Coursework Report: Deep Learning for Data Intensive Science (M2)

Xiaoye Wang (xw453)
Department of Physics, University of Cambridge
xw453@cam.ac.uk

1 Introduction

Large language models (LLMs) have recently demonstrated impressive capabilities beyond traditional natural language tasks, including applications in reasoning, planning, and scientific modeling [1, 2, 3, 4]. Among these emerging directions, leveraging LLMs for time series forecasting presents both a novel opportunity and significant challenges. Recent works such as LLMTIME [5] have proposed specialized tokenization and formatting strategies to adapt numerical sequences for text-based models. In this report, we investigate the use of the Qwen2.5-0.5B-Instruct model for forecasting tasks on the classic Lotka-Volterra predator-prey system.

2 Data Preparation

2.1 LLMTIME Preprocessing for Multivariate Time Series

To prepare the predator-prey dataset for LLM-based forecasting, we implemented a text-based numerical encoding pipeline adapted from the LLMTIME [5]. This is described as follows.

Scaling and Formatting. Firstly, We adopt a rescaling approach. For each sequence $x_t \in \mathbb{R}^2$, we compute a scaling factor α such that the p-percentile of the absolute values in the sequence equals 1:

$$x'_t = \frac{x_t}{\alpha}, \quad \alpha = \text{Percentile}_p\left(|x_t|\right).$$

Each value is then rounded to a fixed number of decimal places, which limits unnecessary token expansion due to long decimal tails. We choose $p = 95$ and precision $= 2$ in the following experiments.

String Encoding. After scaling, each time step is converted into a comma-separated string. Time steps are separated by semicolons to distinguish temporal ordering. The final semicolon is optional but included in our implementation to indicate sequence termination explicitly.

Tokenization. Thanks to Qwen2.5's built-in digit-level tokenization, our formatted strings map naturally to token IDs without additional token manipulation. We avoid inserting spaces between digits, as instructed in our coursework.

2.2 Implementation Overview

We implemented the described preprocessing pipeline in Python. To demonstrate the preprocessing pipeline, two illustrative examples are listed in Table 1.

The word count for this file is 2999 without the declaration section, as reported by Overleaf.

Table 1: LLMTIME preprocessing and tokenization for two trajectories (first 3 time steps each).

Field	Example 1
Original Sequence	`[[0.9499, 1.0406], [0.7406, 0.7795], [0.6822, 0.5644]]`
Scale	`2.8809319`
Formatted Text	`0.33,0.36;0.26,0.27;0.24,0.20;`

Tokenized (IDs)	`15 13 18 18 11 15 13 18 21 26`
	`15 13 17 21 11 15 13 17 22 26`
	`15 13 17 19 11 15 13 17 15 26`

Field	Example 2
Original Sequence	`[[0.9715, 1.0054], [1.0787, 0.8218], [1.2608, 0.6864]]`
Scale	`3.1954854`
Formatted Text	`0.30,0.31;0.34,0.26;0.39,0.21;`

Tokenized (IDs)	`15 13 18 15 11 15 13 18 16 26`
	`15 13 18 19 11 15 13 17 21 26`
	`15 13 18 24 11 15 13 17 16 26`

3 Baseline Study and FLOPs Calculation for Qwen2.5-0.5B-Instruct

3.1 Model Structure and Configurations

In this task, we evaluate the baseline forecasting ability of the untrained Qwen2.5-0.5B-Instruct model [1] on preprocessed time series data. This model consists of 24 decoder layers with a hidden size of 896 and 14 attention heads. It adopts grouped-query attention (GQA) [6] with 2 key-value heads to reduce memory and compute requirements during inference. Each decoder layer includes multi-head attention and a feedforward network with an intermediate size of 4864, utilizing the SwiGLU activation function [7]. The model employs rotary positional embeddings (RoPE) [8] to handle long sequences efficiently and Qwen-specific RMSNorm [9] for normalization. The vocabulary size is 151,936, and the model's embedding and output projection layers are tied. The 'lm_head' layer projects from hidden states to vocabulary logits and is the only trainable component in our setup, where all other parameters remain frozen. A detailed model structure can be found in Appendix A.

3.2 Evaluation of the Untrained Qwen2.5-Instruct Model

To evaluate the zero-shot forecasting ability of the untrained Qwen2.5-0.5B-Instruct model, we used the Lotka-Volterra dataset consisting of 1000 simulated trajectories. The data were randomly shuffled and split into 900 training and 100 test sequences, saved in separate HDF5 files.

For evaluation, each test sequence was preprocessed, with the first 51 steps (context length 510) used as input and the next 20 steps as prediction targets. Then, predictions were generated with greedy decoding. The output tokens were parsed using regular expressions, rescaled to the original scale, and aligned with ground-truth for metric evaluation.

To evaluate forecasting accuracy, we compute six standard metrics: Mean Squared Error (MSE), Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), coefficient of determination (R^2), Normalized RMSE (NRMSE), and Mean Absolute Percentage Error (MAPE). We also visualized the predictions for the first five sequences, with the prediction for sequence 4 shown in Figure 1.

Meanwhie, as shown in Table 2, the untrained model performs poorly across all metrics. The strongly negative R^2 and high MAPE indicate that, without fine-tuning, the model performs worse than a naive mean baseline and produces unstable predictions.

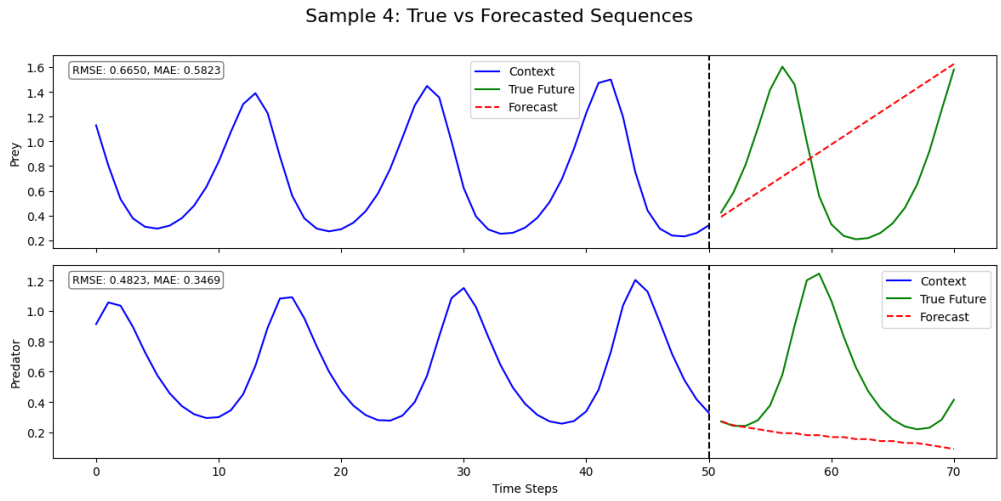


Figure 1: Qwen2.5-0.5B-Instruct's Predictions for sequence 4.

Table 2: Forecasting performance metrics of the untrained Qwen2.5-Instruct model on the Lotka-Volterra dataset.

MSE	RMSE	MAE	R^2	NRMSE (%)	MAPE (%)
3.2828	1.8119	0.7837	-6.6598	21.7710	651.9870

A Short Discussion on Sampling Strategy. In this report, we adopt greedy decoding for prediction, selecting the most probable token at each step without sampling. While the default generation settings of Qwen2.5 and the LLMTIME protocol encourage stochastic decoding, using temperature scaling, logit bias, and top-p sampling to capture forecast uncertainty, we opt for deterministic decoding due to compute and runtime constraints. This simplifies evaluation and enables consistent comparisons, though at the cost of uncertainty-aware inference. We revisit this trade-off in Section 4.6, comparing the untrained and finetuned models.

3.3 FLOPs Calculation Principle for Qwen2.5-0.5B-Instruct

We analyze the floating-point operations (FLOPs) of Qwen2.5-0.5B-Instruct in this section. Following standard FLOPs accounting practice, we omit components with negligible computational cost and simplify GQA to standard multi-head attention (MHA).

3.3.1 Embedding Layer (embed_tokens).

The embedding layer has shape (V_{vocab}, d), where $d = 896$ is the hidden size and $V_{vocab} = 151936$ denotes the vocabulary size. Since embeddings are obtained via direct indexing, the FLOPs incurred are negligible and hence ignored in our estimate:

$$\text{FLOPs}_{\text{embedding}} \approx 0.$$

3.3.2 Decoder Layers (24 × Qwen2DecoderLayer)

Each decoder layer contains a self-attention block with GQA, an MLP block with SwiGLU activation, and two RMSNorms. We analyze these components below.

(a) Self-Attention Module: FLOPs Decomposition. For a standard MHA, given an input $X \in \mathbb{R}^{(B\times L)\times d}$, where B is the batch size, L is the sequence length, the self-attention mechanism consists of the following components.

First, X is linearly projected into queries, keys, and values through three separate weight matrices $W_Q, W_K, W_V \in \mathbb{R}^{d\times d}$. This step produces $Q = XW_Q$, $K = XW_K$, and $V = XW_V$, with each

projection being a matrix multiplication of shape $(B \cdot L, d) \times (d, d)$. The combined FLOPs cost is:

$$\text{FLOPs}_{\text{QKV}} = 3 \times 2 \cdot (B \cdot L \cdot d) \cdot d = 6BLd^2.$$

Next, in the multi-head setting, Q, K, and V are reshaped into shape $(B, L, h, d_{\text{head}})$, where $d_{\text{head}} = d/h$. However, in FLOPs estimation, it is common to flatten the heads and treat them as a single large matrix, and this simplification numerically equivalent to decomposing the hidden dimension d as $d = h \cdot d_{\text{head}}$ and summing over all heads individually. The attention weights are then computed by scaled dot-product between Q and $K^\top$. For each head, this corresponds to a pairwise matrix multiplication over the sequence, with computational cost:

$$\text{FLOPs}_{\text{QK}^\top} = 2BL^2d.$$

Once the attention matrix A is obtained, it is applied to the value matrix via $O = A \cdot V$, with the same cost as above:

$$\text{FLOPs}_{AV} = 2BL^2d.$$

The output tensor O with shape $(B \cdot L, d)$ after reshaping, is then passed through a final output projection using $W_O \in \mathbb{R}^{d \times d}$:

$$\text{FLOPs}_{\text{out\_proj}} = 2(B \cdot L \cdot d) \cdot d = 2BLd^2.$$

Summing the components, the total FLOPs per layer for the attention module is:

$$\begin{aligned}\text{FLOPs}_{\text{attention}} &= 6BLd^2 + 2BL^2d + 2BL^2d + 2BLd^2 \\ &= (6+2)BLd^2 + 4BL^2d \\ &= 8BLd^2 + 4BL^2d.\end{aligned}$$

(b) MLP Block with SwiGLU Activation. The MLP comprises two parallel projections, an element-wise activation, and a down projection. For the two-way projection, The input is projected into two branches using linear layers:

- `gate_proj`: Linear map from $896 \to 4864$, without bias;
- `up_proj`: Same shape, $896 \to 4864$, also without bias.

The FLOPs for each projection is approximated by:

$$\text{FLOPs}_{\text{gate/up}} = 2 \times (B \times L) \times 896 \times 4864.$$

Then, for activation and gating, the `up_proj` output is passed through a SiLU activation:

$$\text{SiLU}(x) = x \cdot \sigma(x), \quad \sigma(x) = \frac{1}{1+e^{-x}}.$$

The activation is then element-wise multiplied with the `gate_proj` output:

$$\text{output} = \text{gate\_proj}(X) \odot \text{SiLU}(\text{up\_proj}(X)).$$

This contributes:

$$\text{FLOPs}_{\text{activation + elementwise mult}} \approx 14 \times (B \times L) \times 4864.$$

Regarding the down projection, the output is projected back to 896 dimensions:

$$\text{FLOPs}_{\text{down}} = 2 \times (B \times L) \times 4864 \times 896.$$

Finally, to calculate the total FLOPs, let $m = B \times L$, $d = 896$, and $d_{\text{ff}} = 4864$, then the total FLOPs of the MLP per layer is:

$$\text{FLOPs}_{\text{MLP}} = 2mdd_{\text{ff}} + 2mdd_{\text{ff}} + 14md_{\text{ff}} + 2md_{\text{ff}}d = 6mdd_{\text{ff}} + 14md_{\text{ff}}.$$

If we ignore the activation cost as a simplification, the MLP FLOPs reduce to:

$$\text{FLOPs}_{\text{MLP}} \approx 6mdd_{\text{ff}}.$$

(c) Layer Normalization (RMSNorm). RMSNorm is a simplified normalization strategy that avoids mean subtraction, in contrast to standard LayerNorm [9, 10]. For a hidden vector $a \in \mathbb{R}^d$, the RMSNorm transform is given by:

$$\bar{a}_i = \frac{a_i}{\text{RMS}(a)} \cdot g_i, \quad \text{where} \quad \text{RMS}(a) = \sqrt{\frac{1}{d}\sum_{i=1}^{d} a_i^2}.$$

The operation involves computing the root mean square of the input vector and scaling each component by a learned weight g_i. Given an input tensor of shape (B, L, d), RMSNorm operates over the d-dimensional vector of each of the $B \times L$ tokens.

To estimate its FLOPs, we consider two components per token. First, RMS computation requires squaring and summing d elements, followed by division and square root, leading to:

$$\text{FLOPs}_{\text{RMS}} \approx 2d + 11.$$

Second, normalization and rescaling involves dividing each element by RMS and multiplying by its weight g_i, yielding:

$$\text{FLOPs}_{\text{scale}} = 2d.$$

Combining the two and over the entire batch, the total FLOPs per token is:

$$\text{FLOPs}_{\text{RMSNorm}} \approx (B \cdot L) \cdot (4d + 11).$$

In practice, the constant offset may be ignored, and the total FLOPs is approximated as $\alpha \cdot B \cdot L \cdot d$ for a small constant $\alpha \in [4, 6]$.

3.3.3 Rotary Positional Embedding (RoPE).

Following the decoder layers, Qwen2.5 concludes with a post-attention normalization and an application of rotary positional embedding. The normalization layer is another instance of RMSNorm and incurs the same FLOPs as analyzed.

For positional encoding, Qwen adopts RoPE, which encodes relative positions by applying learned or fixed sinusoidal rotations to the query and key vectors. For FLOPs, we ignore the FLOPs associated with computing positional encodings, and only account for their addition to token embeddings. Thus, the cost is calculated as:

$$\text{FLOPs}_{\text{RoPE}} \approx B \cdot L \cdot d.$$

3.3.4 Output Projection (lm_head)

After the final decoder layer, the model applies a linear transformation to map the final hidden states to vocabulary logits:

$$\text{FLOPs}_{\text{output}} = 2BLd \cdot V_{vocab}.$$

3.3.5 Overall Forward FLOPs

Combining all components and aggregating over $N = 24$ layers, the total forward FLOPs is:

$$
\begin{aligned}
\text{FLOPs}_{\text{forward}} = \quad & \underbrace{B \cdot L \cdot d}_{\text{positional addition}} \\
& + \sum_{l=1}^{N} \Big[\underbrace{2 \cdot \text{RMSNorm}(B, L, d)}_{\text{pre/post norms}} + \underbrace{\text{AttnFLOPs}(B, L, d)}_{\text{self-attention}} + \underbrace{\text{MLP}(B, L, d, d_{\text{ff}})}_{\text{feedforward}} \Big] \\
& + \underbrace{\text{RMSNorm}(B, L, d)}_{\text{final norm}} \\
& + \underbrace{2 \cdot B \cdot L \cdot d \cdot V_{vocab}}_{\text{output projection (lm\_head)}} \\
= \; & BLd + \alpha BLd + 2BLdV_{\text{vocab}} \\
& + \sum_{l=1}^{N} \Big[2\alpha BLd + 8BLd^2 + 4BL^2d + 6BLd \cdot d_{\text{ff}} + 14BLd_{\text{ff}} \Big].
\end{aligned}
$$

3.3.6 Forward + Backward FLOPs

Assuming backward propagation requires 2× forward FLOPs, the total per training step is:

$$\text{FLOPs}_{\text{step}} = 3 \times \text{FLOPs}_{\text{forward}}.$$

Given the parameters of Qwen2.5-0.5B-Instruct and choosing $\alpha = 4$, we obtain the FLOPs for qwen model for a batch of 4 sequences with context length of 512 tokens, as shown in Table 3.

Table 3: Estimated FLOPs for Qwen2.5-0.5B-Instruct.

Scenario	**FLOPs**	$\log_{10}$ **FLOPs**
Single Forward Pass	2.252×10^{12}	12.353
Forward + Backward	6.757×10^{12}	12.830

3.3.7 Verifying FLOPs Using `calflops`

To cross-check the theoretical FLOPs calculation, we employ the `calflops` package. The measured FLOPs are summarized in Table 4. Given our approximations, a deviation of around 10% is acceptable and lends support to our estimates.

Table 4: Estimated FLOPs for Qwen2.5-0.5B-Instruct using `calflops`.

Scenario	**FLOPs**	$\log_{10}$ **FLOPs**
Single Forward Pass	2.02×10^{12}	12.305
Forward + Backward	6.07×10^{12}	12.783

4 Lora Finetuning for Time Series Data

4.1 Method

We employ Low-Rank Adaptation (LoRA) [11] to finetune the Qwen2.5-0.5B-Instruct model for time series forecasting. In accordance with the standard LoRA setup, only the low-rank adaptation matrices are updated during training, while the base model weights remain frozen. Specifically, we target the query and value projection layers within each Transformer block for LoRA injection, wrapping them with two trainable matrices $A \in \mathbb{R}^{r \times d_{\text{in}}}$ and $B \in \mathbb{R}^{d_{\text{out}} \times r}$. Then the LoRA output is computed as:

$$\text{output} = \text{original\_linear}(x) + \left(xA^\top\right) B^\top \cdot \left(\frac{\alpha}{r}\right).$$

The finetuning is conducted with the standard causal language modeling objective using tokenized numeric trajectories. Input sequences are tokenized into discrete tokens using a sliding window strategy with a configurable context length, and forecasts are generated by autoregressively decoding future values.

4.2 Implementation Details

We implement our pipeline using PyTorch and HuggingFace Transformers. Training is conducted using the AdamW optimizer, which combines the Adam algorithm with weight decay regularization, achieving a similar effect to Adam with L2 regularization but with improved computational efficiency. For the dataset, it is first preprocessed as described in Section 2, including rescaling and numeric formatting. Each trajectory is then tokenized and chunked into fixed-length sequences using a sliding window approach:

$$\begin{aligned} \text{for } i \in \{0, \text{stride}, 2 \cdot \text{stride}, \dots\}: \quad & \text{chunk} = \text{tokens}[i : i + \text{max\_length}] \\ & \text{if } |\text{chunk}| < \text{max\_length} : \text{pad to fixed length.} \end{aligned}$$

To enable LoRA, we freezed all model weights except for the bias term in the output layer. Then, each `q_proj` and `v_proj` layer is wrapped using the `LoRALinear` module. Training is performed with an accelerator-enabled training loop on a single NVIDIA RTX 4090 GPU.

4.3 Baseline Finetuning

We begin our LoRA finetuning study by training the Qwen2.5-0.5B-Instruct model using a default single configuration. Specifically, we used 80% of the trajectories for training and 20% for validation, with all sequences preprocessed and tokenized using the LLMTIME pipeline.

For model optimization, we used a LoRA rank of 4, batch size of 4, learning rate of 1×10^{-5}, and weight decay of 1×10^{-4}. Training ran for 2000 steps with a context length of 512 tokens. A fixed random seed (42) ensured reproducibility. Validation was conducted and checkpoints saved every 1000 steps. This setup serves as our baseline configuration for evaluating the effectiveness of LoRA in finetuning for time series forecasting.

The training dynamics are illustrated in Figure 2, where we observe a rapid initial drop in training loss, followed by stabilization. Correspondingly, the validation loss decreased from an initial value of 0.8974 to 0.5192 at step 1000 and 0.4148 at step 2000.

Figure 2: Training loss curve of Qwen2.5-0.5B-Instruct with LoRA rank 4 over 2000 steps.

However, surprisingly, the basic finetuned model underperforms the original Qwen2.5 almost across all metrics at evaluation (Table 5). This suggests that with a suboptimal learning rate and limited

training steps, finetuning can degrade performance—especially when the base model already shows strong zero-shot ability. This highlights the need for careful hyperparameter tuning, which we address in the following sections.

Table 5: Forecasting performance comparison between untrained Qwen2.5 and baseline LoRA-finetuned model.

Model	MSE	RMSE	MAE	R^2	NRMSE (%)	MAPE (%)
Qwen2.5	3.2828	1.8119	0.7837	-6.6598	21.7710	651.9870
Baseline Setting	6.7802	2.6039	0.7207	-7.0963	31.2879	228.0413

4.4 Parameter Tuning for LoRA

Effect of Learning Rate. To identify a suitable learning rate for LoRA fine-tuning, we conducted a grid search over $\{1 \times 10^{-5}, 5 \times 10^{-5}, 1 \times 10^{-4}\}$ while keeping other hyperparameters fixed: LoRA rank $r = 4$, context length $L = 512$, batch size $= 4$, and total training steps $= 2000$. Figure 3 shows the training loss trajectories across different learning rates. To enhance visual clarity, we applied smoothing and used a log scale on the loss curve. The training loss decreases significantly, with the model trained at 1×10^{-4} demonstrating the fastest convergence and lowest final loss.

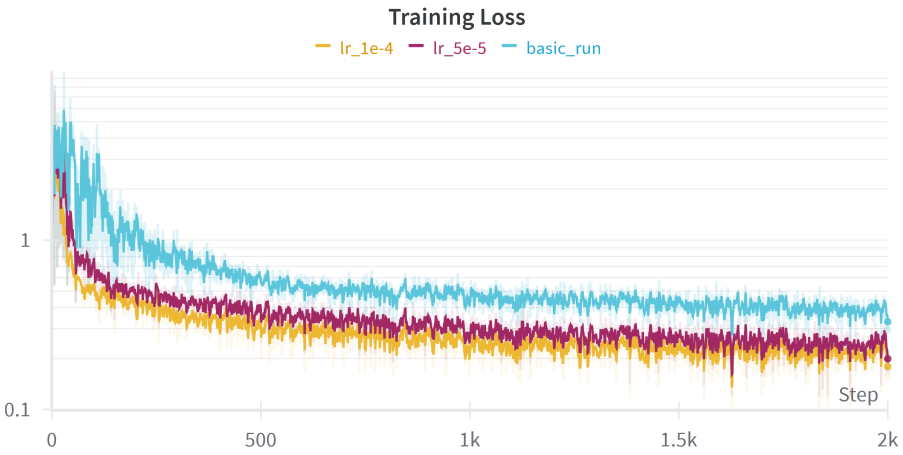


Figure 3: Training loss under different learning rates.

To better illustrate the learning dynamics, Table 6 summarizes the validation loss at initialization, mid-training , and final step for the three learning rates.

Table 6: Validation loss at different steps under different learning rates.

Step	lr = 1×10^{-5}	lr = 5×10^{-5}	lr = 1×10^{-4}
0	0.8974	0.8974	0.8974
1000	0.5192	0.3284	0.2776
2000	0.4148	0.2681	0.2415

Moreover, Table 7 summarizes the evaluation metrics at the end of 2000 steps, showing that the highest learning rate achieves the best performance and significantly outperforms the untrained model. To further qualitatively compare the models, we visualize the forecasted trajectories on the fourth test sequence under each learning rate setting. As seen in Figures 4a, 4b, and 4c, the model trained with 1×10^{-4} captures the dynamics most accurately and smoothly.

Table 7: Forecasting metrics under different learning rates.

Learning Rate	MSE	RMSE	MAE	R^2	NRMSE (%)	MAPE (%)
1×10^{-5}	6.7802	2.6039	0.7207	-7.0963	31.2879	228.0413
5×10^{-5}	0.2282	0.4777	0.1727	0.2972	5.7394	27.5810
1×10^{-4}	**0.0897**	**0.2995**	**0.1408**	**0.8173**	**3.5986**	**21.8218**

Effect of LoRA Rank. To assess the impact of the LoRA rank r, we performed experiments with $r \in \{2, 4, 8\}$ while keeping the learning rate fixed at 1×10^{-5} and the context length at 512. All models were trained for 2000 steps.

Figure 5 shows the training loss curves, where higher ranks led to faster convergence and lower final loss. Correspondingly, Table 8 reports validation losses at steps 0, 1000, and 2000, indicating that larger ranks consistently yielded better generalization.

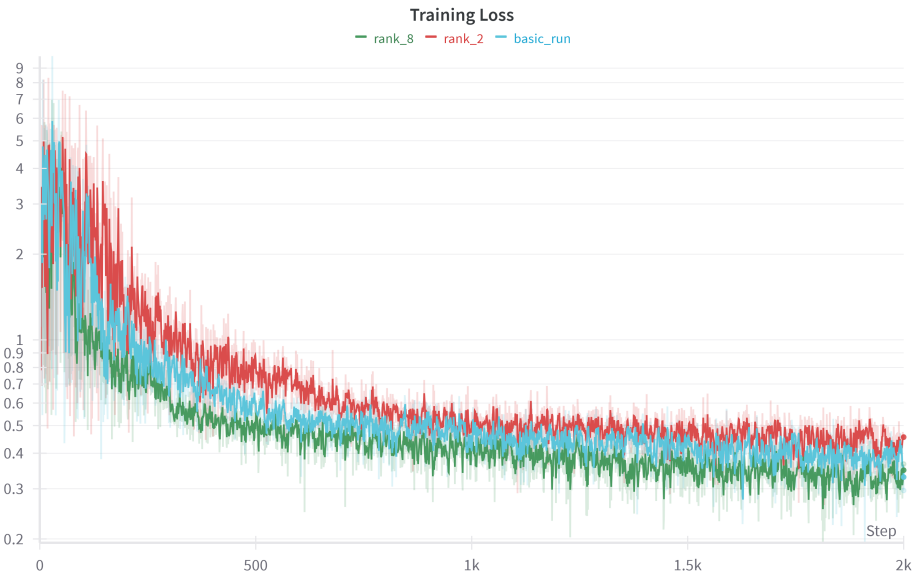


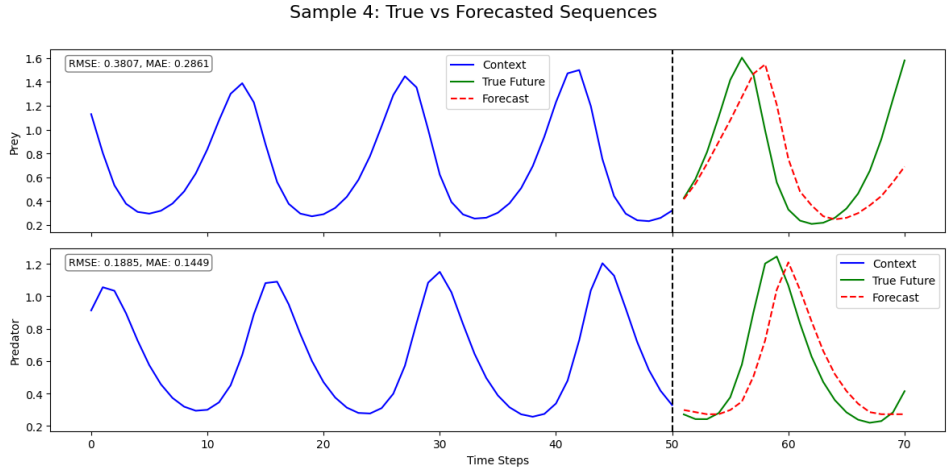
Figure 5: Training loss comparison across different LoRA ranks.

Table 8: Validation loss over training steps for different LoRA ranks.

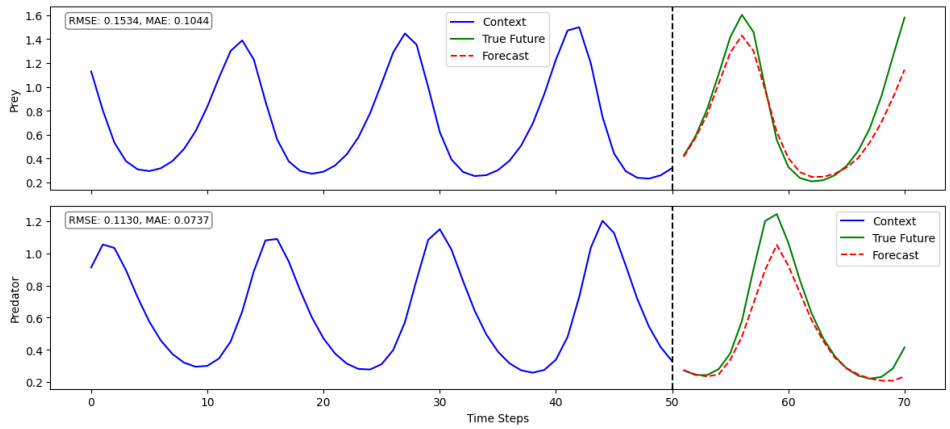
LoRA Rank	Step 0	Step 1000	Step 2000
2	0.8974	0.5723	0.4788
4	0.8974	0.5192	0.4148
8	0.8974	**0.4511**	**0.3586**

Despite showing favorable training behavior, forecasting metrics in Table 9 show that $r = 2$ performs worse than the untrained model, echoing earlier findings with small learning rates. This suggests that under limited training steps, low-rank configurations may struggle to escape local optima inherited from the language pretraining objective. In contrast, $r = 8$ yields clear improvements, indicating enhanced capacity to adapt to the downstream task. Representative comparison for different LoRA ranks are displayed in Figure 6a, 6b and 6c, respectively.

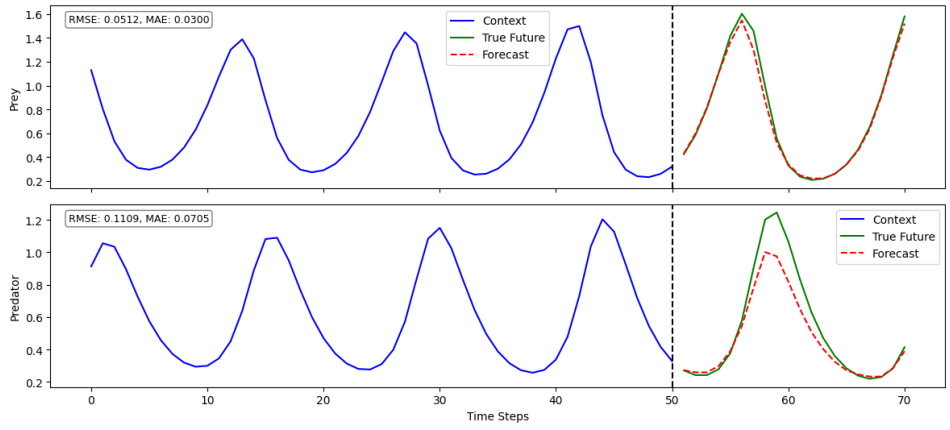
Effect of Context Length. Finally, we examine the impact of context length on model performance. Based on the best configuration found in previous sections, we conduct additional experiments with



(a) $\mathrm{lr} = 1 \times 10^{-5}$

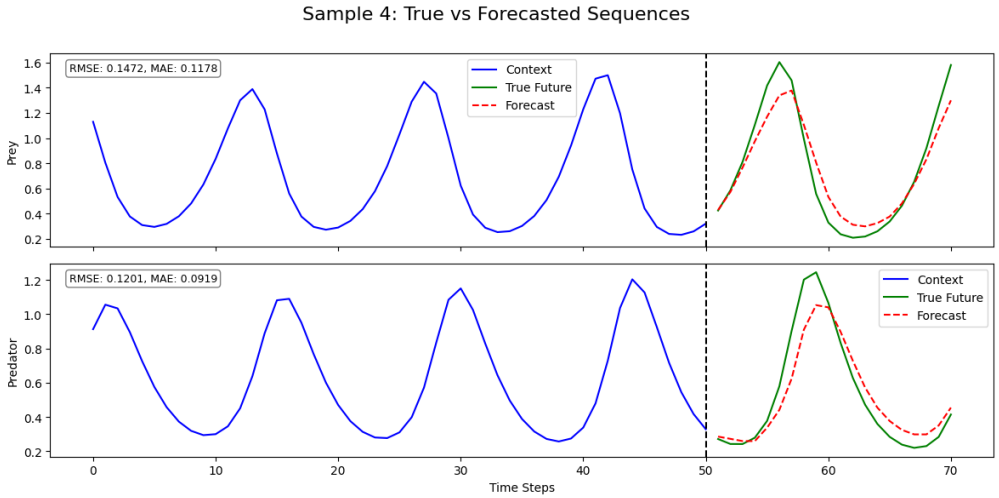


(b) $\mathrm{lr} = 5 \times 10^{-5}$

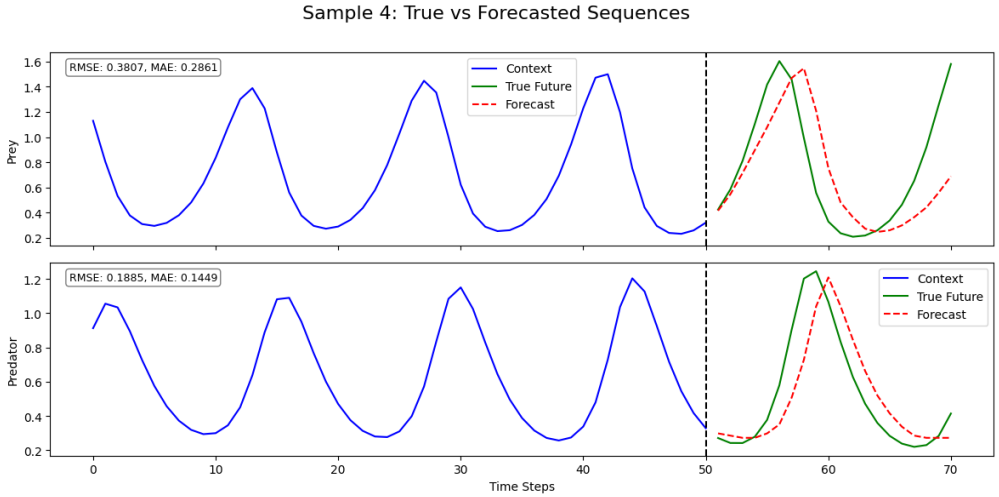


(c) $\mathrm{lr} = 1 \times 10^{-4}$

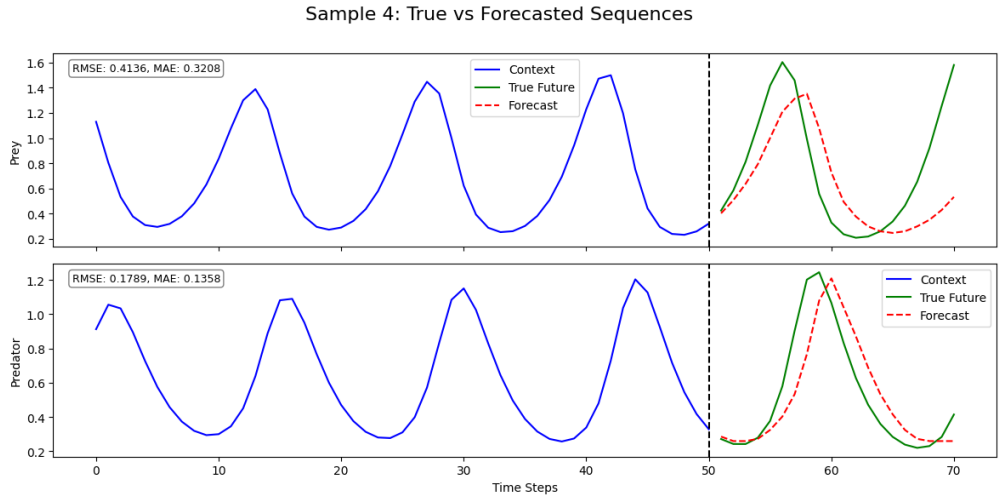
Figure 4: Forecast results on the fourth test sequence under different learning rates.



(a) LoRA Rank $= 2$



(b) LoRA Rank $= 4$



(c) LoRA Rank $= 8$

Figure 6: Forecast results on the fourth test sequence under different LoRA ranks.

Table 9: Forecasting metrics under different LoRA ranks.

LoRA Rank	MSE	RMSE	MAE	R^2	NRMSE (%)	MAPE (%)
2	3.5050	1.8722	0.6731	-5.3531	22.4958	560.6918
4	6.7802	2.6039	0.7207	-7.0963	31.2879	228.0413
8	**0.4249**	**0.6518**	**0.2717**	**0.3086**	**7.8325**	**55.1460**

context lengths set to 128, 512, and 768, each trained for 1000 steps with validation performed every 200 steps

As shown in Figure 7 and Figure 8, both the training and validation losses suggest that a context length of 768 yields the lowest loss values. However, the initially high validation loss and more volatile convergence behavior observed with 768 also hint at potential drawbacks of excessively long contexts. One possible explanation is that longer sequences may dilute or obscure important local temporal patterns, making it harder for the model to retain fine-grained information relevant for near-future prediction. In contrast, shorter contexts like 128 are limited in their ability to capture long-range dependencies, leading to underfitting.

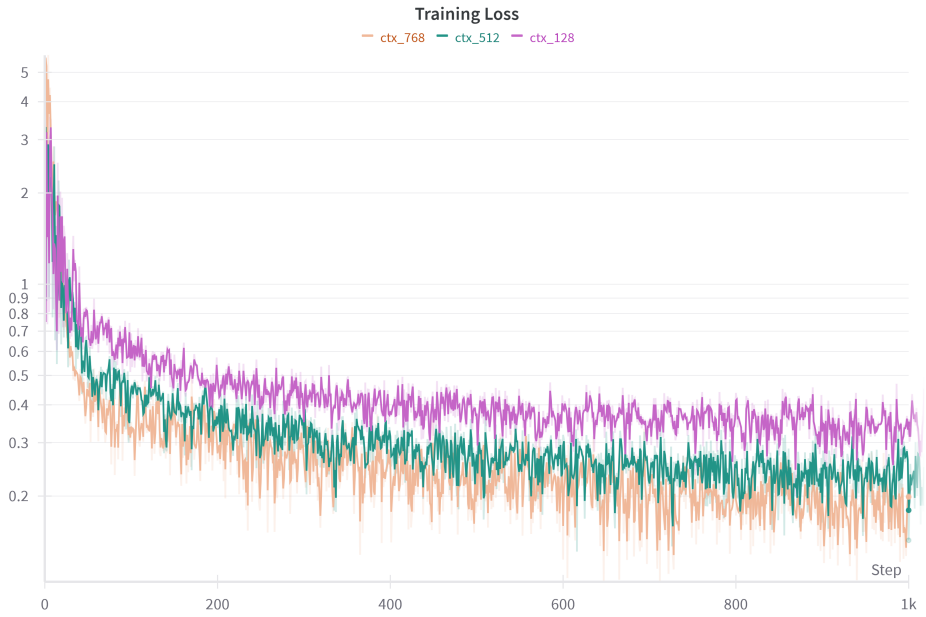


Figure 7: Training loss under different context lengths.

To further validate this observation, we conduct evaluations where 12, 51, and 76 input time steps are used to forecast the next 20 steps. The final evaluation metrics in Table 10 further confirm that 512 tokens achieve the best performance across all six metrics. Thus, we select 512 as the optimal context length for the final model.

Table 10: Forecasting metrics under different context lengths using the best hyperparameters.

Context Length	MSE	RMSE	MAE	R^2	NRMSE (%)	MAPE (%)
128	1.4988	1.2243	0.5790	-1.9545	10.0614	106.4164
512	**0.1344**	**0.3666**	**0.1533**	**0.6996**	**4.4047**	**31.4448**
768	0.7875	0.8874	0.5287	-0.1951	9.8628	87.8303

Figure 8: Validation loss curve under different context lengths.

4.5 Final Model Training

Based on the previous findings, we selected the optimal hyperparameter configuration as: learning rate 1×10^{-4}, LoRA rank 8, and context length 512. For final training, we reused the checkpoint from the context-length ablation experiment and resumed training for 2000 additional steps. After training, the final model achieved the following forecasting performance on the validation set:

Table 11: Forecasting performance of the final model.

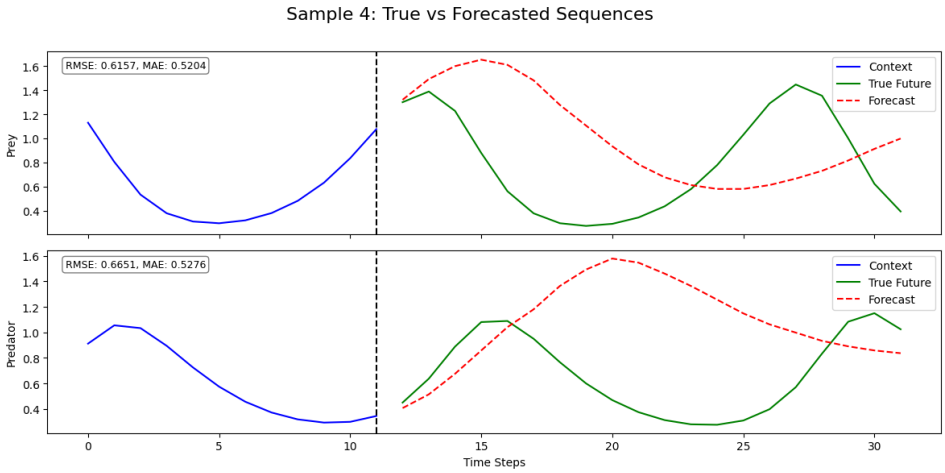
MSE	RMSE	MAE	R^2	NRMSE (%)	MAPE (%)
0.0576	0.2400	0.1044	0.9009	2.8839	12.7867

The final model demonstrates a strong improvement over all earlier configurations, with the highest R^2 and lowest error metrics across the board. This confirms that the selected hyperparameters enable the model to effectively generalize and make accurate forecasts over the Lotka-Volterra sequences. Visualization of the prediction is shown in Figure 10.

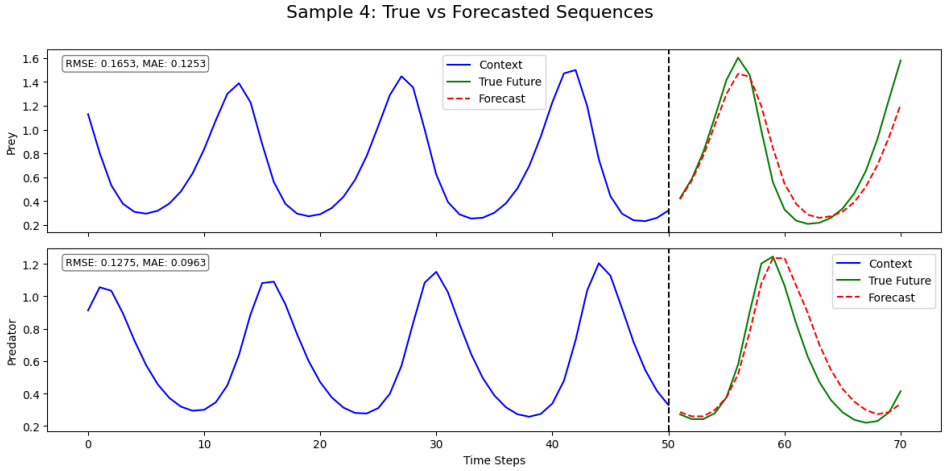
4.6 Ablation Study: Greedy Decoding vs. Temperature Sampling

To evaluate the effect of decoding strategy on forecast accuracy and uncertainty, we perform an ablation study comparing greedy decoding with stochastic sampling. While our main results are based on greedy decoding, LLMTIME recommends probabilistic sampling strategies to obtain more representative forecast distributions.

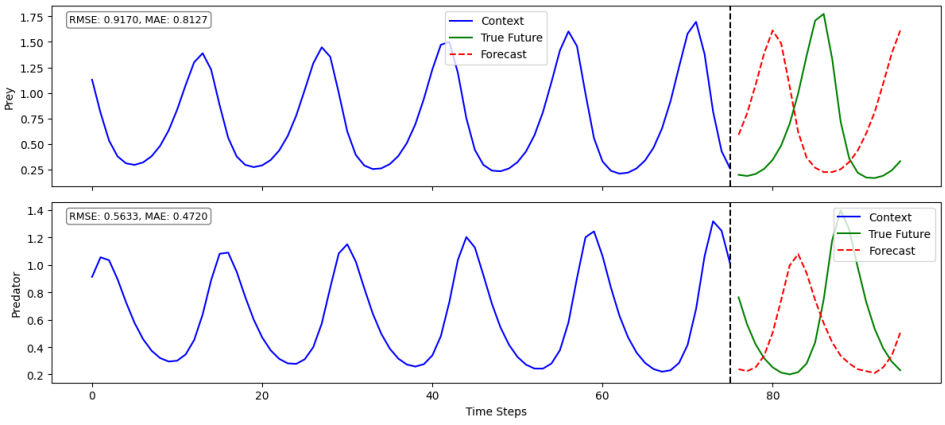
For this comparison, we use the built-in sampling configuration from the Qwen2.5 model with `temperature=0.7`, `top-k=20`, and `top-p=0.8`, and generate 20 samples per sequence. Due to computational constraints, evaluation is performed on the first 10 samples from the test set.



(a) Context length = 128



(b) Context length = 512



(c) Context length = 768

Figure 9: Forecast results on the fourth test sequence under different context lengths.

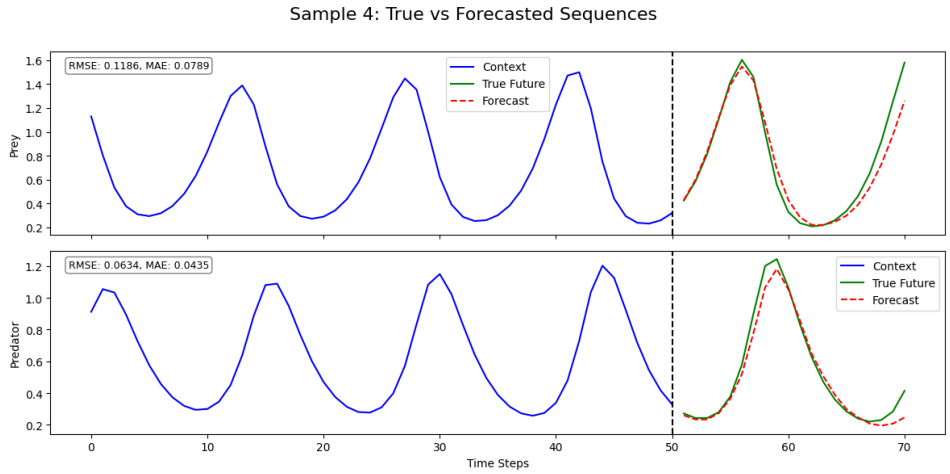


Figure 10: Forecast results on the fourth test sequence by the best model.

Table 12: Forecasting performance and uncertainty under greedy and temperature sampling.

Model	MSE	RMSE	MAE	R^2	NRMSE (%)	MAPE (%)	Avg. Unc.
Qwen2.5 (greedy)	2.3479	1.5323	0.8953	-8.2666	29.36	350.45	–
Qwen2.5 (sampling)	1.7297	1.3152	0.7805	-4.6283	25.20	305.46	0.4065
Best Model (greedy)	0.0691	0.2628	0.1214	0.8540	5.04	15.77	–
Best Model (sampling)	**0.0475**	**0.2179**	**0.1046**	**0.8869**	**4.18**	**15.16**	**0.0682**

Table 12 shows that fine-tuning the model substantially reduces predictive uncertainty. Additionally, applying a probabilistic sampling strategy instead of greedy decoding improves forecast accuracy across all metrics. However, the gains are relatively modest. If computational resources and latency are not primary constraints, sampling-based decoding may offer further performance improvements. To further visualize the differences, we plot the forecast results for the fourth test sequence under temperature sampling from both models in Figure 11a and 11b.

4.7 LoRA FLOPs Calculation

We follow standard FLOPs accounting for rank-r low-rank adapters applied to a linear layer $\text{Linear}(d_{\text{in}} \to d_{\text{out}})$. Each LoRA module introduces an additional forward path via the composition of two projections: $W_A \in \mathbb{R}^{r \times d_{\text{in}}}$ and $W_B \in \mathbb{R}^{d_{\text{out}} \times r}$. The total FLOPs for a single forward pass of the LoRA branch can be approximated as:

$$\text{FLOPs}_{\text{LoRA-forward}} = 2 \cdot (B \cdot L) \cdot r \cdot (d_{\text{in}} + d_{\text{out}}).$$

where the factor of 2 accounts for the matrix multiplications $X \cdot W_A^T$ and $(XW_A^T) \cdot W_B^T$. Then the total training FLOPs for one step become:

$$\text{FLOPs}_{\text{LoRA-step}} = 3 \cdot \text{FLOPs}_{\text{LoRA-forward}}.$$

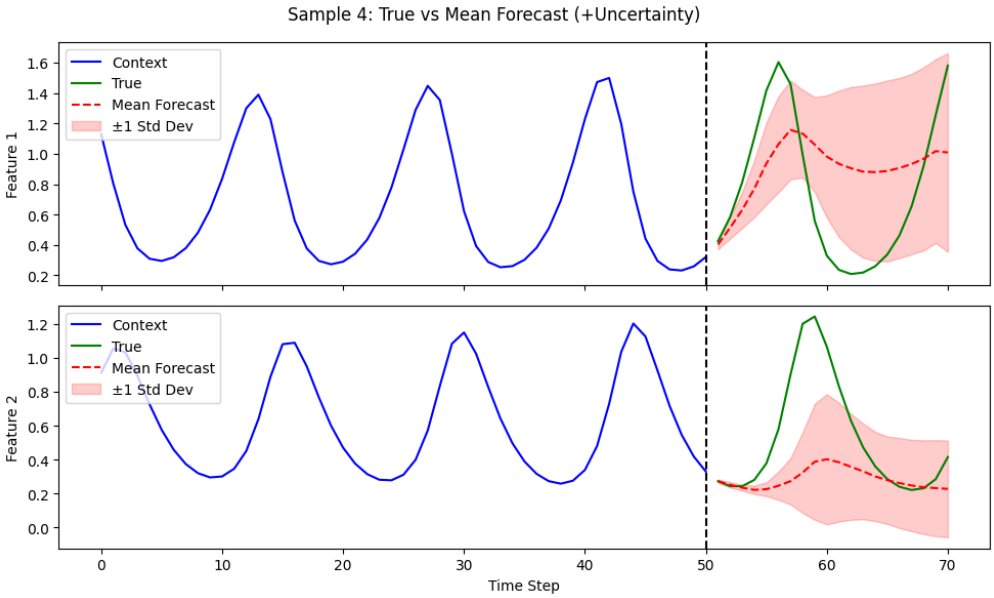
Therefore, for each linear layer wrapped with LoRA, the additional FLOPs during training can be expressed as:

$$\text{FLOPs}_{\text{LoRA}} = 3 \cdot 2 \cdot (B \cdot L) \cdot r \cdot (d_{\text{in}} + d_{\text{out}}) = 6(B \cdot L) \cdot r \cdot (d_{\text{in}} + d_{\text{out}}).$$

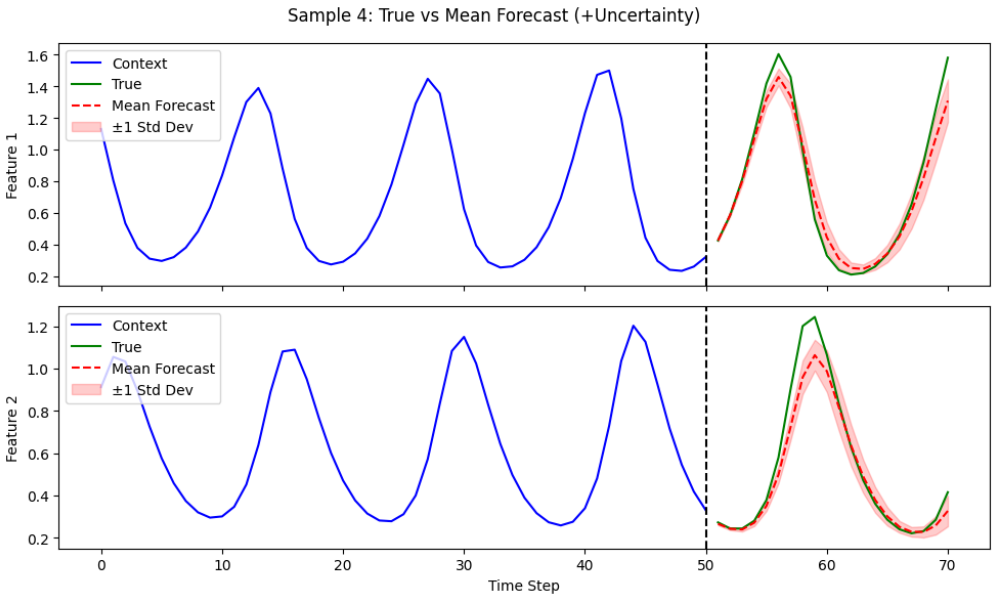
4.8 Report on FLOPs Usage

We summarize the training cost across all experimental settings in Table 13. The overall budget across all nine experiments is approximately:

$$\log_{10}(\text{Total FLOPs}) \approx 16.9995,$$



(a) Original Qwen2.5 model under temperature sampling



(b) Finetuned model under temperature sampling

Figure 11: Forecast distributions on the fourth test sequence using temperature sampling.

indicating that our finetuning strategy is computationally efficient and suitable for medium-scale resource environments.

Table 13: Breakdown of training FLOPs usage across all experiments.

Experiment	Steps	L	r	$\log_{10}$(LoRA FLOPs)	$\log_{10}$(Qwen FLOPs)	$\log_{10}$(Total FLOPs)
Basic Setting	2000	512	4	12.822	16.131	16.131
Learning Rate = 5e-5	2000	512	4	12.822	16.131	16.131
Learning Rate = 1e-4	2000	512	4	12.822	16.131	16.131
LoRA Rank = 2	2000	512	2	12.521	16.131	16.131
LoRA Rank = 8	2000	512	8	13.123	16.131	16.131
Context = 128	1000	128	8	12.220	15.214	15.215
Context = 512	1000	512	8	12.822	15.830	15.830
Context = 768	1000	768	8	12.999	16.014	16.015
Best Setting	2000	512	8	13.123	16.131	16.131

5 Discussion and Conclusion

Discussion. This work explores parameter-efficient fine-tuning for time-series forecasting using Qwen2.5-0.5B-Instruct. By adapting the model under the LLMTIME framework with LoRA, we show that lightweight adaptation achieves strong performance on structured sequences. Ablation studies confirm that tuning learning rate, context length, and LoRA rank significantly affects accuracy. Our best model (lr $= 1 \times 10^{-4}$, rank $= 8$, context length = 512) achieves high forecasting precision with low uncertainty. FLOPs estimation indicates this performance is attainable under modest compute, with total training below 10^{17} FLOPs.

Recommendations for Time-Series Fine-Tuning under Tight Compute Budgets. We recommend the following scheme:

- Use LoRA with moderate rank for a balance of efficiency and expressiveness.
- Limit context length to around 512 tokens for optimal trade-off.
- Greedy decoding is sufficient for a good prediction performance.
- Reuse pretrained or intermediate checkpoints to reduce FLOPs and speed up training.

These insights show that compact LLMs, when adapted effectively, are competitive on numerical forecasting tasks.

Limitations and Future Work While our experiments provide strong evidence for the effectiveness of LoRA-based fine-tuning, several limitations remain that offer directions for future research:

(1) In some cases, fine-tuned models underperform the base Qwen2.5. This suggests that small learning rates or low-rank adaptation may be insufficient to steer the model away from language modeling optima. With access to larger compute budgets, it would be valuable to continue fine-tuning for more steps to better understand the model's optimization dynamics.

(2) We did not explore forecasting under missing or irregular data—a common real-world challenge. Whether LoRA helps or hurts in such settings is worth investigating.

(3) Our LoRA setup is fixed to `q_proj` and `v_proj`. The original LoRA paper considers richer combinations for `Q/K/V/O` with varying ranks. Systematic exploration of these configurations may improve adaptation.

(4) Due to their language pretraining, LLMs often produce verbose or irrelevant outputs in forecasting. Improving prompts or using preference alignment to guide concise generation is crucial for usability.

(5) Finally, we use Qwen2.5's default tokenizer. A task-specific tokenizer for time series data may better suit numerical sequences and improve both efficiency and accuracy.

Future work along these lines—especially with more compute—can yield deeper insights and more deployable models.

References

[1] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

[2] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[3] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[4] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[5] Nate Gruver, Marc Finzi, Shikai Qiu, and Andrew G Wilson. Large language models are zero-shot time series forecasters. *Advances in Neural Information Processing Systems*, 36:19622–19635, 2023.

[6] Joshua Ainslie, James Lee-Thorp, Michiel De Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.

[7] Noam Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

[8] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

[9] Biao Zhang and Rico Sennrich. Root mean square layer normalization. *Advances in Neural Information Processing Systems*, 32, 2019.

[10] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[11] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.

A Qwen2.5-0.5B-Instruct Model Structure

```
Qwen2ForCausalLM(
  (model): Qwen2Model(
    (embed_tokens): Embedding(151936, 896)
    (layers): ModuleList(
      (0-23): 24 x Qwen2DecoderLayer(
        (self_attn): Qwen2Attention(
          (q_proj): Linear(in_features=896, out_features=896, bias=True)
          (k_proj): Linear(in_features=896, out_features=128, bias=True)
          (v_proj): Linear(in_features=896, out_features=128, bias=True)
          (o_proj): Linear(in_features=896, out_features=896, bias=False)
        )
        (mlp): Qwen2MLP(
          (gate_proj): Linear(in_features=896, out_features=4864, bias=False)
          (up_proj): Linear(in_features=896, out_features=4864, bias=False)
          (down_proj): Linear(in_features=4864, out_features=896, bias=False)
          (act_fn): SiLU()
        )
        (input_layernorm): Qwen2RMSNorm((896,), eps=1e-06)
        (post_attention_layernorm): Qwen2RMSNorm((896,), eps=1e-06)
      )
    )
    (norm): Qwen2RMSNorm((896,), eps=1e-06)
    (rotary_emb): Qwen2RotaryEmbedding()
  )
  (lm_head): Linear(in_features=896, out_features=151936, bias=True)
)
```