ANNDL Homework 2 Report - De Novellis De Luca Pettorruso (Team AnotherDeep)

Introduction

The goal of the challenge was to implement and train in Python a neural network and use it to perform classification on multivariate time series. The time series were given in input as a 3D numpy array of the form (x, y, z) where x is the number of time series, y is their length and z is the number of features for each time series.
To implement the network we worked using Python Notebooks on the Google Colab environment, using tensorflow for the implementation of the neural network, scikit-learn for the pre-processing and tsaug for the augmentation.
The first task was to split the dataset given into training and validation and to do it we used the train_test split method of scikit-learn.

Vanilla LSTM

We started the challenge by building a simple Vanilla LSTM network as shown in the laboratory session. The LSTM layer (Long Short-Term Memory layer) is a layer that incorporates a “memory” of the previous data points seen (so the previous time steps of the time series) and so introduces a dependency not only on the current time instant but on the previous one too. The structure was as follows: two LSTM layers of size 128, followed by a Dropout layer with a value of 0.5 and finally a Single Dense layer of 128 neurons and the Output layer. With this structure we reached an accuracy of around 64% on the validation set and 60% on the test set of tha platform.

Preprocessing with scalers

The first thing we did in order to improve the first network was to preprocess the data in order to make it more suitable for our scratch model. For preprocessing the data we used various scalers from the scikit-learn python library:
We first tried the standard scaler, which normalizes the data such that its distribution has mean value zero and a standard deviation of one. Then we tried to use the MinMaxScaler, where the minimum of features is made equal to zero and the maximum equal to one. We also tried the MaxAbsScaler, which scales and translates each feature individually such that the maximal absolute value of each feature in the training set will be one.
All those scalers got the performance of our model worse, and the one that improved and gave us the best results was the RobustScaler, that is built to scale features using statistics that are robust to the outliers, for example removing the median and scaling the data according to the quantile range. With this particular scaling method, our result on the vanilla LSTM model improved by around 2-3%.

Network models

After implementing the pre-processing, we tried the following configurations to improve the model:

Bilinear LSTM

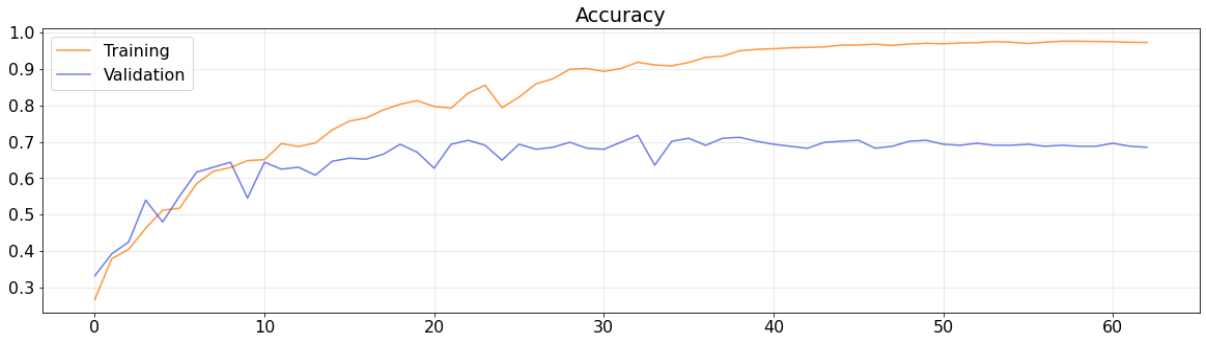
This model has basically the same structure of the first vanilla LSTM model, but the LSTM layers are bidirectional, which means that there is an extra LSTM layer reading the input in the reversed order, starting from the end. Then the output of the ordered and reversed input are combined.
The performance of this model was slightly better than the one of the vanilla LSTM.

1D Convolutional

For the next model we tried a different approach than the previous ones, that is applying convolution to the time series. So basically the model gets the input time series as an 1D image and performs convolution to extract the features and then classification. The first attempt was the same as the laboratory session structured as follows:
Two Conv1D layers of size 128-6 interleaved with a 'Max Pooling' layer and followed by a 'Global Average Pooling 1D' layer. Finally, a single dense layer of 128 neurons and the output layer for the classifier part. We tuned the kernel size a bit and changed it from 3 to 6, to capture more "long term" patterns of the time series and found that it slightly improved the performance of the model.
This model finally reached an accuracy of around 65%.

Modified 1D convolutional with attention layer

We noticed that the complexity of the first 1D Convolutional model could be increased a bit without overfitting, so we added some convolutional layers to the feature extraction part.
The final structure of the model was the following:
Three Conv1D layers of size 100 and kernel 6, a 'Max Pooling' layer, two more Conv1D layers and the Dropout. Finally, a single dense layer of 128 neurons and the output layer for the classifier part.
This final model, after applying augmentation on the dataset (which will be discussed in a later paragraph), reached the final accuracy of 72% on the validation set, and around the same on the test set of Codalab on both phases.



Attention Layer
The attention Layer is a mechanism used to memorize long sequences of information basically encoding the previously received input to save the most important "informations" of the previous data points.
To implement the attention layer we used an online implementation that defined an __init__ method and overwrote the following methods:

-build(): Creates the weights to use for the attention layer;
-call(): The call() method implements the mapping of inputs to outputs. It should implement the forward pass during training.

The “attention mechanism” is integrated with deep learning networks to improve their performance. Adding an attention component to the network has shown significant improvement in tasks such as machine translation, image recognition, text summarization, and similar applications.
In the absence of a mechanism that highlights the salient information across the entirety of the input, the model would only have access to the limited information that would be encoded within the fixed-length vector and miss important “patterns” of the data.

Augmentation

To improve the results already obtained, we decided to perform data augmentation on the training set in order to have more regularized data and prevent overfitting on the test set. At first try, we decided to perform augmentation by trying many functions from python library “tsaug” to add random noise to the original series, time warping 5 times in parallel, random crop sequence with length 300, random quantize to 10-, 20- or 30- level sets, random drift the signal up to 0.1-0.5 with 80% probability and sequence reverse with 50% probability. This method was revealed to be useful and improved the results on the test and validation set, but most of the transformations were not useful and did not provide “meaningful” training samples, so we decided to only use the random noise transformation by augmenting the data with the AddNoise method of the tsaug library. The AddNoise method adds random noise to the time series in a way that the noise added to every time point is independent and identically distributed. When we augmented the train set this way, we achieved 72.56% accuracy on the codalab test set, with very little difference in percentage between the validation and test accuracy.

Other trials

Since the dataset was imbalanced and there were classes such as “breathe” in which our model consistently scored low results, we tried to balance it by giving weights to the classes in order to give more importance during training to the ones with less data, an idea that worked well in the image classification challenge, using the “compute_class_weight” method of the scikit-learn library, that gives an higher score to unrepresented classes of the dataset, and using the calculated weights on the training. However, this method lowered the performance of every model we tried to build a lot, so at the end we decided not to keep it.

Last suggestion was trying to build the network with an architecture similar to the well-known deep learning models like vgg16 or ResNet. In particular, we tried to recreate the supernet with the same number of layers and neurons of the ResNetmodel but substituting the convolutional layers conv2D with conv1D layers. The results were not as good as expected so we decided to keep the original 1D convolutional model that was performing better with a much smaller execution time.