Artificial Neural Networks and Deep Learning - Challenge 2

Samuele Tagliabue - 10563460
Marco Tagliafierro - 10602548

How the problem has been approached

The problem to be solved is a time series forecasting where our forecasting model has to predict the future exploiting past observations in the input sequences.

Firstly, we analyzed the dataset, consisting of 68528 samples for each feature; then we had to decide if we wanted to use a single-shot multivariate forecasting, making the predictions all at once, or an autoregressive method, building the entire prediction iteratively.

After many tests, we decided to go with the autoregressive approach because it performed better and, as a bonus, it would have been more flexible during the shift from phase 1 to phase 2 where the number of samples to predict would have been greater.

Considered models and final model structure

In every model we trained there are some common features:

- We used early stopping to prevent overfitting by monitoring the loss on the validation set and stopping the training when the network didn't improve for 10 consecutive epochs. Another technique we have adopted is the addition of dropout layers: each hidden unit has a probability of 0.5 of being set to 0 during training.
- During the fit process of the model we added the callback *reduce on plateau* that reduced the learning rate of a factor of 0.5 when val_loss stopped improving.
- In order to predict the next values for more than one channel, we used a dense layer with a number given by telescope*num_channels, followed by a reshape layer to obtain a tensor of dimension [none, telescope, num_channels]

We started from a model composed by a *convolution1D* layer followed by a *maxpooling1D* and another *convolution1D* layer, all followed by a *GlobalAveragePooling* layer; this was our starting point on which we worked on.

With this model we noticed that performing autoregression and plotting the predictions we were not able to recognize possible jumps in the time-series of some features, meaning that probably it was too simple.

We have then built two other models composed of two layers, in the first case two LSTM and in the second two BiLSTM, but the results for both were similar to the ones of the first model with poor overall performances.

At this point we decided to increase the telescope of our model to 48 (the initial value was of 1 and we incremented it with steps of 12 until good performances had been achieved) and retrain the models defined before: this gave us a first substantial bump in performances.

After fine tuning, the model that performs better was the one with the BiLSTM layers obtaining a Root Mean Squared Error of 5.04 on the hidden test set (the one used on the evaluation server).

We finally noticed that using only one bidirectional layer gave us better performances and an even lower RMSE of around 3.60.

Pre-processing and oversampling

Before fitting the models, the provided data has been pre-processed in order to normalize the values of the timeseries: this procedure has been done by simply applying a min-max normalization to the train, validation and test sets.
After the normalization, we have applied also a form of oversampling using the provided data to produce new time series in order to fit our models on a bigger dataset; the used instruction is something similar to the following one:

```
data_iterator = tf.data.Dataset
            .from_tensor_slices((X_train, y_train))
            .shuffle(X_train.shape[0])
            .batch(batch_size)
            .repeat()
```

What this instruction does is creating a dataset composed of the slices from X_train and y_train and then setting up an infinite data pipeline by calling *batch* - which combines consecutives elements of the dataset into batches of the specified size - and *repeat*.
This single change, alone, provided a very noticeable improvement in the performance of our models regardless of the actual structure of the model's layers.
Please note that we did not set a seed in the shuffle function since with one set the results were way worse (RMSE greater than 5) and at the same time it's absence produced only slight changes to the overall performance of the model (+/- 0.30 in the RMSE).

Hyperparameters

After we found the best performing model, we started analyzing the impact on performances made by several hyperparameters. The most notable ones are:

- Fit function:
    - steps_per_epoch: since we used an infinite data iterator to be used during the training process, we had to specify this parameter which represents the total number of batches of samples taken from the iterator in each epoch. At the end we set this value to 950
    - validation_steps: same as above but for the validation process, set at the end to 95
- Dataset splitting: several dimensions for training, testing and validation sets have been tested in order to find the best combination; moreover when the optimal model has been found, the test set dimension has been set to 0 in order to have as many samples as possible for the training procedure. We found that a validation set of 3000 samples and a test set of 500 worked best during our tries.
- Different values have also been tested for the following values, increasing and decreasing them until optimal results have been achieved:
    - window, set to 250
    - stride, set to 5
    - telescope, set to 48

In order to evaluate how each one of this hyperparameters affected the overall performances of our model, we used the following metrics:

- MSE, mean squared error
- MAE, mean absolute error
- graphs to visually compare our model's predictions against the expected results in order to have an overall idea of how well/bad it was performing (first graph). Another particularly useful graph has been the one pictured on the right, which allowed us to evaluate the mean squared error in every point of the predictions