REPORT of the 2nd Challenge of ANN2DL
Team Member: **Bianchi Emanuele, Junchong Huang, Manenti Carlo**

- **INTRODUCTION**

The second homework of 'Artificial Neural Network to Deep Learning' (ANN2DL) focused on classifying time series. We were provided with a dataset composed of 2949 samples, each made of 36 time points for all of its 6 features. Unfortunately, we were not given any additional information about the meaning of the feature or the time interval between each time point, except for the name of the classes. At a closer look we noticed how the classes were just the first word of the top songs from Pink Floyd. This clue, being the only information at our disposal, led us to expect a challenge about song recognition. Given a few seconds (36 to be precise) from a Pink Floyd song we needed to assign it the corresponding track.

- **EXPLORATIVE ANALYSIS**

Our first approach was trying to get a deep understanding of the data. At first glance, the dataset was unbalanced; The song 'Sorrow' took one third of the entire dataset with its 777 samples. Moreover, we could not help but notice that features' data spanned with a wide range of values, with some features (like feature 2) ranging from -11585 up to 44394. This could have been a problem during training, so we decided to try to preprocess our data accordingly. Next, we looked for class specific patterns. We observed the trajectory through time of each feature for all the samples of a given song. To aggregate the data for each time point we simply took the mean; although the mean is sensitive to outliers, it could still provide a general idea of the behavior of the features. As expected, the combination of all the six features defined a class specific pattern. Surprisingly, some features patterns seemed more important than others to assign the sample to its song. An example of this is the behavior of feature 1 for the song 'Wish You Were Here' which was flatlined, whereas the same feature in 'Another Brick in The Wall' followed a peculiar pattern.

- **PREPROCESSING**

Normalization vs Standardization vs Raw data

Preprocessing is fundamental even if deep learning can solve end-to-end problems and learn to process only useful data by itself. With proper scaling and outlier removal, the model can learn better and much faster. First, given that the dataset is unbalanced, we performed a stratified split to maintain the same unbalanced composition in both train (80% of the total data) and validation (20%). Then, to assess the best preprocessing approach we compared model accuracy of models trained on normalized and standardization data, while using raw data as baseline. Interestingly, standardization took the upper hand on normalization for both Long Short-Term Memory (LSTM) and 1D Convolutional Neural Networks (1DCNN). Probably standardization preserved data distribution and distances between points in the feature space, thus retaining meaningful information which could have been exploited by the model. Surprisingly, LSTM performances were far better when no preprocessing was performed. It's hard to explain theoretically and maybe that's linked to the peculiarity of the data.

Outliers' removal

Due to an almost complete lack of context for the data, we were not sure if outliers were useful or simply noisy values that should have been removed. So, we wanted to test if "outliers" were meaningful to classify the samples. Hence, we defined as outliers all the samples which had even a single absolute value of Z-index greater than 3. We computed the Z-index relying only on the training set and removed any sample containing at least one outlier. Removing a sample for only a single outlier could be seen as an extreme approach but is necessary to keep a homogenous input shape among all the samples. Finally, we ended up with 1789 samples for training and 456 for validation, losing only 184 samples at total. The validation accuracy of our 1DCNN baseline seemed slightly higher than on raw data but is probably due to a smaller and probably simpler validation set. Furthermore, it is likely that models can learn to handle or even to ignore such extreme values. So, we opted to not remove outliers also because we do not know if they are not representative of our data.

Optimal window size

Up to this point we always worked with a window size of 36 time points, but maybe we could find a better one. Since we needed to classify each provided sample, we thought that reducing the window size is the only feasible approach that would be made on the hidden test. Otherwise, assembling as little as 2 samples, we would have needed to give two labels and then find a way to decide which samples belong to which class. So,

we tried to reduce the window size to 20 and then 15 time points, but it did not yield any improvement. In the end, we decided to keep the original window size.

Feature Selection
At this point we tried something a little more radical. We wanted to know what features mattered the most to differentiate the classes. To do so, we used a simple correlation heatmap which showed us how features correlated with each class. Sadly, no strong correlation was present between classes and any single feature. Then we tried to correlate features with each other to see if some were so similar that we could just remove one of the two. Interestingly, features 6 and 5 were the most correlated features. In the end, removing feature 5 led to no major improvement. We also tested all the possible combinations of 5 features and compared them to the baseline, but due to high variability in the results, no combinations of 5 features can be said to be better than the full model.
A better idea could be to perform this process multiple times, maybe also leveraging k-fold cross validation to make it even more robust

Dimensionality reduction
Playing with features, another idea that came to mind was to exploit PCA, thus reducing the number of features while retaining the most information. The best result was a 34% accuracy, achieved by using all 6 principal components. In the end, we decided to also discard this kind of preprocessing.

- **1D-ResNet and 1D-VGG**

Inspired by the first homework, we wanted to adapt VGG and ResNet architectures to time series classification by changing 2D Convolutional layers into 1D convolutions (Conv1D). Unfortunately, we did not see any improvement over the baseline 1D-CNN. This could be due to lack of proper fine tuning (since we are using an untrained model), which was an essential step to achieve high accuracy levels in the first homework.

- **A FULLY TUNABLE MODEL**

Given that no preprocessing yielded a decent result, we focused our attention on model architectures. Having no information on the data we thought that tunable models could provide a good solution by exploring a vast search space and eventually picking the best hyper-parameters. Starting from the baseline 1D-CNN, we tried to tune simple parameters like the number of filters for Conv1D and the dropout rate using a random search. This simple test resulted in an increase of around 3% in the validation accuracy. Next, we went fully tunable by also making the number of Conv1D, Dense and Dropout layers, with their respective parameters and activation functions, tunable parameters. After 20 search trials the best model was composed of 6 Conv1D, a Global Average Pooling layer and 9 Dense layers of which 3 also had a Dropout layer. This new fully tunable model set a new high of validation accuracy.

Can we do more?
At this point we decided to also introduce skipped connections and regularization in the form of an elastic net for Dense layers. With this new element we hoped to improve the generalization capabilities of the model. However, we needed to define a shared number of filters for all the Conv1D layers to implement skipped connections, thus limiting the search space of hyperparameters. In the end, this new fully tunable model turned out to be surprisingly good, reaching more than 70% accuracy on validation and an astonishing 72.95% on the final hidden test. We also tried to explore extensively the search space performing up to 400 searches to tune the model, but they never improved upon the previous result.

- **TRANSFORMERS**

From the beginning
Transformers are a type of deep learning model that have achieved state-of-the-art results in various natural language processing tasks, such as language translation and text classification, but they have also recently been used for tasks in computer vision. Transformers are based on the idea of self-attention, which allows the model to directly consider relationships between input tokens without the need for recurrence or convolutions. Due to the unknown origin of the data, we assumed that they were composed of sequences of already embedded words (songs). Given that transformers achieved remarkable results in natural language processing tasks, we decided to test this architecture on our data. We started by using a basic model: a transformer encoder with no additional embeddings or positional encoding. However, the results were not satisfactory, with

an accuracy of only 50% at best on the validation set. Next, we attempted to improve the results by adding positional embeddings. We initially tried using some rough learned embeddings that we had written ourselves, but these did not yield reliable results. Therefore, we decided to use sinusoidal positional encoding as described in the original transformer paper*. Furthermore, we attempted to tune the model parameters, such as the number of attention heads, the embedding size, and the feedforward network layer size, eventually we also tried stacking multiple encoders on top of each other. Besides our best efforts, the accuracy on the validation set remained at best around 60%.

A different approach

The poor performance indicated that our hypothesis was wrong, and that the data is likely a normal time series, for which transformers are generally not the first choice due to their lack of explicit support for temporal dependencies. As a result, we decided to focus on more standard architectures for this type of data, such as LSTM and 1D-CNN. These models are better suited to handling sequential data and may be more effective at capturing the underlying patterns in the data. We implemented a Bidirectional LSTM (BiLSTM) and were able to achieve decent results, with an accuracy of around 65% on validation. By stacking multiple layers, we were able to reach 70% always on validation. However, when our model was evaluated on the hidden test set, it saw a slight decrease in performance, resulting in test accuracy of around 68%. Next, we decided to try stacking different architectures together. Our first experiment involved using a BiLSTM network to process the input data, thus obtaining a full sequence of output. We then fed this output sequence into a transformer encoder, as described previously. The results of this approach were encouraging, with an accuracy of around 73% on the validation set, compared to an accuracy of around 68% for BiLSTM alone.

Going deeper

Pushed by the promising results we decided to further extend the model architecture adding a 1D-CNN on top of it. The final architecture of our model consisted of a 1D-CNN followed by a BiLSTM and a transformer encoder, in that order, with skip connections layer between them. This modification resulted in an improved accuracy on the validation set, with a peak of 79% achieved ensembling two similar models with different hyperparameters.

Our best model

We found that the best individual model with this architecture was composed of 5 CNN layers with filter sizes ranging from 256 to 16, followed by two stacked BiLSTM layers and two stacked transformer encoders. When we evaluated this model on the final hidden test set, the accuracy dropped to 73%, which was in line with the performance of our best 1D-CNN model. We would have liked to combine these two approaches by using a more complex CNN block, but at that point we had exhausted both our available submissions and time, so we were unable to make further changes to the model.

- **LET'S GET CREATIVE**

Looking for creative approaches we stumbled upon recurrence plots for times series**. The idea is to generate recurrence plots for each sample and use it to train an image classification model, thus forcing the task into an image classification problem. This would also enable us to leverage our knowledge and models from the previous challenge. We used a simple CNN trained on RGB recurrence plot generated by only the first feature or from all 6 features. In both cases the results were lackluster. One problem could have been the model used, while another could be the way we generate the recurrence plot.

- **CONCLUSIONS**

This challenge enabled us to refine our knowledge of deep learning by applying ourselves to a fully data driven task, where we could not exploit any context information from the data. Considering the time at our disposal we were quite happy with the wide array of approaches explored for both preprocessing and model architectures. In the end, even if many approaches failed to deliver decent results, they were still useful in directing our efforts towards useful approaches and eventually defining our best model.

Reference:

* Attention Is all you need (https://arxiv.org/abs/1706.03762)
** Timage (https://link.springer.com/chapter/10.1007/978-3-030-30490-4_36)