REPORT GRADIENTERS

Matteo Braceschi, Alice Cariboni, Claudio Corradini

Keras tuner

In the first challenge we tried to use Hyperas, a wrapper for convenient hyperparameter optimization, but without success. This time we succeeded in implementing and using Keras Tuner, the optimizer suggested in the README open file.

We did a massive use of the optimizer, to find the best values for the size of the filters, the kernel size and the percentage of dropout when building the model. We also were able to optimize the size of the window, the stride, the size of the batch and the validation split of the data.

We used RandomSearch as a tuning algorithm and each model was tested in 10/20 different trials for 20 epochs, then the best one was retrained for 200 epochs or until early stopping.

We also tested different metrics to evaluate the best model during the tuning, at first mean absolute error, then root mean squared error and r squared that gave us better results.

In order to do so we had to override the run_trial method to set up the hyperparameters and we created an Hypermodel class that is useful to create a model and build the sequences to feed the model given an hyperparameter set.

(Some of us ran the code on Google Colab but others were able to run the code locally on GPU and CPU, so we had different points of view and we could see different errors and behaviors that arose from a different setup and package installation.)

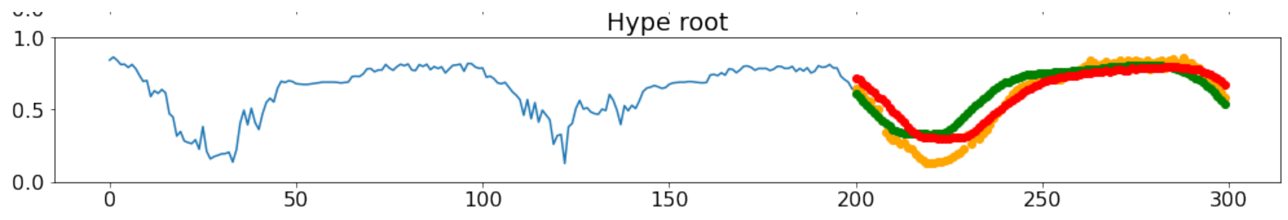
After training a model in different trials with keras tuner we chose the best hyperparameters and trained the same model changing other things like loss function, activation functions and metrics.

One of the first best models found with keras tuner showed even better results after adding the LeakyReLu activation layer in between LSTM layers or in between Conv 1D layers .

Huber loss function

Another specification in our model was the loss function. In the first models we used the Mean Squared Error (MSE), then we tried to change it with the Huber loss function and we noticed that the performance improved. We know that the MSE loss function is great for learning outliers and the MAE loss function is great for ignoring them. While the Huber Loss offers the best of both worlds by balancing the MSE and MAE together. In general, for loss values less than a specific value x, use the MSE; for loss values greater than x, use the MAE. This effectively combines the best of both worlds from the two loss functions. In particular, in our first model, the performance improved from a 7.03 score to a 5.44 score.

Below follows a comparison between the two loss function that we used with the same model trained: Huber(red) and MSE(green)



Autoregressive forecasting vs direct forecasting and more simple model structures

We started training models almost only using autoregressive forecasting because after seeing the results on the model we studied in the laboratory we were convinced it was better than direct forecasting. After changing a few things we started to realize that direct forecasting gives far better results with this dataset. We also started using only bidirectional LSTM or different models mixing all kinds of layers (Conv1D, Bidirectional LSTM, Not bidirectional LSTM). All of them gave better results with direct forecasting.

Due to the increased duration of training and considering the few improvements reached by adding complexity to our model, we started to train simpler models, composed only of convolutional layers. Surprisingly, the score improved by a lot (from 3.9878 to 3.8464). Optimizing with many trials a really simple network with just some design side choice (the use of leakyRelu, Gap and others seen above) it's been the method that gave us the best results.

Attention mechanism

Another way that we followed and that we want to describe due to the fact that we spend a lot of time around it is the implementation of the attention mechanism in our model. We thought that weighted attention networks were the best solution to improve the performance of the model, specially in time series forecasting with long sequential data, and we wanted to discover more about it at the end of the competition. Unfortunately, we don't submit them now because the attention mechanism didn't improve our score compared to only convolutional models.

Firstly, we used LSTM because it worked well with sequential data, which depends more on time than MLP and 1DCNN. And we added a custom layer that implements the attention mechanism. This attempt didn't improve the score, on the contrary it got worse a little.

Another attempt has been done with the MultiHeadAttention layer from the homonymous library and in particular, this time we used the keras tuner to choose the best hyperparameters (such as the num_heads, the number of attention heads). The score remained more or less unchanged but the time for the tuning and training increased a lot.

The last tentative was the Transformer, which is based on the MultiHeadAttention mechanism. We didn’t use the one that we had seen during exercise 7 but we tried to use a transformer found on the web so we could change the model as we wanted. In the end, the best result that we obtained with the Transformer model is 4.5508, not such a bad result but we had better performance in other models, only convolutional and especially when we applied the K-folding.

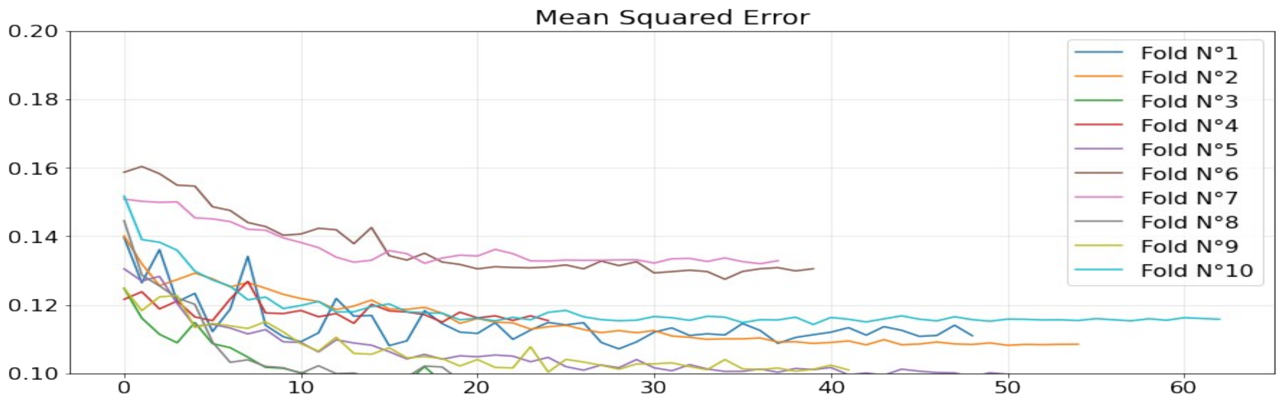
K-folding

The quickness of the training by simple only-convolutional models suggested a further way: we applied the K-folding to obtain a better model.

In detail, we copied our best model 10 times, training each copy with different folds of the dataset (always 9 folds for training and 1 for validation). The final prediction is an average of all the 10 predictions. At first not many improvements have been done, after many trials k-folding different models (the score dropped from 3.8464 to 4-5).The last modification that has been game-changer was the removal of L2 weight regularizers, that has brought our score in our best score of 3.8174.

Our explanation is that the k-folding it’s more than enough to keep generality in our model, and that any other technique to avoid overfitting might only lower the score. Our last trial in this direction has been the removal of the GAP, adding the Flatten layer. Unfortunately the size of the single model did not permit storage in Colab, nor Kaggle or personal devices of the 10 trained models.

Below we show the training of the 10 models, each one with a different configuration of folds for the train-validation:


Mean Squared Error
Fold N°1
Fold N°2
Fold N°3
Fold N°4
Fold N°5
Fold N°6
Fold N°7
Fold N°8
Fold N°9
Fold N°10