Artificial Neural Networks and Deep Learning - Second Challenge 2022

Time Series Classification

Andrea Chizzola, Raffaele Cicellini, Carlo Cominoli
Politecnico di Milano

December 23, 2022

1 Introduction

Time series classification is one of the main applications of deep learning methods. It consists in trying to classify a time sequence based on the values of the features in each timestamp. This task is usually approached with Recurrent Neural Networks, but also Convolutional Neural Networks can be adopted: in this challenge we experimented with LSTM, BiLSTM, Conv1D, also trying to combine them.

2 Data analysis

The training dataset is composed of 2 Numpy arrays:

- x_train is an array of 2429 multivariate time sequences, each of which is composed by 36 timestamps defined by 6 features
- y_train is an array of 2429 labels, one for each time sequence

These arrays were loaded into the script using *np.load*.

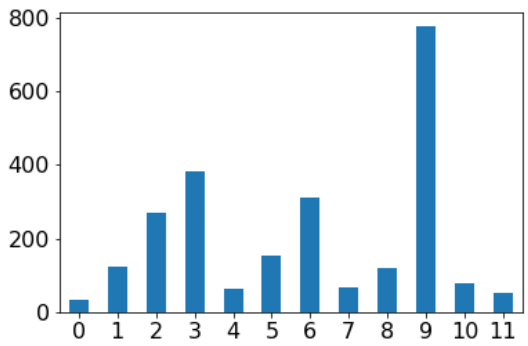

Figure 1: classes distribution

3 Data preparation

3.1 Data splitting

To split the dataset into training and validation we decided to use the *train_test_split* function. We tried both 85/15 and 90/10 splits between training and validation, it was the former, however, to provide the best results. To ensure the inclusion in the training set of samples belonging to all the different classes, we followed a stratified sampling approach.
After the splitting, a scaling transformation was applied to the data. While for most of the models the data was scaled in the [0,1] interval by subtracting the minimum and dividing by maximum minus minimum, in case of BiLSTM the Robust Scaler was used as it showed overall better results.
Given the unbalanced nature of the dataset, we tried using class weights to help the correct classification of less frequent classes. As the results were unexpectedly worse, class weights were not used in the final implementation.

3.2 Data augmentation

To facilitate the learning procedure, we also tried to generate new samples by adding to the original ones a random noise sampled from a Gaussian distribution. However, this augmentation approach led all the selected models to an extremely high level of overfitting, bringing the training loss very close to zero and the training accuracy close to one. As a consequence, the performances on the Codalab test set dropped drastically (16% for the best model that obtained 71% without augmentation).

3.3 Principal Component Analysis

We also tried to apply Principal Component Analysis as another form of preprocessing to perform dimensionality reduction. In fact, using the *corr* function we saw that the 6 features were slightly correlated between them. So, we performed PCA using 3 principal components instead of the 6 original features: this led to slightly better result in local validation accuracy with regard to the best

model, but on Codalab we obtained worse results (only 48.6% against 71.58% of the best model).

4 Training

We performed training monitoring the accuracy and the loss of both the training and of the validation set. We set 200 epochs as maximum, with early stopping and batch size equal to 32. The metrics used to select the best model was the accuracy measured on the validation set to reduce overfitting on training data. Training was done both on Google Colab and on local hardware using a dedicated GPU that was very useful to limit the required training time.

5 Models and techniques

At first, we tried to implement Convolutional Neural Networks with 1D Convolutions, and we extended them reaching a ResNet-style model with Residual Blocks and Conv1D layers. We also tried using a custom model built of Gated Recurrent Unit (GRU) layers. In the end, we tried building Recurrent Neural Networks following the LSTM model and that led us to the final implementation of a custom version of the BiLSTM.

5.1 GRU

We tried experimenting with GRUs, so we built a model composed of 2 GRU layers (128 and 64 units each), interleaved by BatchNormalization and ReLU layers. After these layers, the model was composed of 4 Dense layers (128, 64, 32 and 12 units each) with ReLU (for the first 3) and SoftMax (for the last one) activation function. However, the performances were poor (only 32% on local validation accuracy). We also tried to decrease the number of dense layers at the end, but anyway this didn't make the performances improve by that much.

5.2 1D Convolutions

While traditional Convolutional Neural Network models were developed for image classification problems, a similar approach can be applied on one-dimensional sequences of data, as the model learns to extract features from sequences of observations.
All the different versions we tried followed the same general structure given by a certain number of 1D Convolutional layers, with varying number of filters and kernel size, followed by a Global Averaging Pooling (GAP) layer and a fully connected part dedicated to the final classification. The average validation accuracy settled slightly above 50% and since adding more Conv1D layers didn't provide many improvements we considered a different approach.

5.3 Custom FCN

We also tried to combine different kind of layers to build a custom fully connected network. We used 3 Conv1D layers (128, 256, 128 units each) interleaved by BatchNormalization and ReLU layers, followed by 3 BiLSTM layers (128, 256, 128 units each), finally using a dropout layer, GlobalAveragePooling1D and a single Dense layer with 12 units. However, also this model was not good enough, since we obtained a maximum of 33.33% on local validation accuracy.

5.4 ResNet-style Model

Given the promising results of the 1D Convolutions, we extended the previous model by stacking multiple Residual Blocks, which use skip connections that connect the activations of a layer to further layers by skipping some layers in between. 3 Residual Blocks were used, interleaved by 3 Conv1D layers with kernel size 8, 5 and 3 respectively. Among them also Batch Normalization Layers were used before the relu activation layer. Similarly to the previous case, a GAP layer followed by a fully connected part was dedicated to the final classification. This approach led to both a local validation accuracy and test accuracy of 69%. Unexpectedly, neither the addition of further Residual Blocks nor the use of augmented data

was enough to improve the accuracy due to a strong overfitting.

5.5 LSTM and (Bi)LSTM

We attempted to use LSTM and BiLSTM networks without any convolutional layers and only using 2 dense layers at the end, but the performance was poor. These networks also tended to overfit the data. We then tried using a BiLSTM followed by two pairs of convolution and max pooling layers with another BiLSTM layer, finally a dropout and 2 dense layers. This network obtained 71% accuracy on CodaLab, but the results varied a lot depending on the preprocessing applied. With that network we tried MinMaxScaler, StandardScaler and RobustScaler, and the latter one resulted always the most effective. This type of preprocessing turned out to be the best probably because of the outliers in the dataset.

6 Results

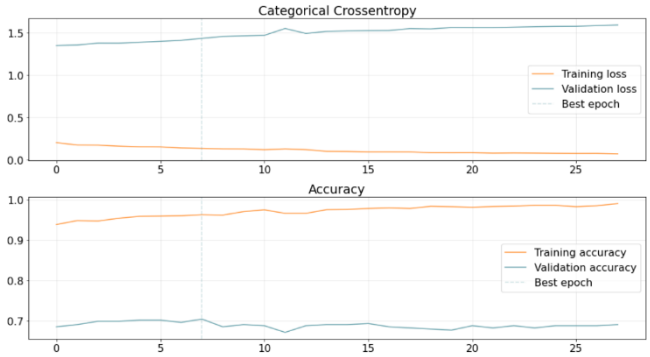

Figure 2: loss and accuracy of the best model

The above results have been obtained using the model that uses BiLSTM described in paragraph 5.5.

The validation accuracy of around 70% is remarkably similar to the accuracy obtained on CodaLab during phase 2, indicating that the model has been able to generalize well without overfitting. This may be due, in part, to the use of the RobustScaler. It seems that this model has been able to effectively generalize to new data, suggesting that it is a strong performer for this particular task.

7 Conclusions

In conclusion it appears that both the BiLSTM with convolutional layers and the model using ResNet have demonstrated good results in the competition, with the ResNet model performing particularly well during phase 1 and while the BiLSTM model performed better during the final phase 2. Given the similar results achieved by these two substantially different models – the first one being a CNN while the latter being a RNN - it is difficult to determine which is the most suitable for the given task. It seems that the difference in performance between the CNN and RNN-based models is not significant in this task.

Tools

- Tensorflow (also gpu version)
- Keras
- Scikit-Learn
- Jetbrains Pycharm
- Google Colab