Artificial Neural Network and Deep Learning

Homework 2: Time series classification

Andrea Seghetto, Francesco Pastore, Oswaldo Morales

December 23, 2022

Introduction

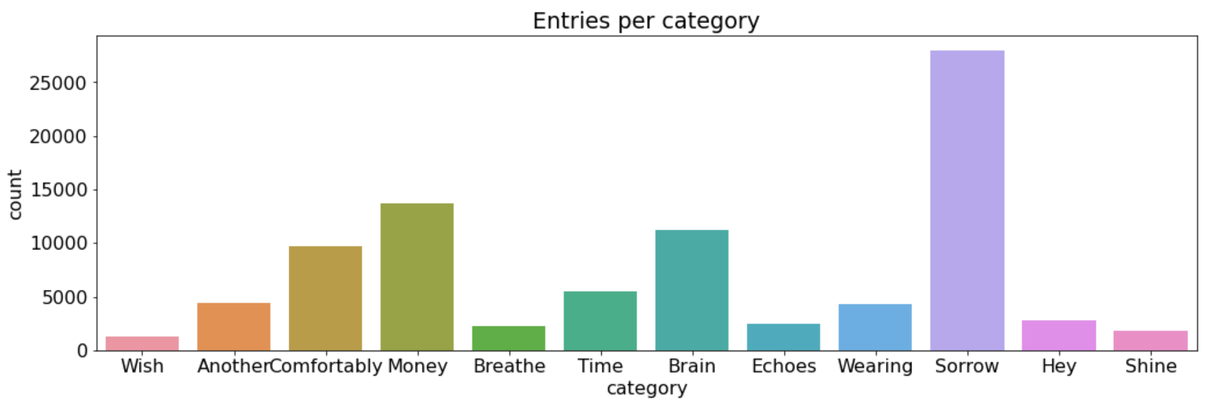
Our goal for the challenge was to design a neural network for time series classification. The database provided contains different time series made up of six features already divided into windows of 36 measurements (window size), which we should classify among the 12 categories assigned.

In the attached zip folder, you can find a custom "dataset.zip" file and nine Jupyter notebooks: one for preprocessing, one for the final ensemble model, and one for each neural network explained below.

At this link there is a shared folder on Google Drive where you can find the same files, in addition to the models already trained for the final ensemble. To run each model with Colab, it is necessary to manually upload the dataset.zip file inside the current working directory.

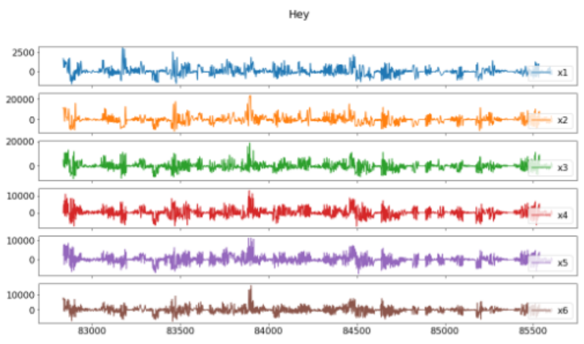
Dataset analysis

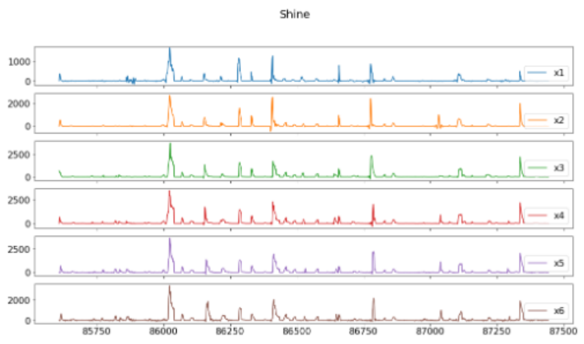
By analyzing the data provided we first noticed that they were unbalanced because one category ("Sorrow") had more samples than the others and some classes were underrepresented like "Breathe" and "Shine".



To overcome this issue, we have tried oversampling (making multiple copies of the same values) and augmentation techniques such as random shift and random scaling. At first, just by applying these methods to single models and training them, we obtained discrete results. Eventually, after training an ensemble that combined models performing these techniques with others that did not, the final results improved.

Since the given data were already divided into windows, we assumed that the samples related to the same window were continuous through time, while samples belonging to different windows had no such continuity relationship.





We have tried from the beginning to play with both the size and stride of the windows, but we find that changes to either of them lead to errors during submission on Codalab. This happened because the window size expected was fixed to 36, as it was in the given dataset.

By visualizing the distribution of the samples in each category, we discovered many outliers to deal with and thought to remove them or apply some normalization techniques to them. Results always got worse, so we decided to leave them as they were originally.

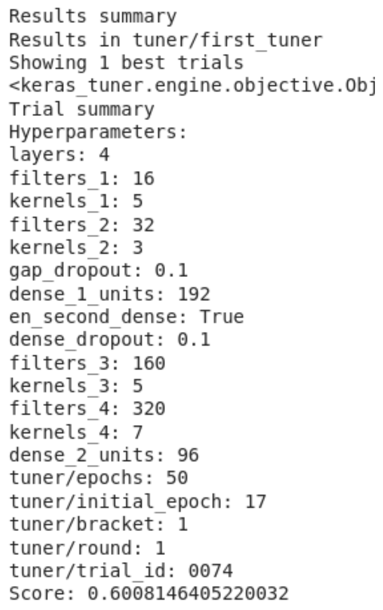
Eventually, instead of using the standard "validation_split" parameter, which does not guarantee data shuffling and stratification, we built our own custom implementation. After proving different split values, we chose 0.2 as the optimal one. We used the same split for each training session to make the results comparable and consistent.

Tuner

Unlike the first challenge, this time we decided to implement and use Keras Tuner to find a model with optimal hyperparameters. When building the model, we use the optimizer to find the best values for the number of convolutional layers, the size of the filters, the kernel size, and the percentage of dropout t

We used HyperBand search as a tuning algorithm, and each model was tested for at most 100 epochs with early stopping. A single-tuner training session required a lot of time, up to an hour and a half because of the high number of hyperparameters involved. For this reason, we tried to limit them as well as the number of trials executed.

For example, once we saw that the best models always had four layers, we avoided tuning the depth of the network anymore. Furthermore, we noticed the suggested dropout rate always varied between 0.1 and 0.4; therefore, we never ran the tuning with a dropout exceeding this value.



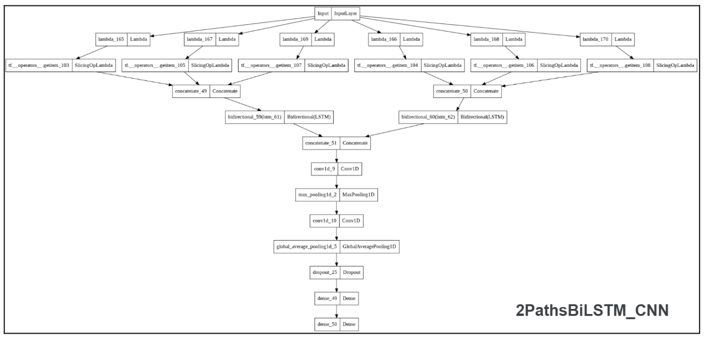
```
Results summary
Results in tuner/first_tuner
Showing 1 best trials
<keras_tuner.engine.objective.Obj
Trial summary
Hyperparameters:
layers: 4
filters_1: 16
kernels_1: 5
filters_2: 32
kernels_2: 3
gap_dropout: 0.1
dense_1_units: 192
en_second_dense: True
dense_dropout: 0.1
filters_3: 160
kernels_3: 5
filters_4: 320
kernels_4: 7
dense_2_units: 96
tuner/epochs: 50
tuner/initial_epoch: 17
tuner/bracket: 1
tuner/round: 1
tuner/trial_id: 0074
Score: 0.6008146405220032
```

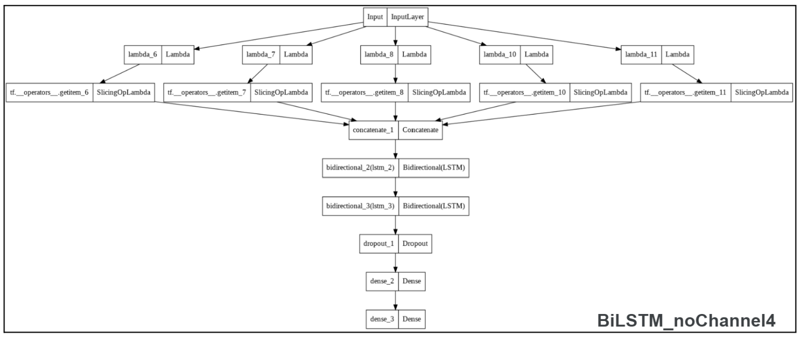
Custom models

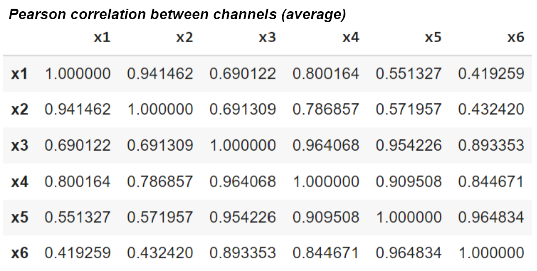
We started training a vanilla LSTM composed of two consecutive LSTM layers with 128 units, followed by a dropout layer with a rate of 0.5 and a final dense layer of 128 neurons. The accuracy of this model was around 0.5.

Then we moved to the 1DCNN model, which is made of two convolutional layers of 128 filters and kernel size 3, a MaxPooling1D between them, a GlobalAveragePooling1D, a dropout of 0.5, and a final dense layer of 128 neurons. At the beginning, this model scored around 0.55, but then we decided to exploit Keras Tuner's hyperparameters, reaching a better score around 0.65.

Afterwards, we attempted a BiLSTM model composed of two bidirectional layers of 128 units, a dropout layer with a 0.5 rate, and a dense layer of 128 neurons. This model outperformed the previous one, scoring around 0.6, and after applying Keras Tuner to it, we got a score around 0.68.


Input | InputLayer
lambda_165 | Lambda
lambda_167 | Lambda
lambda_169 | Lambda
lambda_166 | Lambda
lambda_168 | Lambda
lambda_170 | Lambda
tf.__operators__.getitem_103 | SlicingOpLambda
tf.__operators__.getitem_105 | SlicingOpLambda
tf.__operators__.getitem_107 | SlicingOpLambda
tf.__operators__.getitem_104 | SlicingOpLambda
tf.__operators__.getitem_106 | SlicingOpLambda
tf.__operators__.getitem_108 | SlicingOpLambda
concatenate_49 | Concatenate
concatenate_50 | Concatenate
bidirectional_59(lstm_61) | Bidirectional(LSTM)
bidirectional_60(lstm_62) | Bidirectional(LSTM)
concatenate_51 | Concatenate
conv1d_9 | Conv1D
max_pooling1d_2 | MaxPooling1D
conv1d_10 | Conv1D
global_average_pooling1d_5 | GlobalAveragePooling1D
dropout_25 | Dropout
dense_49 | Dense
dense_50 | Dense
2PathsBiLSTM_CNN


Input | InputLayer
lambda_6 | Lambda
lambda_7 | Lambda
lambda_8 | Lambda
lambda_10 | Lambda
lambda_11 | Lambda
tf.__operators__.getitem_6 | SlicingOpLambda
tf.__operators__.getitem_7 | SlicingOpLambda
tf.__operators__.getitem_8 | SlicingOpLambda
tf.__operators__.getitem_10 | SlicingOpLambda
tf.__operators__.getitem_11 | SlicingOpLambda
concatenate_1 | Concatenate
bidirectional_2(lstm_2) | Bidirectional(LSTM)
bidirectional_3(lstm_3) | Bidirectional(LSTM)
dropout_1 | Dropout
dense_2 | Dense
dense_3 | Dense
BiLSTM_noChannel4



Pearson correlation between channels (average)

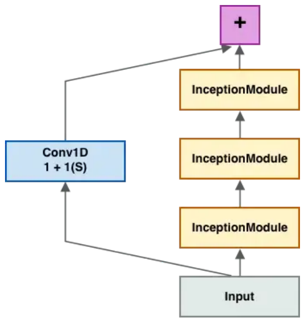
	x1	x2	x3	x4	x5	x6
x1	1.000000	0.941462	0.690122	0.800164	0.551327	0.419259
x2	0.941462	1.000000	0.691309	0.786857	0.571957	0.432420
x3	0.690122	0.691309	1.000000	0.964068	0.954226	0.893353
x4	0.800164	0.786857	0.964068	1.000000	0.909508	0.844671
x5	0.551327	0.571957	0.954226	0.909508	1.000000	0.964834
x6	0.419259	0.432420	0.893353	0.844671	0.964834	1.000000

By computing the Pearson correlation between the channels (averaged over all the measurements), we noted a high correlation between the channels. In particular, we made two models keeping in mind this information and got decent results.

We noted a high correlation between x1 and x2; x3 and x4; and x5 and x6 (higher than 90%). So we created a model that splits the channels into two groups ((x1,x3,x5) and (x2,x4,x6)) in order to analyze them separately in two different BiLSTM layers and then concatenates the results to analyze them in a CNN followed by a dense NN.

Then, we made a simpler model; we noted that channel 4 is the one that is the most correlated with the other channels, so we made a BILSTM model that only uses the channels x1, x2, x3, x5 and x6.

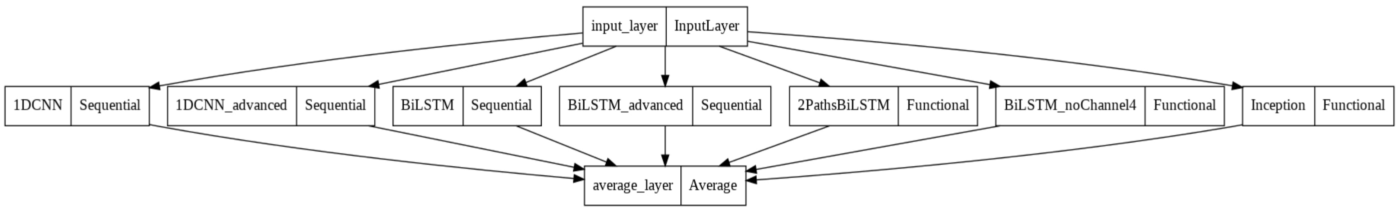
Also, we tried with a network consisting of a series of Inception modules followed by a Global Average Pooling layer and a Dense layer with a softmax activation function, with residual connections at every third inception module. We made a Network consisting of 6 inception modules followed by a dense layer as a classifier. The performance was not so good but we decided to use it for the ensemble because it was different from the others.



Model ensemble

To improve our score, we attempted to combine some of the previously described models into an ensemble model. More in detail, we picked up the seven best performing models, which were: 1DCNN and BiLSTM with just the original dataset, the same ones with oversampling, normalization, and augmentation; the BiLSTM-CNN with two paths; the BiLSTM without channel 4; and the inception-like model.

We have worked on two different ensemble approaches: a simple average of the model's outputs and a weighted average based on the scores obtained on the validation set. Out of the two options, the normal average proved to be more effective, probably due to some implementation problems that arose for the weighted one, and the final score we achieved was around 0.72.



Conclusion

One problem that we encountered was the difficulty of working on data without any information about its origin. No names or details about the features were given, and even the final categories had meaningless names. In the end, we relied on deep learning and its ability to automatically extract features from any dataset.

The final results demonstrate the capability of deep learning to generalize starting from the input data as well as the enormous versatility of deep learning models with respect to the nature of the dataset in use. We have also observed that 1DCNN models, because of the simplicity of their structure, are less powerful but faster to train and converge than LSTM models.

As a final thought, we saw the high potential of the model ensemble. In fact, it enabled us to improve the final accuracy and loss of the individual models into a better model.