POLITECNICO
MILANO 1863

PROJECT REPORT

Multivariate timeseries classification using neural networks

SECOND CHALLENGE IN ARTIFICIAL NEURAL NETWORKS AND DEEP LEARNING

Authors: LUCAS FOUREST, ACHILLE GELLENS, SIMON LESOUEF

Academic year: 2022-2023

1. Introduction

The following is the report of our work on AN2DL's second challenge of the semester. The goal of the challenge was the classification of 6-dimensional timeseries belonging to 12 different classes using artificial neural networks. Namely, we first tried solving the problem using memory-based, recurrent models: standard and bi-directional LSTMs; and proceeded to using 1D-convolutional models, which turned out to perform the best.

We will try to describe the different steps we went through when solving the challenge, the different problems we encountered and the choices we made. There is also a point to be made about the things we did not consider during the homework but could represent leads both for improving upon the submitted solution and, hopefully, perform even better in future situations.

In addition to the attached archive, other pieces of our code are available on the repository: `https://github.com/achgls/multivariate-timeseries-classification.git`.

2. All about the Data

2.1. Summary

The data is made up of N=2429 series of temporal length T=36 and consisting of D=6 features, belonging to K=12 different classes.

It is often useful to know *what* the data is supposed to represent in order to handle it in the most relevant ways. Unfortunately, this is not the case here, so we will treat the data as arbitrary.

2.2. Imbalance issue

Similarly as with the first challenge, there is a strong class imbalance in the given data, with a size difference ratio as large as 23 for the two extreme classes. This will lead to poor hit-rate within the least-represented classes. However, when it comes to global accuracy (the default metric to rank submissions in the CodaLab challenge), it will only be undermined if the test set class distribution is different than this of the training. As it occurs, we assumed from our earlier submissions that the test set distribution seemed, indeed, somewhat similar. Consequently, we chose to not really address the imbalance issue this time, and focus on other aspects of the challenge. We note that there are many cases where it is effectively critical to properly detect the *rare* events, *e.g* frauds, diseases... in which case we would have to account for this issue, as described in the first report, *i.e* weighting loss function and creating balanced data pipelines with resampling methods paired with data augmentation (see 2.4).

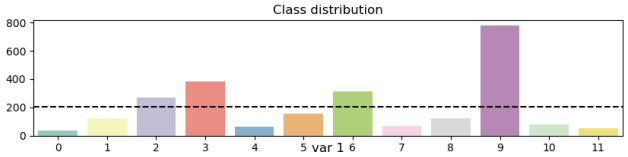

Figure 1: Class distribution in the given data and the average series for each variable and class

2.3. Data rescaling

Instead of min-max normalization or standard-scaling, we chose to use `scikit-learn`'s `RobustScaler` class which scales features based on their IQR, because it is, as its name suggests, robust to outlying values, which prevail in our dataset as shown in Figure 2.

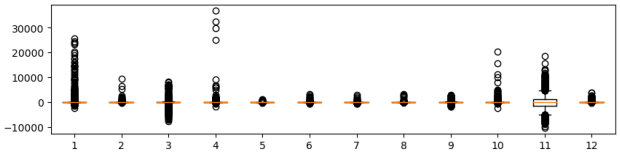

Figure 2: Boxplot of the third variable's values, outliers are the blacks dots, they spread so far out that we can't even see the IQR box and whiskers

It does not make sense to normalize the data dif-

ferently at each timestep. Instead, we want to scale each series in an homogeneous manner across their whole duration. Different variables are likely to take different magnitudes of value, though, so we want to scale each variable differently. We will thus fit the `RobustScaler` object to the $(NT \times D)$ matrix of the concatenated series.

2.4. Data augmentation

In an attempt to make up for the relatively small size of the dataset (especially with smaller classes) and to avoid overfitting, we might want to look into data augmentation applied to time series. Different techniques exist and can be found in [3], [8] and [5], among which the techniques pictured in Figure 3, extracted from [3].

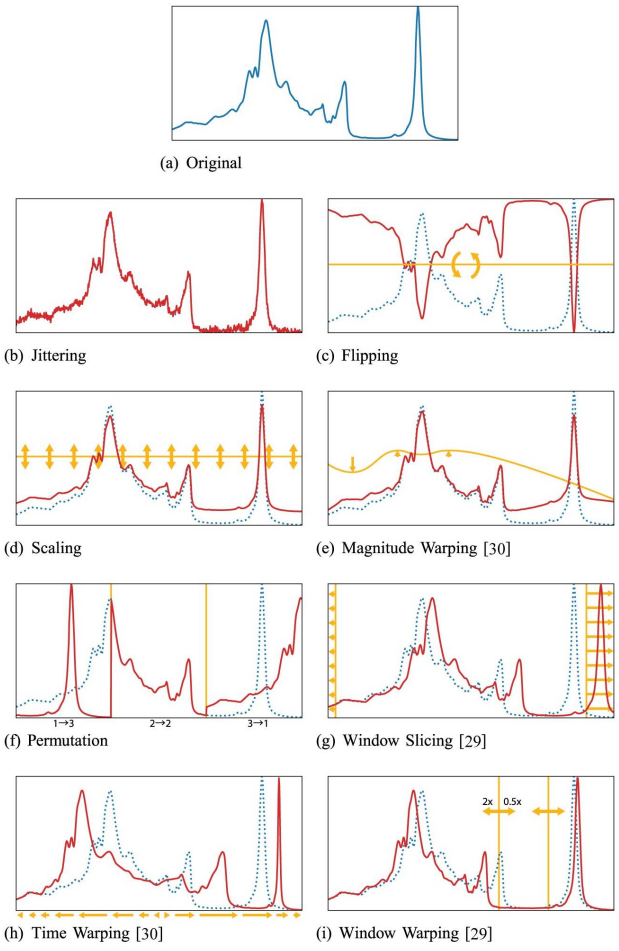

Figure 3: timeseries augmentation techniques

Luckily, there is the open-source `tsaug` [7] Python library that has timeseries augmentation techniques implemented and ready-to-use on `numpy` arrays. The repository [2] also has code snippets and use case examples for the different techniques as described in their related paper [3].

We got to implement some static data augmentation (*i.e* applied before training, after what the training data remains static) and witnessed better results in some cases. However, there are 2 problems: first of all, as we do not know the nature of the data, it is hard to know exactly which transformations are coherent and relevant; also, static augmentation will be less effective at reducing overfitting than real-time augmentation because the network will still be able, after enough epochs, to blindly *remember* the static variations of the data and increase accuracy on the training set while not increasing on validation.

3. Model architectures

3.1. Recurrent memory-based networks

The first intuition when it comes to sequential data is using networks that are able to retain information as they process along the sequence.

RNN - Recurrent Neural Networks are able to do just that by "looping" with themselves: their input at time t is a combination of the t-th input in the sequence and the output from the previous unit (the network at a given timestep/iteration).

LSTM - Long Short-Term Memory networks are the most famous extension to the basic idea of RNNs. They add trainable "gates" that allow to choose what the unit prefers to keep or forget from the previous unit's output. The mechanism behind LSTMs is thoroughly and well-explained in [4]. It yielded considerably better results than simple RNNs.

Bidirectional LSTMs - we tried to introduce bidirectionality in the network, keeping in mind that it doubles the number of parameters. This yielded slightly better results.

Others - there are a lot of variations on the recurrent architectures that stem from RNNs and LSTMs, that we did not get to try and implement. One of the most famous being Gated Recurrent Units (GRU). In [1], these networks are evaluated and compared to standard LSTMs.

3.2. Convolutional models

3.2.1 Why convolutional models ?

Although they are commonly known and used for image anaylsis, convolutional models can work just as well with timeseries. In fact, both types of data

are signals. Images are 2D signals whereas timeseries can be seen as 1D signals. We can exploit the autocorrelation between neighbouring timesteps in timeseries data the same way we exploit the spatial (2D) autocorrelation in neighbouring pixels of an image. As a result, convolutional models are both effective and efficient at capturing information from timeseries. Instead of using Conv2D layers, though, you use Conv1D layer so you have 2-dimensional kernels instead (length or lag, and depth).

3.2.2 Designing a Conv1D network using KerasTuner

We used KerasTuner to find a somewhat optimal set of hyperparameters for a small Conv1D architecture. The code for this tuning process is available on the corresponding notebook on our GitHub, where you will also find the architecture that retrieved the best results during the `BayesianOptimisation` process. Small *handmade* Conv1D models turned out to be the best-performing in the end.

3.2.3 Adaptation of famous architectures in 1D

Another idea we had was to consider using the efficiency of well-known architectures as we did in the previous challenge with famous CNNs structures and transfer learning for image features extraction. Thus, we developed a method to build a ResNet50 like network assembling "basic blocks" (see Figure 4) but using 1DConv keras layers, in order to fit our data.

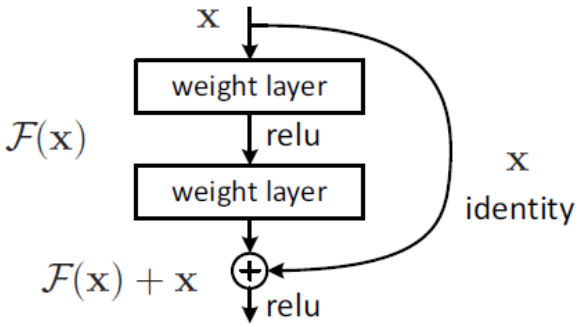

Figure 4: ResNet50 block principle

We did, of course, introduce some Dropout and Batch-Normalization layers to prevent overfitting and for faster convergence. Unfortunately it did not lead to better results than with more "simple and classic" architectures, despite having longer trainings using more time and memory resources. This might be due to the fact that, unlike the case of transfer learning for image classification, the network here is not pre-trained on a huge amount of data (which already makes it a very powerful feature extractor before even being used and completed with FC layers), and it is probably longer and more complicated to train such a model starting from scratch.

4. Leads for improvement

Unfortunately, there are a lot of things that we could not explore in the short time frame of the challenge.

Ensembling techniques - it was something we were sad not to experiment during the first challenge but were hoping to do so this time. Unfortunately, we were once again unable to, with the time we had.

Real-time epoch-wise data augmentation - as explained above, real-time data augmentation (*i.e* randomly augment the training data at the beginning of each epoch) would result in much better generalization as the network would have to learn the actual underlying process in order to improve its performance, as opposed to blindly remember the variations of the training set. Of course, as the data varies at each epoch, convergence will be slower, but it is almost certain that the difference between train and validation accuracy would grow slower than in the static augmentation case.

Kernel methods and feature-mapping - as suggested in the challenge's logbook by E. Lomurno, we could have experimented with feature augmentation, *e.g* using kernel methods to go from feature $[x_1, ...x_6]$ to $[x_1, ...x_6, x_1^2, x_1x_2, ...x_5x_6, x_6^2]$ for instance, which makes it easier to create non -linear separation boundaries.

Attention layers - as suggested by E. Lomurno in the logbook, we could have tried to incorporate attention layers in some ways. Attention is a very *hot* topic in Machine Learning and it is known to be quite consistently effective at improving performance.

Anomaly detection - as said above, the data seems to feature a large number of outlying values. We could have ran some sort of anomaly detection to identify abnormal timeseries as a mean to clean the data. The people behind `tsaug` package also implemented a library for this intent, `adtk` [6].

References

[1] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.

[2] Brian Kenji Iwana and Seiichi Uchida. Bilinear cnn models in tensorflow-keras. `https://scorrea92.medium.com/bilinear-cnn-models-in-tensorflow-keras-801121cc8c4d`, 2019.

[3] Brian Kenji Iwana and Seiichi Uchida. An empirical survey of data augmentation for time series classification with neural networks. *PLOS ONE*, 16(7):e0254841, jul 2021.

[4] Christopher Olah. Understanding lstm networks. http://colah.github.io/posts/2015-08-Understanding-LSTMs/, 2015.

[5] Edgar Talavera, Guillermo Iglesias, Ángel González-Prieto, Alberto Mozo, and Sandra Gómez-Canaval. Data augmentation techniques in time series domain: A survey and taxonomy, 2022.

[6] `tailaiw` and Roy Keyes. Anomaly detection toolkit (adtk) library open-source repository. `https://github.com/arundo/adtk`, 2019.

[7] `tailaiw` and Roy Keyes. tsaug library open-source repository. `https://github.com/arundo/tsaug`, 2020.

[8] Qingsong Wen, Liang Sun, Xiaomin Song, Jingkun Gao, Xue Wang, and Huan Xu. Time series data augmentation for deep learning: A survey. *CoRR*, abs/2002.12478, 2020.