Time Series Models: From Statistics to AI

lecturer: Daniel Durstewitz
tutors: Lukas Eisenmann, Christoph Hemmer, Alena Brändle, Florian Hess, **Elias Weber**, Florian Götz
SS2025

Exercise Sheet 11

Regulations

Please submit your solutions via Moodle in teams of **2 students**, before the exercise group on **Wednesday, July 9th, 2025**. Each submission must include **exactly** two files:

- A `.pdf` file containing both your Jupyter notebook and solutions to analytical exercises. The Jupyter notebook can be exported to pdf by selecting **File → Download as → pdf** in JupyterLab. If this method does not work, you may print the notebook as a pdf instead. Your analytical solutions can be either scanned handwritten solutions or created using LaTeX.
- A `.ipynb` file containing your code as Jupyter notebook.

Both files must follow the naming convention:
`Lastname1-Lastname2-sheet11.pdf`
`Lastname1-Lastname2-sheet11.ipynb`

1 Transformers and Natural Language Processing

You learned in the lecture that the Transformer architecture was suggested as an alternative to RNN architectures (Vaswani et al., 2017) for natural language processing (NLP). While it can be used for or extended to any other time series modeling task, we want to stay true to its originally intended purpose. So in this task, you will implement and train a Transformer to generate language.

Note: While not strictly required, this task greatly benefits from GPU acceleration. If you do not have access to a GPU locally, consider using Google Colab again.

a) You'll first build a dataset class to handle tokenization using characters as tokens. The provided `names.pt` file contains the 1,000 most common German surnames.Your class should:

   - Load the raw data from the file.
   - Define an alphabet $\mathcal{A}$ by collecting all unique letters in the names and adding a special end-of-string (EOS) token.
   - Implement `name_to_indices(self, name, pad=True)`: Converts a name string into a sequence of token indices $x_{1:T}$ (always ending with EOS). If specified, pad with EOS tokens to the maximum name length for consistent input sizes $x_{1:T_{\max}}$. (This is essential to be able to use the default Pytorch dataloader without any modifications).
   - Implement `indices_to_name(self, indices)`: Converts a sequence of indices back into a string (stop at first EOS token).

- Implement `__len__(self)`: Returns the length of the dataset, i.e. the total number of names.
- Implement `__getitem__(self, idx)`: Returns an input sequence $x_{1:T_{\max}-1}$, a target sequence $x_{2:T_{\max}}$ and a boolean mask marking EOS positions (in order to later exclude them from loss calculation).

b) Use `torch.nn` modules to implement the transformer model, which consists of three steps:

- **Input layer: Embedding + Positional Encoding** The model takes a sequence of token indices $x_{1:t}$ as input. Since these are just integer numbers (e.g., "L" $\rightarrow$ 12), we need to map them to meaningful vector representations. This is usually done by one-hot encoding the token indices and then applying a linear layer, that transforms them into an embedding space of dimension M. Fortunately, `nn.Embedding` does exactly that in one module.

  Since Transformers do not inherently understand the order of elements in a sequence, we must add information about the position of each token through a positional encoding. This can really be anything, as long as it is specific to every possible temporal position t, i.e., it does not repeat. In the lecture you learned about the sinusoidal encoding; however, here we will resort to using a learnable positional encoding. That is, a learnable matrix $\mathbf{P} \in \mathbf{R}^{T_{\max} \times M}$. Then, for a given input sequence $x_{1:T}$, slice $\mathbf{P}$ to match the length and add it to the embedding.

- **Transformer Encoder:** Now feed the combined embeddings into the Transformer Encoder stack. This is the heart of the model, responsible for learning relationships between characters in the input. Use `nn.TransformerEncoderLayer` to define a single layer and `nn.TransformerEncoder` to stack multiple of those layers.

  Causal Masking (Very Important!) When generating sequences, the model should only see previous tokens, not future ones. Otherwise, it could simply attend to the ground truth next token.

  To enforce this, use `nn.Transformer.generate_square_subsequent_mask`. This mask, when passed to the `TransformerEncoder` module's forward method, sets attention scores for all "future" tokens to $-\infty$, effectively blocking them during training and generation.

- **Output layer:** The output of the Transformer encoder is still in the embedding space, and so we will use a linear layer that maps from the embedding size M to the alphabet size N.

The final output of our model's forward pass is then interpreted as logits $\mathbf{y}_{1:t} \in \mathbb{R}^{t \times N}$, that is, when applying a softmax function, they receive the interpretation of a probability distribution for every possible subsequent token $k \in$ `tokenize`$(\mathcal{A})$:

$$p_k(x_{t+1}) := p(x_{t+1} = k | x_{1:t}) = \frac{\exp\left(y_{t,k}/T\right)}{\sum_{j=1}^{N} \exp\left(y_{t,j}/T\right)} \tag{I}$$

with some temperature parameter $T \in (0, 1]$. We will then use a cross-entropy loss objective to train the model:

$$\mathcal{L}(x_{1:t}, k^*) = -\log p_{k^*}(x_{t+1}) \tag{II}$$

This effectively maximizes the probability that the next token will be the ground truth token k^*. As mentioned before, we exclude the positions in the sequence that have been padded with EOS tokens from this loss.

c) Finally, implement the generation routine to be used with a trained model to generate new names as follows:

1. Start with one or two initial tokens $x_{1:t}$

2. Compute the probability distribution of the next token $p(x_{t+1} = k|x_{1:t})$ as described above.
3. Sample the next token from this probability distribution using `torch.multinomial`: $\hat{x}_{t+1} \sim p(x_{t+1} = k|x_{1:t})$.
4. Append the new token to the sequence $x_{1:t+1} = [x_{1:t}, \hat{x}_{t+1}]$.
5. Repeat steps 1 to 3 until either the maximum sequence length $T_{\max}$ is reached or the EOS token is generated.

d) Train a model with the hyperparameters specified in the notebook (those are just a rough suggestion that should give you decent results while not taking too long to train, even on a CPU, but feel free to experiment).

Afterwards, generate and print some names by giving one or two initial letters. Explore how the temperature influences the generation. For an initial sequence of your choice, plot histograms of $p_k(x_{t+1})$ for a few values of T.