Artificial Neural Networks and Deep Learning

Caspar Dietz [caspar]*, Gianmario Careddu [GianMario]†, Jody Roberto Battistini [Jody98]‡
Team: JCGTeam
M.Sc. Computer Science and Engineering, Politecnico di Milano
**casparvictor.dietz@mail.polimi.it*, †*gianmario.careddu@mail.polimi.it*, ‡*jodyroberto.battistini@mail.polimi.it*

1. Introduction

The provided time series data came in the shape (2429,36,6). A total of 12 distinct classes had to be classified. We were not given any information on the meaning behind the features.

2. Data Preprocessing

We started by splitting the original data into training (80%) and validation (20%) sets. Furthermore, we decided to use the CodaLab as test set since we are in a competition and not in a real scenario. After the split we saved the sets for reproducibility and consistency. To test different types of preprocessing we used as a benchmark a BD-LSTM model. Details are provided in the following subsections.

2.1. Class Weighting

Since the data set was heavily unbalanced, we started by weighting the loss function during training. We experimented with custom formulas, as well as the implementation of Keras. Unfortunately, both did not improve our accuracy:

Weighting Technique	Accuracy
Without Weights	0.5946
Custom Weights	0.5905
Keras Weights	0.3106

2.2. Oversampling

Next we experimented with oversampling. We started by oversampling the minority classes until all classes had equally many samples. We then compared this approach to the one in which we oversampled the minority classes up to the mean size of all classes. Results were comparable with those of class weighting, so no improvements.

2.3. MinMax and Robust Scaler

To standardise the data we first tried the MinMax scaler feature-wise, but we did not achieve satisfactory results. We noticed that for each feature, there were samples with outstandingly high absolute values, which could potentially be outliers, but without knowing the nature and meaning of the data we decided to preserve everything. We were able to improve validation accuracy using the RobustScalar which removes the median and scales the data according to the quantile range. We scaled the data into the interquantile range 0.25-0.75.

Scaler Technique	Accuracy
Without Robust Scaler	0.5946
With Robust Scaler	0.6316

2.4. Feature Selection

Since we had to no explanation behind the data, we relied on statistics to identify the features that were important. We looked at the variance of the features, discarding the features with less variance. However, holding out features had no improving effect on the validation accuracy.

3. Data Augmentation

Since we had no information on the meaning behind the data, augmentation had to be applied cautiously, not to loose important patterns that we did not know about. We have tried several techniques for augmenting the dataset, such as: (1) adding Gaussian noise to the time series, (2) scaling the time series by multiplying it by a number, (3) changing the period of the time series by dividing it by a number and (4) rotating the time series by shifting it 10 steps in the future. We used these techniques one at a time and discovered that the first technique was the one that gives the best results, however lower accuracy with respect to not using it. We also tried to used all the techniques together, but with worse results. In the following table, we reported the best augmentations we tried together with their accuracies:

Augmentation Technique	Accuracy
No Augm. BD-LSTM	0.5946
Gaussian Noise BD-LSTM	0.5926
All Augm. BD-LSTM	0.5617

4. Architectures

Having identified our best preprocessing, we went onto experimenting with a broad range of different models. Our approach was to steadily increase the complexity of the used models until we find the most suitable.

4.1. Conv1D

The first and most basic model we tried was a one-dimensional convolutional neural network, in which each Conv1D layer was followed by a MaxPooling layer. This allowed us to easily experiment with horizontal scaling of the network (in Fig. 1 we compared the performance when we duplicated or triplicated these layers). Just before the classifier (two dense layers) we also added a GAP layer. As can be seen from Fig. 1, we had not started to overfit and only achieved unsatisfactory accuracies with such simple model.

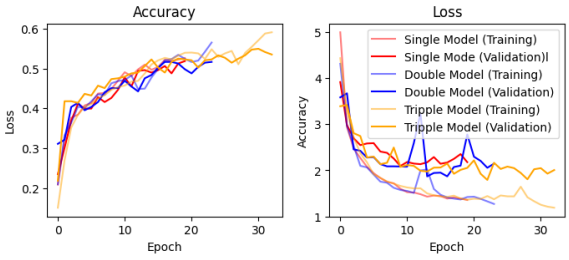

Figure 1. Accuracy and loss when duplicating or triplicating the Conv1D and MaxPooling layers.

4.2. Conv2D

We reframed the time series classification problem into an image classification problem. The core idea was to stack the same copy for each time series multiple times. Before stacking we shifted the series with respect to the previous copy (see Fig. 2). Applying this procedure 36 times to each time series of size 36, we were generating 36x36 "images". We wanted to create a second fictitious dimension to simulate the spatial correlation existing in pixels of images. With this mapping we ended up having six 36x36 images per sample, one per feature. We tried three shifting options: *(1)* shifting the stacked sample by one, *(2)* shifting the stacked sample by the filter size (three in our case) and *(3)* at the i-th stacking step, shift the time series i times with respect to the copy already stacked.

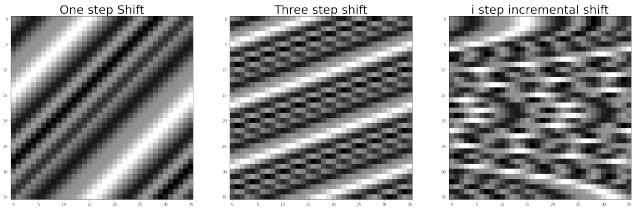

Figure 2. Images generated by the 3 shifting options on sample one feature one

We used EfficientNetV2B3 as feature extractor and a custom dense layer with 256 neurons before the softmax layer as a classifier. To test if some distinguishable patterns were emerging from our transformation, and since the images for each sample were very similar, we have flattened the dataset feature-wise. Therefore each sample with six feature images was decomposed in six image samples. To deal with the required RGB input shape, we fed the same grey scale image into each channel. Each image was individually scaled into the range 0-255, because we wanted to capture pattern associated to the relative evolution of the time series and not on its absolute values. The only shifting option that was generating distinguishable pattern among classes was the third one in Fig. 2. We achieved 0.55 validation accuracy on the flattened dataset. To employ this architecture for the original classification problem we were making six predictions, one per feature of the unknown sample, then selecting the most voted class. The true validation accuracy solving the original classification problem dropped substantially to around 0.3. We experimented with a second approach, using only three features of each sample and feeding each feature image into a different colour channel. This turned out to be very inaccurate. At this point we discarded this approach. In addition to the lower accuracy, the training was much more expensive in terms of GPU time than all the others.

4.3. LSTM

At this point, we scaled a basic LSTM network horizontally to find a model complexity suitable for our use case. In particular we tested a network with either two, three or four consecutive LSTM layers for which we returned the output of each cell. This variable part was followed by a dropout layer and two dense layers for the classifier. As it can be seen from Fig. 3, the

accuracy was not improving over our existing model and the network quickly began to overfit.

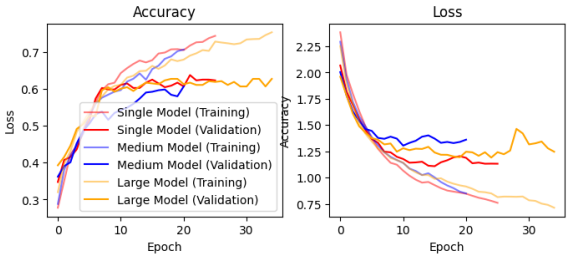

Figure 3. Accuracy and loss when duplicating or triplicating the LSTM layers.

4.4. BD-LSTM

Similar to the previous sections, Fig. 4, shows the accuracy of the base line BD-LSTM compared to models in which we duplicated or triplacated the bidirection LSTM layers in the network. The best results were achieved by two bidirectional layers. We then went on with this configuration and found satisfactory results by introducing dropout layers behind the bidirection LSTM layers, testing different configurations of parameters and hyperparamters with random search, we were able to train models having more than 0.7 validation accuracy.

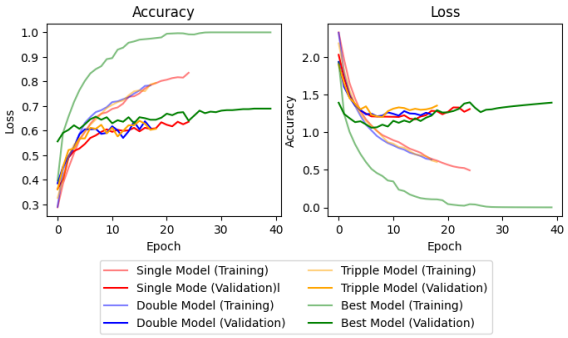

Figure 4. Accuracy and loss when duplicating or triplicating the BD-LSTM layers.

4.5. Transformers

We have tried the transformer architecture used in the paper *Attention is all you need*. As a first attempt we used their architecture without any modification. The learning was slower than our other approaches and the transformer required much more epochs. This configuration was underfitting and the train accuracy was reaching a plateau before even reaching 0.7 accuracy. Since we were using 100 epochs of patience we concluded that the model was too simple. We tried to add complexity by increasing: head size, number of heads, filter dimension, transformers blocks and dense neurons. The increase in complexity was in general leading to more unstable convergence. We dealt with this problem by lowering the learning rates and/or increasing the batche size. After various attempts we were able to increase the validation accuracy up to 0.55. However, we decided to stop at this point because we were able to find configurations with enough complexity to overfit but we were not able to appropriately tune the network, and to understand which parameters were the actual bottleneck for performance. Moreover the training was very time consuming compared to simpler models.

5. Ensemble method and delivered final model

The final step was to ensemble all the best models obtained so far. We ensembled nine BD-LSTM models having validation accuracy greater than 0.7. We started by simply averaging the prediction of the ensembles, then we tried to weight every class prediction of each model in the ensamble by its F1 score on that class, divided by the sum of all F1 scores on that class among the models. Finally we applied tabu search to find a set of weights to compute a weighted average of the models' predictions. Among all the perturbation techniques that we tried, the best one was to randomly perturb the weight by a step in the range [-0.35, 0.35] and saving the solution only if the the validation accuracy had increased. On the validation set we increased accuracy by 3%. Despite the increased performance this technique could be risky to use in real scenarios because it is like training a linear regressor on the validation set and validating it using test data. We also attempt to compute the gradient of the ensembled models to employ gradient descent using Autograd with the jax library. Unfortunately it was not compatible with tensorflow's tensor data structure so we decided to stop. Final model performance and confusion matrix:

min F1	max F1	Avg F1	Valid. acc.
0.2857	1.0	0.6816	79.01%