LLM for time series forecasting

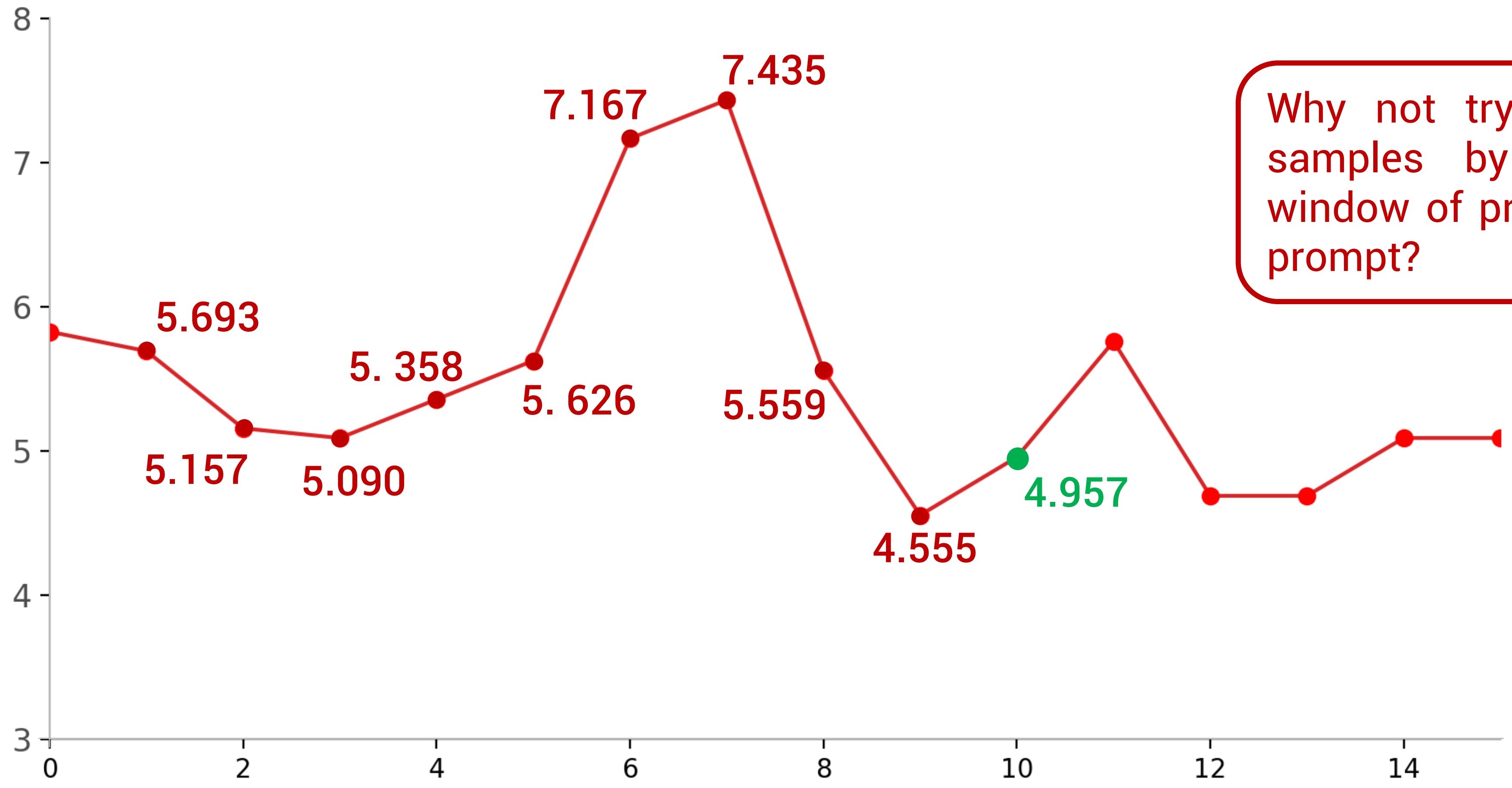
Group: Leonardo Bellini, Lorenzo Grandi, Eugenio Muscinelli

Tutor: Giovanni Battista Esposito

Goal

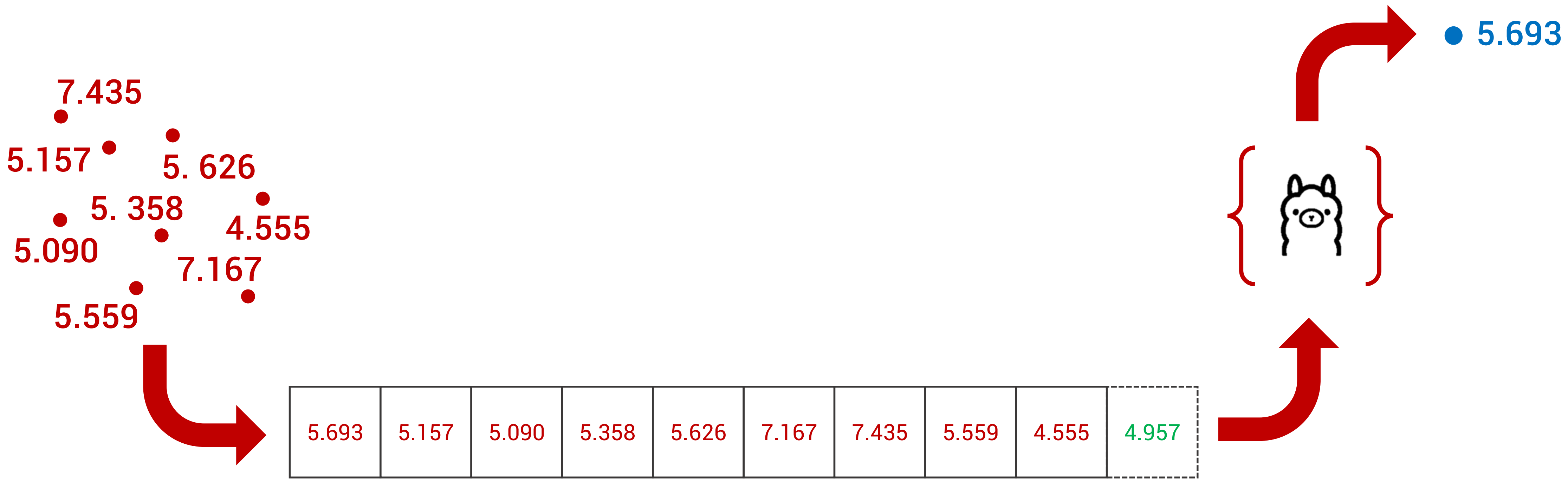
Using the LLama2 model, we want to predict the future samples of a time series representing the data telemetry coming from a node

The starting idea



Why not try to predict future samples by simply using a window of previous ones as the prompt?

The starting idea

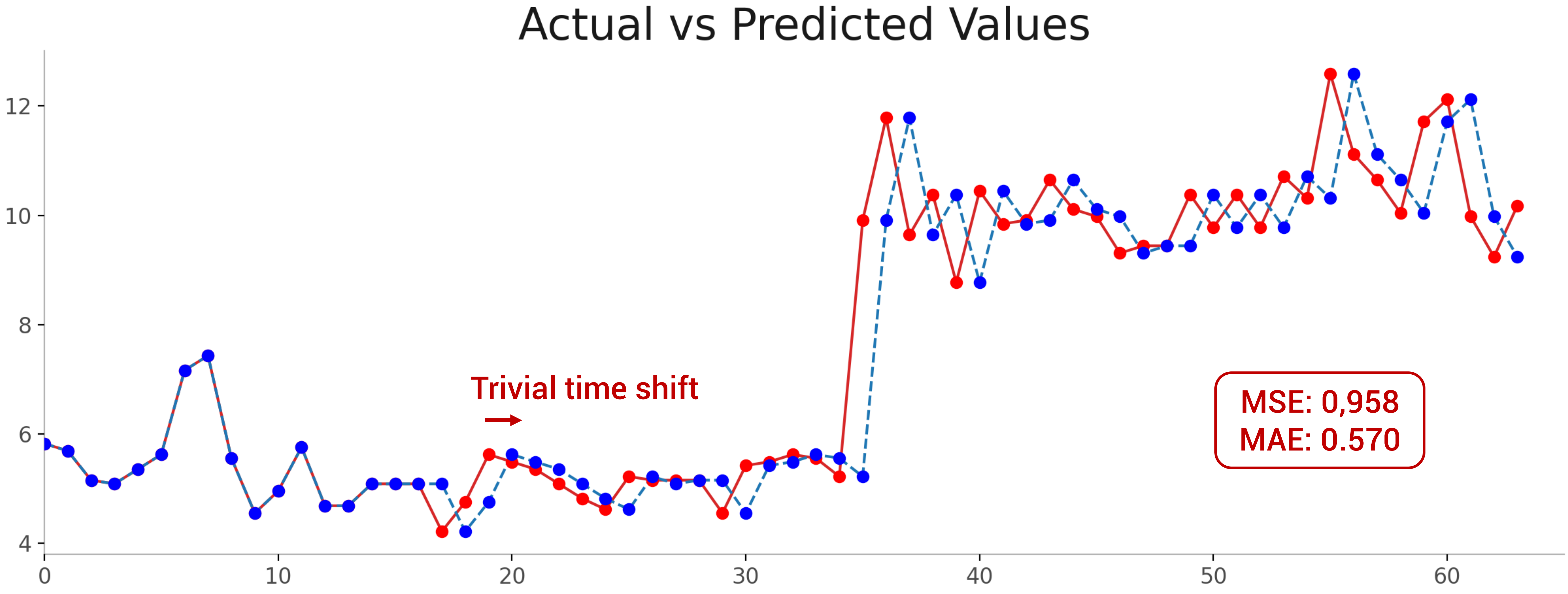


The starting idea

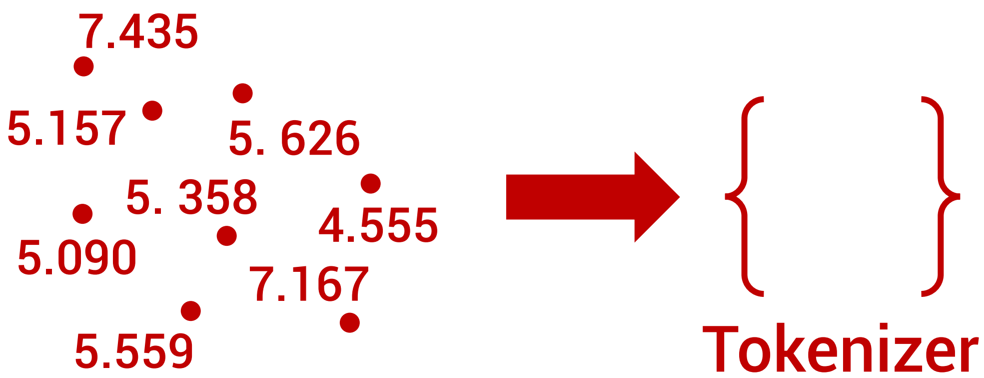
```
MODEL_ID = "TinyLlama/TinyLlama-1.1B-intermediate-step-1431k-3T"
pipe = pipeline('text-generation', model=MODEL_ID, device=0)

for batch_idx in range(num_batches):
    for i in range(batch_idx * batch_size, (batch_idx + 1) * batch_size):
        window = HUFL_df.iloc[i:i + w_dim]
        prompt = window[:w_dim - 1].to_string(header=False, index=False)
        out = pipe(prompt, max_new_tokens=6)[0]['generated_text']
        pattern = r'\b\d+\.\d{3}\b'
        out_float = re.findall(pattern, out)
        out.append(out_float[-1])
```

A window of the time-series is passed to the LLM as a prompt

The results

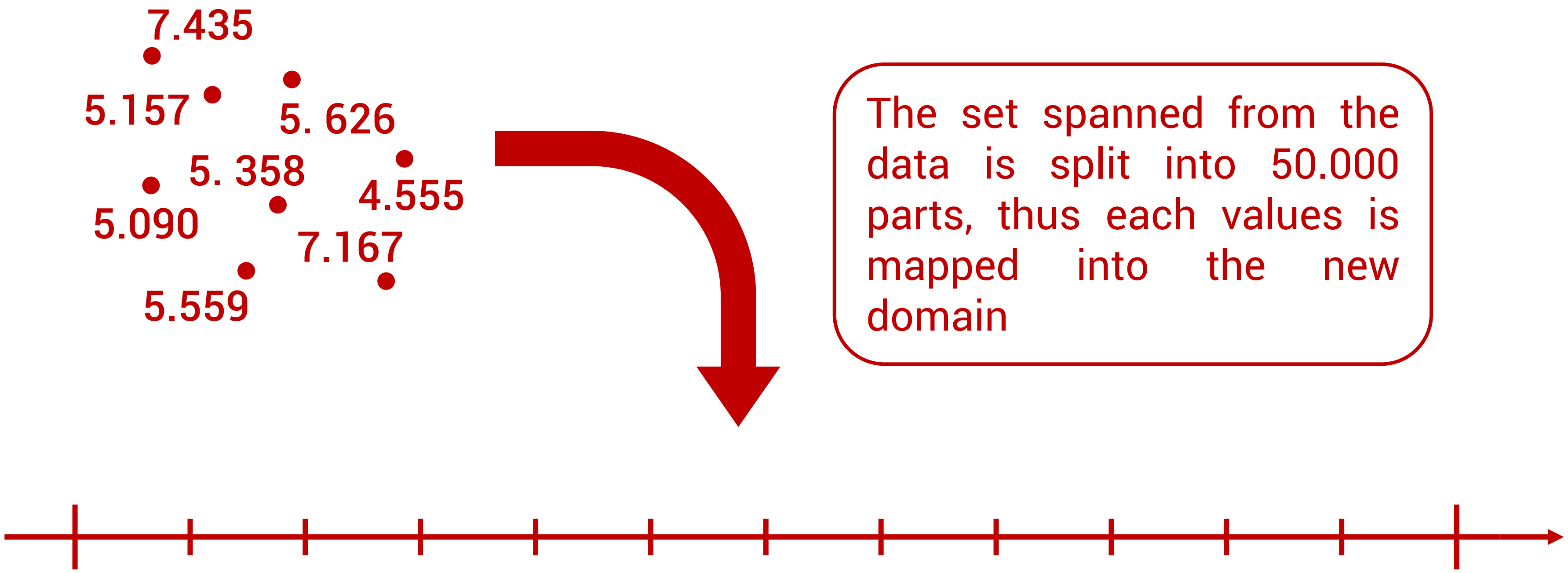

Actual vs Predicted Values
Trivial time shift
MSE: 0,958
MAE: 0.570
12
10
8
6
4
0
10
20
30
40
50
60

The complete network

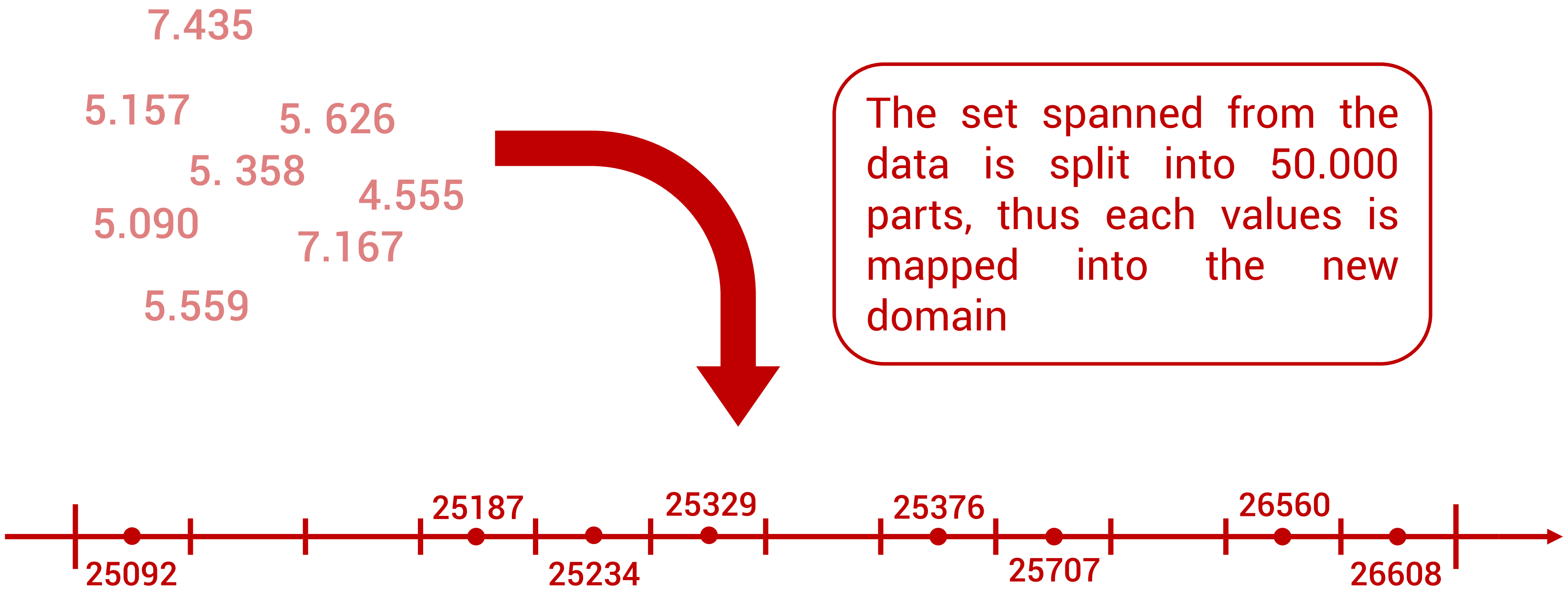


Tokenizer : splits the text into individual words (and subwords) and map each token to a unique identifier. In this way, the human-readable text is converted into a form that the LLM can process effectively.

Tokenizer


7.435
5.157
5. 626
5. 358
4.555
5.090
7.167
5.559
The set spanned from the data is split into 50.000 parts, thus each values is mapped into the new domain

Tokenizer



The set spanned from the data is split into 50.000 parts, thus each values is mapped into the new domain

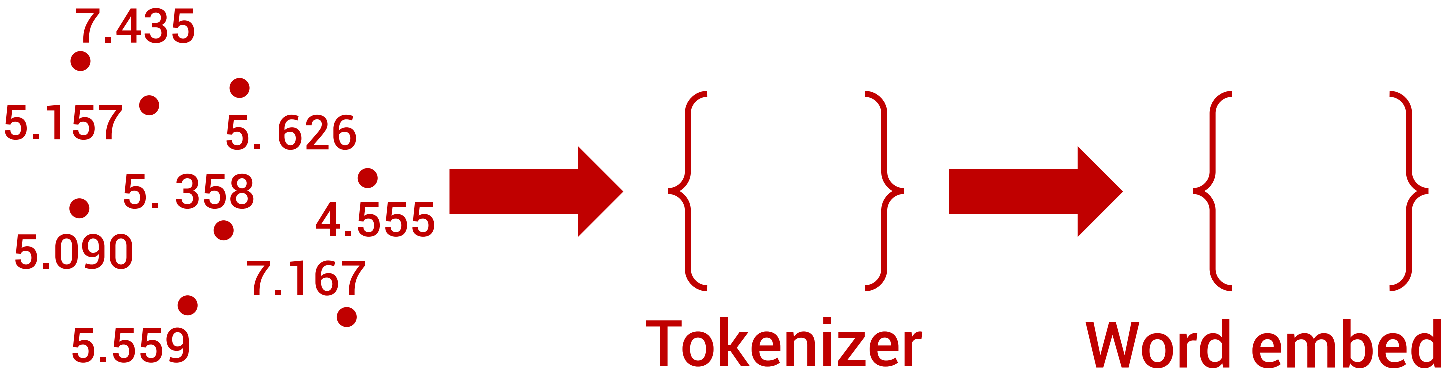
Tokenizer

```python
dataset = pd.read_csv('/kaggle/input/etth1234/ETTh1.csv')
dataset.drop(columns=['date'], inplace=True)
scaler = StandardScaler()
scaled_df = scaler.fit_transform(dataset)
L = 4000
train_df = pd.DataFrame(scaled_df[:L])
valid_df = pd.DataFrame(scaled_df[L:2*L])
test_df = pd.DataFrame(scaled_df[2*L:3*L])
train_HUFL = torch.Tensor(train_df[0].values)
valid_HUFL = torch.Tensor(valid_df[0].values)
test_HUFL = torch.Tensor(test_df[0].values)

class Tokenizer(nn.Module):
    def __init__(self, voc_size, lower_bound, upper_bound):
        super(Tokenizer, self).__init__()
        self.voc_size = voc_size
        self.lower_bound = lower_bound
        self.upper_bound = upper_bound
        self.discrete_values = torch.linspace(self.lower_bound, self.upper_bound, self.voc_size).to('cuda')

    def forward(self, dataset_std):
        quantized_values = (torch.bucketize(dataset_std, self.discrete_values) + 2)
        attention_mask = (~torch.isnan(dataset_std))
        return quantized_values, attention_mask
```

Standardization

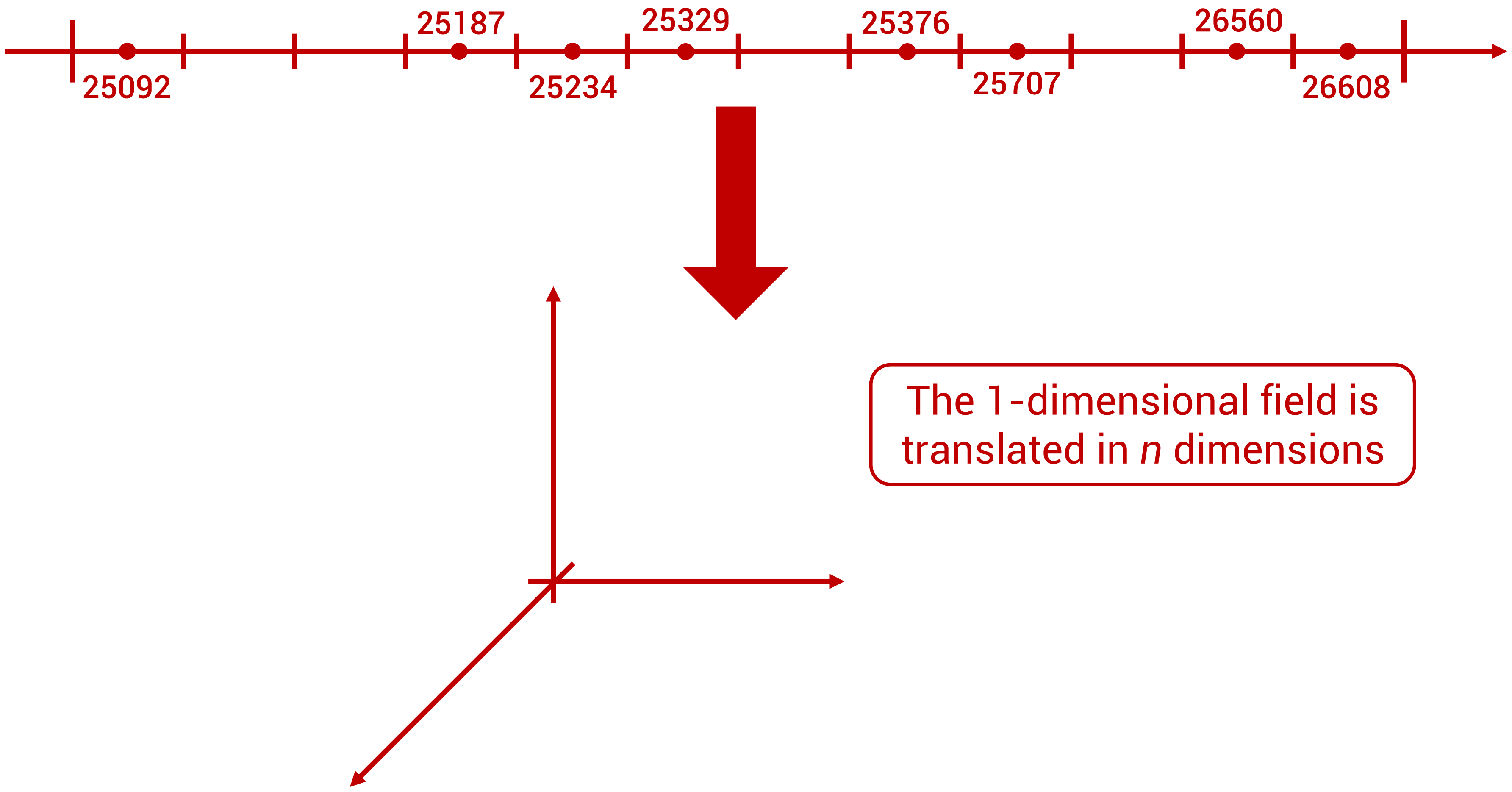
Quantization

The complete network



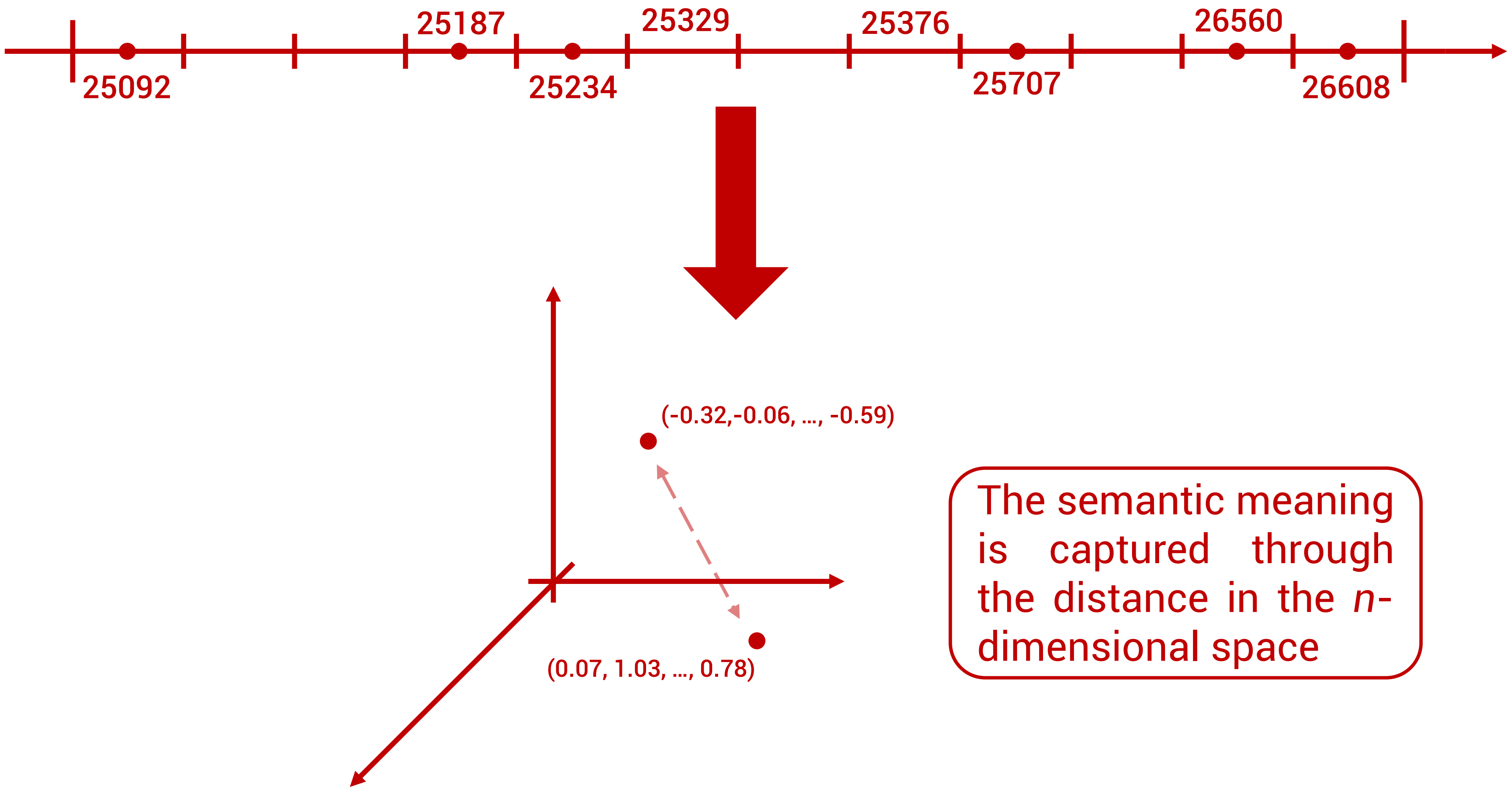
Word embedder : transforms the tokens into dense vectors of fixed size, named embeddings, wich capture the semantic information about the words

Word embedder



The 1-dimensional field is translated in *n* dimensions

Word embedder



The semantic meaning is captured through the distance in the *n*-dimensional space

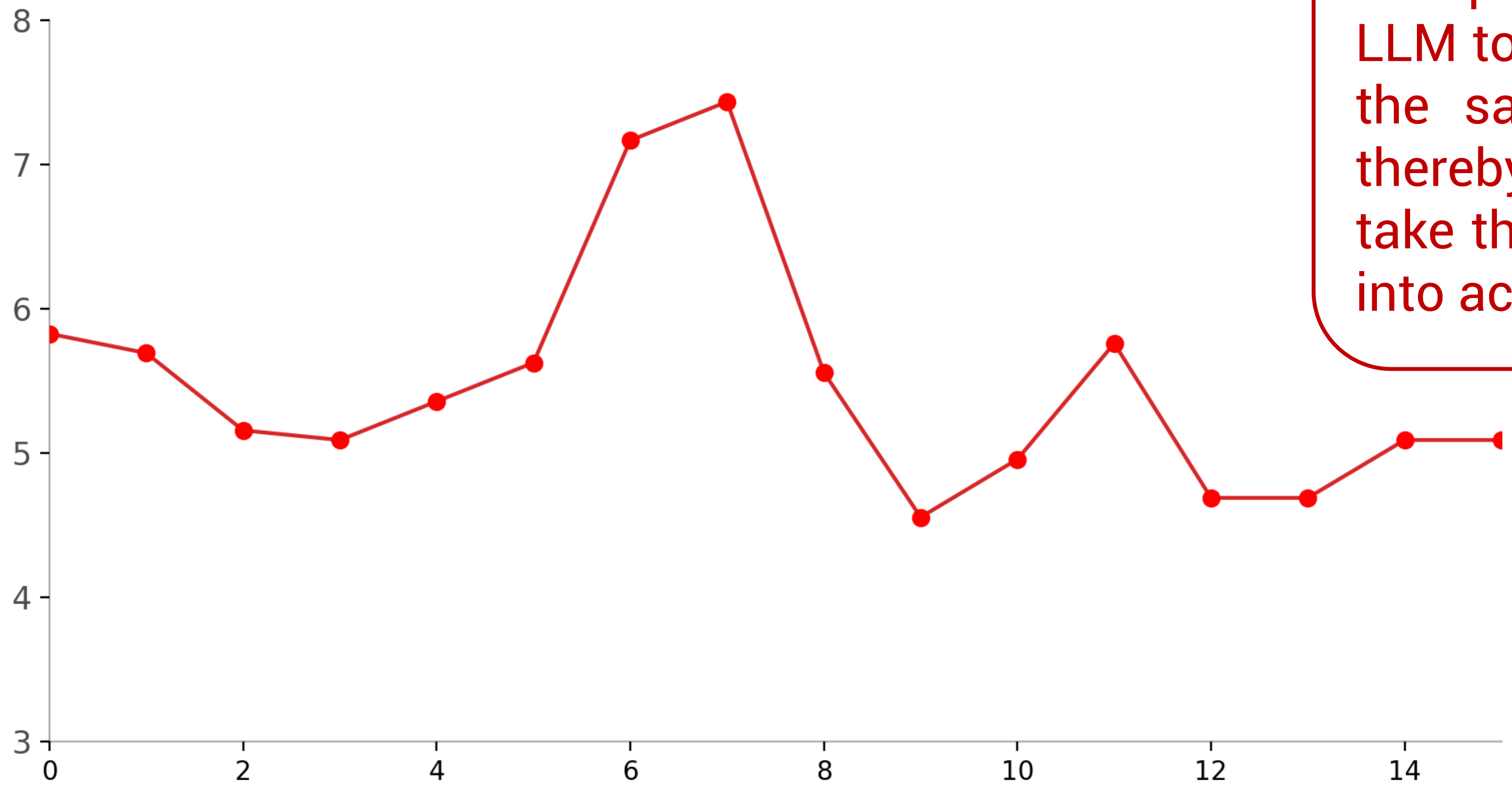
The complete network

Tokenizer

Word embedder

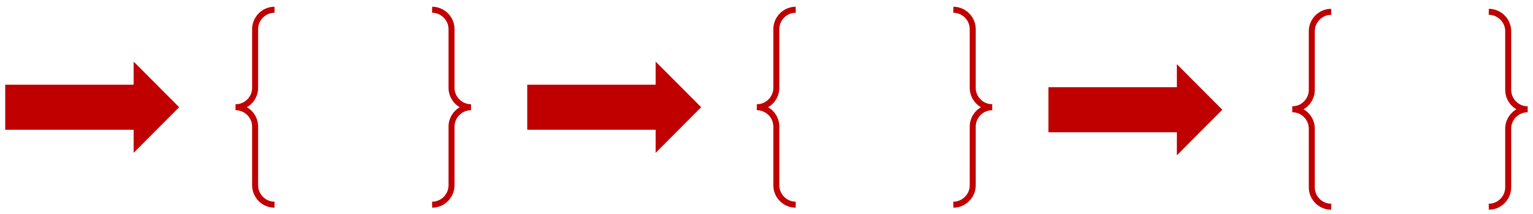
Positional encoder : is a mechanism that helps the model understand the order and position of tokens in a sequence

Positional encoder



The positional encoder allow the LLM to keep the semantic order of the samples in the time-series, thereby enabling predictions that take the history of the time-series into account

The complete network

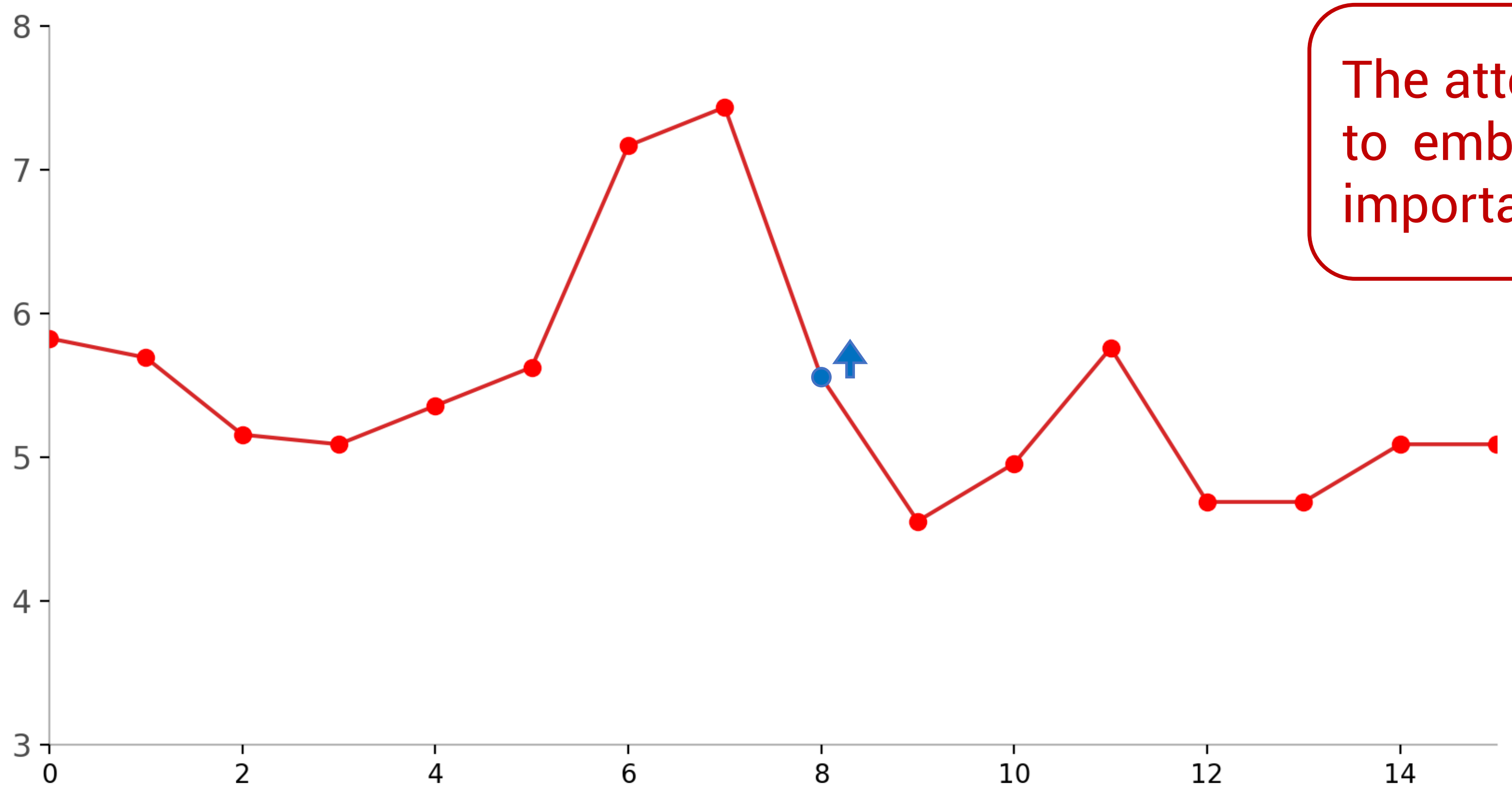


Word embedder

Positional encoder

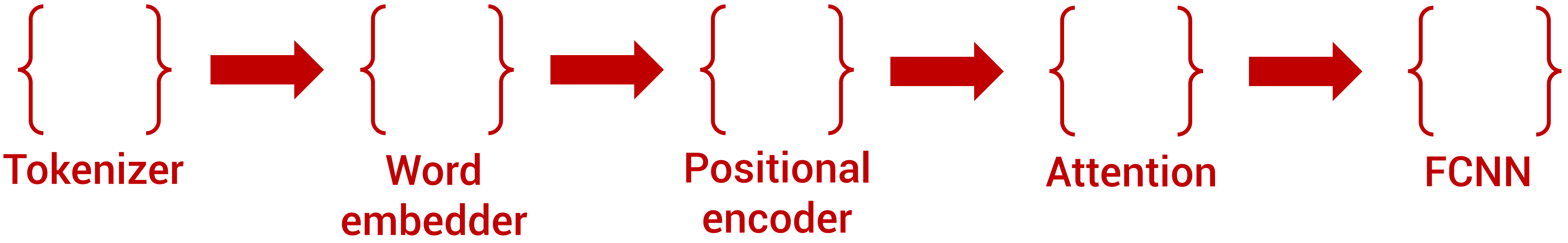
Attention : captures dependencies and relationships within the sequence and regardless of their distance from the current token

Attention layer



The attention layer assigns a score to embedding token related to its importance in the context

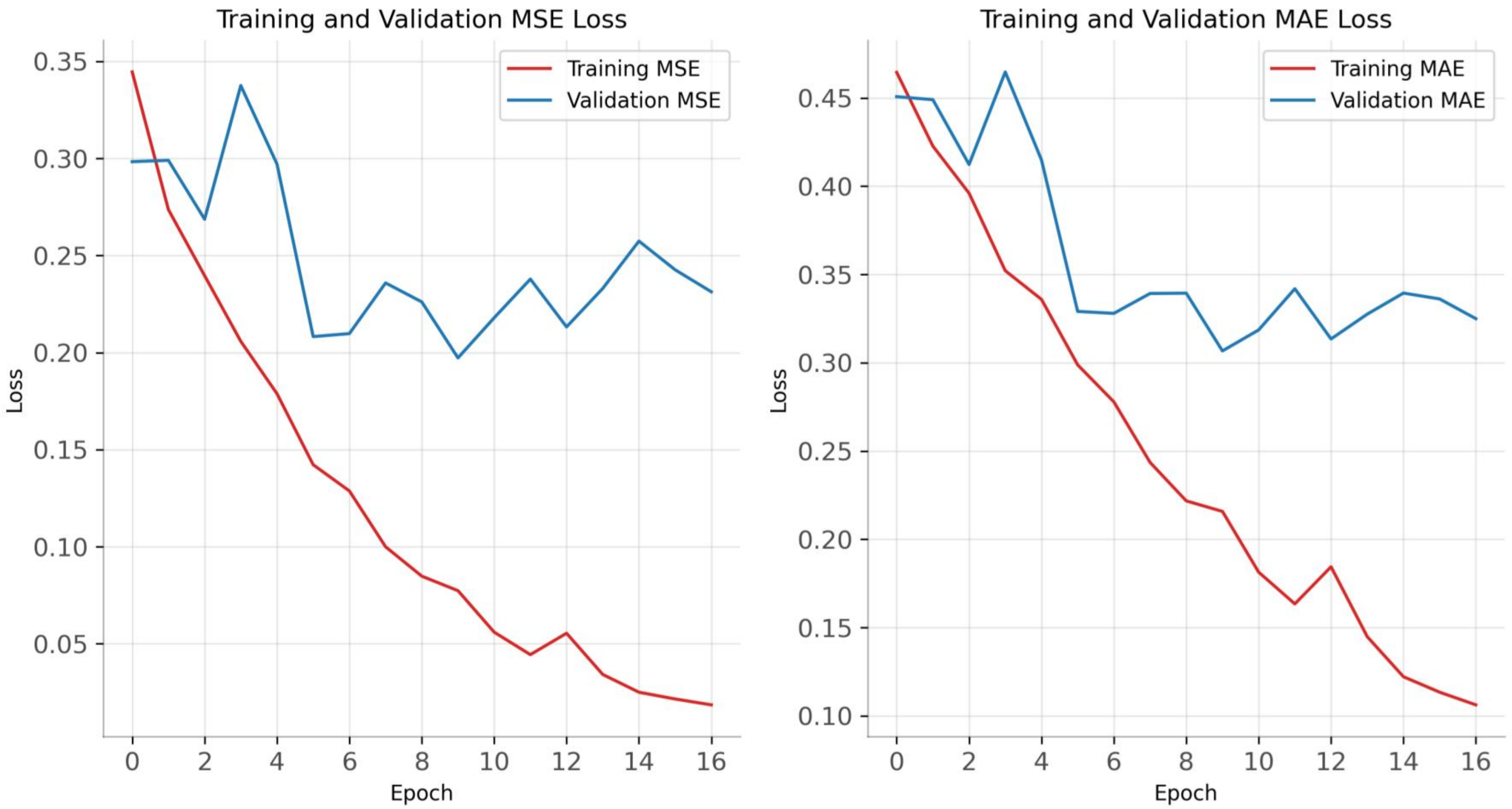
The complete network



Two Fully-Connected NN are trained to map the attention embedding to the predicticted value, acting as a detokenizer

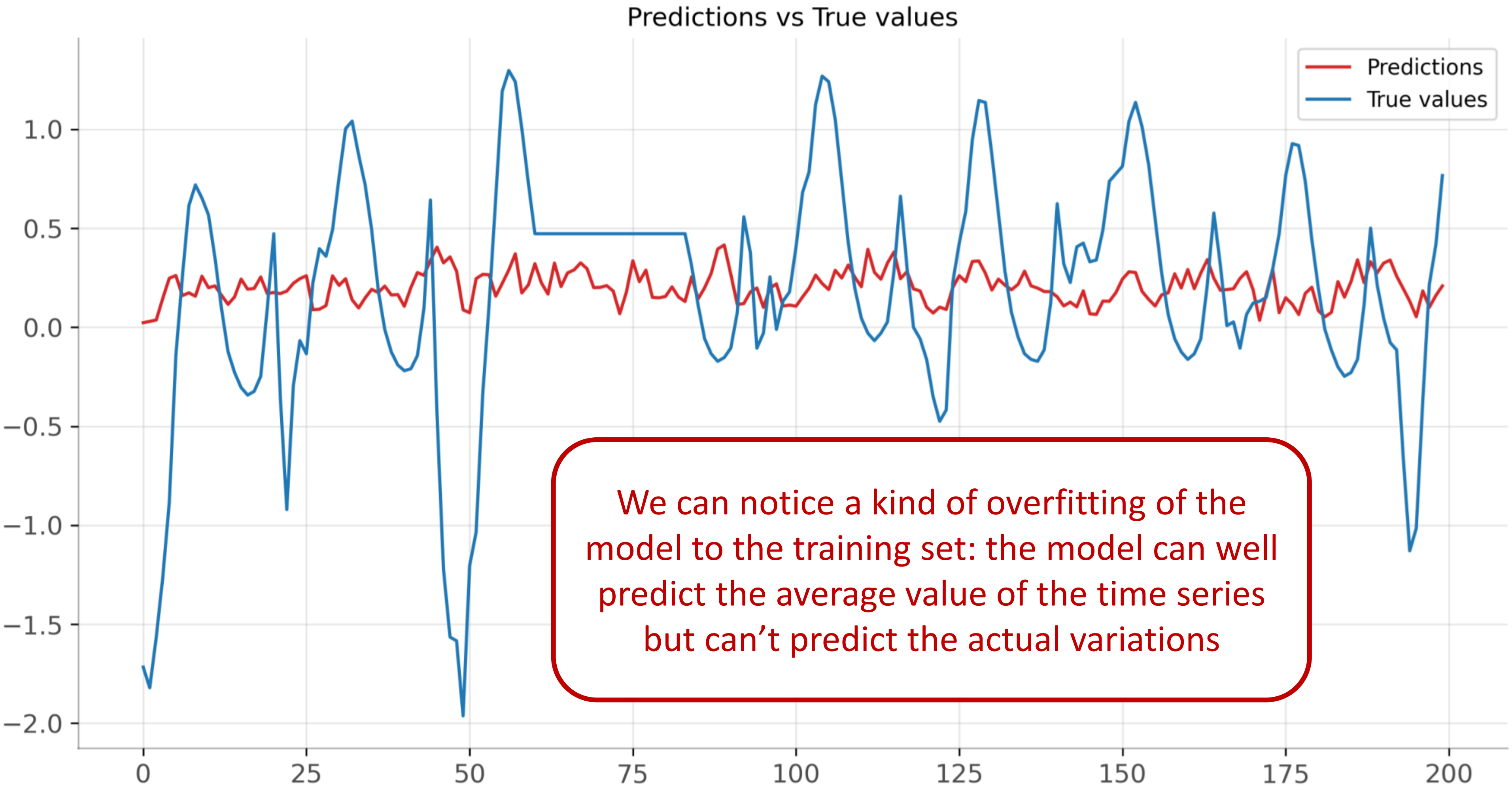
The results

NB: Tested on a second dataset, uncorrelated to the one used in the training phase



MSE: 0,524
MAE: 0,466

The results



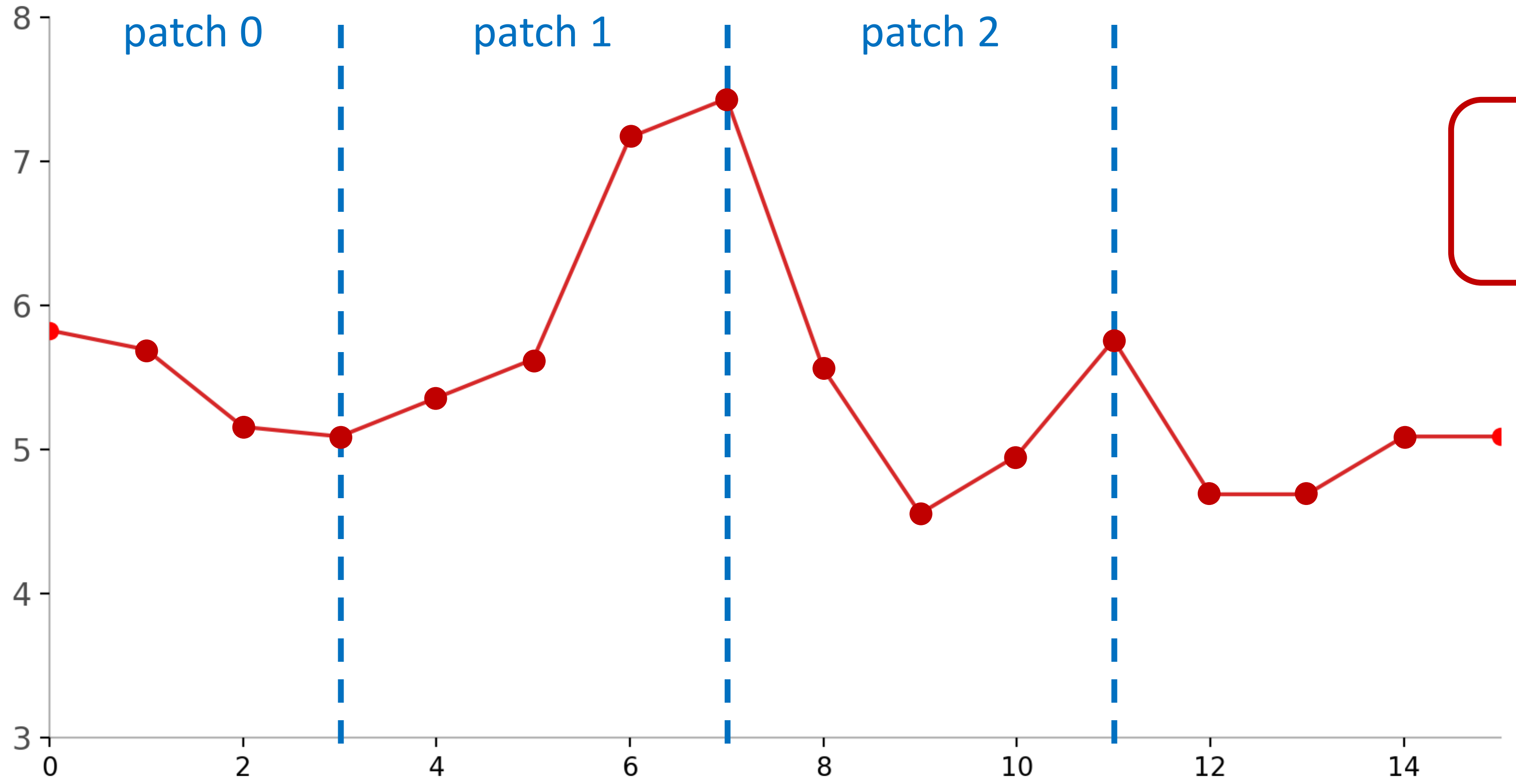
We can notice a kind of overfitting of the model to the training set: the model can well predict the average value of the time series but can't predict the actual variations

The alternative solution

Tokenizer → Replication Padding → Patch embedding → Attention → FCNN

(Ref: Time-llm: time series forecasting by reprogramming large language models)

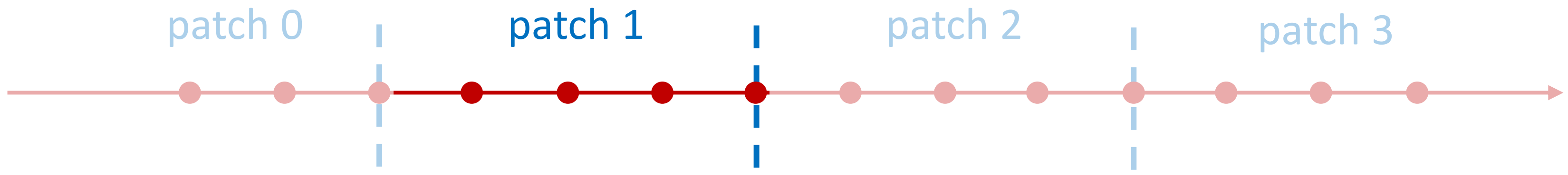
Patch embedding



The series is split in many patches with fixed length

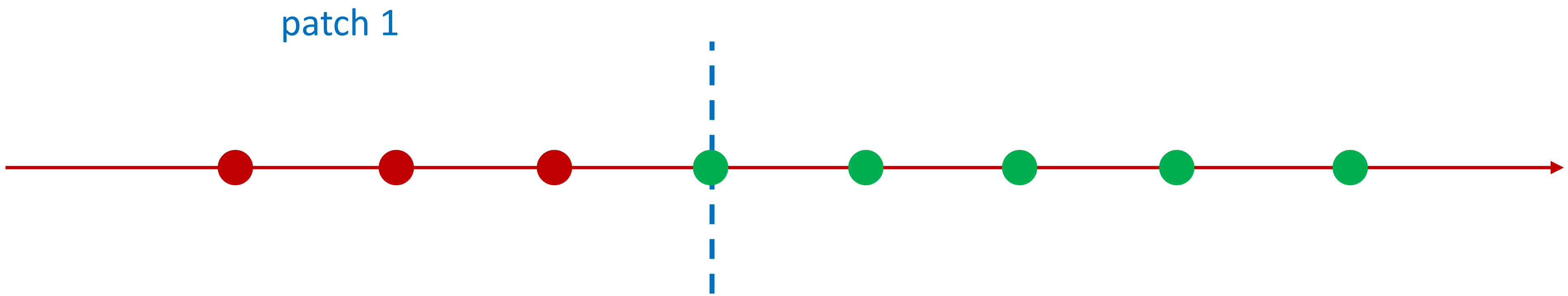
Patch embedding

The series is split in many patches with fixed length

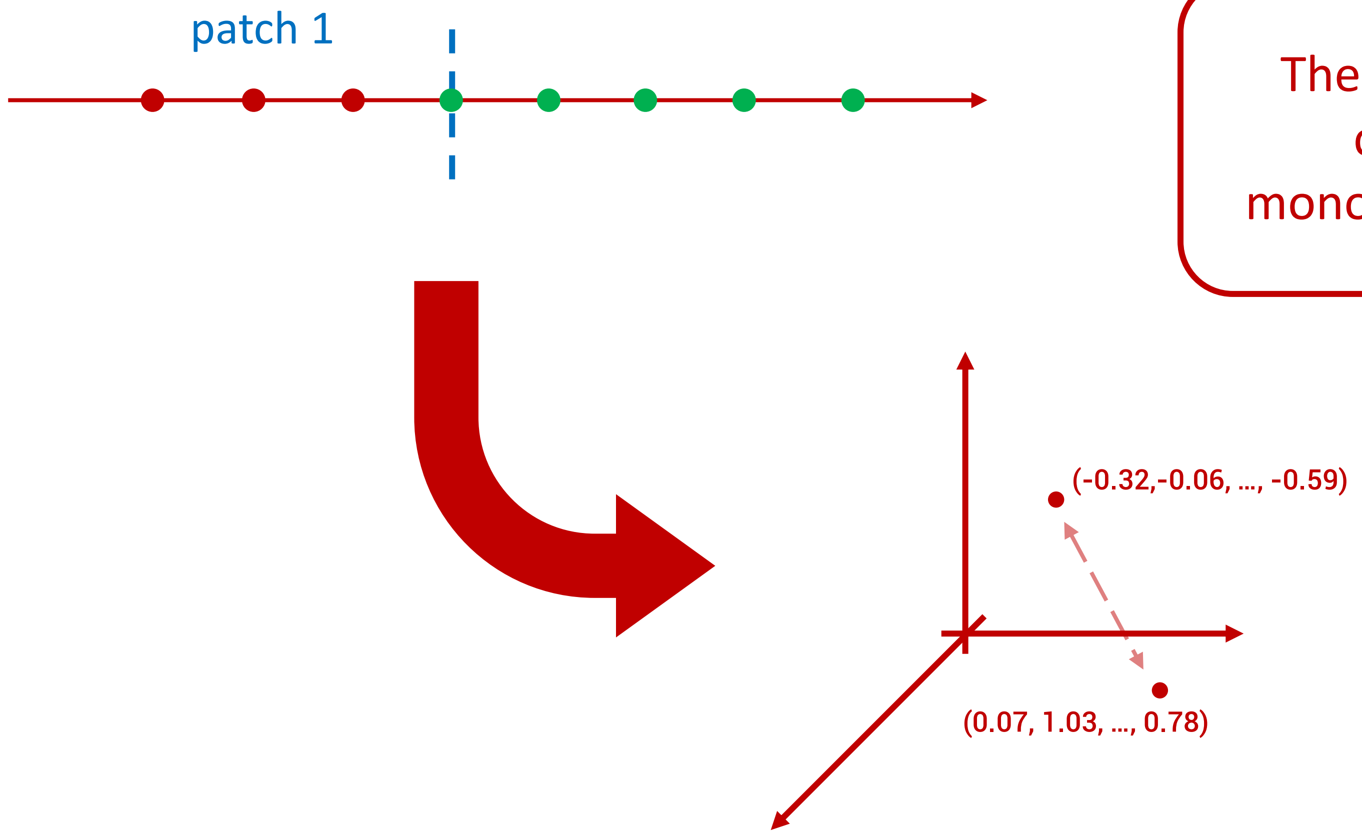


Patch embedding

The previous layer pad and stride the patch, adding the last value to highlight it

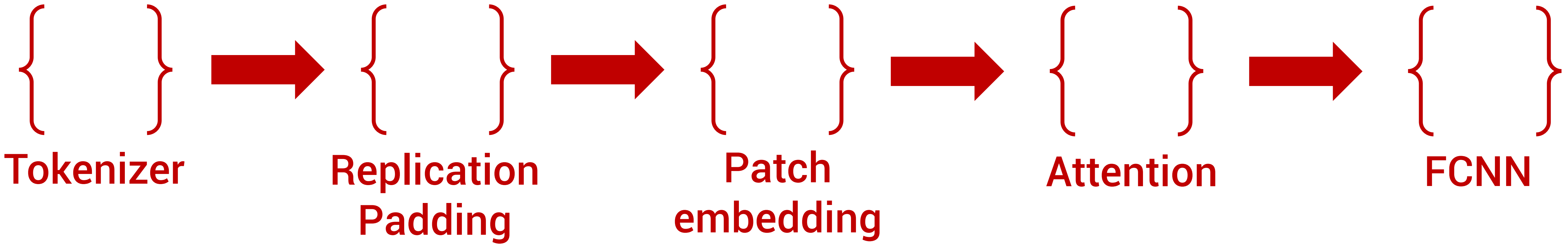


Patch embedding

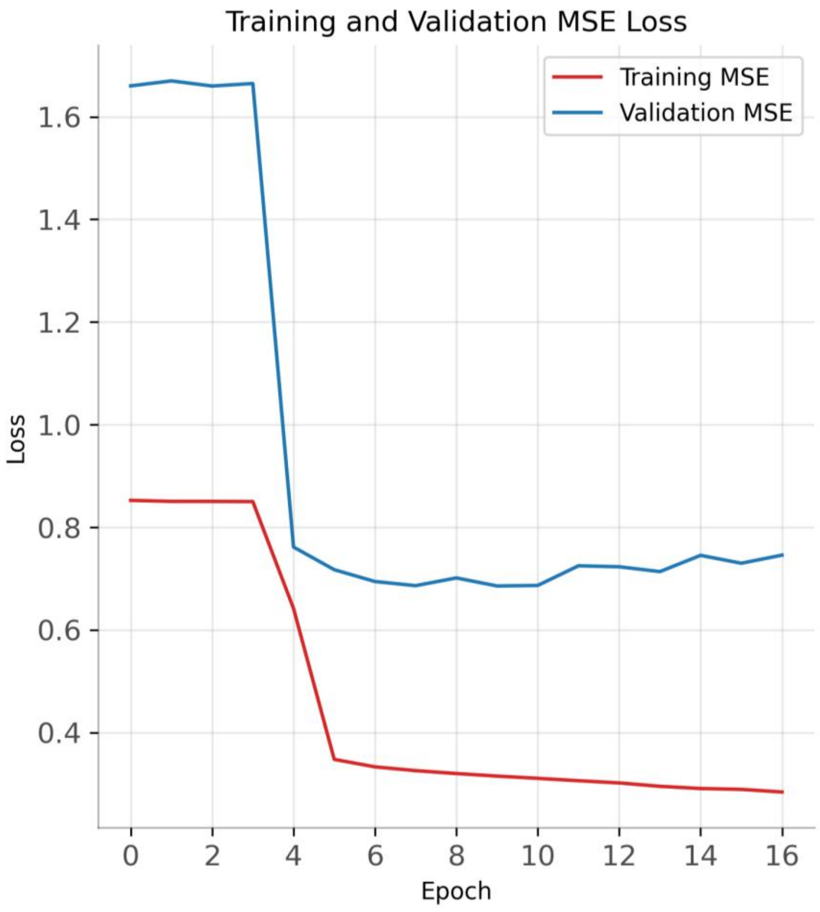


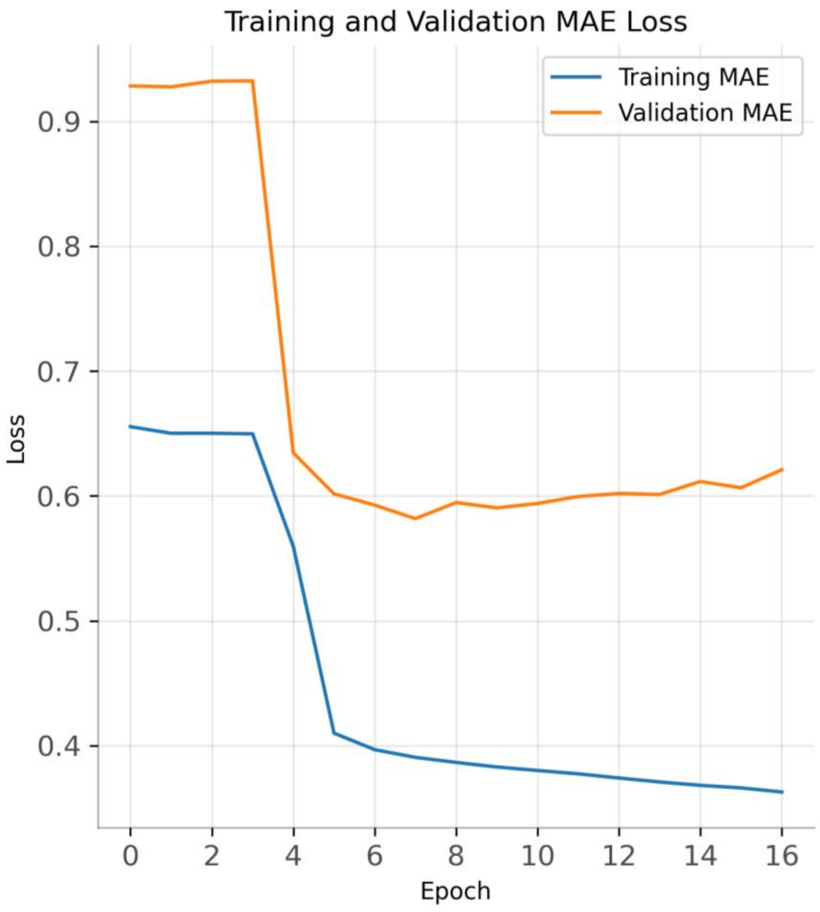
The embedding vectors are computed through a monodimensional convolution

The alternative solution



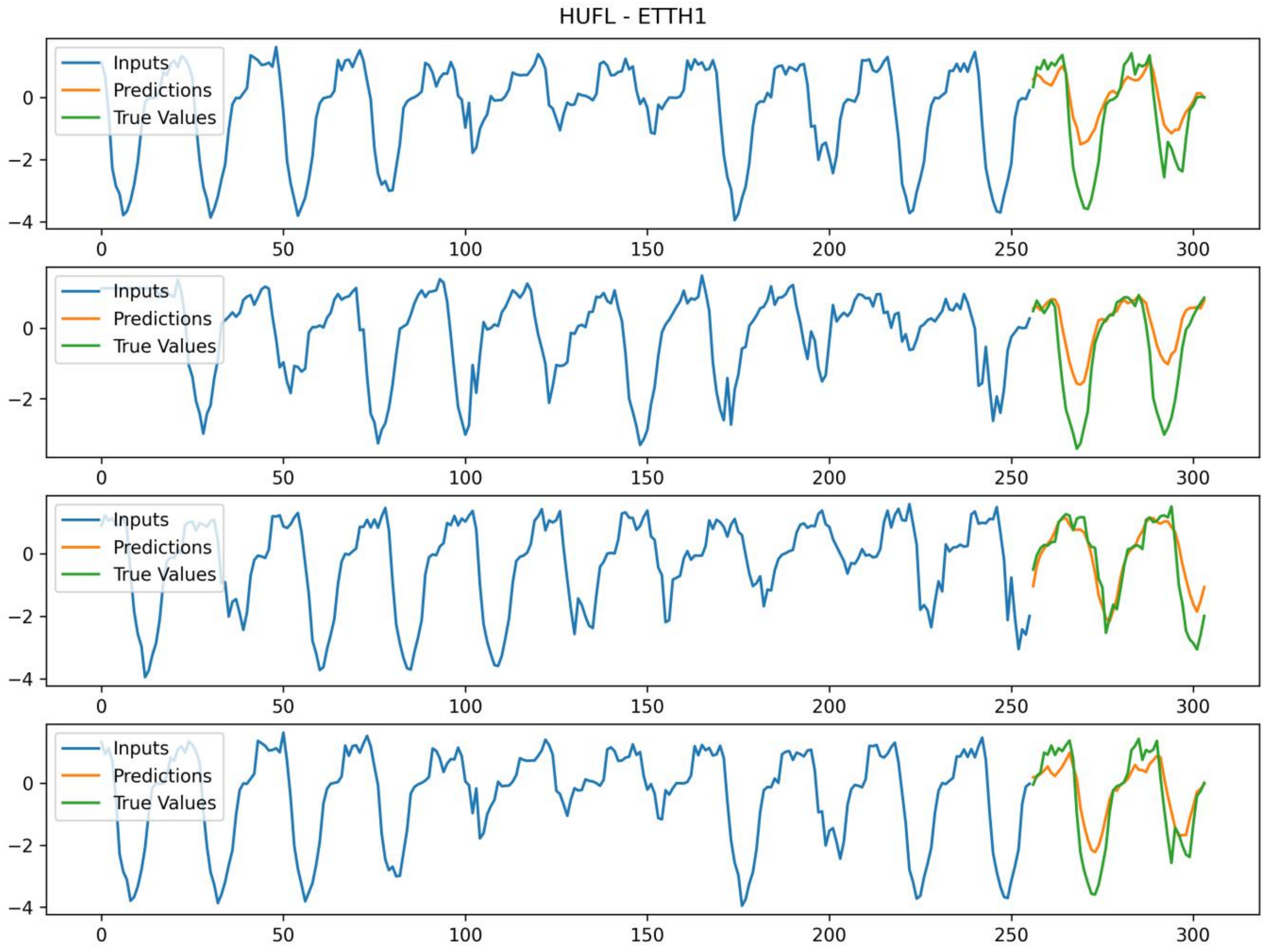
The results





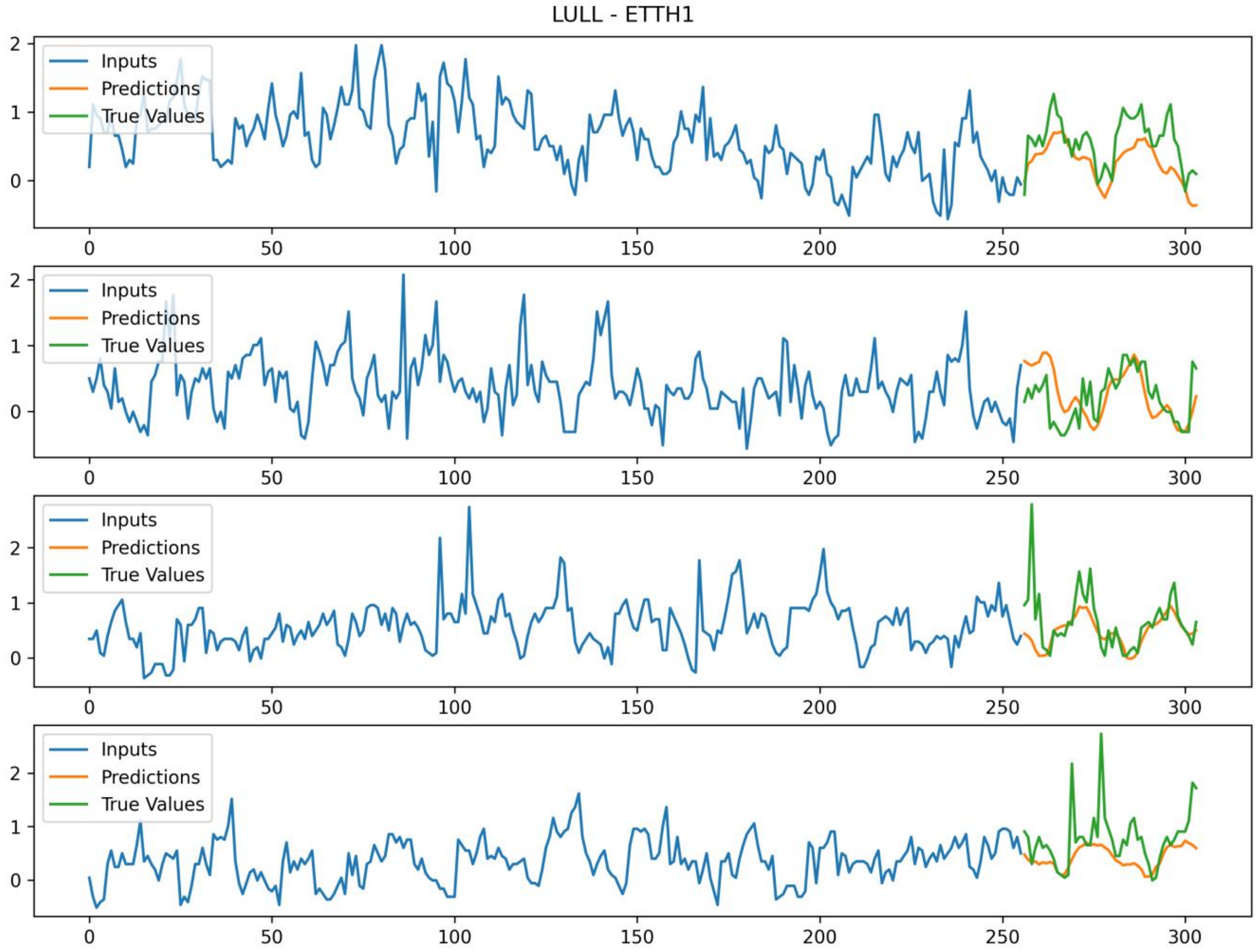
MSE: 0,536
MAE: 0,522

The results



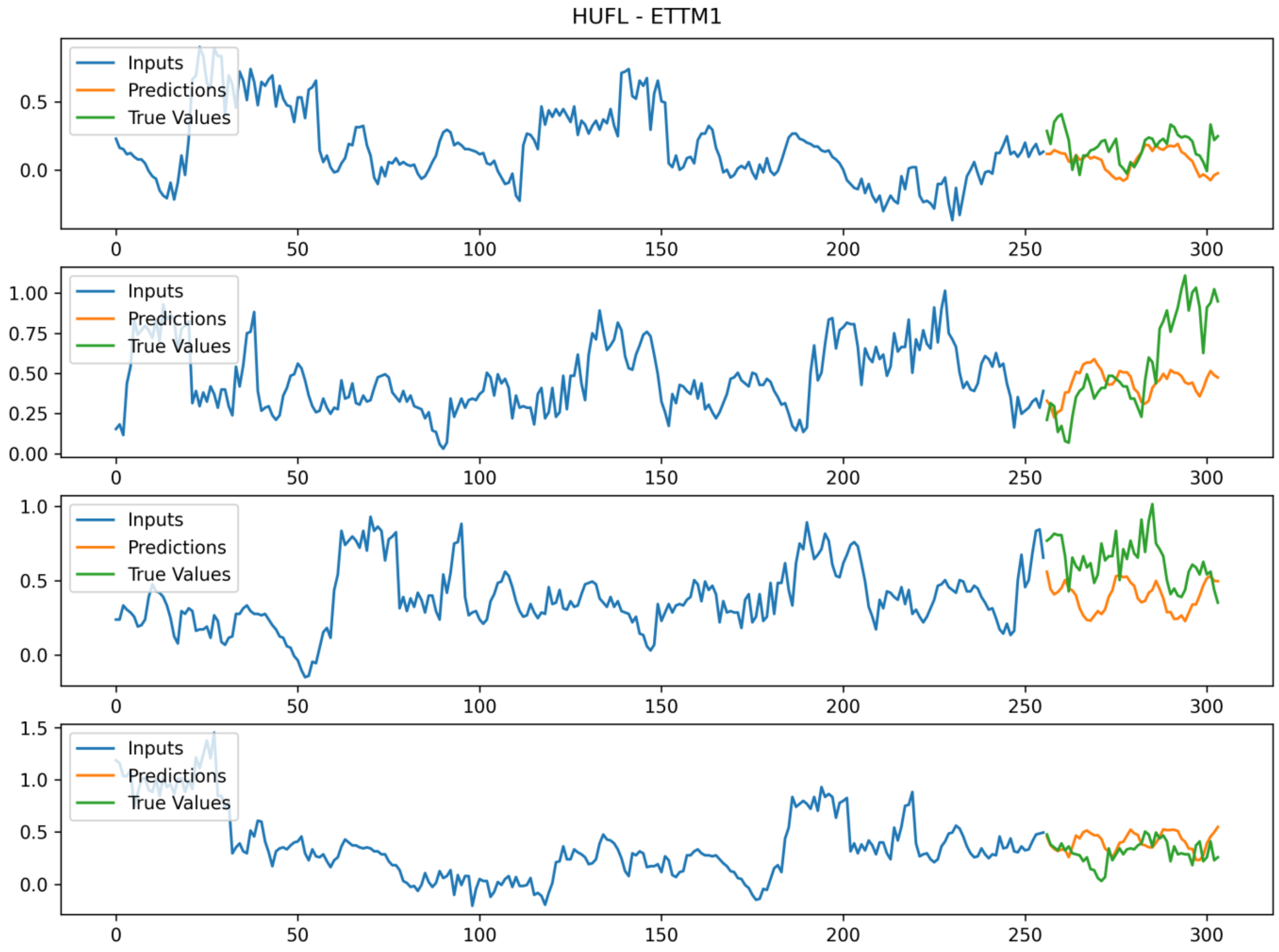
Even with the smallest model the prediction fit the true behavior

The results



Using models with a larger number of parameters would result into better predictions

The results



We tried to evaluate the performances on a different dataset, and the result isn't that different with respect to the previous tests

Possible improvements

From the model viewpoint:

- Larger pretrained model
- Quantization of the parameters
- More complex layers

From the system viewpoint:

- Split the network processing over more GPUs

Thank you for your attention