Challenge 2 Report - ANNDL 2023/2024

(1) Andrea Federici, 10621924
(2) Umberto Salvatore Ficara, 10660214
(3) Alberto Pirillo, 10667220

The purpose of this document is to show the results and explain the passages of our study that were adopted to solve the second ANNDL challenge. Firstly, we describe the time series forecasting task and the solutions that are usually adopted to solve this kind of problem. Then, we discuss in depth the data preprocessing techniques that have been considered, and we devote some time to discuss the different model architectures that we chose. Finally, we talk about the obtained results, together with our considerations on the choices we made.

Problem description

In this challenge, we tackle a *time series forecasting problem*, which involves analyzing time series data using statistics and modelling to predict the future values of a variable of interest based on its historical observations. The underlying assumption is that there is a temporal dependence among observations, implying that the value at a time step is influenced by its past values. This can be used to predict the future values of the time series with a proper approach.

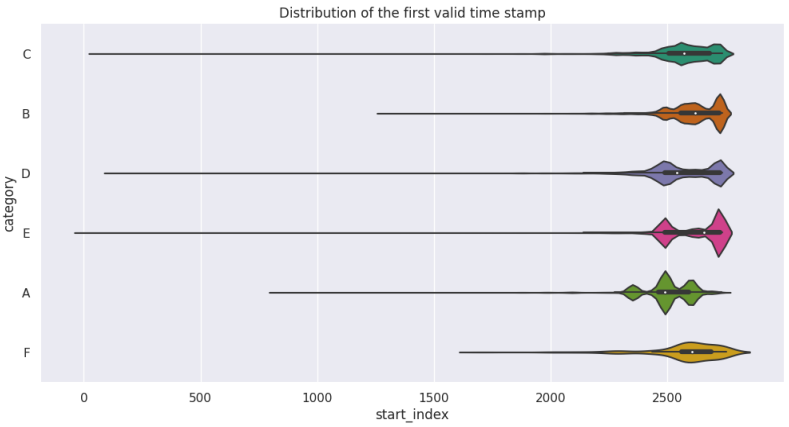
There are various techniques to address a time series forecasting problem, ranging from traditional statistical methods like *ARIMA* and *Exponential Smoothing* to modern Machine and Deep Learning techniques like *Support Vector Regression* (SVR), *XGBoost*, *Recurrent Neural Networks* (RNNs), one-dimensional *Convolutional Neural Networks* (1D-CNNs) and *Transformer neural networks*.

The dataset we were given consisted of two matrices, one containing time series with their respective time steps, and another with information about the start and the end indexes of the sequence.
The time series have a variable length: the maximum one is 2776 whereas the minimum one is 24. This is why the original data has been padded to be stored in a single homogeneous matrix.

Our objective is to develop a forecasting model capable of generalizing enough to transcend the constraints of specific time domains, so that it can be used as a general forecasting model, i.e., not limited to predicting in a single or predefined time context. At test time, the model is required to process a time series of length 200 to predict its first 9 (18 in the second phase) future values. To build our models, we decided to focus on Deep Learning solutions, which are the main topics of the course.

Dataset Inspection and Preprocessing

Our work began by inspecting the provided dataset which consisted of several uncorrelated monovariate time series, belonging to six different domains: “demography”, “finance”, “industry”, “macroeconomy”, “microeconomy” and “other”. Time series inside of the same domain are not related but only collected from similar data sources. On average, the valid length is about 198, and most of the time series have a length smaller than 500.

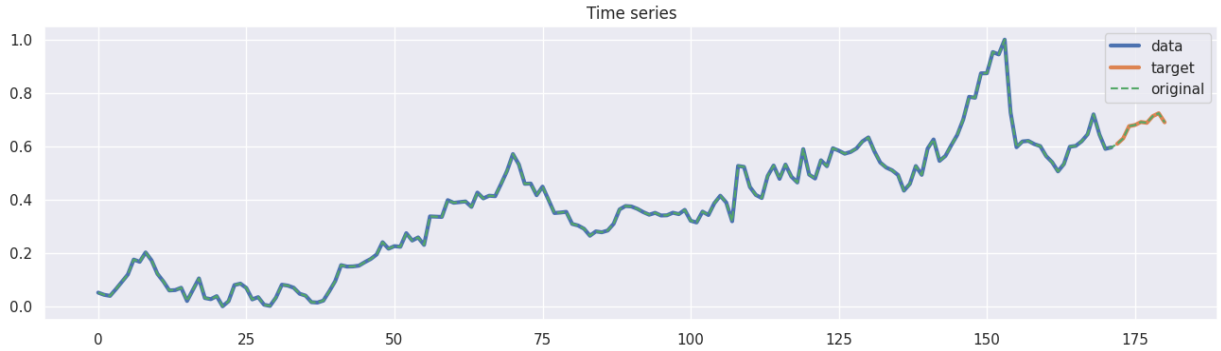


At this point, we needed to create a proper dataset structure so that the problem could be addressed in a supervised way. First of all, we remove all the padding from the time series and store them in a list, which can contain elements of different lengths. In order to match the dimension of the test set, we will build the data such that it has a shape of *(num_samples, 200)* and the targets *(num_samples, 9)*. We called *window* the length of the sequence given as input to the model, and *telescope* the time steps it will forecast.

To do so, we need to process each time series individually: we start from the end of the sequence and check if there are at least 209 (*window* + *telescope*) valid time steps. If this is the case, we add the first 200 time steps to the data and the last 9 time steps to the targets. Then, the window is moved backwards in time by a factor called *stride* (in the end, we used a stride of 50) and the process is repeated. Whenever there are not enough valid time steps in the current sequence (and the sequence is longer than the stride) it

is padded to reach a length of 209 and added to the data and to the targets in the same way, otherwise, it gets dropped. Then, we proceed to the next time series in the dataset.

We used multiple plots like the one below to manually verify that the creation of the window and the targets was consistent. From it, we can see that the original sequence before the splitting and the concatenation of data and target after the splitting perfectly overlap.



Each dataset time series had already been scaled in the range [0, 1]. We experimented with multiple approaches to normalize it further. The goal was to transform the data so that it had desirable statistical properties, like a zero-mean and a unitary-variance. To do so, we can use a *RobustScaler*, which removes the median and rescales the data according to the interquartile range so that the transformation is not sensitive to outliers.

Normally, we would scale each feature individually but the availability of only one feature and the presence of padding made it unfeasible to directly exploit the scikit-learn RobustScaler, so we manually implemented it. We tried two different approaches to do this: the first one consisted in flattening the whole dataset into a single vector, removing padded values and computing the median and the interquartile range and using them to scale the data. The second one consisted in a normalization w.r.t. the time steps: for each time step, we computed each median value and its interquartile range, ignoring padding values. Then, we rescaled each time step individually using this set of values.

However, none of these two normalization techniques have proven to be effective. The model was still capable of learning quite well but reached a lower performance w.r.t. non-normalizing the data. We also tried to rescale each time series individually in the range [0, 1] after the custom RobustScaler normalization, but with no success. Thus, we decided to train the models without applying any normalization to the dataset, since it was the best-performing configuration in our scenario.

At this point, we followed the common practice of splitting the data into train and validation sets, using *stratified sampling* to also consider the category of the original time series. We also discussed preparing our own test set but we then decided to only use the hidden test set on CodaLab.

The Models Journey

Before starting to consider modeling we set the baseline performances (regarding MSE and MAE) considering constant values for all the nine time steps to predict. Specifically, we considered the last time step ‘observed’ and used that value as the constant prediction. We obtained an MSE of 0.0166 and a MAE of 0.0804 with this constant predictor.

The first architecture we experimented with was RNNs, which are designed to learn and capture sequential dependencies in data (i.e. temporal patterns). RNNs have a form of memory which consists of a hidden state that is updated at each time step, which allows them to capture long-term dependencies. We considered both *Long Short-Term Memory* cells (LSTMs) and *Gated Recurrent Units* (GRUs), finding more success in using GRUs (specifically bidirectional GRUs).

We also explored 1D-CNNs, which are very effective in capturing local patterns and features within a sequence. Furthermore, convolutions are translation invariant, which means that they can detect patterns regardless of their position in the input sequence. We did not find much success in the use of 1D-CNNs by themselves. Instead, the combined use of GRUs and 1D convolutional layers proved to be very effective. At this point and with this model (after some hyperparameter tuning), the MSE score on CodaLab was 0.0053.
A significant improvement was made when we inserted *skip connections* in the convolutional layers, effectively building a *ResNet-like feature extractor*, whose output was fed to a single bidirectional GRU layer followed by a dense layer. With this model, we reached an MSE of 0.0048 on the CodaLab test set.

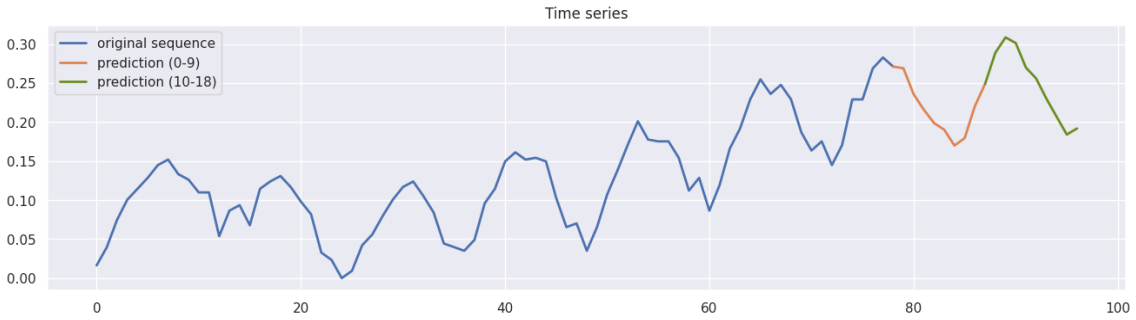
Then, we experimented with *Temporal Convolutional Networks* (TCNs), introduced in this paper. They are a variant of regular CNNs based on *dilated causal convolutions* and skip connections, and they aim at increasing as much as possible the *receptive field* of the network while keeping it as simple as possible. We were able to build a very robust TCN using the code from this library, with a MSE of 0.0061 on both our training and validation sets. To accomplish this, we performed hyperparameter tuning (a random search) using the library Weights & Biases, tuning the number of filters, the kernel size and the dropout rates. However, this model showed a lower performance on the test set, and therefore it was abandoned.

Lastly, we attempted to build a model that included the *attention mechanism*. We implemented it with a custom computation on the input of attention weights, which are then multiplied by the input itself and fed to a feature extractor made of bidirectional LSTM layers and followed by a dense regressor. This model proved very effective since it reached an MSE of 0.0049 on the CodaLab test set.

The last architecture that we considered was the original *Transformer* from the Attention Is All You Need paper. Even though we achieved a validation loss comparable to our best models, this model performed significantly worse on CodaLab, so we opted to not continue development on this path, especially considering that training time for the Transformer models was high.

Building an Autoregressive Model

Considering that the number of samples to predict differed from Phase 1 to Phase 2, we built an *autoregressive* model to create an arbitrary number of predictions from a previously trained model. We did this so that we wouldn't need to retrain a network for Phase 2.
The autoregressive model uses the original time series sequence to generate the first batch of predictions and then uses its output as input to produce another batch of guesses. This process can be repeated as needed. In our case, the autoregressive model produces two batches of 9 samples each, returning 18 predicted samples in total.



Ensemble of models

We expected that combining different architectures (e.g. RNNs to capture long-term dependencies, 1D-CNNs to capture local dependencies and Transformers to identify the most influential part of the input for the prediction) would be very beneficial in an ensemble model.

Ensemble methods are techniques that combine multiple models to achieve better performance. At the end of our work, we decided to use bagging, which consists of training multiple models on different data (or different models on the same data) and combining their predictions. We build the ensemble using our three best-performing models and averaging together their predictions.
This approach proved to be very effective since we saw a large improvement in the performance. The ensemble of models was able to reach an MSE of 0.00430 on the phase 1 CodaLab test set and 0.00903 on the phase 2 CodaLab test set.

Conclusion

The best model was obtained by ensembling the ResNet-like model, the Attention model, and the GRU-Conv1D model. The model provided a MSE of 0.0043 on the phase 1 test set and a MSE of 0.0090 on the phase 2 test set.
Concluding, we would like to discuss a possible approach that we considered during the development phase, but that we decided not to pursue. Such an approach involves creating a model for each category of the time series dataset, and then using that specific model to predict the test data, since the category of the sequence was given. We soon abandoned this idea because of two main reasons. The first one is that the limited number of samples belonging to category 'F' made it difficult to create a robust predictor for that category of sequences. The second reason was related to the time it would have taken to train all the different models, and then to create an ensemble of models for each category.

Contributions

- **Andrea Federici**: Data Preprocessing, Model Selection (Attention Mechanism, Transformer architecture), Autoregressive model construction
- **Umberto Salvatore Ficara**: Data Preprocessing, Model Selection (ResNet-style feature extractor + GRU training)
- **Alberto Pirillo**: Data Preprocessing, Model Selection (GRU + Conv1D & TCNs), Ensembles building

All the work was defined together and shared equally among team members. All the training and hyperparameter tuning was done individually. All notebooks were shared among all group members.