Artificial Neural Networks & Deep Learning

Politecnico di Milano

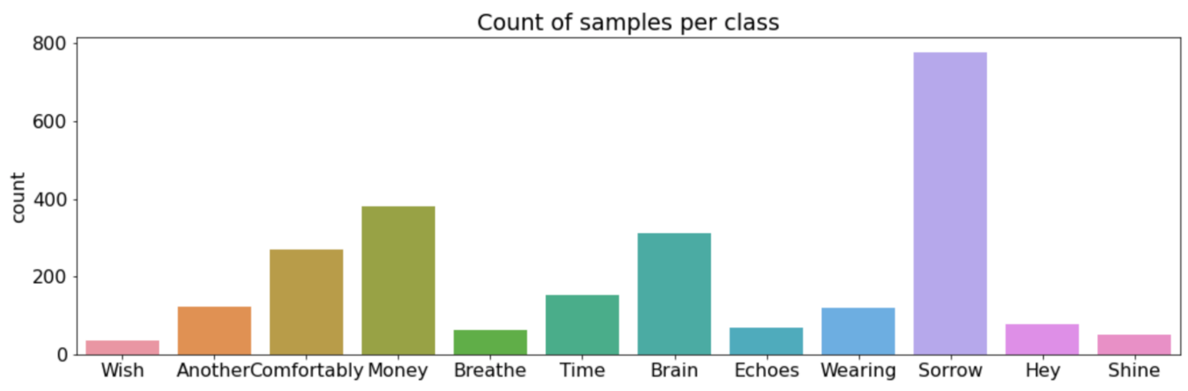
Challenge 2

Valeria Panté (10712755), Luca Tombesi (10865393), Adriana Vella (10864553)

The assigned task is the multi-class classification of time-series data. The paper aims to analyze the dataset, the data pre-processing, the choices and the steps made for the development of the classification model.

1. Dataset

The dataset assigned is composed of 2429 samples belonging to 12 different classes, labeled as *Wish*, *Another*, *Comfortably*, *Money*, *Breathe*, *Time*, *Brain*, *Echoes*, *Wearing*, *Sorrow*, *Hey*, *Shine*. The size of each sample is 36x6, where the first dimension is the window size and the second is the number of features. All the samples are not equally distributed among all the classes, in fact the number of samples for each label is the following:



To avoid overfitting during the training of the model, we chose to split the dataset in two subsets: training and validation. We started with a validation split of 10%, but we noticed that we improved our performances by increasing the percentage up to 20%. A further increase did not result in an additional improvement.

We tried to balance the distribution of samples among the classes by giving a similar number of samples for each class in the training phase, but it wasn't effective because the classes are too unbalanced. We attempted to use several other methods that are discussed below.

2. Preprocessing

In order to improve the quality of the dataset for training, we first shuffled the data as we noticed that in the imported dataset the samples were ordered by classes. This step improved our initial classifier by 5%.

Then we normalized the data. For this purpose we tried different methods (Robust Scaling, MinMax Scaling, normalizing the values in the range [-1,1]), but the one performing better was the Standard Scaling. Thus, we normalized the data feature by feature, subtracting the mean and dividing by the standard deviation. Also the scaling improved our model by another 5%.

In this phase we also tried the following techniques, which did not improve our performances:

- Applying Conditional GAN to generate artificial samples of the classes with smaller cardinalities. We developed a generator and decoder based on Convolutional 1D layers as building blocks of a Conditional GAN that was used in two different ways; first, we generated samples only for the classes with a cardinality less than 150, secondly we did the same just for the classes performing poorly in the test set on CodaLab (*Wish*, *Breathe, Shine*).
- Adding noise to the input to avoid overfitting of the classes with the biggest number of samples. We did so using two techniques: first we tried to simply add some noise taken from a $N(0,1)$ distribution, alternatively we used the *tsaug* library to apply the AddNoise component in order to add random noise up to 1-5% with 50% probability.

These two last attempts decreased the performances, hence they were not used in the final model.

3. Model development

In the first phase of development, we tried to understand which building block best fitted the classification problem at hand, thus we developed some custom RNN, each one based on a different type of recurrent layer: LSTM, BiLSTM, GRU and Conv1D. From this analysis, we opted for the last one as it performed sensitively better.

The first complete network that we developed reached 65% of accuracy of test in CodaLab and was composed of 3 layers of Conv1D with filter size of respectively 8, 5, 3 and kernel size of 128, 256, 128. We ended up with these hyperparameters after several combinations of attempts. After 3 layers, there was a Global Average Pooling layer and only one final Dense Layer with softmax and 12 neurons. We tried to insert more Dense layers with different numbers of neurons and dropouts, but this only decreased the performance.

Then, to upgrade the model, we tried to emulate a ResNet by stacking 3 blocks of 3 Conv1D and BatchNormalization layers each and adding skip connections at the end of each block, maintaining the same final layers. To create the skip connection we first expanded the channels of the output of the previous block to match the size of the current one, then batch-normalized it and finally the two were summed together. This structure reached 70% on the CodaLab test. Removing the BatchNormalization layers inside the blocks gave us a further 1%.

The final result was obtained by adding attention to the previous model. Initially, we inserted a simple Attention layer before the GAP layer, then we improved by using a MultiHead Attention layer with 10 heads and a dimension of 32 for each of them, which proved to be the best configuration of parameters. The attention was performed between the output of the Recurrent Network and the input, whose channels were expanded to match the size. This model had a 72.67% accuracy on the CodaLab test.

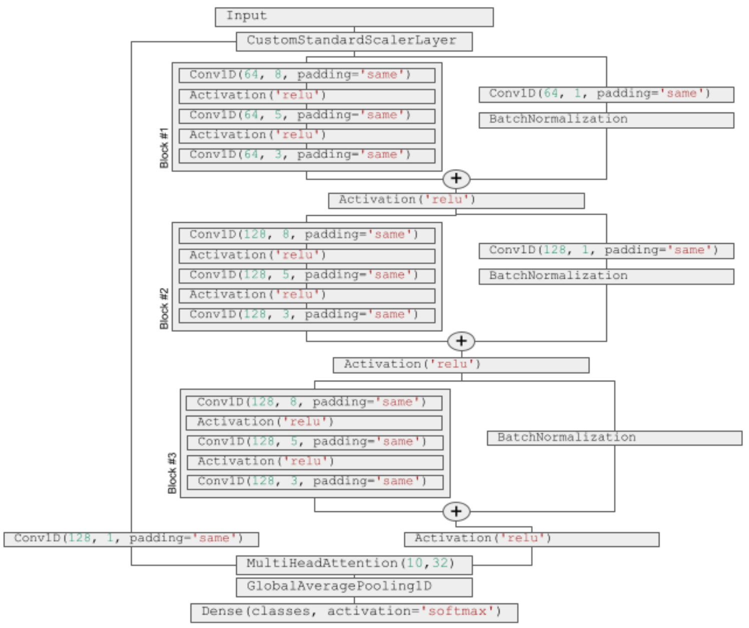
Beside the previous described steps, we attempted several other implementations that resulted in failures:

- Adding LSTM or BiLSTM layers after the Conv1D blocks
- Adding LSTM or BiLSTM layers in parallel to the Conv1D blocks and merging them together by averaging them
- Use an ensemble method, in particular bagging, by bootstrapping to generate 10 folds and fitting 10 models with the same architecture (the best one described above), but each with a different fold of data. The final classification was achieved by majority voting of the models.

- By looking at the distribution of the data per feature, we noticed that some features had similar distributions, so we thought to select only a subset of features. To do so, we did a brute-force approach by fitting the model by all the possible combinations of features in sets of 3,4,5. We compared the performances of each one and we found out that the one containing the features (0,2,4,5) had the best results, but this was not matched in the CodaLab test.
- We changed the window size by splitting the original sample (36x6) into 3 samples (12x6). The model was applied to each of the new samples and the final classification was achieved by majority voting. We chose 3 as the number of splits to have an odd number for tie-breaking between two classes.
- As a final attempt, to solve the unbalanced dataset issue in the learning phase, we tried to add class weights during the model fitting to give more importance to some classes over other ones. We did this in two ways; first we computed the frequencies of samples per class and to find the final weight we did a weighted average. In the other attempt, we increased only the weights of the classes with low accuracy on the CodaLab test (*Wish*, *Breathe, Shine*).

As a final note, we tuned the following hyperparameters for the actual fitting of the model:
- Patience, which was initially set to 20 and then increased to 50, allowing the model to reach better optima; we also increased it further but it didn't change the outcome.
- Batch size, which at the end was set to 64 as we noticed that higher values resulted in worse performances.
- For the learning rate we opted for an adaptive one, with minimum value of 10^{-5} as lower values did not show a better training.

Here, a visual representation of the final structure of the implemented model:


Input
CustomStandardScalerLayer
Block #1
Conv1D(64, 8, padding='same')
Activation('relu')
Conv1D(64, 5, padding='same')
Activation('relu')
Conv1D(64, 3, padding='same')
Conv1D(64, 1, padding='same')
BatchNormalization
Activation('relu')
Block #2
Conv1D(128, 8, padding='same')
Activation('relu')
Conv1D(128, 5, padding='same')
Activation('relu')
Conv1D(128, 3, padding='same')
Conv1D(128, 1, padding='same')
BatchNormalization
Activation('relu')
Block #3
Conv1D(128, 8, padding='same')
Activation('relu')
Conv1D(128, 5, padding='same')
Activation('relu')
Conv1D(128, 3, padding='same')
BatchNormalization
Conv1D(128, 1, padding='same')
Activation('relu')
MultiHeadAttention(10,32)
GlobalAveragePooling1D
Dense(classes, activation='softmax')

```
==================================================================================================
Total params: 672,972
Trainable params: 672,332
Non-trainable params: 640
__________________________________________________________________________________________________
```