Deep Learning: Mini Project 1
Finger Movement Predictor

Pratibha Bhatt, Quentin Desaulty, Axel Manguy

May 17, 2018

1 Introduction

Our aim is to successfully classify EEG data to predict laterality of movement. This is an important application for the construction of brain-computer interfaces. The density of the data appears to be the most crucial challenge to overcome when designing the model. Over-fitting therefore was found to be a common problem while testing models. Three different architectures were used in an attempt to get better accuracy rates: Multilayer Perceptron, Convolutional Neural Network, and Recurrent Neural Network. Finally, a variation of RNN known as LSTM Network is presented, since it yields good results on this data set.

2 Multilayer Perceptron

Since the Multilayer Perceptron was the first historical step in the development of deep learning architectures, our first attempt on finger movement prediction will implement that specific architecture. Several tests were made on the following parameters : The number of hidden layers, the size of the hidden layers and the learning rate of the optimizer.

First try on linear classifier The very first step toward a more complex architecture is building a linear classifier. To do so, we only used a single linear layer. This first simple attempt gave surprising results considering its simplicity. The average result on 10 training gave a **72% accuracy** using *SGD* with a learning rate of 0.005.
Our goal in this section will be, at least, to do better than this very simple classifier.

Trials on hidden layers Using 3 as an arbitrarily starting point of hidden layers number, we built the network with SGD *(Stochastic Gradient Descent)* as optimizer. In order to get better results, we thought about a hidden layer that would extract features, a layer that would have a bigger size than the input. After making some test and comparing with a layer of the same size as the input, we realized that this type of layer was useless and computationally expensive.
Thus we moved to a 3 hidden layers, where each one had the size of the previous one. Deleting a hidden layer didn't change the accuracy of the model. We finally adopt a single hidden layer of size 8. We interpreted that the model was able to extract one feature per channel. Using the same learning rate as for the linear classifier, the performance were comparable Since these results were not as better as we expected, we decided to make attempts on the optimizer.

Testing SGD and optimizer parameters SGD has the peculiarity to be defined by 3 parameters the learning rate, the decay and the momentum. In our case, decay modifications did not give better results, as the learning rate is already low. Momentum, however, allow us to increase our accuracy on some tests but has not an overall impact on performance
The best set of parameters we finally obtain for this model was a learning rate of 0.005 and a momentum of 0.1, allowing us in the best cases to achieve a accuracy of **74%**.

Note that the number of epoch did have a impact on training, as it tends to easily overfit. Experimentally, we observe that starting from 500 epochs, the model overfits at some point. For the purpose of visualization in the test file of the MLP, the number of epochs is setted to 1000.

The final accuracy is close to the linear one and a better implementation is still to be found.

3 Convolutional Neural Network

EEG data was presented in the format of two dimensional tensors arranged in a row. Each 2D tensor was treated as a unique channel, since the data is at different time points.Thus, the data was first modulated from a three dimensional tensor into a four dimensional tensor with each original two dimensional sample, considered as a single channel.

Classification from abstract data, requiring the recognition of possible spatial patterns in a complex grid like data is a task that Covolutional Networks are known to work well on. The intuition behind this is that they use various kernels to convolve the data received, a process that is similar to the use of filters in traditional computer vision applications such as edge detection. Classification using EEG data is a similar task, which is to recognize patterns in the signal that can accurately classify which hand was being used. Biologically, a spatial pattern is expected due to the spatial division of lateral movement. However, the pattern is buried beneath a myriad of signals in the brain, since multiple conscious and unconscious functions are continually performed by it. Additionally, in the EEG, a 3D signal is flattened to represent it as 2D. This results in a very high density of output and contributes to the difficulty.

Therefore, we attempted to solve the problem using a CNN. The architecture had two convolutional layers, with max-pool and dropout. As expected due to the density of the data and the number of possible meaningful patterns embedded in it(which implies a low signal to noise ratio), a very high dropout rate of 0.90 gave the best results.

Different activation functions were tested. In the initial implementation, we used the standard RELU and found a top 1 accuracy on the test data equal to 25 percent.

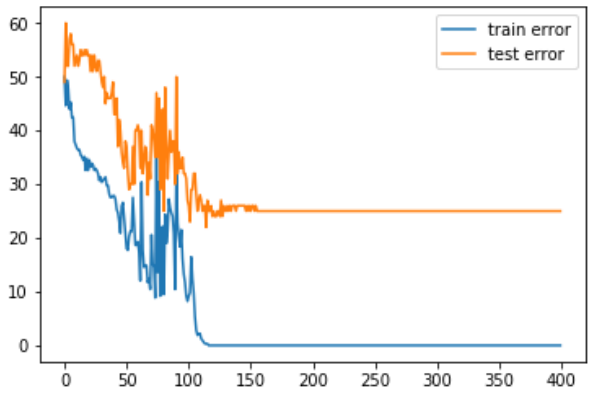

Figure 1: Training and Test error with RELU: SGD with error rate 25 percent

In RELU activation function, the data is basically allowed to pass or not based on an arbitrary threshold of 0. This is similar to how neural activation is actually calculated and processed.

However, the data in EEG signals, when not normalized, can be both positive and negative, depending on the polarity of the signal. A high negative value can also indicate a pattern of neural activations in the opposite direction. This implies that in our normalized dataset(and possibly, after convolution as well), values significantly less than the mean hold valuable information similar to the values significantly greater than the mean. Thus, RELU in the naive first implementation would be ignoring important data.

On testing a number of different activation functions such as Hard soft, ELU and sigmoid; RELU still worked the best by a margin of at least 10 percent (The performances obtained were around 35%). This may, however, be because these functions are not a good representation of a neural activation function. Additionally, they do not give a sparse representation for a dense data set, which could be a limiting factor in this approach, rather than similarity to actual neural activation.

Different optimizers were tested on the convolutional network (SGD, ASGD, Ada delta and Adam). The best results were obtained by SGD with a learning rate 0.01. Therefore, in conclusion, we obtained a model, using CNN, however, the results yielded were still less accurate than state of the art machine learning approaches.

4 Recurrent Neural Network

4.1 Simple RNN architecture

In order to get better results, we need to return to the beginning and take a new look at the data. Since the furnished data are brain records, the time axis can be seen as a sequence rather than a feature dimension. Taking that prior knowledge in account in our model could give us better result, as the model parameters will reflect more the physical reality of the data. Fortunately, there is a whole branch of neural networks that are designed for sequential data. Widely use for Natural Language Processing (NLP), the Recurrent Neural Networks (RNN) make use of the sequential nature of data. In language, the order of words has an impact on the output. In our subject, and since time is sequential by nature, the order of the record could also have an impact on the output.

This being said, the first implementation we tried was the most basic example of RNN. In this first RNN architecture, there are only two linear layers while the output is the probability for each one of the two class (left or right). The sequence is sliced into records and the output of the hidden layer is added to the new record and processed until there is no more record in the sequence. The last output (the one that corresponds to the entire sequence) is taken as the final output of the model. For that model and using clipping on gradient to improve performance, the final error rate of the model is around 23 %.

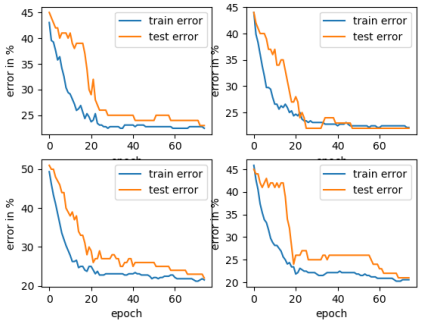

Figure 2: Training and Test error rate on 4 training

At first try the model was nothing better than a random guesser, the problem was that the output was increasing fast and at a point only gives NaN instead of probabilities. This problem known as the gradient explosion problem occurs often on RNN when the optimizer misses the minimum and can't find it. The function clip_grad_norm normalize the parameters before the next optimizer step to help it to not miss the minimum.

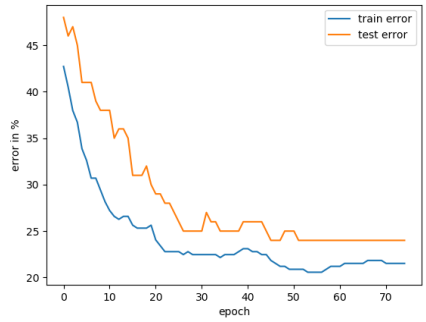

Figure 3: RNN with clip

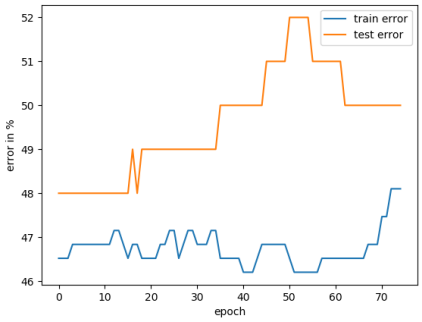

Figure 4: RNN without clip

For the other experimentations, several optimizers (SDG, Adam, RMSprop) have been tried. Some overfit quickly and thus cannot be used. Finally, the one that gives the best results is SGD with 0.1 as learning rate. Other learning rates (1, 1e-2, 1e-3, 1e-4) made the model overfits or reduces the performances. the number of epoch could be reduced since the model doesn't improve after 30 or 40 epoch, but 75 is better for the sake of visualization and convergence. The use of dropout before the softmax layer doesn't give best results although it doesn't reduce the performance either (9 different probabilities between 0.1 and 0.9 have been tried).

4.2 LSTM model

According to the results of the competition where project and data came from, the best result performs 16% of error rate and the second one 19%. Even if the competition was intended for machine learning, it shows there is still a better model to find. Since RNN was a promising field that has better performance than convolution and multilayer perceptron, we decided to search further in that direction. Another algorithm of RNN, more complex but said to have better performance, is the Long Short-Term Memory(LSTM) model. One drawback of the classical RNN architecture we implement is its difficulties to perform on large sequences. On word sequences, it means that simple RNN works for few words while LSTM performs on whole sentences. For our problem, the data can be seen as a sequence of 50 or 500 words. Better results are thus expected with LSTM since it aims to work on large sequences.

Concerning the implementation, there is already an LSTM layer implemented in pytorch so we don't have to implement it on our own. In return for that simplicity, we have to pass the data in a specific format such as sequence length, batch size, and input size. After that LSTM layer (2 layers in fact), there is an output layer with batch normalization that produces the 2 class requested and then a softmax layer to have probabilities. Adding a strong drop out to that architecture gives good results up to 17% error rate.

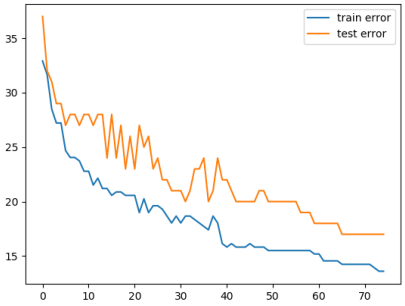

Figure 5: Training and Test error rate on 4 training

For that model, the optimizer has a great impact on performance. At first, using SGD and a too large learning rate, the model was unable to learn and wasn't better than a random guesser. With correction of the learning parameter, the model was able to learn but the result was disappointing. With Adam as optimizer, the model gives good results but not always better than simple RNN. Finally, the pytorch documentation indicates an optimizer that first appears in an article on RNN : RMSProp. This optimizer gives indeed better results around 19% error rate and up to 17% when combined with other techniques.

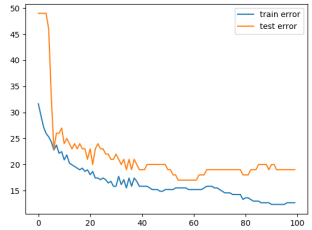

Figure 6: LSTM and RMSprop

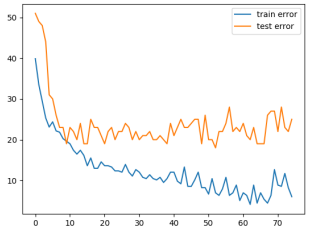

Figure 7: LSTM with Adam

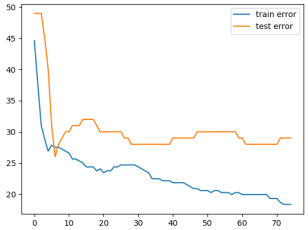

Figure 8: LSTM with SGD

To get the better performance possible several other techniques have been tried. One first thing to take into account is that LSTM is difficult to train. Moreover, it has a strong tendency to overfit. To prevent that phenomenon a dropout layer with a strong probability and a batch layer normalization has been implemented. Especially for the dropout, the probability have an influence on results. The better result was obtained for 0.9 of dropout probability (same test procedure as for RNN). To prevent overfitting, the optimizer and learning rate being definitively chosen, we tried to reduce the hidden size. Starting at 128 and decreasing by steps of 20, we manage to get better performances with less overfitting for a hidden size of 65, closer to the sequence length and the batch size. Finally, we tried L1 and L2 regularization with no noticeable error rate improvement.

5 Conclusion

In terms of performance, the final result doesn't outperform the best machine learning implementations. However, it gives satisfying results with an average performance of 19%. Considering the task of predicting movements, 19% is an interesting result that could find some applications. The fact that data came from real human brains makes them noisy and models find difficulties in learning on such data. The very reality of the data explains by itself the disappointing performance of deep learning model. As always, data quality is the underlying condition of machine learning performance.
In terms of method, we started with a very basic architecture and as we increased progressively depth and complexity, we saw the performance arise. If the first model (Multilayer Perceptron) gave an average error rate of 26%, the use of more complex architecture with the Convolution Neural Network gave slightly better results with 25% error rate. Afterward, taking a new look at the data and taking into account the sequentiality of records led to a 23% performance. Finally, making the network learn and memorize better on longer inputs, gave the best results with a 19% average performance.
About the test procedure, for each model, we tried to evaluate the impact of numerous parameters such as the optimizer, the learning rate or the use of more advanced features (Dropout L1/L2 regularization...).
The final outcome of that project is a better understanding of neural network architecture as well as a practical sense on how to use neural networks on real-world data.