AN2DL - HOMEWORK 2

ARTIFICIAL NEURAL NOOBS: Luca Fornasari, Nicolò Fasulo, Tommaso Aiello

About the challenge

We needed to perform a **time series classification** task. We were given an imbalanced dataset composed of 2429 samples belonging to 12 classes. Each sample was characterized by 6 time series with length 36 ‘timestamps’.

Data Preparation

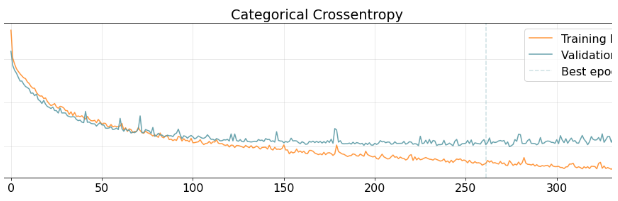
We first had to load the data using the numpy.load function. Then we decided to **standardize**, using the StandardScaler function, each of the 6 columns. In order to standardize also the input of the ‘predict’ function in the model.py we used the **‘pickle’** library in order to transfer the Scaler object from our notebook.
Then we decided to split the data set into training and validation sets (**Hold Out** 20%).
In this case we used the original window of 36 values so we opted not to build any subsequences and not to do any data augmentation.

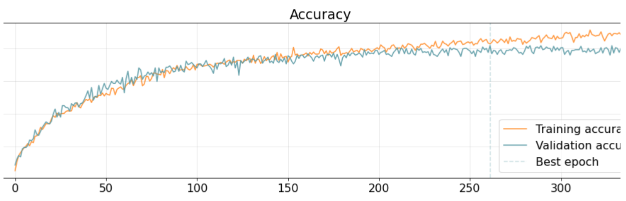
Ensemble Model

Before diving into the architecture that we implemented, it is important to clarify that we decided to use an **ensemble model** in order to take advantage of each model’s best features and capabilities to correctly classify different classes. We have decided to implement 4 models: 1) CNN with conv1d layers 2) RNN with bidirectional Lstm layers 3) Hybrid model with Conv1d and Lstm layers 4) ResNet style model with conv1d layers

CNN with conv1d layers

This model is composed of an input layer followed by a **Conv1D** layer (64 units and ReLu as activation function) followed by a Max Pooling layer followed by another **Conv1D** layer (64 units and ReLu as activation function). After that we put a **Global Average Pooling** layer and a Dropout layer (0.5). After that there is the classifier part, composed of a **Dense** layer (64 units) and then a Dense output layer with **‘softmax’** as activation function.



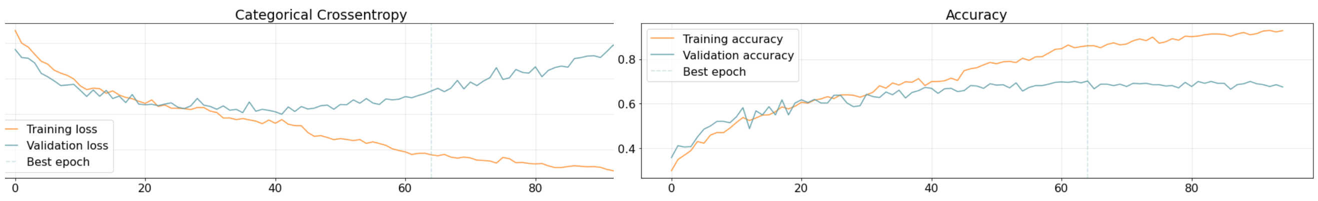


The model has 18508 parameters and it reached a validation accuracy of 0.7078 after 337 epochs.

RNN with bidirectional Lstm layers

This model is composed by an input layer followed by three **bidirectional LSTM** (with respectively 256, 128 and 64 units) and a dropout layer (rate of 0.5) as feature extractor;

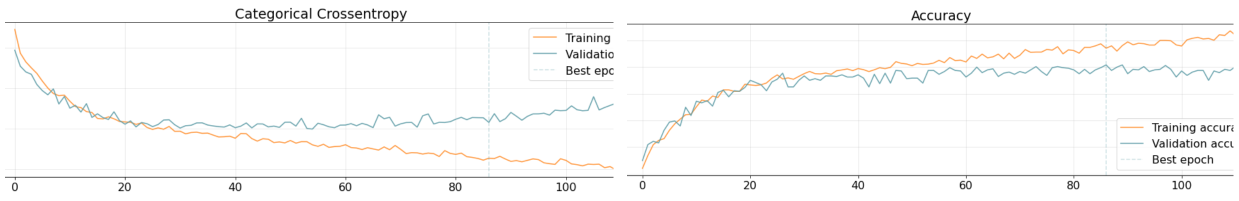
after that, the classifier part is made of a **Dense** layer (with 64 units and **ReLU** as activation function), a dropout (rate of 0.5) and an output dense layer (Softmax as activation function). In total, the model consists of 1,368,396 (trainable) parameters.



As shown in the images above, the model reached a validation accuracy of 0.7016 in 65 epochs, even though it was already slightly overfitted with a training accuracy of 0.86.

Hybrid Model with Conv1d and Lstm layers

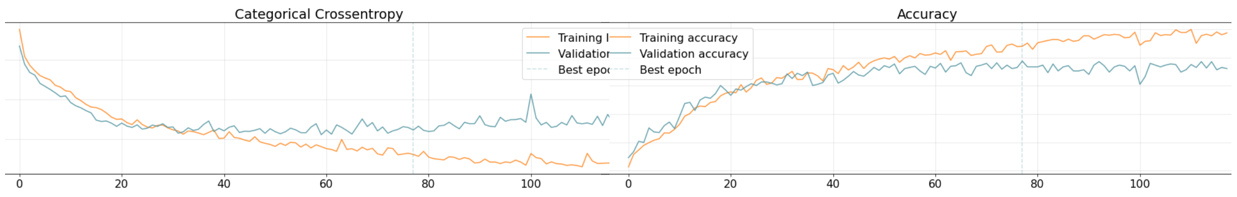
This model is composed of an input layer followed by a **Conv1D** layer (64 units and ReLu as activation function) followed by a **Max Pooling** layer followed by another **Conv1D** layer (64 units and ReLu as activation function). After that we put a **Lstm** layer in order to try to mix both convolutional and recurrent layers and then a Dropout layer (0.5). After that there is the classifier part, composed of a **Dense** layer (64 units) and then a Dense output layer.



The model has 51532 parameters and it reached 0.7078 val_accuracy in 112 epochs because it stopped since we used **Early Stopping** with patience equal to 25.

ResNet style model with Conv1d

This model is composed of two ‘resnet style’ blocks. The input of each block is processed by **3 Conv1D** and by a single **Conv1D** layer in parallel. The output of the block is composed of the sum of the two ‘paths’. At the end of the two blocks there is a **Global Average Pooling** layer and a Dropout layer (0.5) and then the classifier part composed by two **Dense** layers (64 and 32 units) with **L2 regularization** penalty. And then the presence of the output layer with ‘softmax’ as activation function.



The model is composed of 24824 parameters and it reached a val_accuary of 0.6872 in 118 epochs and it stopped since we used **Early Stopping** with patience = 40

Alternatives that we tried

We initially implemented a model that achieved 74,6% accuracy on the test set during phase one of the competition on Codalab. This model had more or less the same architecture as our final model, but we used a different **preprocessing** approach. Specifically, we used a **window** size of 27 and a **stride** of 3 to create sequences, and we also divided the input X of the 'predict' function in the 'model.py' file into 4 **subsequences** of 27 'timestamps' each by **slicing** X. However, even thought this model gave us the best result on Codalab, we ultimately decided not to use this model, during phase 2, for two main reasons:

1) We made a mistake by building the sequences before splitting the dataset into training and validation sets which led to our models achieving a val_accuracy of 88-90% because they were validated on samples that were partially equal to the ones in the training set.
2) We noticed that the padding length was too high because of the window and stride chosen, meaning that most of the subsequences were full of zeros.

We tried to fix these issues and still **divide** the sequences into subsequences trying different values for 'window' and 'stride', but this resulted in worse performance. We then decided to **increase** the length of the sequences to 48, filling the remaining part of the time series with the first 12 values of the same sequence. However, this also resulted in slightly worse performance on the test set. We also tried **normalizing** instead of **standardizing** the data, but this led to worse results because after some **exploratory data analysis** we realized that the **peaks** in the time series were important for classifying the different classes.
We also attempted to improve the model through **data augmentation** by creating new samples adding a **Gaussian noise** and by adding a Gaussian Noise layer to the model. However, these approaches did not improve the performance regardless of the magnitude of the variance we chose.
We also tried to **reduce the dimensionality** of the input using a **forward feature selection** method, but we end up having better results using all the features. On the other hand, even though we were skeptical because of the small and imbalance dataset, we tried **increasing the dimensionality** by adding a new feature that was a **non-linear** combination of the other features, and by adding a new column containing relevant statistics (mean, variance, median, standard deviation,...) of each time series, but we still got better results just by using the original six features.
To tackle the problem of having an imbalanced dataset we tried to add **'class_weight'** in the **'model.fit'** function but in this case it didn't work as well as it did during the first competition. (No free lunch)
We also tried to use a different type of recurrent layer such as **GRU** instead of **LSTM** layer. Finally, we attempted to implement an **attention** mechanism by adding a **MultiHead Attention** layer to our models, but this didn't result in an improvement in validation accuracy beyond 55%.

Conclusions

Our implementation obtained a result of 0.7436 on the part of the test set of the phase1 and a result of 0.7338 on the test set of the phase2 of the competition.