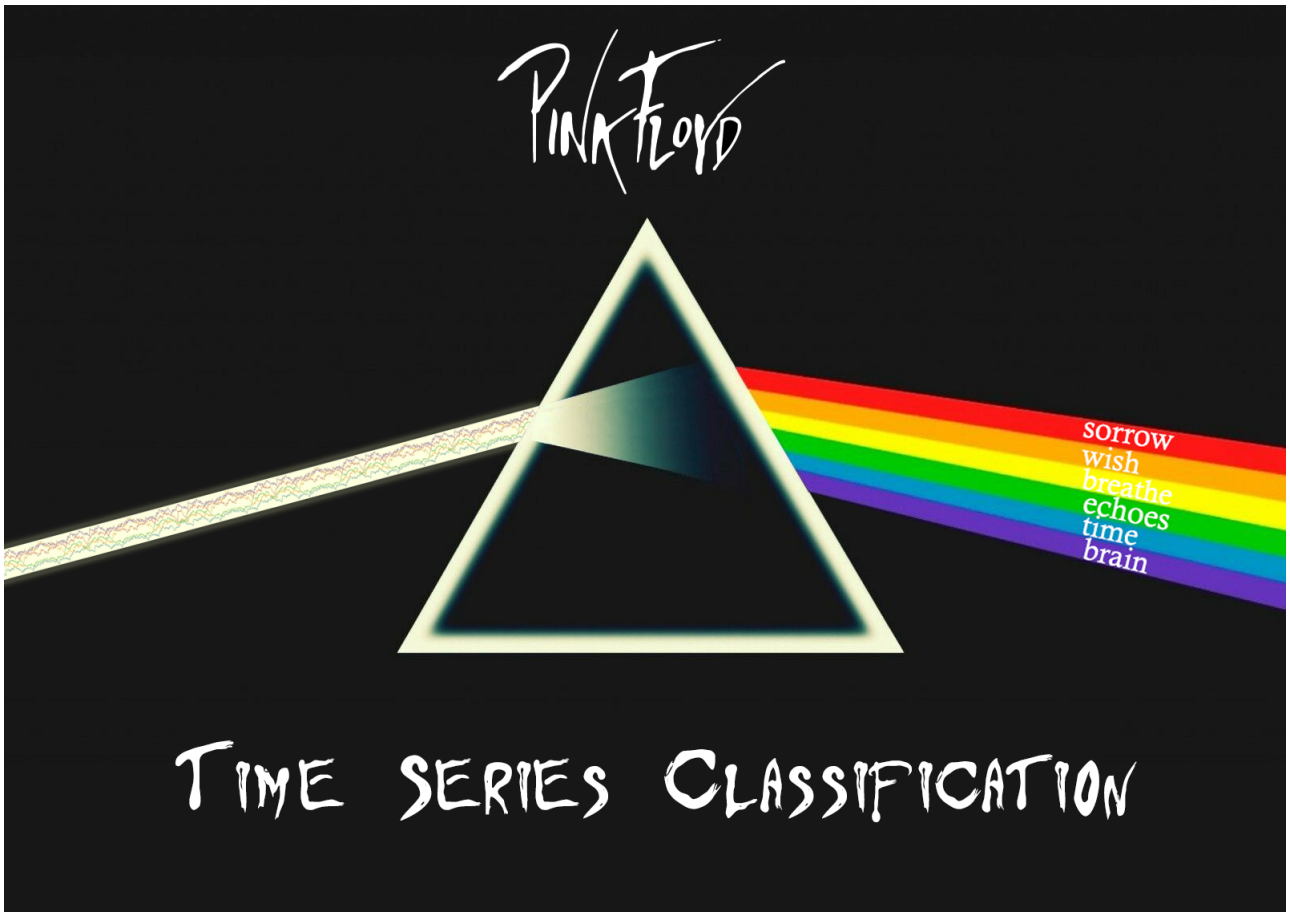
Project Report · Artificial Neural Networks and Deep Learning · Homework 2

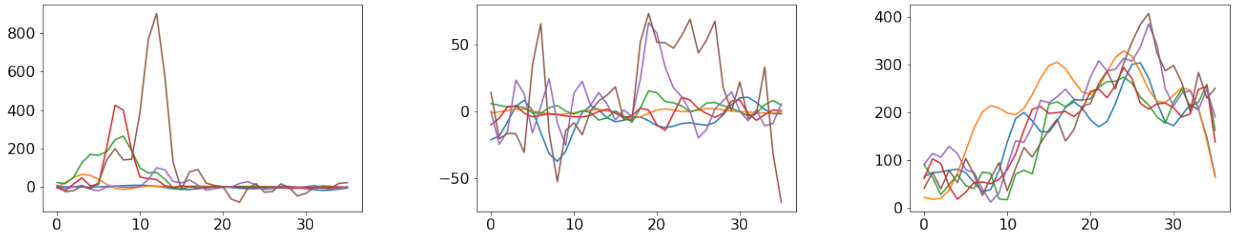


Cariati Leonardo [10588999]
Cutrupi Lorenzo [10629494]
Zardi Enrico [10659549]
Group: Pietort

Contents

1 Dataset study

The given dataset is a collection of time series samples of unknown source. Each sample contains 6 features and each one is a sequence of 36 real numbers. There are 12 classes and the dataset is imbalanced: the less represented class has 34 samples and the most represented one has 777 samples, while in total the samples are 2429. The goal is to use this dataset to train a neural network capable to classify this type of data.



Three samples, belonging to class 1, 5 and 9 respectively

The dataset used for testing the models after the submission seems to be proportionally imbalanced like the one we're using for training. Since the goal is to reach the highest accuracy, we considered more important to have a higher score in the most represented classes rather than the others. For this reason, applying class weights or oversampling to balance classes didn't seem to be a good choice. Effectively, applying these methods we ended up with more balanced models that had more or less the same accuracy between all the classes, but had worse overall accuracy both in the validation and the submission test set. For this reason we decided to give each sample the same importance, regardless of the class it belonged.

Having such an imbalanced dataset could lead us to pick a non-representative validation set if created improperly. That's the reason why we created a method that, for each class, added 80% of the original samples to the training set and 20% to the validation set. This granted our model to be validated with a dataset proportionally equal (classwise) to the training one, giving a consistent accuracy value for our model, which differed to the submission accuracy by less than 1%.

2 Model choice and preprocessing

We tested different approaches and different network types in order to understand how to better classify these time series. Firstly, we started with very simple networks composed by only a couple of layers each, trying LSTM's and Bidirectional LSTM's. Bidirectional seemed to work better than classic LSTM, but actually both were performing bad, probably because neural networks using these type of layers work better when studying long sequences, which was not our case. For this reason we decided to test simple Convolutional Neural Networks with 1D layers, and it revealed to be the best-fitting architecture for our classification problem. Moreover, we noticed that using a simpler architecture was faster to train and gave better results than a complex model with different convolutional layers, so we ended up using the following configuration for our task:

Layer (type)	Output Shape	Param #
Input (InputLayer)	[(None, 36, 6)]	0
conv1d_2 (Conv1D)	(None, 36, 256)	4864
layer_normalization_1 (Layer	(None, 36, 256)	512
max_pooling1d_1 (MaxPooling1	(None, 18, 256)	0
conv1d_3 (Conv1D)	(None, 18, 128)	98432
global_average_pooling1d_1 (	(None, 128)	0
dropout_2 (Dropout)	(None, 128)	0
dense_2 (Dense)	(None, 128)	16512
dropout_3 (Dropout)	(None, 128)	0
dense_3 (Dense)	(None, 12)	1548

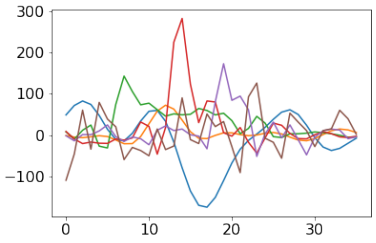
Total params: 121,868
Trainable params: 121,868
Non-trainable params: 0

This architecture allowed us to start with a decent 69% of submission accuracy, training it with the initial dataset without any modification.

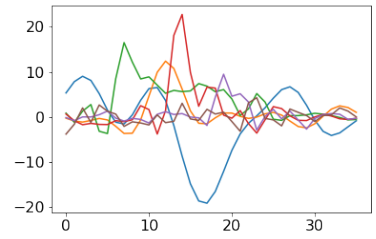
To confirm that in the problem we were facing a simpler architecture performed better, we tested different (and more complex) architectures found online, such as ResNet1D and Transformers for Time Series Classification, but both models gave us poor results and were very slow to train.

2.1 Normalization

Having a look at different samples, we figured out that some samples belonging to different classes had a totally different amplitude of values and there was a lot of variance in the data: some of them were showing values between 0 and 200 (with some peaks reaching a value of 1000), some others got values belonging to the interval [-40;0] and others were showing values extremely greater fluctuating between 0 and 2000. Moreover, looking at specific samples we noticed that feature values could vary a lot during time and were frequently showing peaks in the highest and lowest values. For example, in the image (a) showing the original sample, the red feature has for most of the time a value near to 0 and suddenly shows a peak of 300. This behaviour can be problematic for the training phase, so we decided to use normalization to overcome this problem.



(a) Original sample

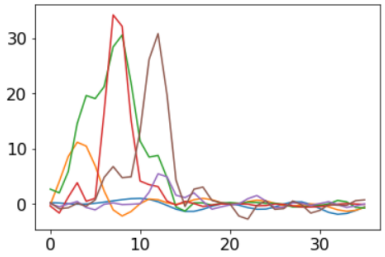


(b) Rescaled sample

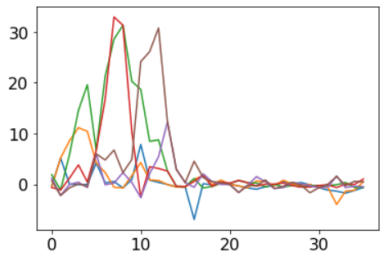
In order to deal with this lack of regularity, we tried different classical rescaling techniques like min-max scaling and max absolute value scaling, but that only led to disastrous results, worsening the accuracy. We also tried *StandardScaler* which gave good results, but in the end we decided to rescale the values of each class according to their respective median. To do so, we used the preprocessing function *RobustScaler* contained in the sklearn library, and that allowed us to train our model with a normalized dataset, performing better than before. Unfortunately, we got some issues when we tried to submit our model because we couldn't manage how to put our preprocessing in the *model.py* file, and that led us to waste a couple of submits. To overcome this problem, we decided to write the same preprocessing function manually, both in the notebook used to train the neural network and in the *model.py* preprocessing part, which allowed us to rescale the test set as we desired, resulting in an improvement of around 2% in the submission accuracy, compared to the one without rescaling.

2.2 Augmentation

Having the original dataset only 2429 samples and being completely unbalanced between all classes, we decided to use augmentation in order to improve our model's performances. The basic idea was to modify the original sequences in such a way that the newly created sequences had different values but the same shape as before.



(a) Original sample



(b) Augmented sample

Going deep into the web, we found different ways to perform augmentation on time series: originally, we included a *GaussianNoise* layer in our architecture which allowed us to add a random noise to each sample used to train the CNN and, since the model was able to generalize better, it improved its performances. After this promising result, we decided to change the way of performing augmentation, and we started using the

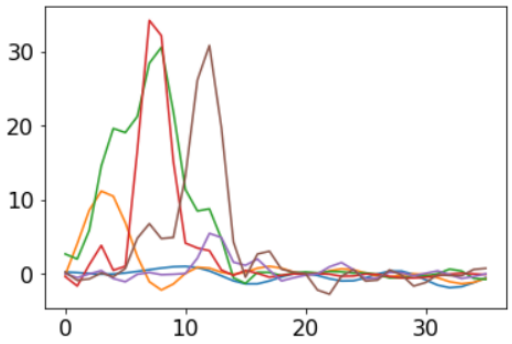
TSAUG library since it allowed us to apply a heavier augmentation on the original samples. Playing with the many parameters the library provides, we decided to use this configuration:

```
augmenter = (
                tsaug.AddNoise(scale=(0.2, 0.7)) @ 0.5
                 + tsaug.TimeWarp(40) @ 0.3
                 + tsaug.Pool(size=2) @ 0.5
                )
```

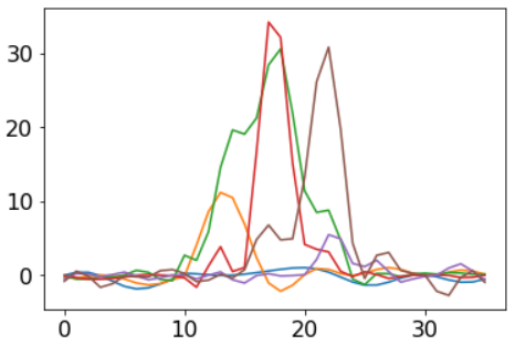
This combination, especially because of Time Warp, made the model very slow to train, not much in the sense of time required for each epoch, which was quite fast, but in the sense of epochs required to achieve an interesting training accuracy. Indeed, to have a training accuracy of around 85%, we had to train the model for, more or less, 600 epochs. Despite that, using such augmentation gave us the best result we ever achieved, with a validation accuracy of 75% and a submission accuracy of 74.73%.

2.3 Shifting window

Another particular technique used to augment time series data is to cut samples in different time intervals and shuffling those fragments in order to create a brand new sample to use in the training process. Since we do not know the nature of the dataset, we've chosen to apply a gentle transformation consisting in cutting our samples in 2 time windows (containing all the 6 features) and swapping them. Considering each class score in the codalab leaderboard, we opted to add more samples, with multiple cuts, of the worst scoring classes, and fewer samples of the already good scoring classes to our training dataset in order to try to increase further more the accuracy of our model. The training dataset size obtained this way increased dramatically with 5 times more samples, slowing a lot the training phase. The results we got by training our model with this new dataset were not particularly good, reaching a validation and a submission accuracy almost identical to the previous one, so we opted to remain with the original dataset augmented with the *TSAUG* library as it was faster to train with. We even tried to merge different samples of the same classes by taking time windows of different samples and combining them, but as before it only led to a slower training phase for identical performances, so it was not worth the effort.



(a) Original sample



(b) Sample shifted of 10 units

3 Final phase

We decided to submit the 3 models that performed better in the development stage, and they maintained more or less the same results as before, all of them gaining around 74% accuracy. One thing to note is that the model which received the highest score in the first phase is the one with the lowest score in the second phase, meaning that the heavier augmentation didn't actually improve the overall accuracy compared to the one with only noise. Maybe, a more meticulous study of the augmentation parameters provided by *TSAUG* library could have led us to find a better combination of values that would have increased performances.

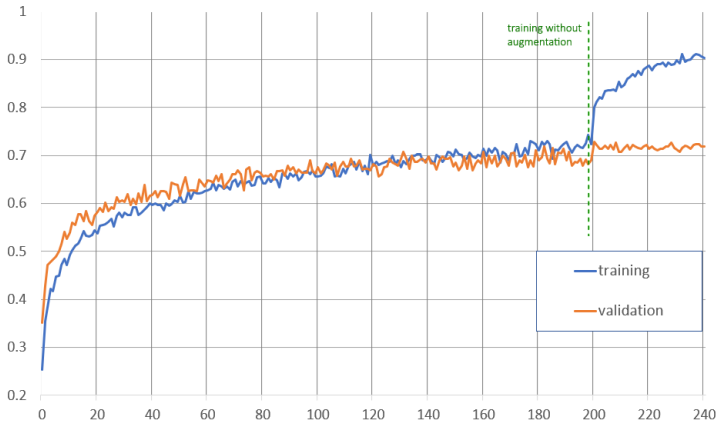


Figure 4: Accuracy of the best model.

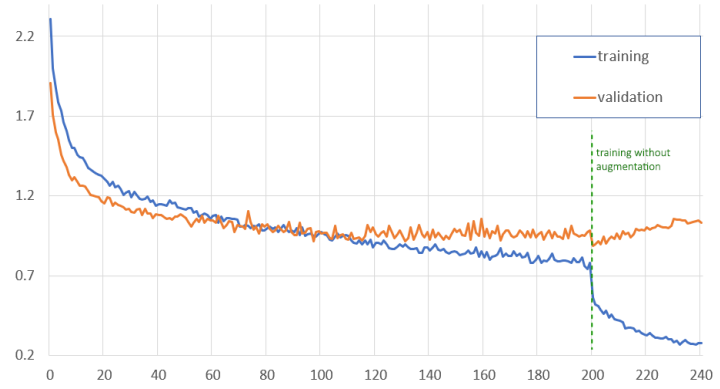


Figure 5: Loss of the best model.

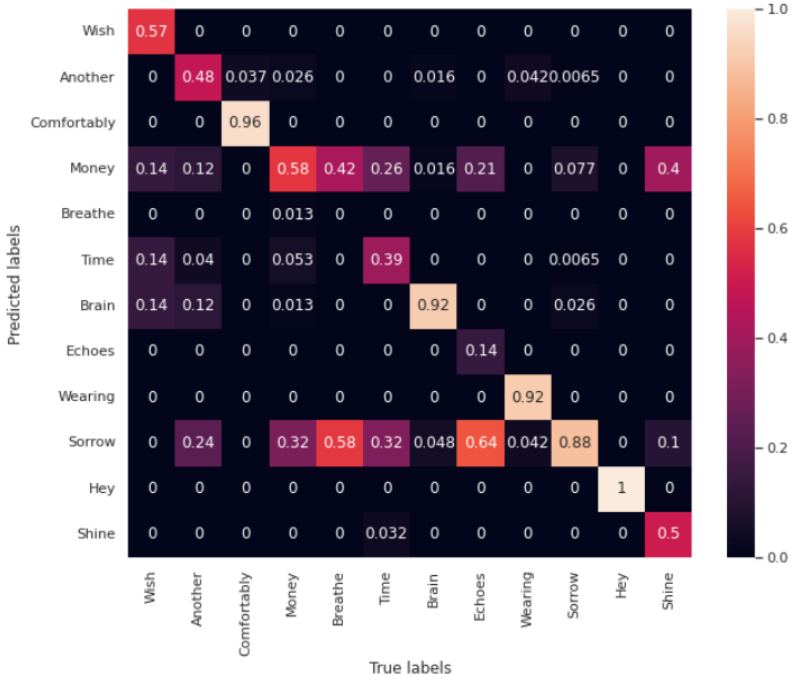


Figure 6: Confusion matrix of the best model.