Prediction of Epileptic Seizures using Machine Learning and Deep Learning models

Master's Thesis submitted to the

Faculty of Informatics of the *Università della Svizzera Italiana*

Master of Science in Informatics

Major in Artificial Intelligence

presented by

Alessia Ruggeri

under the supervision of

Prof. Cesare Alippi

co-supervised by

Dr. Daniele Grattarola

September 2019

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Alessia Ruggeri
Lugano, 5th September 2019

Contents

Acronyms

Adam adaptive moment estimation

AI artificial intelligence

AUC area under the curve

CNN convolutional neural network

DRE drug-resistant epilepsy

ECC edge-conditioned convolution

EEG electroencephalogram

FCNN fully-connected neural network

GCN graph convolutional network

GNN graph neural network

iEEG intracranial electroencephalogram

LSTM Long short-term memory

ML machine learning

MRI magnetic resonance imaging

MSE mean squared error

NN neural network

μV nanovolts

RBF radial basis function

ReLU rectified linear unit

RNN recurrent neural network

ROC receiver operating characteristic

SVM Support Vector Machine

Tanh hyperbolic tangent

Chapter 1

Implementation

In this chapter we present the tools that have been used for the implementation of the project, some details about the data, the data preprocessing procedure, a description of the models' configurations and experiments conducted and the produced results.

1.1 Tools

All the computation for this project has been conducted on a university server equipped with Ubuntu SMP version 16.04.1 and a NVIDIA TITAN Xp graphics card for the deep learning training processes. A brief description of the software tools used for the project will follow.

Python [1] The programming language used for this project is Python v3.6. Python is a general purpose language, known for its ease of use and understanding; however, thanks to the addition of dedicated libraries for data analysis and predictive modeling, in the last few years it has become the reference and most-used language for data science.

Numpy [2] Numpy is an extremely popular and useful library for scientific computing with Python. It allows to easily handle multidimensional data through matrix representation and to perform operation between them thanks to its broadcasting functions. Numpy has been used in all the project implementation's steps to handle and manipulate the data.

Pandas [3] Pandas is an open source library which provides efficient, flexible and easy-to-use data structures and data analysis tools for Python. Pandas is built on top of Numpy library and it is suited to handle almost any kind of data, representing them in a handy tabular form. We mainly used Pandas in order to store the results from the experiments.

matplotlib [4] Matplotlib is a 2D plotting library and it represents one of the most common visualization tools for Python. Its `pyplot` module provides a MATLAB-like functional interface and a wide degree of customization of the generated figures. All the plots in this thesis has been generated using matplotlib library.

scikit-learn [5] Scikit-learn is a well-known, simple and efficient library for data analysis and machine learning in Python. It is open-source and is built on NumPy, SciPy, and matplotlib libraries. It provides several useful tools for data preprocessing, model selection, classification, regression, clustering and dimensionality reduction. In this project, it has been heavily used for the data preprocessing and to implement the classic machine learning models.

XGBoost [6] XGBoost is an optimized library for distributed gradient boosting, designed to be highly efficient, flexible and portable. It implements a tree-based gradient boosting model which has been used for the gradient boosting experiments.

Tensorflow & Keras [7] [8] The framework we used in order to build deep learning models is Tensorflow v2.0. Tensorflow is one of the most popular frameworks for machine learning and deep learning; it is open-source and it provides a flexible ecosystem of tools, libraries and community resources to easily build machine learning models. On top of Tensorflow, we used Keras high-level neural networks API. Keras was originally a library separated from Tensorflow, providing ready-to-use tools for fast experimentation and developing of neural networks models by running on top of TensorFlow, CNTK, or Theano. With the update to version 2.0 of Tensorflow, Keras has officially become part of Tensorflow API. We mainly used Keras library on top of Tensorflow for the implementation of deep learning models in order to generate high-level and easy-to-understand code. This choice was reasoned by the fact that this thesis project is also related to the medical field, so we tried to make the code readable also by people which are not specialized in the data scientist field.

Spektral [9] Spektral is a Python library for graph deep learning, based on the Keras API. It provides a simple but flexible framework for creating graph neural networks (GNNs) by making available several ready-to-use, but still highly customizable, graph-based deep learning layers. It also implements functions for the creation of the functional connectivity network from a data stream. In this project, Spektral library was used for the generation of functional connectivity graphs and for the implementation of graph-based deep learning models.

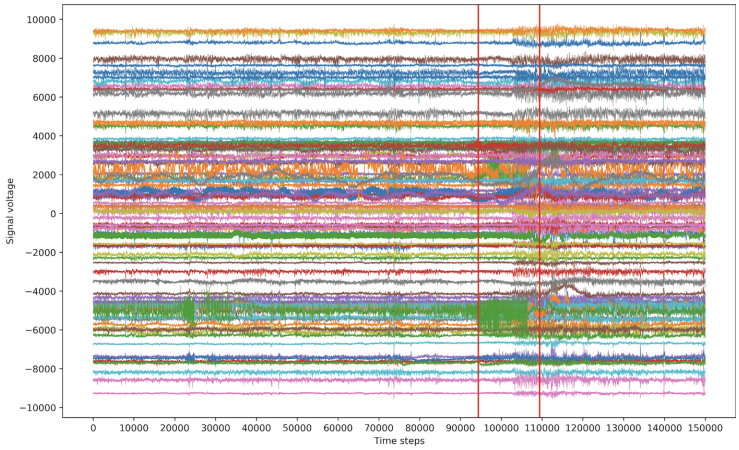
1.2 Data analysis

For this project, we were provided with 24 hours of iEEG data generated from real measurements on a patient suffering form epilepsy. The data contains three seizures, all happening during the first three hours of recording; therefore just the data regarding the first three hours have been used in the project. The brain activity has been measured using 90 electrodes and a sampling frequency of 500 Hz, so each hour contains 1 800 000 time steps of measurements.

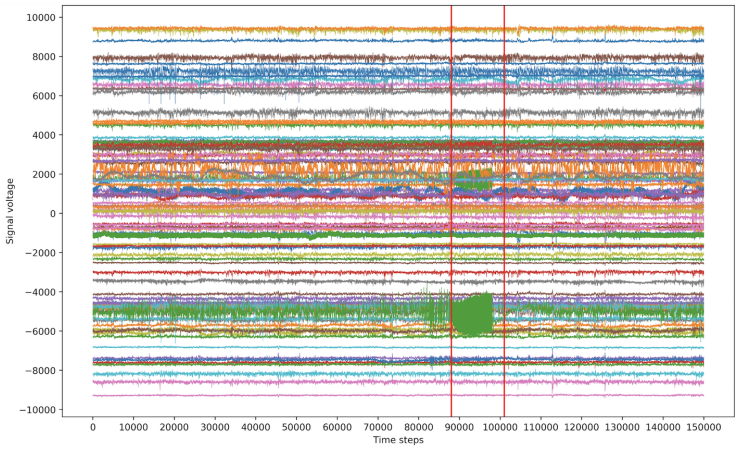
First thing, we identified the position of the seizures in the data and their length. As already mentioned, the data contains three seizures in total, each one having a duration between 13 000 and 15 000 time steps, for a total of 42 000 time steps of seizure. This corresponds to 26 to 30 seconds of duration for each seizure and 84 seconds of seizure in total. Since the most important data for this project were the seizure time steps and we did not need a huge amount of non-seizure time steps, during all the project we considered only a portion of data around each one of the three seizures, obtaining 450 000 time steps (15 minutes) to work with between seizure and non-seizure data in total. This should give an idea of how much limited the amount of useful data was for this project.

Some basic statistical measurements have been applied to the iEEG data in order to familiarize with its features. The measured voltage in the iEEG vary between about −9 500 and 9 800 μV (microvolts), covering a range of about 19 300 μV. Each electrode's signal oscillate through time with a standard deviation of around 82 μV and the various electrodes signals cover different areas of the iEEG voltage range based on their placement on the brain surface. In Figure 1.1 the plots of the iEEG around the three seizures is shown. The start and end times of the seizures in each plot are indicated by red vertical lines.

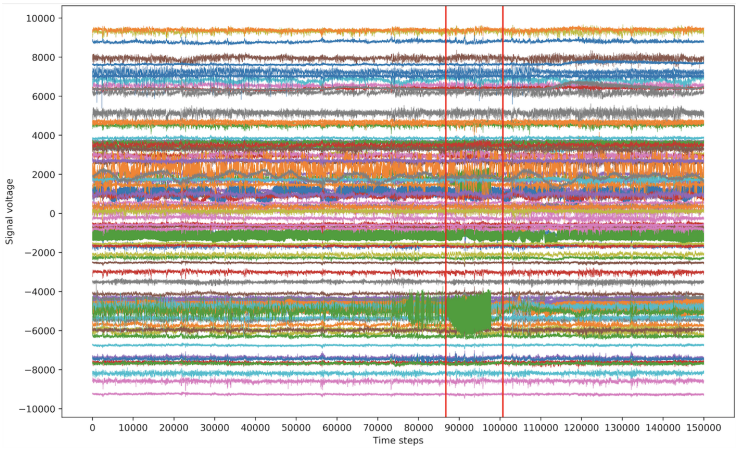
Change following 3 iEEGs to high-quality corresponding images



(a) First epileptic seizure



(b) Second epileptic seizure



(c) Third epileptic seizure

Figure 1.1. Plots of the iEEG around the three epileptic seizures (we suggest to zoom-in the figures in the pdf file)

Looking at Figure 1.1, it seems like the iEEGs is not able to explicitly show the presence of a seizure, since the dynamic of the electrodes signals does not seem to undergo a big change in correspondence of the beginning of a seizure. This behaviour suggests that the seizure could not be directly represented by the dynamic of the signals, but it could be represented by the non-linear relations between the electrodes signal's dynamic through time.

The amount of positive time steps, that are the ones inside the seizure portions, corresponds only to the 9.3% of the data, with the remaining 90.7% of data being negative time steps; therefore we are dealing with very unbalanced data (see Figure 1.2). The extreme unbalance of the data, together with the severe restriction in the amount of data, make this dataset very difficult to work with, but at the same time very similar to a real-world scenario.

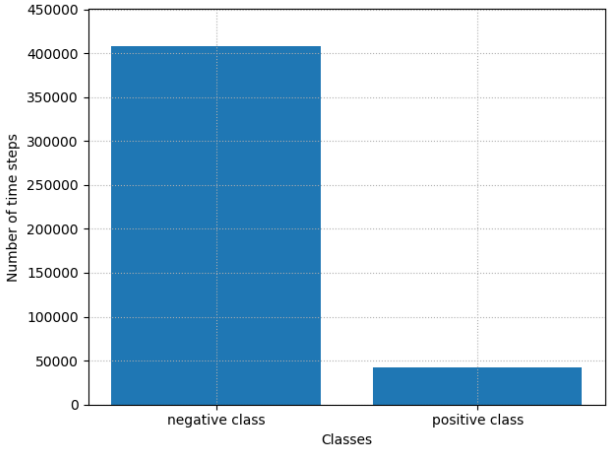
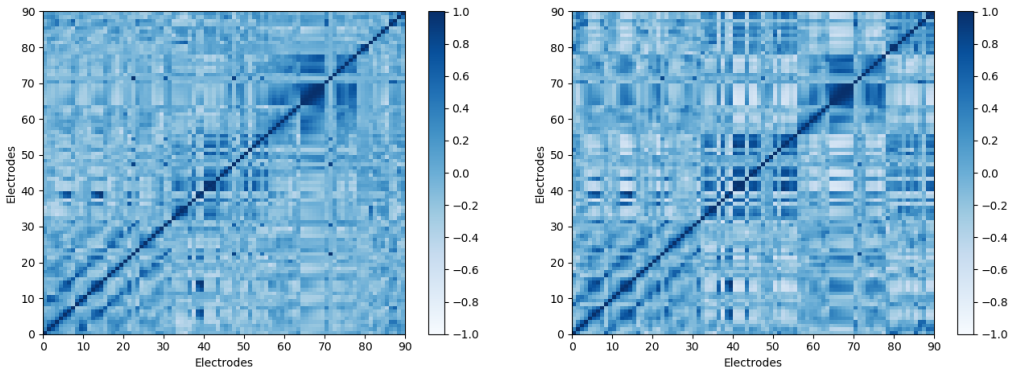


Figure 1.2. Histogram of the number of time step for each class

In order to have an idea of the linear relation between electrodes values, we computed the Pearson correlation coefficient on two sequences of 5 seconds each, one of negative time steps and the other of positive time steps. In Figure 1.3 we show the related correlation heatmaps. As you can see from the figures, the correlation heatmap related to the positive time steps reveals a higher linear relation between electrodes values if compared to the correlation heatmap related to the negative time steps. The correlation heatmaps, however, present only the linear relations between data, while we believe that there are hidden non-linear relation between electrodes signal through time which could be crucial for the identification of epileptic seizures.

To have another confirmation of the presence of some linear relation between electrodes signals, we computed the standard deviation of the oscillation of each electrode,



(a) Correlation heatmap of 5 seconds of negative time steps

(b) Correlation heatmap of 5 seconds of positive time steps

Figure 1.3. Comparison of correlation heatmaps related to negative and positive time steps

but this time we computed the mean of the standard deviation both over the negative time steps and over the positive ones. The average standard deviation of the oscillation of each electrode over the negative time steps was around 80 μV, while the one over the positive time steps was around 93 μV. This shows that, during an epileptic seizure, electrodes signals tends to have wider fluctuation margins, but the difference from the standard fluctuation margins does not seems to be enough to identify a seizure.

1.3 Data preprocessing

Once we familiarized with the dataset, we preprocessed it in order to shape it to the right form to be the input of machine learning and deep learning models for the different prediction tasks. As already mentioned in Section **??**, we divided the data in a training set and a test set, using two seizures' data in the training set and the remaining seizure's data in the test set. We sacrificed the validation set for lack of data. The data was prepared to be used by different models for the three prediction tasks described in Section **??**. Depending on the model and on the task at hand, a sample could represent a time step, a sequence of samples or a sequence of graphs.

1.3.1 Time steps as samples

This data configuration has been used only for the problem case of detection on a time step; therefore it as been used as input to random forest, gradient boosting, SVM and dense neural network. In order to prepare data with each time step represent-

ing a sample, no additional operations where required, since the dataset was already provided as a big sequence of time step - target couples. Since SVMs and neural networks are sensible to the scaling of the data, in those cases a `StandardScaler` from scikit-learn library has been added. The `StandardScaler` standardize the features of a sample such that the data distribution will have a mean value equal to 0 and a standard deviation value equal to 1. The standardization is computed as:

$$z = \frac{x-\mu}{\sigma} \qquad \text{with} \qquad \mu = \frac{1}{N}\sum_{i=1}^{N}(x_i) \tag{1.1}$$

$$\text{and} \qquad \sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i-\mu)^2} \tag{1.2}$$

where z is the standardized version of x, μ is the mean of x and σ is the standard deviation of x.

1.3.2 Sequences of time steps as samples

This data configuration has been used for the problem cases both of detection on a sequence and of prediction on a sequence. The sequences of time steps generated has been the input to standard convolutional and LSTM neural networks. In order to prepare data with each sequence of time steps representing a sample, the dataset has been converted in a set of sequences with a single target associated to each sequence. To do that, we implemented a function that considers different parameters.

The `look_back` parameter indicates the number of time steps contained in each sequence; in other words it represents the amount of time steps at which the model can look at in order to predict a target.

The `target_steps_ahead` parameter determines how far in the future we want the target time step of a sequence to be, starting from the position of the last time step of the sequence. This means that if we set `target_steps_ahead`=0, the model will use the sequence's features to predict the target associated with the last time step of the sequence, that corresponds to the problem case of detection on a sequence. Alternatively, if we set `target_steps_ahead`>0, the model will use the sequence's features to predict the target associated with a time step outside the sequence and in the future, that corresponds to the problem case of prediction on a sequence.

The `stride` parameter sets the size of the distance between the beginning time step of two consecutive sequences. For instance, if `stride`=1, the sequences will begin time

steps at indices $i_0, i_1, i_2, i_3, ...$, while if `stride`=3, the sequences will begin with time steps at indices $i_0, i_3, i_6, i_9, ...$

The `subsampling_factor` parameter adds a downsampling effect to the number of sequences by keeping a factor of negative samples with respect to the positive ones. This means that if, for example, `subsampling_factor`=5, for each positive sample we will keep only five negative samples. This is useful in order to decrease the unbalance in the data. The downsampling effect is applied only if the parameter `subsample` is set to *True*.

The function we implemented to generate the sequences of time steps considers one time step index at a time, it takes the previous `look_back` time steps to create the sequence and it looks `target_steps_ahead` time steps ahead to identify the target to assign to the sequence. After that, it consider the next time step index, which is `stride` time steps after the current one, and repeats the process. After having created all the sequences, the function discards some random negative sequences in order to respect the proportion between positive and negative samples given by the `subsampling_factor`.

In order to create this data configuration, we preprocessed the data by first applying a `StandardScaler`, like in the previous case, and then using the sequence generator function both on the training set and the test set, using the same parameters for both of them. The only difference was the use of `subsample`=*False* for the test set, since during the testing of the model we do not need to compensate the unbalance of the dataset.

1.3.3 Sequences of graphs as samples

This data configuration has been used for the problem cases both of detection on a sequence and of prediction on a sequence. The sequences of graphs generated has been the input to graph-based convolutional and LSTM neural networks. In order to prepare data with each sequence of graphs representing a sample, the first part of the preprocessing process has been the same as for the sequences of time steps: we first applied a `StandardScaler` and then used the sequence generator function both on the training set and the test set, without subsampling the test set. After that, an additional operation was required, that is the transformation of the sequences of time steps into sequences of graphs.

As already mentioned in Section **??**, the sequences of time steps can be transformed into sequences of graphs by computing the functional connectivity between electrodes and generating a corresponding graph for each time window. To do that, we considered one sequence at a time and we divided it into a fixed number of time windows, each one containing the same number of time steps. For each time window, we built a

graph having one node for each electrode and we used the correlation values between electrodes signal in that time window as edge attribute. The number of edges was limited by keeping only the edges with most significative correlation values. Through this process, we generated a graph for each time window in the sequence, therefore the sequence of time steps was transformed in a sequence of graphs. The target remained the same as for sequences of time steps.

This procedure has been implemented with the function `get_fc` from `Spektral` library, which does exactly what just described: it takes a sequence of time steps as input, it divides it in time windows, it computes the functional connectivity between electrodes inside each time window and it outputs the resulting sequence of graphs. The function `get_fc` allows to configure some parameters, among which `samples_per_graph` determine the dimension of the time windows to divide the sequence and `percentiles` controls which edges in the graphs will be removed. To be more specific, `percentiles` is a tuple of two numbers between 0 and 100 and it affects the number of edges in the graph by causing the removal of the links with correlation value between the two percentiles.

1.3.4 Cross-validation

In order to have representative results, we performed k-fold cross validation for all the models and the tasks. Cross validation is a useful technique to evaluate machine learning models, especially when there is a limited amount of data at disposal and we cannot rely on the validation set. In order to perform k-folds cross validation, the dataset is divided into k subsets (folds), of which $k-1$ folds are used as training set and the remaining fold as test set. The training-testing process is performed k times, so that each fold is used as test set one time and it is included in the training set all the other times. To perform cross validation with our data, we divided the dataset into three subsets (3-folds cross validation), each one containing one of the three seizures, and we prepared the three corresponding training and test sets, each time using two folds as training set and the remaining fold as test set.

1.4 Experiments

As already mentioned in Section **??**, a big amount of experiments have been conducted in order to apply the chosen machine learning and deep learning models to the three cases of the problem of epileptic seizures prediction. These experiments helped to identify the best-performing configuration for each model on the task at hand. This

section will present the models architectures and parameters that have been tested and the best configuration that has been selected for each model, which generated the results described in section 1.5. The best configuration was chosen by looking for the best trade-off between the evaluation metrics over the 3-fold cross-validation.

1.4.1 Support Vector Machine experiments

The SVM model has been tested on the problem of detection based on a single time step. We did not make a lot of experiments using SVM, since the poor results immediately suggested that this machine learning algorithm is too weak to handle the complexity of the seizure prediction task.

In order to test the SVM model, we used the `svm.SVC` implementation from scikit-learn library. We kept all the parameters to the default values, except for the `gamma` and `class_weight` parameters. The SVM has been tested with a penalty parameter $C = 1$ and using a Gaussian radial basis function (RBF) kernel. The `gamma` value was set to `'scale'`, that corresponds to $\frac{1}{F \cdot var(X)}$, where F is the number of features and $var(X)$ is the variance of the input features. We tested both a balanced and an unbalanced version of the SVM by setting the `class_weight` parameter to `'balanced'` or leaving it to `None` and we obtained slightly better results using the balanced configuration of the model. Table 1.1 presents the parameters of the **??** model that performed better.

Parameter	Value
C	1
kernel	`'rbf'`
gamma	`'scale'`
balanced	`True`

Table 1.1. Parameters of best-performing SVM model

1.4.2 Random forest experiments

The random forest model has been tested on the problem of detection based on a single time step.

In order to test the random forest model, we used the `ensemble.RandomForestClassifier` implementation from scikit-learn library. We kept all the parameters to the default values, except for the `n_estimators`, the `max_depth` and `class_weight` parameters. The `criterion` parameter to measure the quality of a split was set to `'gini'`, which corresponds to the Gini impurity function. We tried several values for

n_estimators, which is the number of trees in the forest, and for max_depth, which is the maximum depth of the tree, in order to find the best balance between the two. We tested both a balanced and an unbalanced version of the random forest classifier by setting the class_weight parameter to 'balanced' or leaving it to None and there was no resulting difference between the two configurations. Table 1.2 presents the parameters of the random forest model that performed better.

Parameter	Value
n_estimators	20
max_depth	8
criterion	'gini'
balanced	True/False

Table 1.2. Parameters of best-performing random forest model

1.4.3 Gradient boosting experiments

The gradient boosting model has been tested on the problem of detection based on a single time step.

In order to test the gradient boosting model, we used the XGBClassifier implementation from xgboost library. We kept all the parameters to the default values, except for the n_estimators, the max_depth and scale_pos_weight parameters. The booster parameter was set to 'gbtree' in order to use a tree-based booster. As for the random forest model, we tried different values for n_estimators and for max_depth in order to find the best balance between the two. We tested both a balanced and an unbalanced version of the random forest classifier by setting the scale_pos_weight parameter to $\frac{\text{num\_negative}}{\text{num\_positive}}$, which are respectively the number of negative and positive samples, or by leaving it to None and there was no resulting difference between the two configurations. Table 1.3 presents the parameters of the gradient boosting model that performed better.

Parameter	Value
n_estimators	20
max_depth	4
booster	'gbtree'
balanced	True/False

Table 1.3. Parameters of best-performing gradient boosting model

1.4.4 Dense neural network experiments

The dense neural network model has been tested on the problem of detection based on a single time step.

In order to test the FCNN model, we used the `Dense` and `Dropout` layers implementation from tensorflow.keras library. We investigated several configurations of the model by trying different values for the number of dense layers (`depth_dense`), the number of units in each layer (`units`), the activation function to use (`activation`), the amount of ℓ_2 regularization to apply to the kernel (`kernel_regularizer`) and the dropout rate to apply between the dense layers (`dropout`). During the training of the model, we used the `class_weight` parameter to compensate the unbalance of the dataset.

The model configuration which obtained the best results is composed by three fully-connected layers, with a dropout layer after each one of them. The first two dense layer have 512 units, while the third one has 256 units, and they all use the ReLU activation function and the ℓ_2 kernel regularization. After the three dense layers, there is a fourth dense layer with only 1 unit and a sigmoid activation function in order to output the prediction. Table 1.4 presents the parameters of the dense neural network model that performed better.

Parameter	**Value**
epochs	20
batch_size	32
depth_dense	4
units	512 / 256
activation	`'relu'`
kernel_regularizer	`l2(5e-2)`
dropout	0.4
class_weight	$\{0: \frac{N}{\texttt{num\_negative}}, 1: \frac{N}{\texttt{num\_positive}}\}$

Table 1.4. Parameters of best-performing dense neural network model

1.4.5 Convolutional neural network experiments

1.4.6 LSTM neural network experiments

1.4.7 Graph-based LSTM neural network experiments

1.4.8 Graph-based convolutional neural network experiments

1.5 Results

Notes

Topics

- Intro (big summary of the thesis)
    - Problem statement
    - What (GNN, LSTM, etc.)
- Background (theoretical explanation of models used and SOTA)
    - Machine learning methods
    - Neural Networks
    - Dense
    - LSTM
    - Conv
    - GNN
    - Seizure prediction
    - EEG
    - Functional connectivity network
    - State of the art
- Methods (description of all the work done and motivations, but no numbers)
    - Baseline
    - Graph / contribution
- Experiments/Implementation (technical description of all the work done with parameters and numbers)
    - Environment (software, server, framework)
    - Data + preprocessing

Architectures (params)

Results

- Conclusions

  Comments

  Future work

Project steps

- Baseline

  Detection

  — Random Forest

  — Gradient Boosting

  — SVM

  — Dense

  — LSTM

  — Convolution

  Prediction

  — LSTM

  — Convolution

  —— Stride $>$ 1 and bigger look_back

  —— Parameters search

- Graph Neural Networks

  GNN + LSTM

  GNN + Conv1D

  TODO Pooling

  — **TODO** Hierarchical (Rex Ying)

  — **TODO** Decimation (Bianchi)

- Baseline

  TODO Cross Validation on 3 seizures

  TODO Bayesian optimization

TODO Different functional connectivity nets

TODO Big data

TODO Structural connectivity (KNN)

Missing experiments: Detection with graph-based models, cross-validation on detection tasks

1.6 General considerations

1.6.1 IEEG plots

The ieeg plots give an overall idea of the type of data to deal with and of the shape of the patient's seizures. On 24 hours of recording, there are three seizures in the first 3 hours. The data is divided in clips of 1 hour each, containing 1800000 timestamps each (the signal is sampled at 500Hz). The three seizures found have a duration between 13000 and 15000 timestamps each (between 26 and 30 seconds).

The ieeg doesn't explicitly show the presence of a seizure; indeed the dynamic of the electrodes signals doesn't change in correspondence of the beginning of a seizure. For example, in the first seizure the signals dynamic remains pretty stable during the first half of the seizure and suddenly changes at the beginning of the second half of the seizure. This behaviour suggests that the seizure is not directly represented by the dynamic of the signals, but it is represented by the non-linear relations between the electrodes behaviours.

The ieeg plots show an interesting thing: in all the three seizures, there are two electrodes that are more dynamic exactly during the seizure. In the plots they are always represented with green colour.

- classic plot (with y values): electrodes signals positioned at -5000 and 2500.
- norm plot (y normalised): electrodes signals positioned at 8th position from the top and 33th position from the bottom.

1.7 Machine learning classic models

1.7.1 Experiments with classic methods

The three machine learning classic methods used to perform experiments are random forest, gradient boosting and support vector machine. All the three methods have been

implemented using scikit learns modules; for gradient boosting also xgboost library has been used. For all the experiment just portions containing seizures of the entire dataset have been used to train and to test data in order to speed up the process and to reduce the huge difference of availability of data between the two classes.

Classic methods have been used only for the detection task, since they are not powerful enough in order to deal with the prediction task, at least with this kind of data.

Parameters:

- random forest:

  number of estimators: 100

  maximum depth: 10

  both balanced and non-balanced class weight

- gradient boosting:

  number of estimators: 100

  maximum depth: 10

- svm:

  gamma: scale

  both weighted and non-weighted classes

In all the experiments, seizure 2 and 3 have been used as training set, while seizure 1 has been used as test set.

Maybe try to cross-validate by exchanging datasets for training and testing.

Results: All the experiments had almost the same results:

- Loss: around 0.15
- Accuracy: around 0.85
- Roc auc: around 0.65

Only the unbalanced random forest was able to get almost decent results:

- Loss: 0.11
- Accuracy: 0.85

- Roc auc: 0.73

The results are very bad and show that the classic methods are too weak to be able to solve this problem having to deal with such a complex and unbalanced dataset. Indeed, the area under the Roc curve is around 0.65 for the majority of the models, which means that the predictions are made almost in a random way.

Maybe try using some different metric (ex. F1)

The predictions have also been plotted in order to have a better idea of what was predicted wrong and where and also the plots show a non-logical distribution, even if in some cases (for example in the non-balanced experiment with the random forest) the predictions in the seizure area seem to match with the targets. Also in the gradient boosting experiment, the model seems to start predicting 1 (there is a seizure) in the same timestamps in which the signals in the ieeg start to be more dynamic (almost in the middle of the seizure 1).

1.8 Deep learning classic models

1.8.1 Dense neural network

A dense neural network have been built in order to deal with the seizure detection task. After some hyperparameters tuning, the final structure of the network was the following:

Parameters: These are the parameters used in the network's layers.

- activation: tanh
- kernel regularisation: L2(5e-4)
- class weight: {0: len(y_train) / n_negative, len(y_train / n_positive}

Structure: These are the layers used to build the network.

```
Dense(units: 512)
Dropout(0.5)
Dense(units: 512)
Dropout(0.5)
Dense(units: 256)
Dropout(0.5)
Dense(units: 1, activation: 'sigmoid')
```

In order to compile the model, binary cross-entropy has been used as loss and Adam as optimiser.

Initially the structure was composed by less layers and each one with less units (ex. 128, 156), but the model wasn't powerful enough so both the number of layers and units have been increased. The kernel regularisation and the dropout layers have been added in order to avoid overfitting on training data. For the activation of the dense layers, both tanh and ReLu have been tried, but for some reason tanh works way better. Since the two classes of the problem, that are "seizure" (1) and "not-seizure" (0), are very unbalanced due to the small presence of seizure timestamps with respect to the number of non-seizure timestamps, class weight has been used to train the network. The weight assigned to each class is inversely proportional to the number of timestamps of that class with respect to the total number of timestamps.

Initially, in order to preprocess the data, the MinMaxScaler from scikit-learn library was used, but then it was substituted by the StandardScaler because this one led to better performances. The data are also shuffled in order to help the network to generalise.

Initially, as for the machine learning classic methods, just portions containing seizures of the entire dataset have been used to train and to test data. In order to do some test, a modification of the initial dataset has been tried: the dataset has been modified so that each clip is trimmed ending with the end of the seizure and starting 100,000 timestamps before the end of the seizure. The intention of this test was to eliminate the useless bias generated by the timestamps after the seizure. However, after testing it, for some reason the results got a lot worse, so the old portions of the dataset have been used for the next experiments.

Results After all the tuning described above, the best results that the dense network got on the detection task were the following:

- Loss: 0.87
- Accuracy: 0.79
- Roc auc: 0.74

These results are not extremely good but they are ok considering the complexity of the problem. The dense network behaved in a similar way to the unbalanced random forest, looking at the results. Looking at the plots of the predictions, they don't seem to be very promising, but in order to better understand the behaviour of the prediction, also the running mean have been plotted and it explicitly shows that the network is able to understand where is the seizure.

A strange thing happens in the predictions: in the test prediction plot, there is a sort of a "hole" of zero predictions right after the end of the seizure.

1.8.2 LSTM neural network

I built an LSTM network in order to deal both with the detection and the prediction tasks.

For this network, I introduced subsampling to the data preprocessing. The subsampling function take as input the subsampling factor, which represent how many negative examples we want to keep with respect to the positive ones. In this way, we can consistently reduce the disparity in the amount of positive and negative examples, preserving all the positive ones. The subsampling was performed both for the detection and the prediction task. The subsampling factor was set to 2, so that for each positive samples there are two negative ones. The stride in the subsampling was set to 1 in order to keep all the (positive) samples.

Another function was used in order to generate the sequences of data and labels to give as input to the network. It takes the parameters "look_back", which is the length of the input sequence (how many samples do I look behind), and "target_steps_ahead", which represents how many steps ahead to predict (how many samples there are between the end of the input sequence and the future sample I want to predict). Given this two parameters, it creates the pairs input_sequence - target accordingly.

For the detection task, the parameter "target_steps_ahead" was set to 0, that is the target corresponding to the last sample of the input sequence. For the prediction task, obviously the parameter was set to some values higher than 0, as we want to predict the target of a sample in the future, that wasn't inside the input sequence.

Detection task For the detection task, after some hyperparameter search and tuning, the best result was given by these parameters (experiment 147):

```
epochs:        10
batch_size:    64
depth_lstm:    1
depth_dense:   2
units_lstm:    256
reg:            l2(5e-1)
activation:    relu
batch_norm:    True
dropout:        0.4
```

```
class_weight:  {0: 0.4682, 1: 3.0}
look_back:     100
stride:        1
predicted_timestamps:   1
target_steps_ahead:     0
subsampling_factor:     2
```

The network was build in this way:

```
LSTM(units: 256)
BatchNormalization()
Dropout(0.4)
Dense(units: 256)
BatchNormalization()
Dropout(0.4)
Dense(units: 1, activation: 'sigmoid')
```

The results on an input sequence of length 100 were:

- Loss: 0.2632
- Accuracy: 0.9059
- Roc auc: 0.9263

Prediction task For the prediction task, after some hyperparameter search and tuning, the best result was given by these parameters (experiment 20):

```
epochs:        10
batch_size:    64
depth_lstm:    1
depth_dense:   2
units_lstm:    256
reg:            l2(5e-1)
activation:    relu
batch_norm:    True
dropout:        0.4
class_weight:  {0: 1.1561, 1: 7.4074}
look_back:     200
stride:        1
predicted_timestamps:   1
target_steps_ahead:     2000
subsampling_factor:     2
```

The network was build in this way:

```
LSTM(units: 256)
BatchNormalization()
Dropout(0.4)
Dense(units: 256)
BatchNormalization()
Dropout(0.4)
Dense(units: 1, activation: 'sigmoid')
```

The results on an input sequence of length 100 and a target steps ahead of 2000 were:

- Loss: 0.3098
- Accuracy: 0.9454
- Roc auc: 0.9337

The network that performed better was the same for both the detection and prediction tasks, even with the same regularisation parameters. What changed was, of course, the distance of the sample predicted and the length of the input sequence. The LSTM, in both the situation, wasn't able to deal with more than 200 samples in the input sequence (the results were way worse with a higher value). For the prediction task, the network was able to successfully predict until 2000 steps ahead, which correspond to 4 seconds. Higher than that, the prediction became almost random.

1.8.3 Convolutional neural network

I built a CNN in order to deal both with the detection and the prediction tasks.

For this network, as for the LSTM, I introduced subsampling to the data preprocessing (see LSTM).The subsampling was performed both for the detection and the prediction task. The subsampling factor was set to 2, so that for each positive samples there are two negative ones. Another function was used in order to generate the sequences of data and labels to give as input to the network (see LSTM).

Detection task For the detection task, we used the same parameter we found for the LSTM, except for the look_back, that in the case of the convolutional can be much higher. The best result was given by these parameters (experiment 2):

```
epochs:         10
batch_size:     64
depth_conv:     5
depth_dense:    2
filters:         512
kernel_size:    5
reg:             l2(5e-1)
activation:     relu
batch_norm:     True
dropout:         0.4
class_weight:   {0: 1.1561, 1: 7.4074}
look_back:      1000
stride:         1
predicted_timestamps:    1
target_steps_ahead:      0
subsampling_factor:      2
```

Run again experiments for detection with CNN using less filters and smaller kernel size

The network was build in this way:

```
Conv1D(filters: 512)
MaxPooling1D()
Conv1D(filters: 512)
MaxPooling1D()
Conv1D(filters: 512)
MaxPooling1D()
Conv1D(filters: 512)
MaxPooling1D()
Conv1D(filters: 512)
MaxPooling1D()
Flatten()
Dense(units: 256)
BatchNormalization()
Dropout(0.4)
Dense(units: 1, activation: 'sigmoid')
```

The results on an input sequence of length 1000 were:

- Loss: 0.1550
- Accuracy: 0.9374

- Roc auc: 0.9870

Prediction task For the prediction task, I began with some hyperparameter search and tuning using the same parameters used for the LSTM as initial model. The best result was given by these parameters (experiment 29):

```
epochs:        10
batch_size:    64
depth_conv:    5
depth_dense:   2
filters:        256
kernel_size:   5
reg:            l2(5e-1)
activation:    relu
batch_norm:    True
dropout:        0.4
class_weight:  {0: 1.1560, 1: 7.4074}
look_back:     1000
stride:        1
predicted_timestamps:   1
target_steps_ahead:     2000
subsampling_factor:     2
```

The network was build in this way:

```
Conv1D(filters: 256)
MaxPooling1D()
Conv1D(filters: 256)
MaxPooling1D()
Conv1D(filters: 256)
MaxPooling1D()
Conv1D(filters: 256)
MaxPooling1D()
Conv1D(filters: 128)
MaxPooling1D()
Flatten()
Dense(units: 256)
BatchNormalization()
Dropout(0.4)
Dense(units: 1, activation: 'sigmoid')
```

The results on an input sequence of length 100 and a target steps ahead of 2000 were:

- Loss: 0.2598
- Accuracy: 0.9314
- Roc auc: 0.9274

For the prediction task, the network was able to successfully predict until 2000 steps ahead, which correspond to 4 seconds. Higher than that, the prediction became almost random.

CNN notes

Bibliography

[1] Python. URL https://www.python.org/. [Online; accessed 16-August-2019].

[2] Numpy. URL https://www.numpy.org/. [Online; accessed 16-August-2019].

[3] Pandas. URL https://pandas.pydata.org/. [Online; accessed 16-August-2019].

[4] matplotlib. URL https://matplotlib.org/. [Online; accessed 16-August-2019].

[5] scikit-learn. URL https://scikit-learn.org/. [Online; accessed 16-August-2019].

[6] Xgboost. URL https://xgboost.readthedocs.io/en/latest/. [Online; accessed 16-August-2019].

[7] Tensorflow. URL https://www.tensorflow.org/. [Online; accessed 16-August-2019].

[8] Keras. URL https://keras.io/. [Online; accessed 16-August-2019].

[9] Grattarola Daniele. Spektral. URL https://github.com/danielegrattarola/spektral. [Online; accessed 16-August-2019].