LLM fine-tuning for time series annotation

Philippe Helluy

IRMA, Université de Strasbourg, Inria Tonus

August 29, 2025

Outlines

How it works (roughly...)

Fine tuning of Qwen LLM for time series annotation

How it works (roughly...)

Very brief history

- Artificial neural networks have existed for a **long time** ([?], late 1950s).
- Ups and downs, then **Yann LeCun** ([?], handwriting recognition, 1989).
- *Attention is all you need* [?], invention of **transformers** at Google.
- Without huge **computing power**, it would not work.

Completion is all you need

- Principle: we are given the beginning of a text. The task is to predict the next word. Example: "the cat eats the ..." (we must guess "mouse").
- Words (or *tokens*):

t_1	t_2	t_3	t_4	t_5	$t_6 = t_m$
.	cat	the	eats	tomcat	mouse

- Corpus: "the cat eats the mouse.", "the tomcat eats the mouse.", "..the cat eats.", "..the mouse eats.", "..the tomcat eats.", *etc.*
- Remark: all sentences have $\ell = 6$ words (we pad with the filler word ".").

Digitization

- Encoding: to each word (or *token*) we associate a vector of $m = 6$ dimensions

$$"." = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad "cat" = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad "the" = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad \textit{etc.}$$

- Embedding into a lower-dimensional space of size p (to account for synonyms, among other things). For example $p = 5$. The embedding E_{w_0} is a function from $\mathbb{R}^m$ to $\mathbb{R}^p$.
- Each token t_i is represented by a vector v_i. The embedding parameters w_0 are unknown.

$$v_i = E_{w_0}(t_i).$$

Encoder

- A sentence is thus represented by a "tensor" of ℓ numerical vectors stacked together:

$$r_0 = v_{i_1} v_{i_2} \dots v_{i_\ell}$$

  It is therefore an object in a space of dimension $N = \ell \times p = 30$.
- The sentence goes through k layers of transformers T_{w_i}, which are mappings from $\mathbb{R}^N$ to $\mathbb{R}^N$ with unknown parameter vectors w_i

$$r_i = T_{w_i}(r_{i-1}), \quad i = 1 \dots k.$$

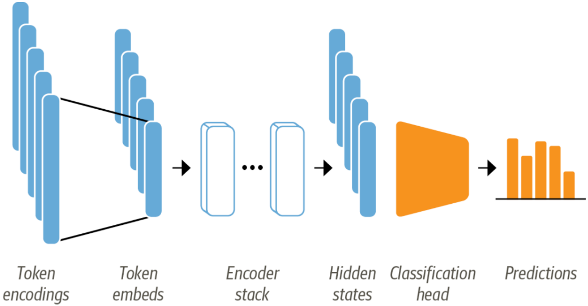
- The deeper we go into the layers, the more abstract the representation r_i of the initial sentence becomes. The vector r_k ("latent state") contains the information extracted by the network from the initial sentence r_0.

Decoder

Finally, the decoder allows us to predict a probability vector p in $\mathbb{R}^m$: p_i is the probability that the next word is m_i.

$$p = D_{w_{k+1}}(r_k).$$

In summary (more details in [?, ?]):



Training

- Choosing the form of the functions E_{w_0}, T_{w_i}, $D_{w_{k+1}}$: a trade-off between cost, efficiency, and simplicity. For now, it is as much an art as it is a science.
- Historically, several possible architectures: CNN, RNN, LSTM, transformers. Evolution is strongly linked to available computing power.
- The parameter vector w, of size s, is **unknown**.
- Training consists in optimizing these parameters so that the model best reproduces the sentences from the corpus.
- This is the most difficult part of the computation: it requires a supercomputer, specialized processors, and costs millions of euros.
- Orders of magnitude for GPT-3: $\ell = 2000$, $m = 50000$, $p = 20000$, $s = 170$ billion...

Inference

- Once the parameters w are computed, inference is fast.
- A network can be retrained for a specific task at a reduced cost (*fine tuning*). For example, Copilot and ChatGPT are specialized versions of GPT-x ($x = 3, 4, 5$)
- The *pre-prompt* is essential to obtain high-quality results.
- For computational cost reasons, **ChatGPT does not learn in real time**. Its short-term memory is therefore limited to a few thousand words.

Example with PyTorch

- PyTorch is a free software library developed by Huggingface to manipulate neural networks.
- Minimal example for creating a toy LLM from scratch with PyTorch: `https://github.com/phelluy/DLAA_2025/blob/main/mini_llm.py`
- Don't expect this to give interesting inference ! It just shows the main structure of a training procedure.

Fine tuning of Qwen LLM for time series annotation

Objectives

- Teach a small LLM (Qwen2.5 0.5 billions parameters) to describe a time series
- Example of prompt:

```
Describe the time series in three sentences.
First sentence: describe trend (increasing/decreasing/flat).
Second sentence: noise intensity (low/medium/high).
Third sentence: approximate localisation of global maximum (beginning/middle/end)
and global minimum (beginning/middle/end).
Put the description in a JSON format with the following pattern
<json>{ "trend": <sentence1>,
"noise": <sentence2>,
"extrema": <sentence3> }
</json>
Series: [02, 01, 00, 03, ... ]
```

- Tokenization is essential: the values in the series are scaled to the range 00..99.

Necessity of training

- Small LLM fails to answer properly. It generates total garbage.
- Example of output:

```
series name: \"series\"
time interval: 1 day
``` To provide a detailed analysis of the given time series data, I will break it down into its com
### Time Series Description:
The provided time series consists of daily values for several variables over a period of one year (
### Trend Analysis:
- **Increase**: The values increase steadily from 01 to 04, then decrease again.
- **Decrease**: The values decrease from 04 onwards
...
```

- Let's try to improve this with supervised fine tuning
```

Practical methodology

- Generate a dataset of correct examples with a large LLM (Mistral, ChatGPT, etc.)
- Use 90% of the dataset for training and the 10% left for evaluation.
- Apply a supervised fine tuning (SFT) procedure on a small LLM from this dataset.
- In order to reduce the cost we adopt a LoRA approach. [1]
- Now go to `https://github.com/phelluy/DLAA_2025` and follow the README file.
- The Colab GPU memory is limited, I had to reduce the size of the computations. For full results see [?].

[1] The LoRA (Low-Rank Adaptation) approach in supervised fine-tuning (SFT) freezes the original model weights and injects small trainable low-rank matrices into certain layers (typically linear projections in attention/FFN). This drastically reduces the number of parameters that need updating, making fine-tuning large models much more memory- and compute-efficient while still achieving strong adaptation.

Bibliographie I