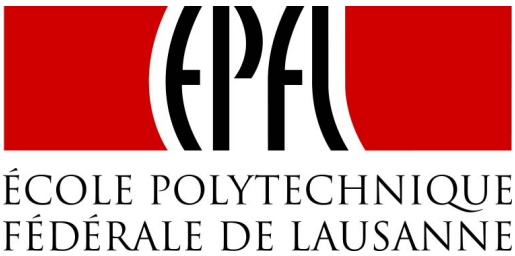


Ecole Polytechnique Federale de Lausanne

Deep Learning Course

Group number 31

Prediction of finger movements from EEG recordings

Authors	*Scipers*
Wildi Thibault	250411
Pavliv Maxim	246967
Weber Antoine	229195

Lecturer François Fleuret

May 31, 2018

1 Introduction

The goal of the project was to design and train a Neural Network to predict the laterality of finger movements based on EEG recordings. The dataset was divided into a train set of 316 recordings and a test set of 100 recordings. A total of 28 channels were measured during 0.5 s at 100 Hz or 1 kHz resulting in either 50 or 500 time samples per channel for each recording.

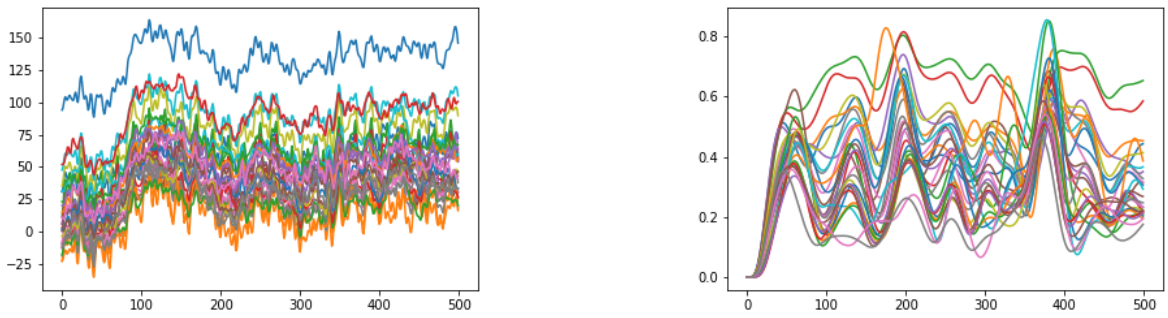
2 Baseline

To have a first glance at the complexity of the given task, we tried to implement some basic machine learning algorithms on the dataset to have an idea of the performances we should expect from our Network and set a baseline to assess whether our Network performed poorly or well.

One should consider that applying Machine Learning algorithms to such a dataset without processing it is a bit of a nonsense. Hence, we pre-processed the signal and extracted features, as expecting good classification results by simply pushing a raw signal into an LDA or SVM classifier is not a good idea. The implemented pre-processing was the following :

1. Center and normalize the signals
2. Bandpass filtering to remove noise
3. Rectification
4. Lowpass filtering to keep only the envelope of the signal

Figure 2.1 illustrates our results.



(a) Raw EEG data *(b) Preprocessed EEG data*

Figure 2.1: Raw and Preprocessed EEG data

After some research, we found that the most relevant characteristics of such bio-signals can be retained using simple features such as the Mean Absolute Value, the Number of Slope Changes and the Waveform Length [1]. The number of zero crossings can also be extracted but we decided to rectify the signal (i.e take the absolute value) which means that such a feature would always be 0. With such features, we obtained the results illustrated on Figure 2.2.

[1] https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1206493&tag=1

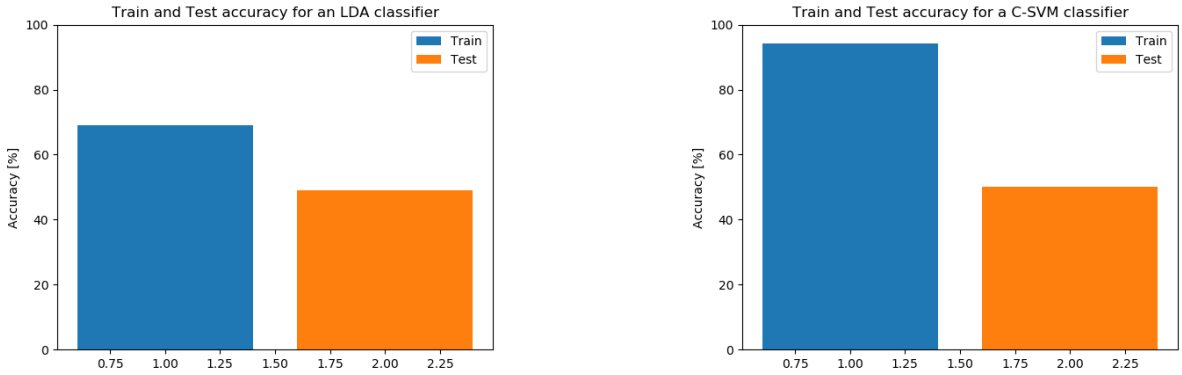
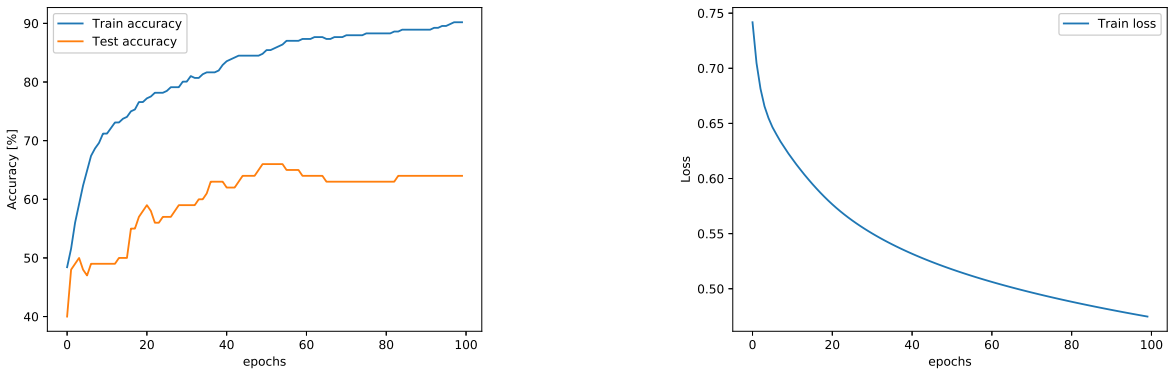


Figure 2.2: Classification results of an LDA and SVM classifier

Both results illustrated in Figure 2.2 allowed us to deduce that the given task was not easy to solve. Indeed, LDA performs a dimensionality reduction followed by linear separations, which may be the reason why the results are not optimal but SVM is already a powerful algorithm used in a large variety of problems. However SVM scored only around 90% accuracy on the train set, and around 50% on the test set. Our choice of features/pre-processing could potentially be optimized, but this was not the goal here, the goal was only to have an overview of the task.

We also tried to implement a neural network with only one hidden layer (flattening the dataset, layer of size 14000x2) we typically obtained around 90% of accuracy on the training set and 60% on the testing set. This is the most basic and primitive neural we could design. It's results are illustrated in Figure 2.3.



(a) Different accuracies using only 1 Linear layer

(b) Train loss using only 1 Linear layer

Figure 2.3: Results of the one hidden-layer network

As the dataset was used in a machine learning competition in 2004 (so pretty old, the performances were quite low at this point) we can also check what performances the winner achieved. On the website of the competition, we can see that the winner, *Zhiguang Zhang*, scored an accuracy of 84% on the test set. On one side he's the winner of this competition, so his accuracy should set the maximum performance we can hope to achieve, but on the other side, it was in 2004 and deep learning has made incredible improvements since then.

Finally, from this baseline, we concluded that if our network reached at least 70% of accuracy on the testing set without over-fitting the training set, it would be a descent score. Considering the different results we cited before, we knew that an accuracy of >95% on the test set would be hard/impossible to achieve.

3 Development of our Neural Network and learning characteristics

First, we should mention that the pre-processing steps used for setting the baseline were not used anymore. Indeed, the purpose of a deep network is that it will learn the appropriate preprocessing steps and the relevant features by itself. At first we fed the raw data to the network. After some time, we implemented a centering and normalization of the data hoping that it will help the network to generalize. Figure 3.1 illustrates the typical shape of such centered and normalized data.
As the task was a binary classification problem, the group directly implemented a CrossEntropy loss.

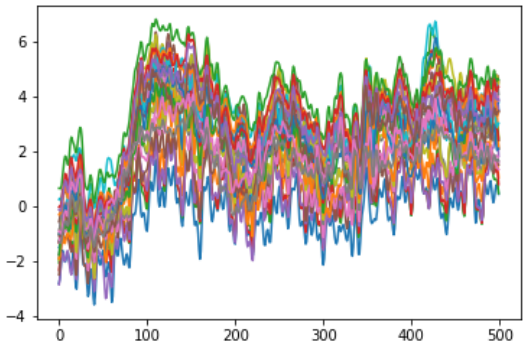


Figure 3.1: Centered and normalized EEG data

Initially, we built a network containing 10 layers (alternating convolution and pooling layers), followed by 3 or 4 fully connected layers. This first network reached about 100% accuracy on the training set but barely went over 55% on the testing set. This was a clear indication that the model was over-fitting. There were too many parameters to optimize considering the small amount of data we had. We then tried adding drop-out layers, but it didn't increase the performance in any noticeable way.

At this point we implemented normalization of the dataset as a pre-processing step (as described above). This helped but still didn't give us the results we expected. We reached the conclusion that our dataset was way too small for such a big network, as we only had a maximum of 500 points (containing noise) in the time-domain and 316 recordings for training. Therefore we proceeded to drastically simplify our network, "trading" train accuracy for test accuracy.

Having so few data-points seemed to be a problem. To overcome it, we thought of performing data augmentation. To do this, each recording of 28x500 would be segmented into 10 "sub-recordings" of 28x50. This would effectively increase the number of recordings by a factor of ten, giving 3160 training data. However performing such an augmentation is questionable. If the information in the data is only contained at specific locations or that the whole signal must be available for it the be interpreted correctly, and such partitioning could induce an irreversible loss of information. In our case, implementing this method did not increases our performances, we therefore decided to keep working on the full length signals.

We then tried to implement simpler neural networks with only 2-4 convolutional layers and one (or two) fully connected layers. The goal was to extract the interesting features with the convolutional and pooling layers, and then to classify them with the fully connected layer(s). The parameters of the layers, had then to be tuned to reach the best possible performances: highest accuracy in testing without over-fitting the training set. To avoid over-fitting we used drop-out, checked the number of free parameters (in the range of some thousands only). We also implemented batch-normalization

layers to avoid vanishing gradients.

We tried different convolutional layers with first implementing a kernel width of 3. Considering the 1kHz dataset, this kernel width was too small. Indeed, to be able to detect interesting trends in the data, the kernel width had to be increased. The different convolutional and pooling operations were always executed along the time domain as the goal of such layers in our architecture was to extract useful features (specific trends at characteristic locations, edges, etc). The first convolutional layers were done along each channel, resulting in a 1D kernel. However, to diminish the number of channels, 2D kernels (spanning across channels) had to be implemented. At this point we were working with *TensorFlow* and its *Keras* API as its simple structure made us able to test a wide variety of designs without having to tediously implement them in *PyTorch*. After some tuning of architectures, decent results were obtained considering the baseline computed with LDA and SVM : between 65% and 75% of accuracy for the testing set (on average, the exact numbers fluctuate between each run) and around 95% of accuracy for the training set. Such results were obtained using the following architecture :

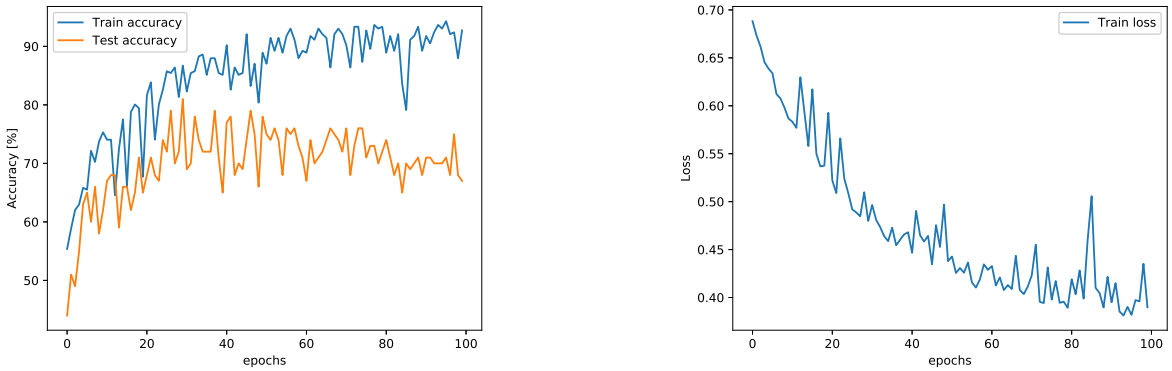
1. Maxpool layer, kernel width = 5

2. Convolutional layer, kernel width = 21 with 5 output channels

3. Maxpool layer, kernel width = 5

4. Flattening then inserting in one Fully connected layer directly connected to the output

The activation function used was a ReLu and the optimizer was Adam. At this point, we didn't manage to improve our results by tuning the parameters of the network and of the learning process anymore. We then implemented this architecture into *PyTorch*. However, by observing the evolution of the accuracies through the different training epochs, we observed high fluctuations of the testing accuracy. This is illustrated in Figure 3.2.



(a) Evolution of the accuracies through the epochs

(b) Evolution of the train loss through the epochs

Figure 3.2

We notice that a problem arises from around the epoch ≈ 40 on. The reason may be that the optimizer is bouncing across narrow valleys, therefore the learning rate has to be reduced to perform smaller steps and converge to the bottom of one of these "valleys". We then reduced the learning rate from the 20th epoch and onwards, which allowed us to gain several percent of accuracy. From this point, after some fine tuning and a little grid search, we arrived to our final architecture.

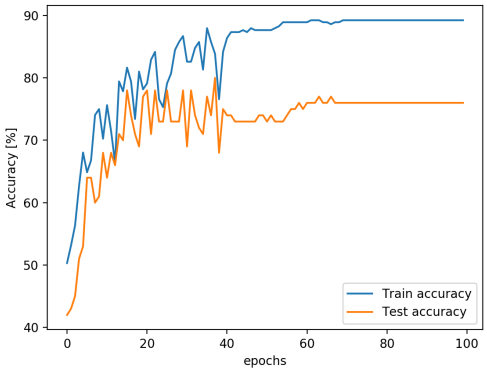
4 Results

The final architecture performing the best accuracies on Pytorch is the following :

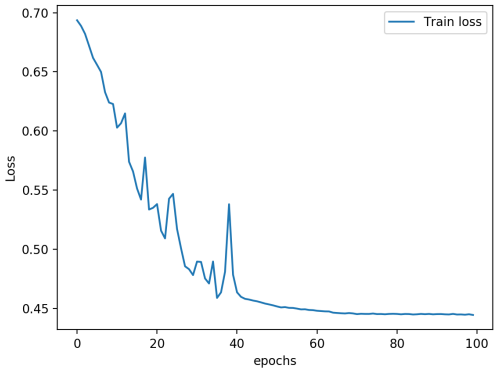
```
nn.Sequential(
        nn.Conv1d(in_channels = 28, out_channels = 28, kernel_size = (5),stride=5),
        nn.SELU(),
        nn.BatchNorm2d(28),
        nn.Conv1d(in_channels = 28, out_channels = 20, kernel_size = (5),stride=5),
        nn.SELU(),
        nn.BatchNorm2d(20),
        nn.Dropout(0.3),
        nn.Conv1d(in_channels = 20, out_channels = 5, kernel_size = (5),stride=5),
        nn.SELU(),

        Flatten(),
        nn.BatchNorm1d(20),
        nn.Dropout(0.5),
        nn.Linear(20,2),
        nn.Softmax(dim=1))
```

The architecture is first defined with 3 different Convolutional layers, all three with a kernel size of 5 and a stride of 5. Batch normalization is done in between each layer. Dropout were added at strategic points to minimize over-fitting. These locations were found as a result of grid searching. While training, the learning rate is reduced twice, at epoch 40 and 70.

This architecture converges to a score of around 76% accuracy on the test set and around 90% on the train set. Such results were satisfying as it beats the baseline we computed prior to implementing the architecture without over-fitting: the difference between the training accuracy and the test accuracy is acceptable. Figure 4.1 illustrates our final results.



(a) Evolution of the accuracies through the epochs



(b) Evolution of the train loss through the epochs

Figure 4.1

5 Conclusion

We achieved decent results compared to the baseline we set with traditional machine learning algorithms, but our score which is under 80% is not excessively satisfying, knowing what today's neural networks are capable of. The main problem was having such a few number of datapoints: a complex network would over fit, and a small one perhaps wouldn't be able the exact all the required features. Implementing deep convolutional networks could potentially score very high accuracies, but to implement such seep networks, one would need a lot more datapoints.