Artificial Neural Networks and Deep Learning Homework #2 Report

Alireza Nadirkhanlou | alireza.nadirkhanlou@mail.polimi.it
Ghodrat Rezaei | ghodrat.rezaei@mail.polimi.it
Riccardo Tomada | riccardo.tomada@mail.polimi.it

Introduction

In this challenge, we were given a multivariate time series classification problem. The given dataset consisted of time series data divided into seven different channels. The task was to predict the next sequence of this data into a finite horizon (to be precise, 864 timesteps).

Our efforts to perform this task could be summarized as below:

- Trying the architectures already introduced in the lab sessions, in order to establish a baseline performance
- Using more advanced models introduced during the lectures
- Attempting to implement more sophisticated, state-of-the-art architectures and adapt them to our problem

Next, we will describe the environment in which we ran our experiments, and the changes we had to make to the code provided in the lab sessions in order to optimize the usage of the environment's resources.

The Environment

Similar to our submissions for the first homework, we decided to use kaggle again for this challenge.

As for the data preprocessing, we experimented with both min-max and std-mean normalizations. We used a train-validation-test split of 0.70-0.15-0.15.

Regarding the preparation of the input sequences, we first tried the `build_sequences` function present in the lab notebooks, but doing so, we encountered an overflow of memory when we were trying to build sequences with relatively great window sizes but small strides. As a result, we needed a smarter and more efficient way of building sequences. We found our solution in this tutorial, in which a `WindowGenerator` class is built to deal with the generation of windows. Using this class, the windows are only generated when the model requests the next window from it, and this way it solves our memory overflow problem.

However, we did not completely discard the `build_sequences` function, because our `WindowGenerator` was not compatible with parts of the lab notebook code after the training, and in order to make our own lives easier, we used this generator for the evaluation and testing part of our experiment.

For more information about the smaller details of our environment and settings such as callbacks, etc., you can refer to the notebook attached along with this report.

Convolutional Bidi-LSTM Model (+ Autoregression)

This architecture is the same one presented in the lab sessions, and we chose it in order to establish a baseline performance for our future attempts.

The main part of the architecture consisted of two consecutive blocks of Bidi-LSTM + Conv1D layers, the former followed by a MaxPooling1D layer and the latter followed by a GlobalAveragePooling1D layer. The layers are further connected to a Dropout layer, a Dense layer, and another Conv1D layer.

We experimented with the following parameters during our attempts with this architecture:

- Window size (`INPUT_WIDTH` in the notebook)
- Telescope (`OUT_STEPS` in the notebook)
- Sequence stride (`STRIDE` in the notebook)
- Batch size (`BATCH_SIZE` in the notebook)

First, we tried predicting the entire output sequence (telescope size 864). The results we achieved were not promising, and the predictions were almost a straight line, unable to predict the oscillations of the data. However, we noticed that the stride can noticeably affect performance, and we were able to achieve slightly better results (in terms of loss) by changing the stride value.

In addition to direct forecasting, we also tried to use the autoregression technique to see if the results would improve. In this attempt too, we failed to get acceptable results. We also observed that the predicted values diverged when the telescope was too little.

Attention Mechanism

In our second attempt, we used the transformer encoder-decoder architecture with the attention mechanism.

This architecture consisted of two main parts: the encoder blocks and the decoder blocks. Each encoder block, in turn, consisted of an attention+normalization part and a feed-forward convolutional part. The attention part of the encoder has a `MultiHeadAttention` layer with a parameter that determines the number of *heads*, and it is one of our hyperparameters in this architecture. The decoder block in our architecture is simply a multilayer perceptron (i.e. a dense layer).

Our main challenge in this part was the memory consumption. Even with our `WindowGenerator`, the model would run out of memory given windows with relatively big width (about 2000 steps), even when the telescope was small (around 100). This was due to the fact that the `MultiHeadAttention` layer performs cross-attention across the input vector, and thus the memory size increases by the second power of input width (window size).

Time2Vec

One further improvement we made was the addition of a Time2Vec layer (found here), which is a time embedding layer, to our architecture, which led to slightly better results with negligible overhead.

Finally, after experimenting with many different settings and hyperparameters, we were able to achieve our best performance using this architecture and the following hyperparameters:

- `BATCH_SIZE`: 32
- `INPUT_WIDTH`: 800
- `OUT_STEPS`: 10

- `STRIDE`: **10**
- `ATTENTION_HEADS`: **12**

You can see the predictions (before denormalization) in **Figure_1**.

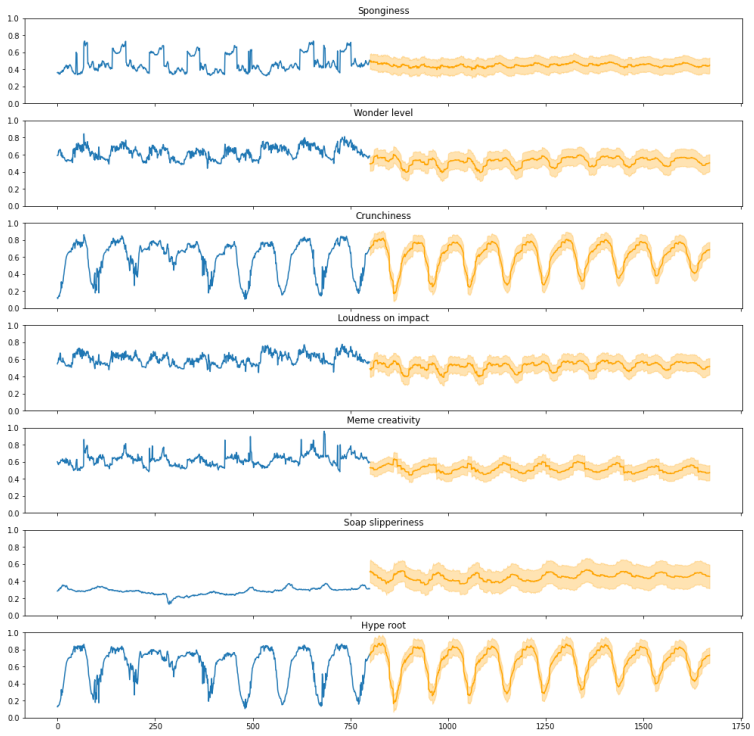

Figure_1 -Autoregressive forecasts of the Attention + Time2Vec model

Further Attempts

In addition to what has already been mentioned, we tried other methods and architectures, some of which we could not implement, and some did not improve our performance.

Seasonal Decomposition

Knowing that our data is a time series, we tried performing seasonal decomposition on our data in order to decompose it into trend, seasonality and residual values. Our strategy was to train the seasonality (which was not a simple recurring pattern) and the residuals separately, and in prediction, add them back together. Unfortunately, the results were not promising, and we gave up this idea.

Informer Architecture

After we faced memory limitations with the transformer, we came across Informer, which is an efficient implementation of the transformer architecture, in the paper presented here (and its implementation here). We made some adjustments to the source code in order to be able to run it in the Kaggle environment, but even with this architecture, we faced the same OOM errors that we encountered with the original transformer model.