky ReLu was briefly tested and didn’t present notable improvements, it was thus dismissed.

-Code architecture

Our code has been designed to reconduct as much parameters as possible to the outer layer. As such from the main .py file, one can easily test different set ups.

II-The networks

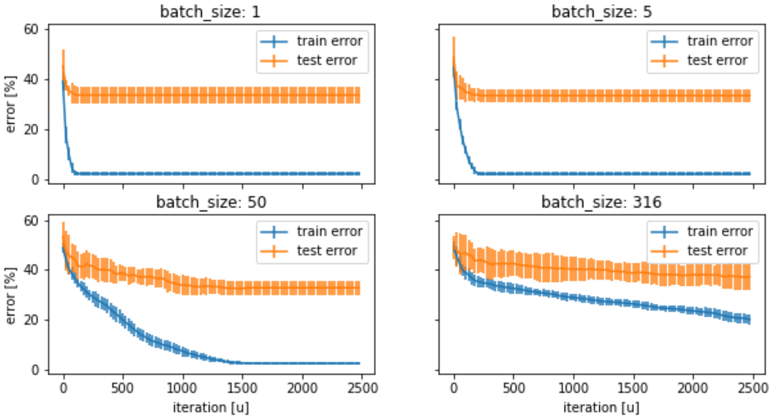
A/ First neural network

Let’s start with a rather simple neural network. This network is composed of just one hidden layer. We thus have two linear classifiers (Lin) and one ReLu in the following order: Lin->ReLu->Lin.

To begin we put 50 neurons in the layer.

-Tuning mini-batch size

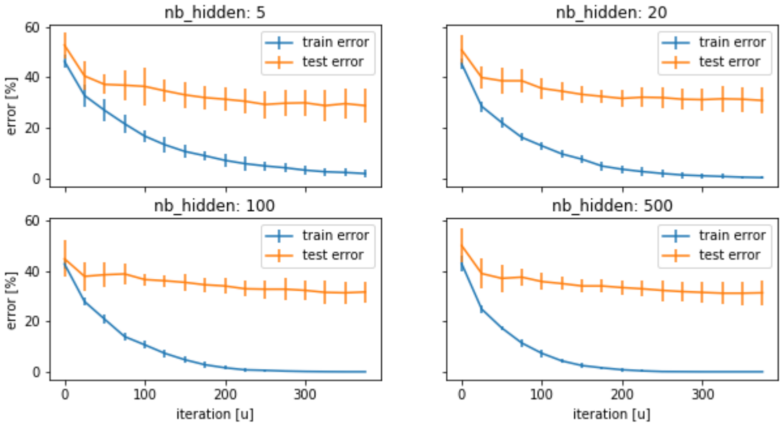
We start with a stochastic gradient descend method as optimization method. We search for an appropriate size of mini-batch. We test with batches of size 1, 5, 50 and 316. The batch size of 1 performs the real stochastic gradient descent. The batch size of 316 performs a regular gradient descent. The best solution is usually somewhere in between with a so-called mini-batch SGD. Smaller batches are valuable for: computation time on large datasets, and randomness. An approach with more randomness will be better at avoiding local minima.



Overall the end results for the batch of sizes 1, 5 and 50 are similar. We choose a batch size of 5 for the rest as it has the lowest variance. On a side note, all the experiments highly overfit. This is because we trained the network extensively. A method consists of stopping the training soon enough. We will not use this. However, we will use other approaches to avoid overfitting later on. Note that we have plotted with relation to the number of iterations. However, this is not a fair approach as the number of parameter updates is not the same. The SGD will perform 316 times more updates than the GD. This is translated in computation time. The computation cost of one SGD update is much smaller than that of a GD though.

-Tuning size of hidden layer

We now try to tune the size of the hidden layer:



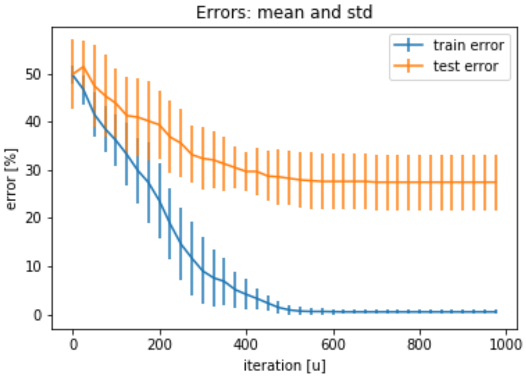
As expected, with a smaller hidden layer, there is more variance.
Though the results are close, and we cannot take serious conclusions, the best number of hidden neurons here is 5, with a mean error rate of 28.7%. Either way, the additional neurons are not necessary in this case. It seems that this dataset does not need many parameters, at least not for one layer.

Overall this network is not terrible, but it cannot yield very strong results. We thus move to another network before any further considerations.

B/ Second neural network

We will now validate a second neural network that is slightly more complex. It includes one convolutional layer, followed by a max pool, and two hidden linear layers. After each layer is a ReLu. We thus have the following operations: Conv -> Maxpool -> ReLu -> Linear -> ReLu -> Linear -> ReLu -> ReLu.

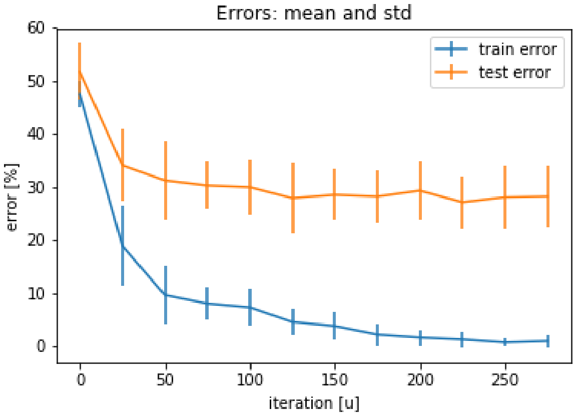
Let's look at a first validation. We used batches of size 5, a convolutional layer of size 50 with a kernel of size 5, a max pool with a kernel of size 5, and hidden layers of size 15 and 5. The convolutional layer was chosen such as to double the number of channels. The other values are inspired from the results obtained on the previous network.



At 27.5% mean final error rate, this network is not significantly more efficient than the previous one. As expected the learning is slightly slower as their more to train.

<u>-To ADAM:</u>

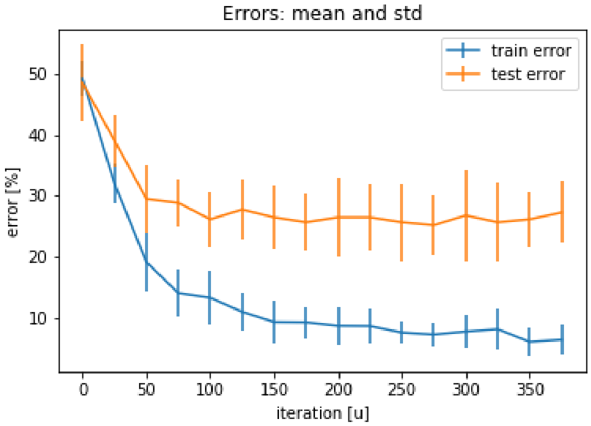
Let's switch from the SGD to the ADAM method. ADAM is meant to improve SGD by means of momentum, and adaptive updates. Momentum can help 'pass through' barriers and reduces oscillation. ADAM updates each coordinate separately. This is a strong approach to handle the anisotropy of the mapping.



The Adam optimizer converges much faster. The final result is similar.

<u>-Drop-out:</u>

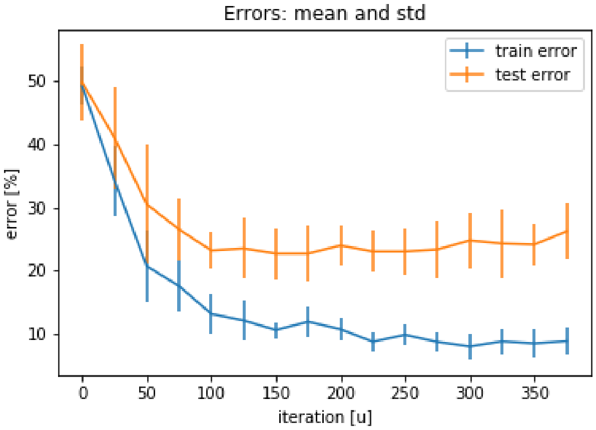
As stated before there is an obvious point where the testing error stops decreasing whereas the training error still does. To avoid this overfitting, we will use dropout. Dropout removes a percentage of neurons in a layer randomly for each iteration. This way the neurons never have the same inputs and outputs and do not learn too much in accordance to what other neurons learn. We test different values of this parameters to assess its effect.



As expected, the training error take more time to converge and never reaches a level of low variance. Again, the error rate is about the same as previously.

<u>-L2-regularization:</u>

Another method to avoid overfitting is L2-regularization. With it, a value proportional to the norm of the weights is added to the loss function. Let’s also make use of it. We chose a lambda parameter of 0.001.



Finally these result are much better, with a mean at 24.2%.