Time series Forecasting for Neural Networks

Cyril Mergny internship at LISN

supervised by Lionel Mathelin and Bérangère Podvin

1 Neural Network models for time forecasting

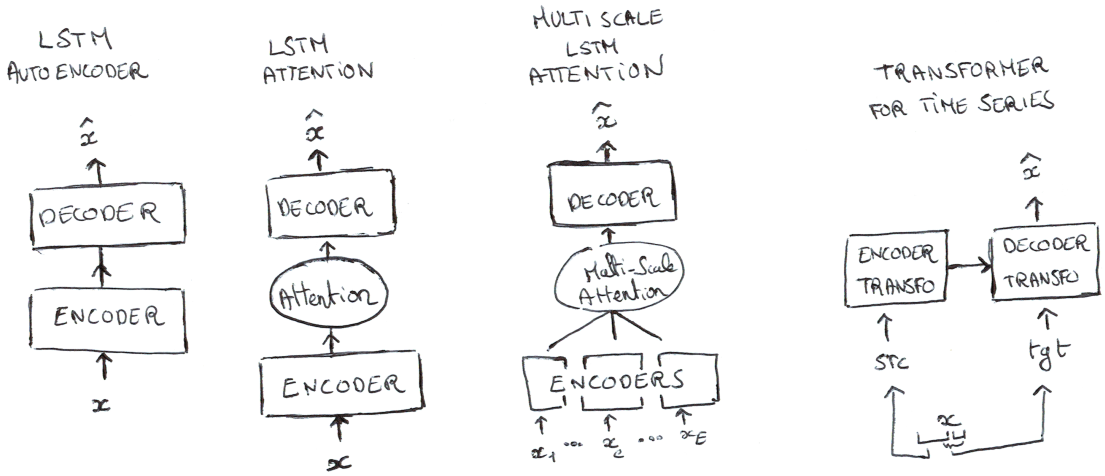

Figure 1: Over-simplified diagrams of the different models used in this study. These are given here for a general understanding of the models' behavior. For further and more rigourous representations, see figures below.

1.1 Classic LSTM Auto-Encoder

An instance of the LSTM Auto-Encoder model is called in this project using the lines:

```
# Import class
from models.LSTM_AE import LSTM_EncoderDecoder
# Initiate model
model = LSTM_EncoderDecoder(E, H).to("cuda")
```

where E is the number of modes and H the size of the LSTM hidden states.

LSTM Network have been built to solve the vanishing gradient problem of RNNs. The LSTM cell consist of multiple operations called gates. For time series forecasting, we require a sequence to sequence model. The LSTM auto-Encoder is the corresponding model for sequence to sequence problems.

We summarize the layers composing this LSTM in Figure (2). The model takes as an input an array $\mathbf{X}$ representing the time series $x_e(t)$ of each mode. The encoder hidden memory (the hidden state and cell states) are initalized to zeros. The LSTM outputs a hidden state h_S which will used to initalized the decoder memory. We input the last term of the time series, and according the target length desired T the decoding process will be repeated T times. The ouput of the LSTM decoder is a hidden states of size (1, N, H) thus it needs to be expressed to the input dimension E with the linear layer. The output $\hat{x}(t_{S+i})$ and memory h' of the decoder is then fed back to itself, making T prediction ahead in time of the input. Hence the ouptut shape is (T, N, E).

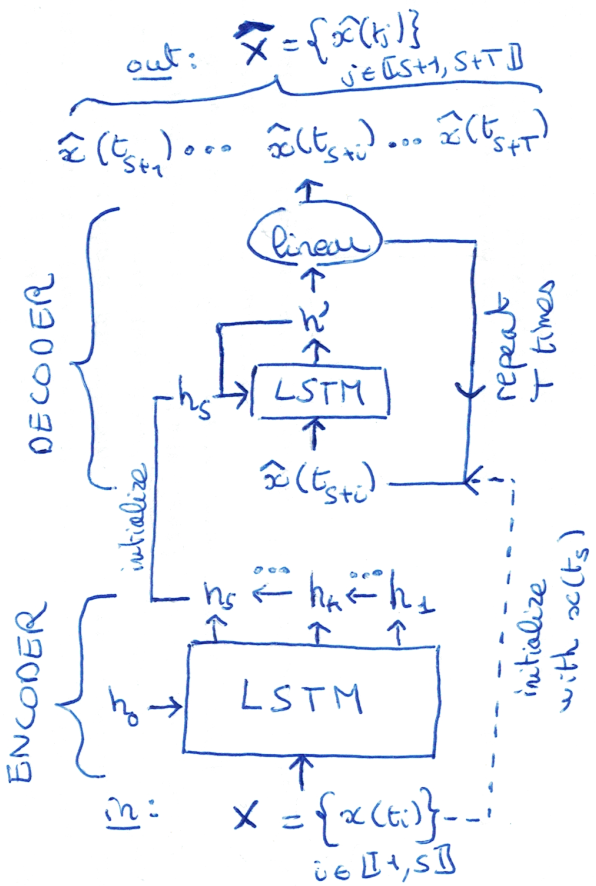

Figure 2: Diagram of the LSTM Auto-Encoder model. The input time serie is encoded into a hidden state h_S which then use to initalize the decoding process. Input X is of shape (S, N, E), the hidden states h_i each have shape (1, N, H), and the output has shape (T, N, E).

1.2 LSTM Auto-Encoder with temporal Attention

An instance of the LSTM Auto-Encoder with Attention model is called in python using the command:

```
# Import class
from models.LSTM_A import LSTM_Attention
# Initiate model
model = LSTM_Attention(E, H).to("cuda")
```

where E is the number of modes and H the size of the LSTM hidden states.

A current issue that is more common in time series than NLP is due to the length of the time series. In general our inputs $x_e(t)$ contains a few hundreds of time points. Which is more than one order of magnitude higher than sentences in general (this one has 18). For the classical Auto-Encoder described in the first section, each hidden state h_k will pass its information to the next one h_{k+1}, and only the final state wille be transmitted to the decoder. However for long length time series, it is possible that the information start to vanish the more states there are. The proposed solution to this issue is to use an Attention mechanism on all the hidden states, to compute a linear combination of them. This Attention then outputs a new hidden state $\tilde{h}$ that is used by the decoder. This process is repeated for all time step to predict, hence the linear combinations ocurring during the Attention mechanism is specific to the timestep to predict t_{S+i}. One of the main avantages of this model compared to the classical LSTM Auto-Encoder is also its hability to represents the weight given to each previous time step to predict the current one (see figure).

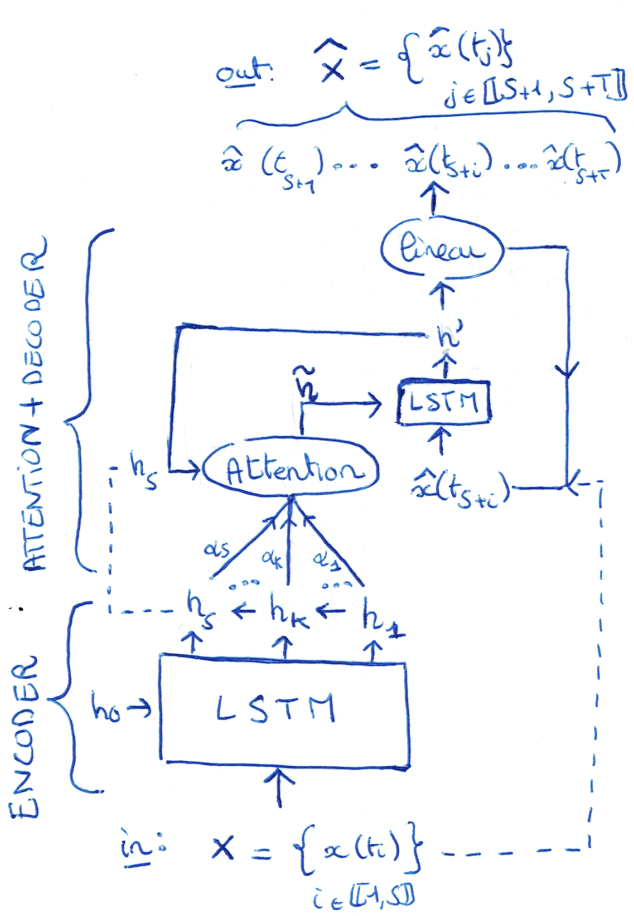

Figure 3: Diagram of the LSTM Auto-Encoder with temporal Attention model. The Attention layer uses a linear combination of all the encoder's hidden states h_k instead of the last one. Input X is of shape (S, N, E), the hidden states h_i each have shape (1, N, H), and the output has shape (T, N, E).

1.3 Multi-Scale Attention model

An instance of the Multi-Scale Attention model is called in python using the command:

```
# Import class
from models.LSTM_A import LSTM_Attention
# Initiate model
model = LSTM_Attention(E, H).to("cuda")
```

where E is the number of modes and H the size of the LSTM hidden states.

The kind of multivariate data can vary on very different spectrum of frequencies and amplitudes. Especially, in this study, the higher the POD mode, the higher the frequency. This implies that when considering the training of hundreds of modes ($E >> Ec$), the model will be inputed time series that evolves on too different timescales making the training of many modes impossible for the previous models. A solution to this is inspired from the Multi-Scale Music article, uses Multi-Scale Attention on multiples LSTMs. The idea is that because each mode varies on such different time scale it requires its own LSTM to train to understand it properly. Thus our input is split according to each of its modes, and each are given to a different LSTM. As for previously these LSTM will output a hidden state h_k^e. Then to make the prediction of mode $x_e(t)$ we look at its interactions with all the modes. This is done with the Attention mechanism that will make a linear combination of all the hiddden states h_k^e. Then (for the specific mode to predict) the new hidden state $\tilde{h}$ is given to the decoder which works as the model above. The full Attention + Decoder process is computed for each mode and repeated $T \times E$ times.

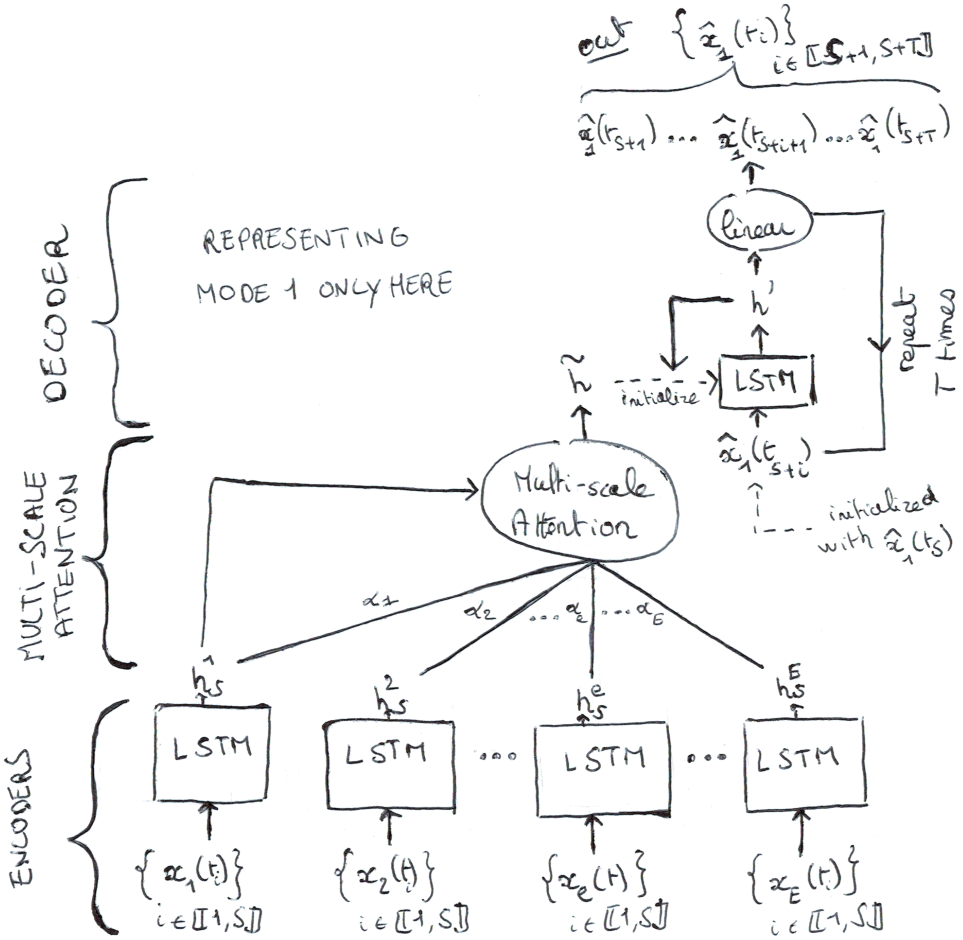

Figure 4: Diagram of the Multi-Scale Attention LSTMs model for mode 1 only. The multivariate time serie is split into each of its indivual mode that are the inputs of LSTM encoders. Then a Multi-Scale Attention mechanism uses a linear combination of these modes to create the mode specific hiddden state $\tilde{h}$. Finally the decoder works as previously stated. The input X of shape (S, N, E) is split into E inputs of shape (S, N, 1).

1.4 Trasnformers for time series

An instance of the LSTM Auto-Encoder with Attention model is called in python using the command:

```
# Import class
from models.transformers import Transformers
# Initiate model
model =  Transformers(d_model, nhead).to("cuda")
```

where d_{model} is the embedding size and n_{head} the number of heads of the Transformer.

LSTM were used for NLP and widely spread to time series datas. Recently, Transformers Networks (Attention is all you need) have surpassed any other Neural Networks, LSTM included for almost every NLP tasks and start to become predominant in computer vision too. One might safely assume, that they should be also widely used for time series. The reality, is that from the litterature I've read, no team of researcher seems to agree on a proper architecture model for transformers in time series. Although, some articles shows model that seem to work for their dataset, they don't seem to exist a general consensus on what to use for transformers when dealing with time series. One of the main difference with NLP is that each point $x^e(t_i)$ is a word for the NLP that will be embedded in much higher dimension, and a scalar (dimension 1) for our case that can take an infinite amount of different values. This means that to use the transformers layers we need to first discretize or signal, and more importantly express it in much higher dimension(usually $d = 512$ for transformers). For this model, the input is split into a source and target arrays which are embedded in higher dimensions by the linear layer. We then use the same layers than (cite article) with a positional encoding and the transformers encoder and decoder layers. The prediction is sent iteratively to the input for multistep predicitions.

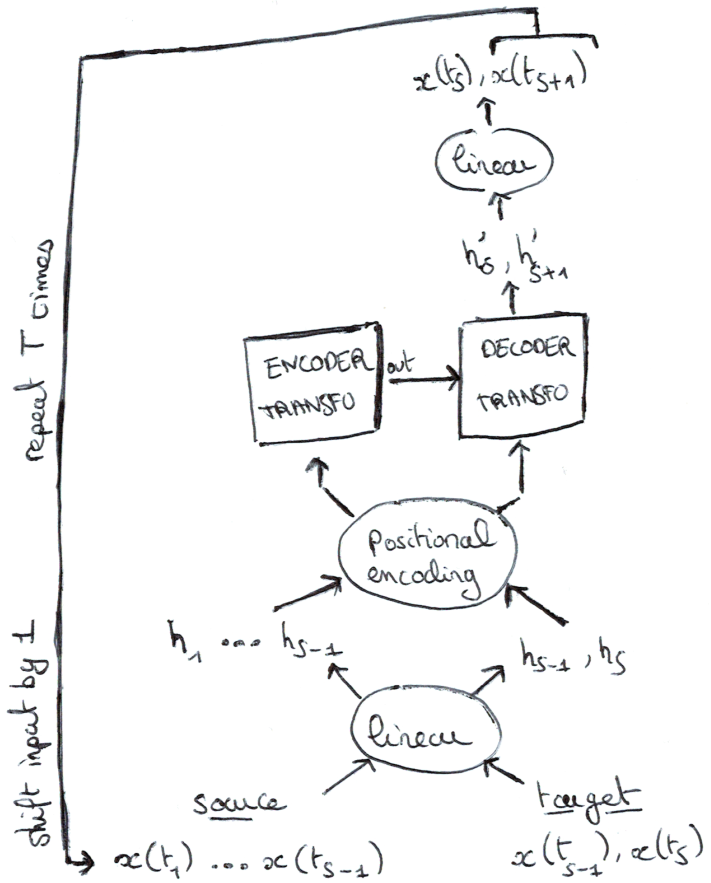

Figure 5: Diagram of a Auto-Encoder Transformer model for time series. The input X is cut into a source and target arrays, which respectively feed the encoder and decoder layer. The positional encoding process is necessary to induce the temporal coherence of the arrays. Prediction is sent iteratively to the input for multistep predicitions.

2 Training and predictions

The training is called in the main.py script after building the model by the lines:

```
import trainer
# Initiate the trainer with model & data
trainer = trainer.Trainer(model, data)
# Start the training loop
trainer.train(epochs, bs, lr, saving_dir)
```

where *epochs* is the number of epochs, *bs* is the batch size, *lr* the learning rate and *saving_dir* the path to save the model.

During training the loss function is computed using the mean square error (i.e. euclidian distance) between the predicted vector $\hat{\mathbf{x}}(t)$ and the target vector $\mathbf{y}(t)$.

$$d_{Eucl} = \sqrt{\sum_{i=1}^{i=T} \left(\hat{x}(t_{S+i}) - y(t_{S+i})\right)^2} \tag{1}$$

The model will updates its weight through backpropagation of the training set loss function. For validation purposes, an other loss is also computed for the validation set, which will show if our model overfits. The trained model is saved when it had the lowest validation loss.

```
""" ... cut of the trainer class """

for ep in range(epochs):
    # Training loop
    self.model.train()
    self.test_loss[ep] = self.step(input_train, self.data.
    y_train, n_batches_train, training=True)
    # Evaluation loop
    self.model.eval()
    with torch.no_grad():
        self.valid_loss[ep] = self.step(input_valid, self.data.
    y_valid, n_batches_valid)
    # Every 10 eps check best valid loss & save
    if self.valid_loss[ep] < self.best_loss and ep%10==0:
        self.model.save(path+'best_model')
        self.best_loss = self.valid_loss[ep]
# ...
```