M2 coursework report; Word Count:

Yihao Liu; CRSid:yl2063

April 2025

1 Introduction

Large Language Models (LLMs) have shown potential beyond natural language processing, notably in numerical reasoning and time series forecasting. In particular, untrained LLMs can forecast time series when suitably tokenized [1]. This project investigates the application of LoRA (Low-Rank Adaptation) to fine-tune Qwen2.5-Instruct [2] for forecasting dynamics governed by the Lotka-Volterra predator-prey equations.

All experiments were conducted under a strict compute budget, expressed in floating point operations (FLOPs). This report presents our implementation, experiments, and insights, with the goal of evaluating the trade-offs between accuracy, compute cost, and model configuration for LLM-based time series prediction.

2 Compute Constraint

This coursework imposes a strict compute budget of 10^{17} floating point operations (FLOPs) across all reported experiments.

Only operations listed in Table 1 are counted; others (e.g., tokenization, memory access) are excluded. FLOPs are calculated for the forward pass, with backpropagation assumed to cost exactly twice as much. In addition, only training a model are counted towards the FLOPs budget; evaluation and inference are excluded.

3 Load and preprocessing the data

3.1 Dataset Structure

The dataset used in this coursework is stored in HDF5 format (`lotka_volterra_data.h5`) and contains time series data simulating predator-prey population dynamics based on

Table 1: FLOPS Accounting for Primitives

Operation	**FLOPs**
Addition / Subtraction / Negation (float or int)	1
Multiplication / Division / Inverse (float or int)	1
ReLU / Absolute Value	1
Exponentiation / Logarithm	10
Sine / Cosine / Square Root	10

the Lotka-Volterra equations.

It consists of two components:

- `/trajectories`: A 3-dimensional array of shape $(1000, 100, 2)$, where:
  - The first dimension indexes 1000 independent predator-prey systems,
  - The second dimension corresponds to 100 time steps per system,
  - The third dimension contains two variables: prey population (`[:,:,0]`) and predator population (`[:,:,1]`).
- `/time`: A 1-dimensional array of length 100, representing the shared timeline across all systems.

3.2 Preprocessing the data

To make the multivariate numerical time series compatible with the Qwen2.5-Instruct tokenizer, we implemented a text-based preprocessing pipeline adapted from the LLM-TIME scheme [1]. The raw population values of the predator-prey dataset were first scaled and then serialized into structured text sequences.

3.2.1 Scaling and Precision

Each numerical value x_t was rescaled to $x'_t = \frac{x_t}{\alpha}$, where α is a scaling constant selected to ensure that most values fall within a consistent numeric range (e.g., $[0.00, 9.99]$ in our implementation). The scaled values were then rounded to a fixed number of decimal places (e.g., two in this coursework), ensuring uniform token lengths and predictable tokenization.

3.2.2 Multivariate Encoding

To represent both prey and predator populations over univariate time, we adopted a format that encodes multivariate time series as token-friendly strings. The encoding follows these rules:

- Different variables (prey and predator) at the same timestep are separated by a comma ‘,‘,
- Different timesteps are separated by a semicolon ‘;‘.

A short example with two variables over three time steps is encoded as:

```
0.25,1.50;0.27,1.47;0.31,1.42
```

3.2.3 Data splitting

The dataset was randomly partitioned into training, validation, and test sets using a 7:1.5:1.5 split ratio. The indices of the 1000 trajectories were first shuffled with a fixed random seed to ensure reproducibility.

- The **training set** was used for model training.
- The **validation set** serves to evaluate models' forecasting ability.
- The **test set**, although not used in the majority of models' performance evaluation, provides an unbiased inference of the final model's generalization ability.

3.2.4 Numeric Tokenization

Each numeric value, in the form of string, will be tokenized into separate digits and punctuation marks by the Qwen tokenizer. This text-based representation serves as input for both training and inference. Please note that our range $[0.00, 9.99]$ converted every value into exact three digits, so that the numbers of tokens generated from the string-form sequences are always consistent.

3.2.5 Examples

To verify the correctness of the LLMTIME preprocessing and tokenization pipeline, we examined the second sequence from the training set as a representative example. The first 40 characters of the preprocessed text sequence, as well as the first 40 corresponding token IDs produced by the Qwen2.5 tokenizer, were printed.

The preprocessed sequence was a semicolon- and comma-separated string representing scaled predator-prey values over time:

```
0.92,0.74;0.56,0.77;0.34,0.70;0.23,0.59;
```

After applying the tokenizer, the numerical string was mapped into a sequence of token IDs, preserving the numeric structure:

```
[15, 13, 24, 17, 11, 15, 13, 22, 19, 26, 15, 13, 20, 21, 11, 15,
13, 22, 22, 26, ...]
```

In both cases, we confirmed that the total sequence length was 999, ensuring that the preprocessing and tokenization stages were consistent in their output dimensions.

During preprocessing, the final semicolons were intentionally omitted to avoid creating an unnecessary empty token after tokenization. As a result, the total number of characters (and thus tokens) in a final string is 999 instead of 1000.

4 Model Architecture, LoRA implementation and FLOPs Accounting

4.1 Model Architecture

The Qwen2.5-Instruct model is a decoder-only transformer architecture. In this coursework, we use the 0.5B parameter variant, which consists of 24 stacked transformer blocks, each with a hidden size of 896 and 16 attention heads. The vocabulary size is 151,936. The model architecture proceeds as follows:

Input Embedding. Each input token is embedded into a 896-dimensional vector using a learned embedding layer. The dimension of embedding is also called hidden dimension.

Transformer Block. Each of the 24 decoder layers contains:

- **RMSNorm:** Root Mean Square Layer Normalization is applied both before and after the attention and MLP sublayers. It is defined as:

$$\text{RMSNorm}(x) = \frac{x}{\sqrt{\frac{1}{d}\sum x_i^2 + \epsilon}} \cdot \gamma$$

- **Self-Attention:** Although the model internally uses Grouped Query Attention (GQA), for the purpose of FLOPs calculation we follow the coursework simplification and treat it as standard Multi-Head Attention. The attention mechanism computes:

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V, \quad \text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

where W^Q, W^K, W^V, W^O are learned projections. The projection dimensions are:

  - W^Q: 896×896
  - W^K, W^V: 896×128 (reflecting grouped attention structure)
  - W^O: 896×896

- **Feedforward (MLP) with SwiGLU:** The MLP block uses two linear layers with an intermediate SwiGLU activation:

$$\text{SwiGLU}(x_1, x_2) = \text{SiLU}(x_1) \cdot x_2$$

  where SiLU is defined as $\text{SiLU}(x) = \frac{x}{1+e^{-x}}$. The feedforward dimension is expanded to 4864, and the linear projections are:

  - $W_{\text{gate}}, W_{\text{up}}$: 896×4864
  - W_{down}: 4864×896

- **Residual Connections:** Each sublayer uses a residual connection added elementwise to the input.

Output Layer. The final output is passed through an RMSNorm layer and projected via a linear output layer to match the vocabulary size of 151936.

4.2 LoRA implementation

LoRA is a parameter-efficient fine-tuning technique that introduces trainable low-rank matrices into pre-trained transformer architectures. Instead of updating the full weight matrices of a large model during fine-tuning, LoRA freezes the original weights and injects two smaller matrices $A \in R^{r \times d}$ and $B \in R^{d \times r}$ such that their product approximates the necessary weight updates:

$$W_{\text{adapted}} = W_0 + \Delta W = W_0 + BA$$

where W_0 is the frozen pre-trained weight, and $r \ll d$ denotes the LoRA rank.

In this project, LoRA was applied to the `q_proj` and `v_proj` linear layers inside the self-attention modules of Qwen2.5-Instruct. For each such layer, the original projection is wrapped in a `LoRALinear` module that adds the low-rank path during the forward

pass while keeping the base matrix frozen:

$$\text{output} = \text{Linear}(x; W_0) + \frac{\alpha}{r} \cdot B(Ax)$$

Only the matrices A and B are trained, while the backbone transformer remains unchanged.

4.3 FLOPs Accounting

We implemented a modular FLOPs estimation framework in Python. The design closely mirrors the architecture of the Qwen2.5-Instruct model and allows us to compute training or inference costs under various hyperparameter settings.

The computation is broken down into the following components:

- **Embedding Layer:** Only the addition of positional embeddings is counted. Each token contributes one addition per hidden dimension.
- **Multi-Head Attention:** We account for:
  - Q, K, V projections: each modeled as a matrix multiplication with cost $m \cdot n \cdot (2k - 1)$.
  - Scaled dot-product attention: includes $QK^\top$, softmax (estimated as 12 FLOPs per element), and the multiplication with V.
  - Output projection: another matrix multiplication.
  - Optional LoRA: if enabled, low-rank projections are applied to `q_proj` and `v_proj` with cost $2 \times 2 \times B \times L \times d \times r$.
- **Feed-Forward Network (FFN):** The FFN consists of two linear layers with an intermediate dimensionality defined by the `ffn_ratio`. The activation function, SwiGLU, is modeled as 14 FLOPs per element.
- **Normalization Layers:** RMSNorm FLOPs are estimated as $4d + 11$ per token.
- **Output Projection and Loss:** The final hidden states are projected to vocabulary logits with FLOPs $\sim (2d - 1) \cdot V$. Cross-entropy loss is approximated as $11 \cdot V$ per token.

The top-level function `flops_for_experiment(...)` supports full control over key variables:

- `num_steps`: number of optimizer steps or inference passes.
- `batch_size`: number of sequences per batch.
- `seq_len`: sequence length (context window).
- `hidden_dim`: model hidden size.
- `num_layers`: number of Transformer blocks.
- `num_heads`: number of attention heads.
- `ffn_ratio`: expansion factor for the FFN module.
- `r`: LoRA rank (set to 0 if LoRA is not used).
- `training`: boolean flag to determine if the backward pass should be included (as $2\times$ forward).

This modular design enables us to rapidly explore the FLOPs impact of different experimental configurations and ensures all reported results respect the coursework's compute constraint.

5 Methodology of Model Evaluation

5.1 Context-Target Splitting

We evaluate the model in an autoregressive forecasting setting by splitting each tokenized input sequence into a context segment and a target prediction segment. In our experiments, the context ratio is set to be 0.7, so that 70 percent of the tokens will be used as model input. The rest is used as target and we only predict the first 100 tokens for each sequence, so that the evaluation time is acceptable. As a result, there are 700 context tokens for prediction and 100 target tokens for ground truth.

5.2 Autoregressive Generation and Loss Computation

We use `model.generate()` with greedy coding to produce tokens conditioned on the context. This is an ideal method as it generates future tokens based solely on past context. It appends one new token at each step and feeds it back into the model for the next prediction.

The output scores (logits) are returned for each step, allowing us to compute the per-token cross-entropy loss against the ground truth target tokens. The average token-level is reported per sequence.

5.3 Numerical Decoding and MSE Calculation

To evaluate the accuracy of the model's numerical forecasts, we first decode the generated token IDs into text using Qwen tokenizer's built-in `decode` method. This converts the predicted token sequence into a string following the LLMTIME format[1].

Next, we manually parse this string to extract numerical values. Specifically, we split the string by semicolons (‘`;`‘) to separate individual timesteps, and then by commas (‘`,`‘) to isolate the prey and predator values at each timestep. Each component is then converted to a floating-point number using Python's `float()` function. Any malformed or non-numeric substrings are safely ignored.

This post-processing method return real numbers of the predicted prey-predator systems. The same decoding procedure is applied to the ground truth target tokens. We then compute the Mean Squared Error (MSE) between the predicted and true numeric sequences. The model may return illegal elements instead of a number, so we ignore these illegal elements and then calculate MSE for sequences with the same length between predicted system and ground truth system only.

In this work, we select MSE as the primary evaluation metric, as it directly reflects the discrepancy between the predicted and ground-truth trajectories at the sequence level. Meanwhile, the average token-level validation loss is used as a secondary indicator to provide insights into the model's token generation behavior.

5.4 Preliminary Evaluation of an untrained model

The first step before formally conducting experiments is to make an initial trial evaluation of the performance of the untrained `Qwen2.5-Instruct` model without any LoRA adaptation. This experiment serves as a baseline on the model's ability to forecast scientific time series in the LLMTIME format.

Figure 1 shows the resulting cross-entropy loss curves, indicating a general upward trend over prediction steps, which should be expected because of the nature of forecasting errors.

Notably, the loss curves exhibit an oscillatory pattern due to the presence of formatting tokens such as commas and semicolons, which are easy for the model to predict.

To further analyze forecasting behavior, we visualized predicted versus ground truth

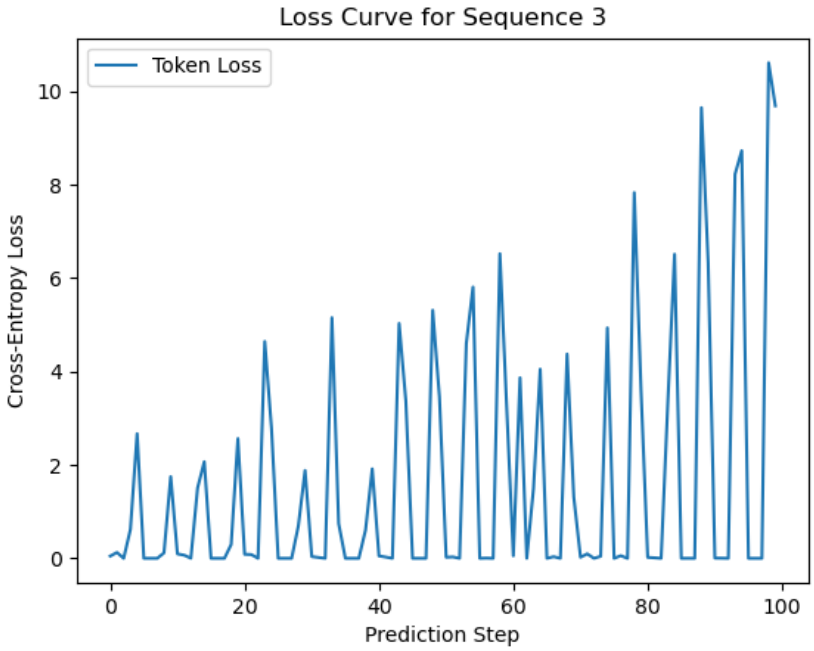

Figure 1: Validation token-level cross-entropy loss curve for Sequence 3.

trajectories. Even in the best-performing case in Sequence 9 of the validation dataset, as shown in Figure 2, the predicted trajectory diverges from the ground truth shortly after the start of prediction.

Figure 3 shows the predictions of Sequence 7, which significantly diverges and fails to capture the true dynamics of the system.

The evaluation of the untrained model on the validation set gives an average MSE of 0.2702 and an average cross-entropy validation loss of 1.785. These values will be compared with later experiments.

Overall, this baseline evaluation highlights the possibility and limitations of using an unadapted large language model for scientific forecasting tasks and motivates the use of parameter-efficient fine-tuning techniques such as LoRA.

6 Experiments

Formal experiments include the fine-tuning and training of`Qwen2.5-Instruct` using LoRA. We explored ranges of hyperparameters over learning rates and LoRA ranks. In addition, by selecting the best hyperparameter ranges, the model was trained with different context length. Finally, a longer experiments was conducted with all the best choices.

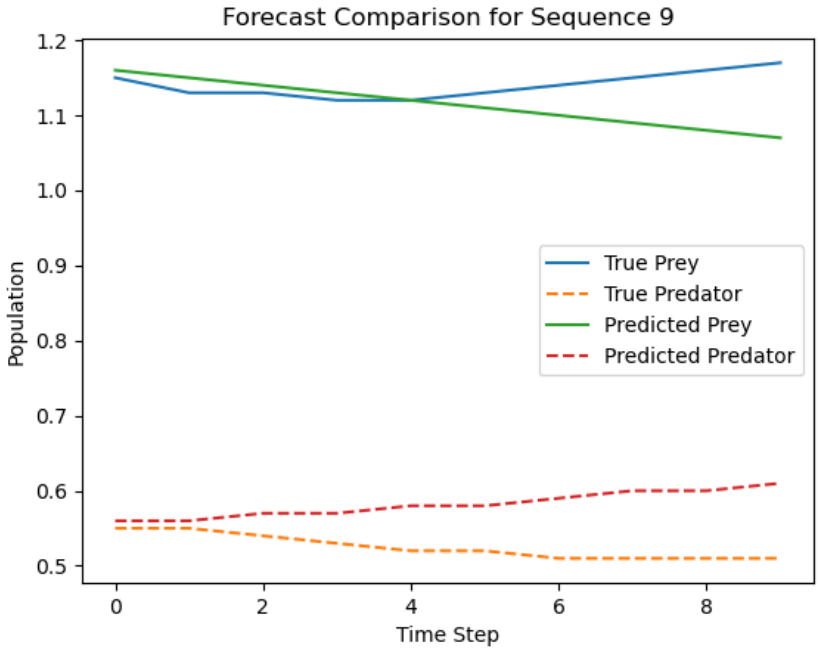

Figure 2: Forecast comparison on Sequence 9

6.1 Training Methodology

6.1.1 Data Chunking

The first step before training a model is to further process the training set. It is important to convert all the data in the form of fixed-length inputs, so that we can facilitate batch training using PyTorch's `DataLoader`, which ensures efficient parallelization. The tokenized sequences are sliced into overlapping segments of fixed maximum context length, with a specified stride (256 tokens in this coursework). For each segment, if its length is shorter than this fixed context length, the sequence is padded on the left using the model's designated padding token. This ensures that all input tensors have consistent shape. The resulting padded segments are then stacked into a tensor to be consumed by the model during training.

6.1.2 Training Procedure

The training procedure starts with the injection of LoRA adapters into the query and value projection layers of each self-attention block. Only the low-rank parameters are updated while the base model remains frozen.

When the model is ready, the chunked dataset is loaded into batches (batch size is 4 in this coursework).

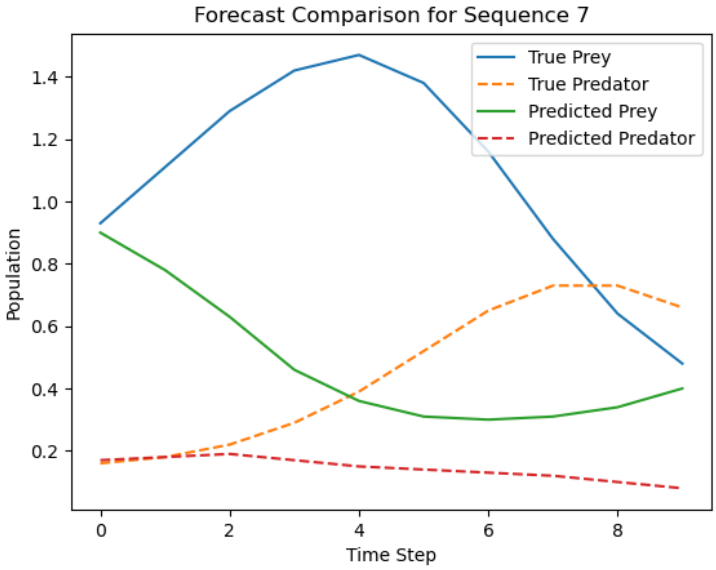

Figure 3: Forecast comparison on Sequence 7

Each training batch is passed through the model using the standard forward interface `model(input_ids, labels=...)`. This method is different from the `model.generate(...)` interface used during evaluation. The training-time forward pass enables a form of *teacher forcing*, in which the model is given the entire input sequence and asked to predict the next token at each position. Cross-entropy loss is computed between these predictions and the true next-token labels. In addition, to prevent the model from learning from padded positions, all padding tokens in the label tensor are masked by setting their value to `-100`, following the HuggingFace convention for ignored loss positions.

The model is optimized using the Adam optimizer, with a batch size of 4 and a selected number of steps.

6.2 A Trial Experiment with default hyperparameters

We trained the Qwen2.5-Instruct model with a LoRA rank of 4 and a learning rate of 10^{-5} for 500 optimization steps. This is a trial experiment, served to validate the training pipeline and provide an initial sense of the model's ability to learn the dynamics in the LLMTIME-formatted forecasting task.

6.3 Experiments with Hyperparameter Tuning

6.4 Experiments with different context length

6.5 Final Experiment with longer optimizer steps

7 Conclusion

References

[1] N. Gruver, M. Finzi, S. Qiu, and A. G. Wilson, "Large language models are zero-shot time series forecasters," 2024.

[2] Q. Team, "Qwen2.5 technical report," 2025.