Machine Learning Course, CentraleSupélec, 2017/2018

Project Report

Mélisande Teng, Joël Seytre

melisande.teng@student.ecp.fr, joel.seytre@student.ecp.fr

1 General setting

1.1 General

On the kaggle challenge (leaderboard at `https://www.kaggle.com/c/dreem-deep-sleep-power-increase/leaderboard`), our team name is inspired from a well-known meme:

Figure 1: *How the Machine Learning community feels since 2012*

Our code is available at `https://github.com/joelseytre/dreem`.
We chose to work in Python, using Keras on top of a GPU-enabled tensorflow build.

Note: throughout the report, we will specify what is implemented in our final code as well as what we went through and how it compares. We will try to be as explanatory as possible, with some "educational remarks" here and there that try to be pedagogical (as required for the project).

1.2 Context

Of all the stages of sleep, deep sleep is the most restorative one, yet, its quality can be affected by stress, or aging, and it can only be monitored with brain activity. Dreem headband proposes to stimulate deep sleep thanks to synchronized closed-loop sound stimulations, proved to enhance deep sleep quality. Sensors on the headband measure brain activity through EEG, and respiratory rate, to determine when it is a good time to send a stimulation.
In this challenge, we predict the impact of a stimulation. The accuracy of the prediction is measured in terms of root mean-squared error.

1.3 Our final pipeline

The pipeline from the untreated `.csv` files to the final `.csv` containing our predictions is as follows:

- The data is split into EEG, respiration and meta data (which implements a One-Hot Encoder for user IDs), normalized separately for EEG and respiration data, and then split further into a training and validation set;
  (see Section 2)

- The EEG data as well as the user encoding are fed to a feed-forward Neural Network which predicts the power increase we are trying to model.
  The activations of the last layer are stored for future usage - we have effectively extracted features from the EEG data;
  (see Section 3)

- Those features are put together with the user encoding, the result of a Fast-Fourier Transform of the respiration data as well. We then train a Gradient-Boosting Regressor on the obtained data.
  (see Section 4)

2 Pre-processing

2.1 Dividing the dataset

We divide the dataset in 3 main sets:

- The EEG data
- The respiration data
- The rest of it under the label "meta-data"

In order to cross-validate our results, we divide our $50,000$ 8-second recordings into a training set of $40,000$ data points and a validation set of $10,000$ data points. This ensures that we can test our solution without submitting it but most importantly it allows us to tune our Neural Network and our Gradient-Boosting Regressor.

2.2 Normalization

We noticed that normalization had a sizeable impact on the results.
We started by normalizing each feature into `[0;1]` using `scikit-learn`'s prepocessing tools but then noticed that doing that broke the temporal continuity of the signal.
That is why in the end we normalized the entire dataset, separately for the EEG and the respiration data. To that end we substracted their mean value to the respective datasets and then divided by their variance.

Note: Normalizing into `[0;1]` can cause a lack of robustness with respect to outliers, which is why we divide by the mean and not the maximum.

2.3 One-Hot Encoding on Users

We noticed that the users are the same in the training set and the test set (there are 143 of them).
In order to avoid an arbitrary order where user n°54 could be assumed "closer" to user n°55 than user n°88, we implemented a One-Hot Encoding: the $50,000$ x 1 column containing the user ID is converted to a $50,000$ x 143 matrix where each row contains only zeroes except for a 1 in the column corresponding to the user ID.

2.4 Fast Fourier Transform

We applied FFT on EEG and Respiration data. The Fourier transform converts waveform data in the time domain into the frequency domain, breaking down real signals into mathematical functions that can be more easily handled. The advantage of Fast Fourier Transform is essentially computational efficiency.

3 Using a Neural Network to extract features

3.1 Purpose

Neural Networks can be an efficient way of extracting features automatically, without hand-crafting them.
We tried to see what the performance of a neural network might be as a regressor for this task. We first implemented a deep convolutional neural network (CNN) that took in the 3 separate categories of data as separate branches and concatenated the sub-networks. We thought that 1-D CNNs might be well suited for such a temporal task.
We were not satisfied with the results - we then used the network to extract features that would better represent the data than the raw inputs.

Note: since we benefited from a GPU installation, we thought it would be a good chance of using it! Fast computations guaranteed

3.2 Architecture

Ironically (given our Kaggle team name), we did not implement any CNN in our final solution.
After a lot of trial and error, we found that a very simple structure worked well to extract the activations.
The architecture is as follows:

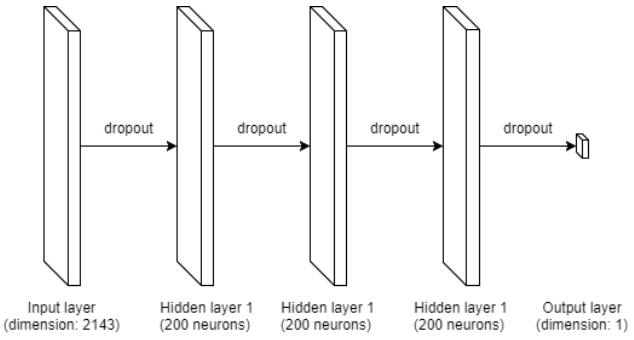

Figure 2: *Our network architecture: pretty straight forward...*

3.3 Tuning the network

Fine-tuning the network played a big role in improving our results.

- **Regularization:** Over-fitting is an issue that comes up frequently when trying to use neural networks. Indeed, the number of parameters is usually "pretty large" and the deep neural structure makes it possible for the network to learn non-convex solutions quite easily.
  We made some attempts with the built-in regularization tools of `L1` and `L2` weight

decay, but noticed that it didn't help us in the way we wanted. In the end the use of dropout (with a factor 0.5) was enough.
Dropout, helps prevent overfitting[1]. The idea behind dropout: a fraction (here 50%, which results in the most regularization) of the activations of a given layer are randomly put to 0 during training time (not during test time! to compensate, activations are averaged at test time). This forces redundancy in the network, which causes robustness and increases regularization.
With regularization in mind, we also reduced the number of epochs in order to use early-stopping: the idea behind it that the network first learns general features and then goes into specific features that cause over-fitting. The idea is to stop before that. The final number of epochs was determined by looking at the evolution of the validation error and stopping when it starts to stagnate while the training error is still going down.

- Given the fact that the challenge is evaluated based on the Root Mean Square Error (RMSE), we use the keras built-in Mean Square Error loss;
- For the optimizer we use Adam, which has recently been established as one of the best optimizers for training neural networks.
- We chose to use `ReLu`(Rectified Linear Units) as non-linear activation functions as a result of trial and error. We tried `ELu`as well as `tanh` and the `sigmoid` function.

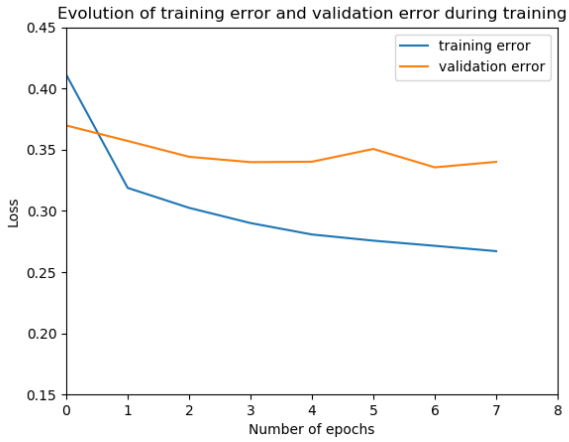

Figure 3: We stop when the validation error stagnates, even though the training error is still going down.

3.4 Educational Remarks

On Regularization In presence of a large number of features, regularization helps tackling overfitting by penalizing the magnitude of coefficients of features for better prediction.

[1]http://www.cs.toronto.edu/ rsalakhu/papers/srivastava14a.pdf

`keras` offers various built-in weight regularizers, `l1`, `l2`, and `l1-l2`. We will go quickly though the three options in this paragraph.
L1 regularization, also known as LASSO, adds penalty equivalent to absolute value of the magnitude of coefficients in the objective. With LASSO regularization, we get sparsity in the coefficients.
L2 regularization, also known as weight decay, or Ridge, adds an extra term to the cost function that penalizes squared weights, keeping the weights small, and helps to stop the network from fitting the sampling error. L2 is chosen over L1 for data that is not at all sparse, and where we don't expect regression coefficients to show a decaying property.
L1-L2 regularization (ElasticNet) combines both L1 and L2 methods, at the cost of an added hyperparameter to tune.

On Neural Networks Neural Networks are inspired biological neural networks in animal brains, the original goal being to solve problems in the same way that a human brain would. Though resemblance with the human brain can seem rather far-fetched, one can still see some logic in the sense that no one actually understands what deep learning algorithms exactly do, just as no student can read in the teacher's mind (unfortunately for us!).
A neural network is based on a collection of connected nodes called artificial neurons, that

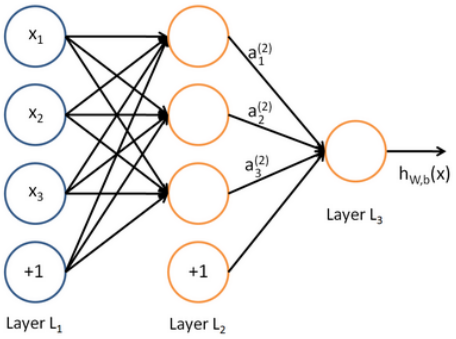

Figure 4: *Basic architecture of a neural network*

form layers (input - hidden (multiple) - output) (Fig. 4[2]). As in a synapse, artificial neurons can transmit a signal to the other neurons they are connected to. Usually, a different function is applied to the data in each layer. The outputs of the nodes are called activations.
Some nodes are not connected to the previous layer, they are called "bias" nodes and act as intercept terms in the regression. Now, to put it simply, the artificial neurons are detectors that turn on in response to the activations received from the previous layer. If all the detectors turn on, there is a good chance that your prediction was accurate. Neural networks are useful and popular because there exist some good tools for building and putting a lot of detectors together.

On our implementation We can note that our network is not that deep, with only 3 hidden layers.
We attempted some solutions with a deep neural network that would convolve over the temporal dimension, but it seems that the results were not as good in creating relevant

[2] `http://ufldl.stanford.edu/wiki/index.php/Neural_Networks`

features.
Our hypothesis: the fact that the window of time before and after the impulse is fixed (4 seconds before, 4 seconds after) makes the "sliding-window" part of a CNN irrelevant since we do not need to detect phenomenons with very different temporal scales.

Figure 5: *The famous internet meme, partially credited by Google researchers for the name of their neural network's architecture Inception.*
It seems that we didn't need to...

4 Gradient boosting

We tried Gradient Boosting with `sklearn.ensemble.GradientBoostingRegressor` and `lightgbm.LGBMRegressor`. In the end we used the latter one as it had settings that suited us better (especially early-stopping), and as it is slightly faster.

4.1 Parameter tuning

4.1.1 sklearn

Parameters for the best prediction obtained with `sklearn.ensemble.GradientBoostingRegressor` were tuned as follows:

- As the accuracy of the prediction is assessed with RMSE, `loss` is the built-in least-squares.
- `n_estimators` is 300. It seemed there was no significant difference when putting a very large number of estimators (500-1000), and that it only lead to a huge increase in computational time.
- `learning_rate` is 0.1. A smaller learning rate is always better; however, the number of estimators needs to be increased as the learning rate is lowered to have a robust model, and it can be computationally intensive. As `sklearn` doesn't have early stopping built-in we kept learning rate at default.
- `subsample` is 0.8. Choosing a subsample less than 1 reduces the variance.

In order to test several parameter combinations at the same time, we found it quite convenient to use `GridSearchCV` to specify a range for the parameters and find the best parameter combination.

4.1.2 lightgbm

`lightgbm.LGBMRegressor` performs as well as `sklearn`'s gradient boosting module, but is faster and has a built-in early stopping option (no need to set the number of estimators). We set the early stopping option to 50 to prevent overfitting.
We ended up using lightgbm because it suited us better, with the same settings found for the `sklearn.ensemble` gradient boost regressor. Our final submission score was 0.51942.

4.2 Educational Remarks

On Gradient Boosting Gradient Boosting is a very useful technique for regression problems, and uses Gradient descent and Boosting. Boosting is a method that was originally designed for classification problems, but that can be extended to regression problems. The idea is to produce a series of weak learners (learners that have accuracy only slightly better than chance) to get a strong learner. The predictions of each weak learner are usually combined according to the following rule: after a weak learner is added, the data is reweighted, and examples that are well predicted lose weight, and examples that were mispredicted gain weight.
In Gradient Boosting, models are trained sequentially, gradually minimizing the loss function using Gradient Descent method. Therefore, learners are constructed in such a way that they can be maximally correlated with the negative gradient of the loss function.

5 Conclusion

In this report we explain the process behind the *CNNs, CNNs everywhere* team of the Dreem Kaggle challenge that closed on Jan. 21st, 2018.
Thank you for your attention and we wish you many successful convolutions (unlike this project).