Artificial Neural Networks Homework 2 Report

Residual Sum of Students

Juliette Daniel, Davide Rigamonti, Raul Singh

INTRODUCTION

This is a report aimed at analyzing the work of our group through the second challenge held for the Artificial Neural Networks course at Politecnico di Milano under the supervision of Prof. Mattucci during the month of December 2022.

THE OBJECTIVE

The project assignment consisted in using the tools that were seen in the course lessons in order to solve a time series multiclass classification problem centered around the recognition of 12 different classes. We decided to use the recommended frameworks (TensorFlow and Keras) in conjunction with Python Jupyter Notebooks, which allowed us to tackle the problem in a familiar and effective way; as a collaborative versioning tool we used GitHub and we also used Google Drive as a shared storage for bigger files. In addition, we had to import the source code for the `train_test_split` function (from the scipy GitHub) and slightly modify it in order to have a non-randomized and stratified split for our dataset (which is a missing feature from the function); this was done in order to deterministically apply data augmentation and feature splitting. Lastly, this report was written utilizing LaTeX.

THE JOURNEY

Basic Models

We started out using the lab notebook about time-series classification as a base blueprint, since the problem we have to deal with falls into the same category. We set up three different approaches to time-series classification: Long Short Term Memory (LSTM), Bidirectional Long Short Term Memory (BiLSTM) and a 1-dimensional Convolutional Neural Network (CNN).

We tried out all three models using the same hyperparameters of the lab's notebook, with a 0.2/0.8 split between validation and training, without doing pre-processing or other modifications, just to have a quick glimpse of the performance difference between the models. The results were somewhat unexpected since both the LSTM and BiLSTM got to a certain validation accuracy immediately after the first epoch but never improved from that point. The LSTM scored 0.4 on validation accuracy, however, after evaluating performance on single classes it turned out that most of them had an accuracy of 0. The BiLSTM model showed an improvement compared to the previous one. Validation accuracy was around 0.55 and only class 4 scored 0 accuracy. To our surprise, the CNN model improved a lot during the training phase and got to about 0.6 accuracy. The accuracy on classes 4, 5 and 7 was really bad, but it was generally performing better than the previous models. This last model also surpassed the other two in training speed by being much lighter, therefore quicker.

Given the surprising results, we decided to focus our initial effort on the CNN. We did not discard the other two models, in fact we had the goal to constantly try to improve them since we predicted that the CNN would have failed to achieve a very high accuracy due to its structure's limitations compared to LSTMs which are built with this kind of task in mind.

Preprocessing

We then tried adding some preprocessing before our model. We first attempted to normalize by rescaling our data into a [0, 1] or [-1, 1] interval along different dimensions. We also tried applying this normalization considering different scopes for the identification of the maximum and minimum, but in general the outcomes were poor since most classes showed similar, or even worse results than the previous case.

We also instead tried standardization by fitting our data around a normal distribution with mean 0 and deviation 1. We immediately noticed a significant improvement when standardizing over the feature axis and, when combined with the CNN, we managed to obtain a validation accuracy around 0.70.

However, some classes still had an accuracy of 0 and we felt like we still needed to look for further improvements.

Class balancing

As for the previous challenge, we tried class balancing by means of adding weights for different classes to the loss function during training. The expected result would be to obtain better accuracy on the classes with fewer samples; however, while we succeeded in achieving a more distributed class accuracy, this was at the cost of a heavy drop in overall performance as the validation accuracy dropped to 0.30.

Therefore, we decided to discard this approach and try new techniques instead.

Transfer learning and fine tuning

As we were looking for a new approach, we tried using transfer learning to improve our performance. The model we chose to use for transfer learning was ResNet50. Unfortunately, the results were not satisfactory as, not only did the model have a long training time due to its heavier weight, but the accuracy was worse than the previously tried techniques.

We also tried experimenting with fine tuning at different levels by considering different groups of layers but the results were similar and unsatisfactory once again.

This is why we tried changing our approach and focused on data augmentation.

Data Augmentation

We felt like data augmentation could have improved the generalization capabilities of our model and its accuracy overall; in addition, due to the strong imbalance in the number of samples for some particular classes, some classes had far better predictions than others. Therefore we thought that data augmentation aimed at the less populated classes could expand their samples and improve the accuracy overall.

Unfortunately this wasn't the case as, with all the different data augmentation methods that we tried, the results were similar and not impressive at all; when aimed at the less performing classes their performance was undoubtedly improved, but at the cost of reducing the performance of the other classes with a net loss that went from 1% to 10% overall accuracy for different models, hyperparameters and DAUG techniques.

Even applying a small amount of data augmentation to all of the classes was detrimental to the accuracy of the model, resulting in consistently worse results for most of the tested combinations of techniques.

However, implementing the augmenting functions by hand has been a great learning experience: we decided to implement a function able to add gaussian noise by specifying its mean and variance, a function to reverse a customizable portion of a series and a cutmix function between samples of the same class with various customizable options; all of the augmentation functions can be found in the provided notebook and can target specific features and/or classes.

Window tuning

Since our model wasn't improving anymore, we tried an approach seen during lab sessions: dividing the dataset in sequences by specifying a window size. We developed a small function that takes some data and generates sequences from each sample, with the specified length. The function was very minimal and allowed only for a stride of size 1, while the window size was customizable.

At first, our validation accuracy improved a lot on all three models showing suspiciously good results. Thanks to the experience acquired in the previous challenge, we were able to immediately identify the issue: by applying the function also on the validation dataset, we were getting fake good results. The same thing happened with data augmentation in the previous challenge, so we immediately addressed the problem. We then applied the window function only on the training dataset and not on the validation one.

We tried many window sizes with all models at our disposal. Large windows (between 28 and 35) showed a slight improvement on every model, while smaller windows did not impact accuracy much, and in some cases it was even worse. After some more tinkering, we got the best result using the CNN model with a window size of 34, which scored an astounding 0.74 validation accuracy, which translated to a 0.72 on the leaderboard. The two LSTM models improved a bit more, but did not come close to the CNN's accuracy.

We extended the function to make it possible to have a stride greater than 1, add padding to the new windows and extend the windowing outside of the given windows of size 36 by considering all of the samples in the dataset as a continuous time series. With these new tools we tried different combinations of parameters to see if we could improve the model even further; however, after a lot of tinkering, we achieved the best results using a window size of 34, stride of 1, no padding and the original window splitting (basically our previous setup).

In the end, by trying out different numbers of hyperparameters for our various models, we were able to consistently get 0.74-0.76 validation accuracy using the CNN in conjunction with our tried and tested window size of 34 and stride of 1.

Feature tinkering

Inspired by a recommendation found in the logbook we decided to try and see how tinkering with the features would affect the results of our models.

To do so we implemented three main functions: a function able to remove one or more features from the dataset, another function that would rescale between 0 and 1 a specified subset of features over the current window and generate a new parallel feature containing the mean of the values of the original feature over a given range for each specified feature, and lastly one function able to create for each specified feature, a new feature where the value of the series at time t was given by the difference of the values of the original feature at time t and t-1 (imposing 0 for the first value).

Unfortunately the results weren't satisfying since they didn't lead to any meaningful improvement in prediction accuracy; however, it was interesting to see how many of the features could be transformed or outright removed without impacting on the overall performance as a lot of information contained in the dataset remains the same, even if presented differently.

Attention

As a last resort, we tried to implement a self-attention mechanism by utilizing the `keras-self-attention` library, which offers a handy sequential layer to implement directly inside the network structure.

After experimenting a bit with the position of the layer we noticed better results when placed after the convolutional layers of our best model; while, this addition didn't translate in an immediate improvement in validation accuracy we noticed that the models that implemented the self-attention mechanism were more robust and were more prone to preserving the performance from our local validation to the CodaLab test set, thus indicating a "better understanding" of the features and more generalization power.

Having more time at our ready, we would have certainly liked to experiment more with attention and self-attention mechanisms, however, this was a last-minute addition that resulted in a 1% accuracy increment on the CodaLab test set.

Overall we feel like we didn't really explore this particular path in-depth as we are sure that it could have improved our model even more since it's basically state of the art technology.

THE RESULT

Our final network is composed of a Keras Normalization layer which standardizes our data around a normal distribution with mean 0 and deviation 1, this particular preprocessing layer is fit on the whole training set beforehand as it needs a reference for the mean and the variance. After the preprocessing section we proceed to the feature extraction section, which is composed of three one dimensional convolutional layers (counting 512, 256 and 128 units respectively) with 3x3 filters using "same" padding, interleaved by one dimensional max pooling layers. Then, we have the sequential self-attention layer and the global average pooling layer, followed by a dropout layer and by a two-layer classifier where the first layer counts 128 "ReLU" units and the second layer is the classic softmax output layer counting 12 = #classes units.

Even though we experimented with much more complex designs, it's a bit underwhelming to see such a simple model (and a purely convolutional one too!) succeed over the other ones; this shocking result may have different reasons at its core as some particular weaknesses come clear when we compare our convolutional model with the LSTM-based models: even after pushing the training epochs to the very maximum, overfitting is almost inevitable and most of the critical classes perform much worse w.r.t. the best classes while, with the models that include components specifically made to work with time-related data, we observe more balance between the accuracy results of the various classes.

In addition, by analyzing the raw dataset, it's clear that it doesn't represent a "clean" multivariate time series but there might be some time-independent extra information that needs to be extrapolated and might throw off the LSTM models, justifying their poor performance in comparison with the convolutional models.