ANNDL Challenge 2 Report

Alessio Facincani ⊗ Mousa Sondoqah ⊗ Mohanad Diab

Looking at the data

For the first approach, it is always best to look at the data we will have to work on. This is a time series classification problem, and we are provided with a time series composed of six features. The data is split into sequences with a window size of 36, and the windows have no overlapping, thus using a stride of the same size as the window. This windowing provides a data amount of 87444 samples, giving a total of 2429 windows, and each window is associated with one of the 12 classes. A visualization of the data is provided in Figure (1).

Dataset imbalance

From Figure (1) it can be seen that the given data is very unbalanced. In particular, the first class #0 has a ridiculous amount of data (34 data, just 1.4%) compared to the most present class #9 (777 data, almost 32%). Training models on this data provided bad results, especially in the small classes, often obtaining 0% accuracy on the hidden test set for some of those classes.
In these cases, the main approach to deal with dataset imbalance is to use the `class_weights` attribute in the fitting function, but in our case, it provided a significant drop in accuracy, so we discarded it and tried different approaches.

Making a network

For the hyperparameter optimization we used the `keras-tuner` package (script `Tuning.ipynb`). The tuner was programmed to test three possible model structures at once: Vanilla LSTM, Bidirectional LSTM, and 1D Convolutional. For each structure, different configurations are tested, in terms of the number of structure-specific layers, number of neurons per layer, number of dropout layers, number of pooling layers, and number of fully connected layers. (Note: we also tried to mix structures, like using LSTM layers together with Conv1D layers, but the results were plain bad, varying from incapable-to-learn to incapable-to-generalize, missing any sweet spot in the middle.)

The initial tuning run was performed with the Hyperband algorithm, to save time exploring the huge variety of combinations that these structures together potentially provide (based on its programming, there are almost 10 billion possible combinations of parameters, excluding the learning rate of logarithmic sampling!). The first tests proved the Conv1D structure to be the overall best performing, while LSTM and BiLSTM overfit quickly, so we discarded those and tested even more thoroughly the combinations for the Conv1D structure.

In the end, the model we chose turned out to be quite similar to the one seen in Lab 7 and used for the same kind of classification problem, which is a rather simple model, especially compared to the previous challenge - but this seems to be often the case with time-structured data. This is a sign that tuning hyperparameters might not be the only way to go to find a better model, but many other factors are in play. For this challenge, data preprocessing, such as scaling, and changing the stride had much more effect on the final accuracy than any other different model we could find using the tuner.

Scaling

Generally, data scaling is needed **so the data is well spread in the space and algorithms can learn better from it**. For this challenge, we had two variables to deal with to guarantee the best scaling method, the first is the **direction of scaling**, so we had to make the decision whether to scale by feature or by sequence or even by activity. The second decision we had to make was the **scaler type**, and we had plenty of options to choose from (MinMaxScaler, StandardScaler, RobustScaler, etc.). After deep data investigation and some trials, we chose to go with RobustScaler, scaling the data feature by feature. What the RobustScaler does is to remove the median and scale the data in the range between the 1st quartile and the 3rd quartile.

Why did RobustScaler work here? it’s known that RobustSscaler can handle the presence of the outliers in the data, and it can be seen in Figure (1) that the shows some abnormal jumps in some occasions, and identifying those extremes as outliers or no, this would still influence of the performance of the StandardScaler while computing the empirical mean and standard deviation. And since the median and the interquartile are used to compute the RobustScaler, this would explain the fair performance of the RobustScaler for this problem. Figure (2) shows the distribution of the different features after applying the RobustScaler.

Building Sequences with a smaller stride

The dataset was provided having the shape of `(2429, 36, 6)`, which represents the accumulation of the 87444 timestamps of a group of 36 sequences, each sequence is labeled with one defined class. The original structure of the given data has no overlapping within those sequences, which means that each sequence has a unique set of timestamps. Passing the original shape of the data into the model, the model showed a performance of around 67% of accuracy, so we thought that changing the structure of the data would help the model to better recognize the patterns for the different classes. This task was kind of challenging due to the shape of the data, so we chose to construct a dataframe to better handle the dataset exploiting the different features of such data structure. Next, we customize a sliding-window function to build sequences with different strides and window sizes, by passing the corresponding dataframe, however, the window size was always fixed to 36 in order to preserve the input shape to be compatible with the shape of the hidden test set.
Building sequences with a stride smaller than the window size would present overlaps in the consecutive sequences, such reframing of the data can improve the performance of the models if it’s used wisely, we dealt with it as another hyperparameter in the network, to avoid possible overfitting due to huge overlaps. After generating sequences with different strides [4,9,12,18,36], we ended up achieving the best performance building sequences with a stride of 12. A visualization to better describe the process of building sequences with smaller stride can be found in Figure (3).

Final model

After searching for the best configuration for our 1D TCN model, we had to perform the final training for the model, as shown in Figure (1). We trained the model then using `ReduceLROnPlateau` to better adapt for the learning rate during training. Wrapping everything up, we submitted this model to CodaLab obtaining an accuracy of 74.5% in both the first and the second phases. From the confusion matrix we could see that the detection of the breath class was the hardest point for all the models, obtaining only 20 % accuracy for that class. Class wish was the least number of data samples, at the end it achieved an accuracy of 53% with the help of the customized sequences, while class shine with few samples got 44% of accuracy. This shows that the amount of given data is not always a problem, probably the data provided for breath class was not representative enough for its class.

Failed attempts

Class weights

One of the effective ways to deal with imbalance datasets is to use class_weights argument in the training function so more attention will be given to those classes with less number of samples, unfortunately this approach showed horrible results for this task.

Hybrid models

Exploiting the advantages of different models can help in ending up with a structure which can perform well on the given data, in our case; we tried to build models which are composed of TCN (1D-Conv) and LSTM together, but such model overfitted on the data although we tried to better tune different layers together, so at the end we preferred to pass this approach.

Removing Outliers

Outliers are those data points that are significantly different from the rest of the dataset. They are often abnormal observations that skew the data distribution, and arise due to inconsistent data entry, or erroneous observations.

To ensure that the trained model generalizes well to the valid range of test inputs, it's important to detect and remove outliers. For this challenge we tried a couple of statistical and machine learning models to detect for those anomalies and remove them before feeding data into the model. Unfortunately such removal didn't seem to work at the end, this can be explained due to the possibility that the removed data still have information in which the model needs to work and they are not necessarily outliers.

Random Observations

Once we started playing with the challenge, we experienced a very weird pattern for the validation accuracy, after deep investigation we figured out that such an issue occurs while splitting for training-validation sets using the **`validation_split`** argument in **`model.fit`** function**.**

This problem was solved by simply using the **`train_test_split`** function which showed a significant improvement for the validation accuracy.

Appendix

Figures

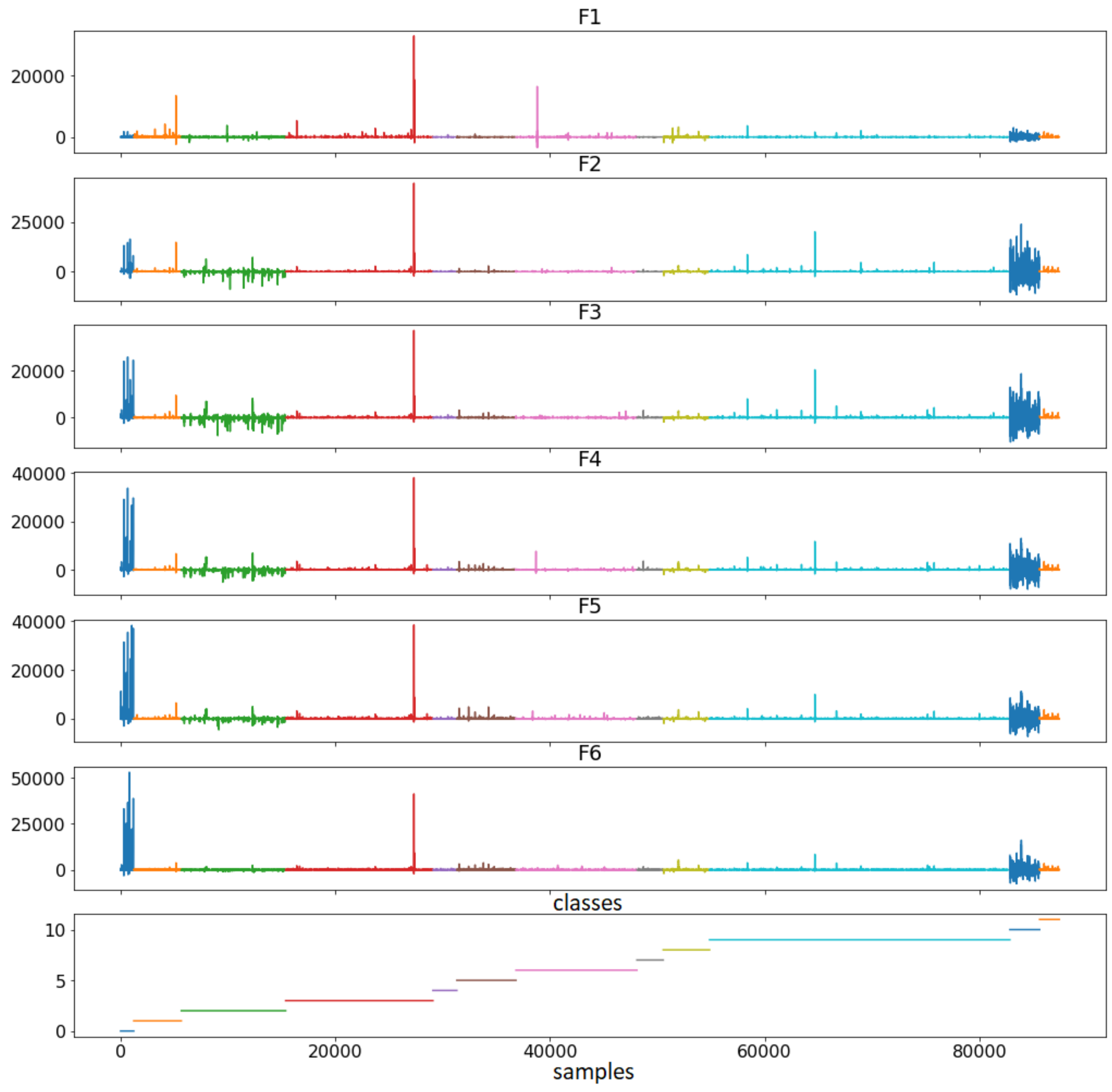

Figure 1: the six features plotted along the time axis, with different colors for each of the 12 classes.

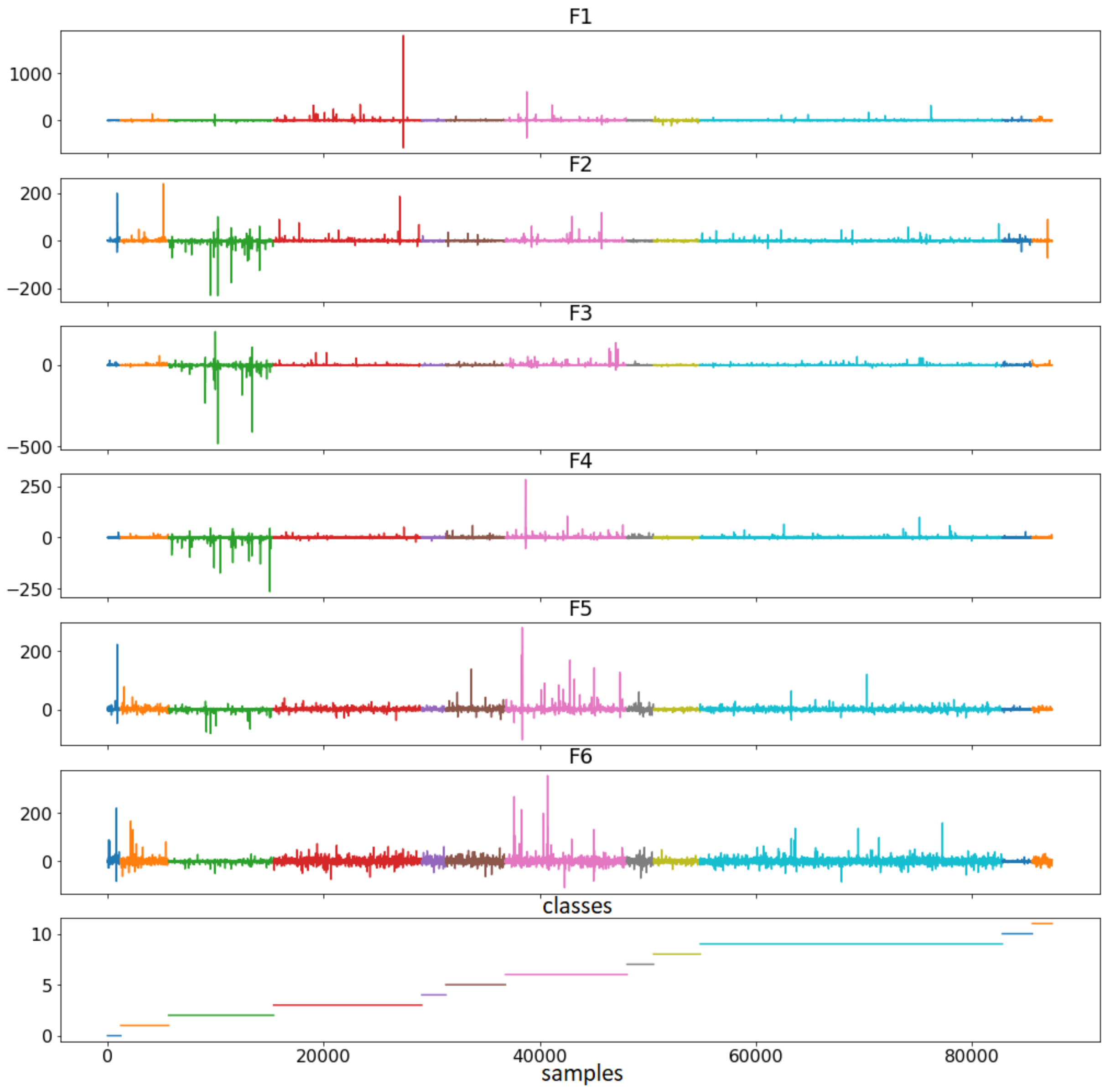

Figure 2: the six features rescaled with RobustScaler.

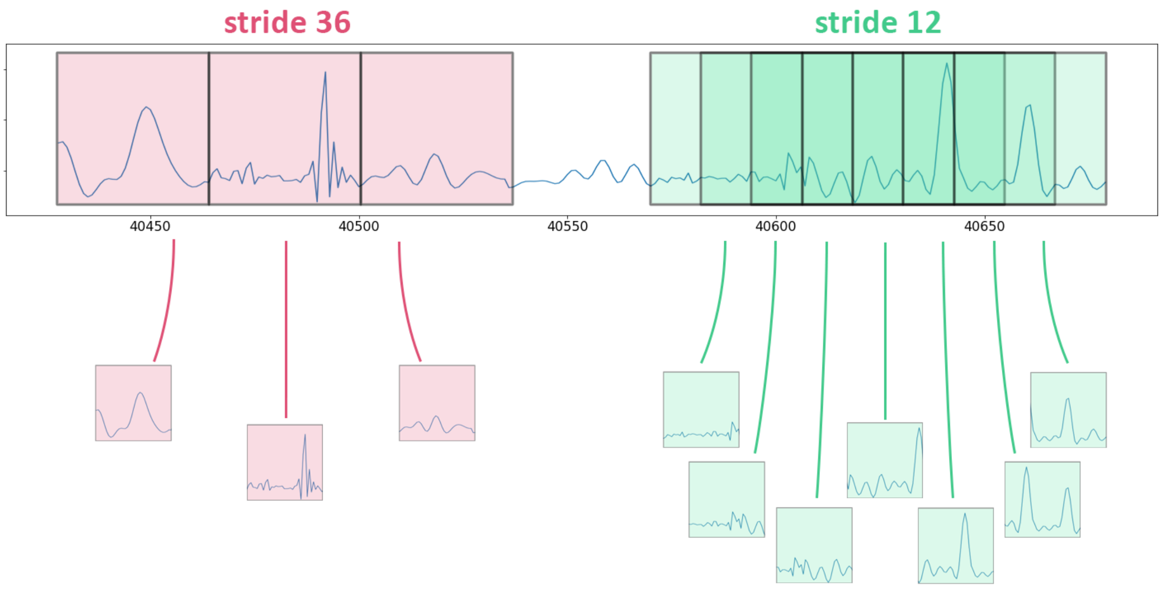

Figure 3: visualization of the stride change using the same window size of 36. From the same amount of samples, with a smaller stride we obtain more windows, thus more data to work with. The time series samples in the figure are taken from the first feature of class #6.

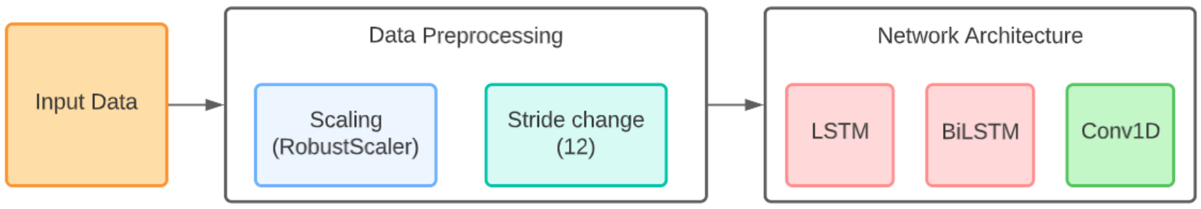

Figure 4: training pipeline overview, showing the main steps in data processing and model choosing The red cells models were tried but not used at the end due to their performance.

Figures list