M2: Deep Learning Coursework Report

Alik Mondal (am3353)
Department of Physics, University of Cambridge

April 8, 2025

Word Count: 2541
Declaration of Autogeneration Tools: I declare that I have used ChatGPT and Perplexity AI as autogeneration tools for assistance with ideas, explanations, and drafting text for this coursework. These tools were utilized in accordance with the course guidelines to supplement my understanding and enhance the quality of my submission.

1 Introduction

This coursework explores the feasibility of using large language models (LLMs) for time series forecasting, with a focus on the Lotka–Volterra predator-prey system. We employ the Qwen2.5-Instruct model, enhanced via Low-Rank Adaptation (LoRA) fine-tuning, and preprocess the numerical time series data using LLMTIME-style formatting to make it compatible with LLM input formats. All experiments are conducted under a strict compute budget of 10^{17} FLOPs.

We begin by evaluating the zero-shot performance of the pre-trained instruction model using both instruction-based and normal prompting strategies. This is followed by LoRA-based fine-tuning for 5,000 optimization steps, allowing us to compare its performance with the zero-shot baseline. We then carry out hyperparameter optimization over LoRA rank, learning rate, and context length, while tracking FLOPs usage throughout. Finally, we train a model with the best hyperparameter configuration for 10,000 optimization steps, ensuring all training remains within the compute constraint. All results are logged for comprehensive analysis.

2 Data Loading and Preprocessing

2.1 Lotka-Volterra System

We use the provided dataset of the Lotka-Volterra system, which includes population values for both prey and predators over time. The dataset, provided in HDF5 format, contains 1,000 different predator-prey systems, each with 100 time points and 2 population variables (prey and predator), as shown in Figure 1. The execution of the dataset extraction code is modularized in `src/data_create.py`, where an 80:20 temporal split of the data is performed for forecasting performance comparison over future time steps.

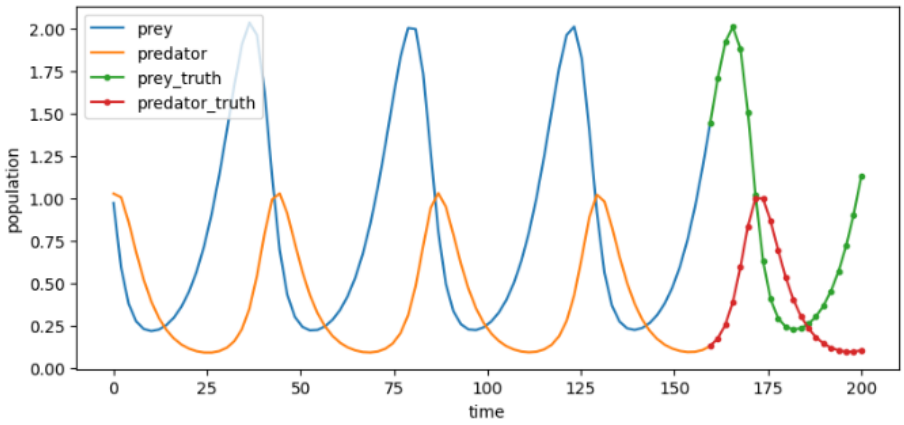


Figure 1: Prey-predator population dynamics of a single Lotka-Volterra system, in forecasting format.

2.2 LLMTIME Preprocessing

Following [1], the time series is scaled and formatted into a structured prompt. Our implementation extends the original LLMTIME approach to handle multivariate time series data for Qwen 2.5.

2.2.1 Scaling and Precision

Instead of scaling by a fixed constant, we use a data-dependent normalization strategy. For each input sample, we compute an offset based on the minimum value of the series, extended downward by a fraction β of the range. The series is then scaled using the α-quantile of the offset-adjusted values:

$$x'_t = \frac{x_t - \text{offset}}{\text{scale}} \tag{1}$$

where $\text{offset} = \min(x) - \beta \cdot (\max(x) - \min(x))$ and $\text{scale} = \text{Quantile}_\alpha(x - \text{offset})$.

We use $\alpha = 0.99$ and $\beta = 0.3$ in our experiments. This adaptive normalization brings most values into a compact range (typically between 0 and 10) while preserving temporal structure. The normalized values are then rounded to a fixed number of decimal places (default: 3) for consistent token formatting.

2.2.2 Multivariate Encoding

To encode multivariate time series data (such as predator-prey population dynamics), we adopt a structured text format where each timestep contains comma-separated variables, and timesteps are separated by semicolons.

An example of an encoded series for the model is as follows:

```
0.25,1.50;0.27,1.52;0.30,1.48;0.34,1.42;0.39,1.35;0.46,1.26;
0.54,1.17;0.64,1.08;0.75,1.00;0.87,0.95; ...
```

This encoded representation is used as the input prompt for the LLM after tokenization. At inference time, predicted values are denormalized using the stored offset and scaling parameters to recover the original value range.

2.3 Prompting

For zero-shot forecasting with the pre-trained model, we experiment with two different prompting strategies.

First, given that Qwen 2.5 is an instruction-tuned model, we adopt a ChatGPT-style prompting format to provide explicit context and guide the model's output. The prompt is structured as follows:

```
 <|im_start|>user
I have time series for prey and predator populations.
The data is formatted as: {encoded_series};
Predict the next {forecast_length} points in the same format as below:
{prey_11, pred_11; prey_12, pred_12; ...}
JUST PREDICT DON'T SAY ANYTHING
<|im_end|>
<|im_start|>assistant
```

This style was used only for zero-shot inference, where we observed that contextualizing the generation task improved the quality of the next-token predictions. However, for an analogous training scheme, we used a localized preprocessing schema with smaller context windows and fixed-length forecast targets. This approach, though structurally simpler, did not yield significantly better training results.

The second approach involves directly feeding the encoded time series into the model without any instruction format for zero-shot. In this case, overlapping chunks of tokenized sequences are used for next-token prediction in fine-tuning settings. This method is better suited for autoregressive fine-tuning workflows.

Both prompting are implemented in `src/forecast.py`.

2.4 Tokenization

We preserve Qwen's natural tokenization process, allowing it to effectively handle numerical strings, including decimal numbers, ensuring they are tokenized appropriately.

In the tokenization process, we take into account both the maximum sequence length and an overlapping stride mechanism to ensure that context is retained across chunks. The input text is divided into chunks, each not exceeding the context length, with an overlap of `context_length // 2` tokens between consecutive chunks. This overlap helps the model retain context from one chunk to the next. The primary function for tokenization processes a list of texts by encoding them into tokens using the Qwen2.5 tokenizer, dividing the tokenized text into manageable chunks of the specified maximum length, and applying padding where necessary to ensure each chunk reaches the maximum length. The resulting chunks are then returned as input IDs, ready for the model to consume. This approach ensures that the input sequences are appropriately tokenized and chunked for both inference and training, while maintaining continuity between chunks.

```
Encoded-Sequence-0:  0.571,0.772;0.506,0.776;0.446,0.761;0.397,
0.728;0.359,0.682;0.333,0.629;0.317,0.574; ...
Tokenized-Sequence-0:  tensor([15, 13, 20, ..., 24, 19, 26])

Encoded-Sequence-1:  0.46,1.132;0.313,0.965;0.256,0.754;0.24,
```

```
0.581;0.246,0.456;0.269,0.371;0.311,0.316;0.37 ...
Tokenized-Sequence-1:  tensor([15, 13, 19, ..., 17, 24, 26])
```

2.5 Train/Test Splits

We randomly sample sequences for training and validation with an 80/20 split, where 80% (800 trajectories) are used for training and 20% (200 trajectories) for validation. In addition to this random sampling, for forecasting tasks, we perform a temporal split of 80/20, where the first 80% of the time points are used for training and the remaining 20% are reserved for validation. This time-based split ensures that the model is tested on future time points.

3 Model Architecture

3.1 Qwen2.5-Instruct

Qwen2.5-Instruct is a multilingual instruction-tuned large language model designed to follow complex user prompts more effectively across languages. Building on the Qwen2.5 architecture, it incorporates several key innovations aimed at improving efficiency and generalization.

The model architecture consists of:

- **Transformer backbone** using Grouped Query Attention (GQA), which allows for efficient attention computation across multiple heads while reducing memory usage. For FLOPs estimation, we assume a standard Multi-Head Attention (MHA) implementation instead of GQA.
- **SwiGLU** (Switchable Gated Linear Units) as the activation function, improving performance over standard ReLU or GELU activations by enabling better gradient flow.
- **RMSNorm**, a root mean square normalization technique that stabilizes training without relying on mean subtraction like traditional LayerNorm.
- **Rotary Positional Embeddings** (RoPE) to encode positional information in a way that generalizes better to longer sequences. For FLOPs estimation, we ignore the operations required to compute the positional encodings themselves, but include the cost of adding them to the token embeddings.
- **Loss Function:** The model is trained using the standard causal language modeling (CLM) loss, implemented as a token-level cross-entropy loss to predict the next token given the previous context.

In our experiments, we utilize the 0.5B parameter version of Qwen2.5-Instruct quantized to 4-bit precision. This quantization significantly reduces memory requirements while preserving model accuracy, enabling inference on consumer-grade GPUs with 24 GB memory and allowing us to stay within our local compute constraints. Additionally, we assume that the backward pass requires twice the number of FLOPs as the forward pass.

3.2 LoRA Fine-Tuning

We use LoRA to efficiently adapt the model by inserting low-rank decomposition matrices into attention projection layers, `q_proj`, `v_proj`. This approach significantly reduces the number of

trainable parameters while maintaining adaptation effectiveness. Instead of updating the full weight matrix, we decompose the update into two smaller matrices and only train them, while freezing the original model weights. Given a weight matrix $W \in \mathbb{R}^{d\times k}$, LoRA parameterizes the update ΔW as a low-rank decomposition:

$$\Delta W = BA \tag{2}$$

where $B \in \mathbb{R}^{d\times r}$, $A \in \mathbb{R}^{r\times k}$, and $r \ll \min(d, k)$ is the rank of the decomposition. In our implementation, we set the scaling factor $\alpha/r = 1$ by choosing $\alpha = r$, so no explicit scaling is applied to the LoRA update.

Our initial LoRA configuration:

- Rank: 4
- Context Length: 512
- Learning Rate: 1e-5
- Batch Size: 4

4 Training Strategy and FLOP Budgeting

4.1 Compute-Efficient Design

The coursework imposed a strict compute budget of 10^{17} FLOPs for both training and inference. FLOP costing followed specific guidelines: basic arithmetic operations such as addition, subtraction, and multiplication count as 1 FLOP; backward passes are estimated as twice the cost of the forward pass; activation functions like ReLU or GELU cost 1 FLOP; and more complex operations such as exponentiation or logarithms cost 10 FLOPs. Given this, accurate accounting of FLOPs was crucial in both model design and experimental planning. However, the local compute environment posed limitations, and our heavy reliance on CSD3 significantly slowed the development cycle. Job queuing on CSD3 often took several hours even for single-GPU submissions, which severely restricted our ability to iterate quickly and validate changes, making compute-efficient choices a practical necessity rather than just an academic constraint.

We froze the base model weights and only trained LoRA adapters, systematically experimenting with smaller LoRA ranks (2, 4, 6) to limit parameter updates with different learning rates. We also varied the context length (128, 512, 768) through later experimentation to assess the trade-off between sequence resolution and computational cost. Batch sizes and the number of training samples were reduced, and the training loop was capped at usually below 10,000 steps. Additionally, we restricted validation to a maximum of 5 samples per training step to keep testing overhead minimal. These cumulative optimizations allowed us to maintain compliance with the FLOP limit while still achieving meaningful adaptation and evaluation of our models.

4.2 FLOP Calculation

We estimate the floating point operations (FLOPs) for a forward and backward pass through the Qwen2.5-0.5B model (regular and with LoRA) based on architectural configurations.

4.2.1 Token Embeddings

The token embedding layer includes the addition of learned token and positional embeddings. For each token, this involves a simple vector addition of length equal to the hidden size. Therefore, the FLOPs per token are:

$$\text{FLOPs}_{\text{embedding}} = \text{hidden\_size}$$

For a sequence length of `seq_len` tokens and a hidden size of 896, this results in:

$$\text{Total}_{\text{embedding}} = \texttt{seq\_len} \times 896 \text{ FLOPs}$$

4.2.2 Self-Attention Layers

We approximate the FLOPs of the Grouped Query Attention (GQA) mechanism by treating it as standard multi-head attention. Each attention layer includes:

- Linear projections for queries, keys, and values,
- Dot-product attention computation,
- Softmax normalization,
- Output projection.

These operations are performed for each head and repeated across all transformer layers. For 24 hidden layers and 14 attention heads, the total attention FLOPs are estimated as:

$$\text{FLOPs}_{\text{attn}} \approx 24 \times \Big(\text{QKV projections} + \text{QK}^\top \text{ attention scores} + \text{softmax} + \text{output projection}\Big)$$

4.2.3 Feed-Forward Network (FFN)

Each transformer layer contains a SwiGLU MLP, consisting of:

- Two linear projections (up and gate),
- Element-wise SwiGLU activation,
- One linear down-projection.

Each projection involves large matrix multiplications, and activations are applied element-wise. The total FLOPs for the FFN across all 24 hidden layers is approximately:

$$\text{FLOPs}_{\text{ffn}} \approx 24 \times (2 \times \text{hidden\_size} \times \text{intermediate\_size} + \text{activations} + \text{down projection})$$

For hidden size = 896 and intermediate size = 4864, this evaluates to:

$$\text{FLOPs}_{\text{ffn}} \approx 24 \times (2 \times 896 \times 4864 + \text{activations} + \text{down projection})$$

4.2.4 RMS Layer Normalization

Each transformer layer uses RMS normalization (RMSNorm) twice — once before the attention block and once before the MLP. For each RMSNorm operation, the following computations are performed per input element:

1. Compute the square: 1 multiplication,
2. Sum of squares: (elements − 1) additions,
3. Division for the mean: 1 division,
4. Square root of the mean: ∼10 FLOPs,
5. Element-wise normalization: 1 division per element,
6. Element-wise scaling: 1 multiplication per element.

Thus, the total FLOPs per RMSNorm operation for `elements` input values is:

$$\text{FLOPs}_{\text{RMSNorm}} \approx 4 \times \text{elements} + 10$$

For 48 RMSNorm operations (2 per layer across 24 layers), the total normalization cost is:

$$\text{FLOPs}_{\text{layernorm}} = 48 \times (4 \times (\texttt{seq\_length} \times \texttt{hidden\_size}) + 10)$$

4.2.5 Total Forward and Backward Pass

Assuming the backward pass requires twice the number of FLOPs as the forward pass (a common approximation), the total computational cost per training step can be expressed as:

$$\text{FLOPs}_{\text{total}} = \text{FLOPs}_{\text{forward}} + 2 \times \text{FLOPs}_{\text{forward}} = 3 \times \text{FLOPs}_{\text{forward}}$$

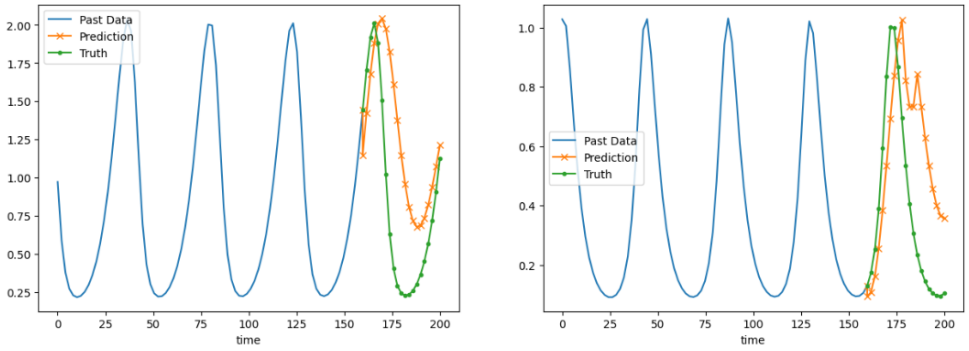
Note: In this estimation, we assume a batch size of 1 for simplicity. In practice, FLOPs scale linearly with batch size.

4.3 Zero-Shot Baseline (Untrained Qwen 2.5)

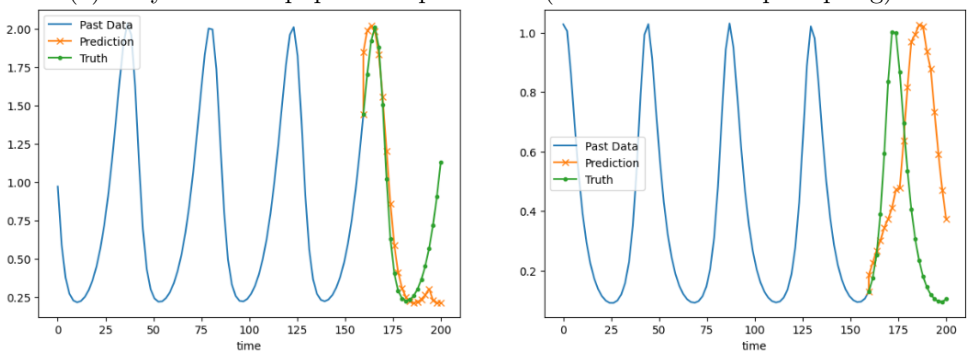
We evaluate Qwen2.5 without fine-tuning as a baseline. In this evaluation, we perform only one forward pass with probabilistic sampling, using a temperature of approximately 0.2.

We use two different prompting styles for this baseline, as discussed previously in the Prompting section of this report: one basic/direct and the other instruction-based.

The performance predictions for both prompting styles are shown in Figure 2a and Figure 2b for the prey and predator populations, respectively.



(a) Prey-Predator population prediction (instruction-based prompting).



(b) Predator population prediction (direct prompting).

Figure 2: Zero-shot baseline forecasts for prey and predator populations using Qwen2.5 with different prompting strategies.

4.4 Default LoRA Fine-Tuning

4.4.1 Training Configuration

The default training configuration is provided in the `src/config.yaml` file. The main parameters include:

- **Training steps**: 5000
- **Batch Size**: 4
- **Context Length**: 512
- **LoRA Rank**: 4
- **Forecast Length**: 21
- **Learning Rate**: 1e-5
- **Validation Limit**: 5

The LoRA training script is located in `src/lora_train.py`, which handles model loading, modification, and data loading upon initialization. The model can be trained using the `.train()` method, which only takes input the model hyperparameter configuration.

We train the LoRA-adapted model for 5000 training steps, with a validation limit of 5 at each step to reduce computational overhead and time. The model and optimizer states are checkpointed every 250 iterations. Since we average out the validation loss at every training

iteration, we observe a much smoother averaged validation curve, o contrast to the stochastic training curve as shown in Figure 3. This is in contrast to the more stochastic training curve, with both losses decreasing and stagnating around 0.25 for the given hyperparameter bounds.

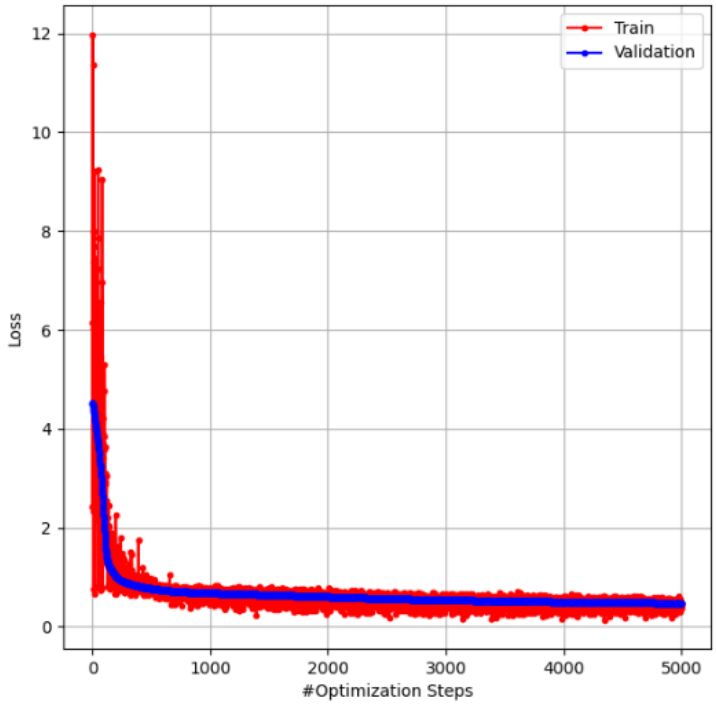


Figure 3: Training and averaged validation loss curve over 5000 steps for the default LoRA fine-tuning setup.

The prediction quality is not satisfactory, as we observe repeated words (particularly Chinese characters) and redundant token generation in the numerical forecasts for most validation system samples, despite the loss curve showing a smooth decline. We performed extensive cleaning and post-processing of the generated output texts even before decoding, yet only around 10% of the validation samples remained viable—most with significant truncations.

We attempted forecasting using both deterministic and probabilistic decoding strategies, combined with instruction-based and basic prompting techniques; however, the outputs remained largely incoherent and nonsensical in general and at scale. We suspect a case of catastrophic forgetting during training, where the model relies heavily on its pre-trained priors, showing a strong bias towards generating Chinese characters. We get forecasts like these for the better ones:

```
0.239,0.312;澳IGN灶职权;0.26,0.281;杂瑞腾;0.291,0.259; ...
```

Figure 4: Default LoRA Fine-tuned Model output (around best ∼10% of the entire validation set).

Subsequent hyperparameter optimization yielded better results in terms of loss; however, the quality of generated forecasts unfortunately did not improve.

All training and output results are logged in the `log/slora_error.log` and `log/slora_output.log` files.

Figure 5 shows the prey-predator forecasting performance.

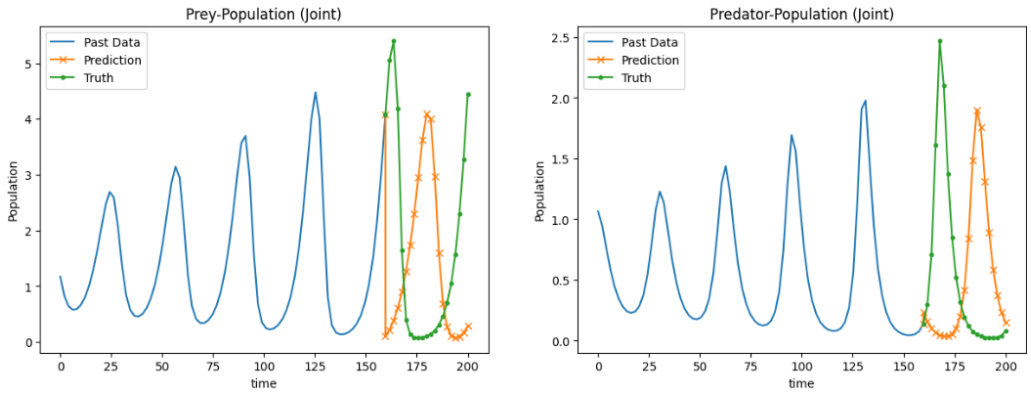


Figure 5: Default LoRA Fine-tuned Model for forecast trained on 5000 steps.

4.4.2 LoRA FLOPs Calculation

For each LoRA module, there are two trainable matrices:

- Matrix A of shape (d_{in}, r)
- Matrix B of shape (r, d_{out})

Assuming $d_{\text{in}} = d_{\text{out}} = \text{hidden\_size} = 896$, and considering that the LoRA adapter is applied to both the Q and V projections in each transformer layer, the number of LoRA modules is:

$$\text{num\_lora\_modules} = 2 \times \text{num\_layers} = 2 \times 24 = 48$$

The number of FLOPs required for the LoRA backward pass is approximated by:

$$\text{FLOPs}_{\text{lora\_backward}} = 2 \times \text{FLOPs}_{\text{forward}} \times \frac{\text{lora\_params}}{\text{qv\_params}}$$

where:

$$\text{lora\_params} = 2 \times \text{hidden\_size} \times r \times \text{num\_lora\_modules}$$
$$\text{qv\_params} = 2 \times \text{hidden\_size} \times \text{hidden\_size} \times \text{num\_layers}$$

We effectively scale the backward pass FLOPs by the ratio of LoRA parameters to full Q and V parameters. This results in a significant reduction in training computation when fine-tuning with LoRA adapters instead of full-model fine-tuning.

Thus, the total FLOPs for a LoRA training step are:

$$\text{FLOPs}_{\text{LoRA}} = \text{FLOPs}_{\text{forward}} + \text{FLOPs}_{\text{lora\_backward}}$$

4.5 Empirical FLOPs Comparison (Regular vs LoRA)

We conducted empirical FLOPs measurements on the Qwen2.5-0.5B model with the default configeration (context length: 512, batch size: 4, lora rank: 4) for 5000 training steps. Below is a summary of the total estimated FLOPs:

- **Forward Pass FLOPs (per step):** 1.564×10^{12}

- **Regular Training FLOPs (F + B) (per step):** 4.7×10^{12}
- **LoRA Training FLOPs ($F + B_{LoRA}$) (per step):** 1.59×10^{12}
- **Total FLOPs over 5000 Steps:**
  - Regular: 2.34×10^{16}
  - LoRA: 7.96 $\times 10^{15}$ (using updated ratio)

Component	Regular Training	LoRA Training
Embedding Layer	1.8×10^6	1.8×10^6
Attention (per layer)	1.14×10^{10}	1.14×10^{10}
MLP (per layer)	5.36×10^{10}	5.36×10^{10}
Layer Norm	7.3×10^6	7.3×10^6
Transformer Layer	6.5×10^10	6.5×10^10
Final layer	1.1×10^9	1.1×10^9
Forward Pass	1.56×10^{12}	1.56×10^{12}
Backward Pass	3.1×10^{12}	3.4×10^5
Total Training Step	4.69×10^{12}	1.59×10^{12}

Table 1: Empirical FLOPs per training step: Regular vs LoRA (r=4, seq=512, bs=4)

The use of LoRA adapters (with just 215,040 additional parameters) reduces total training step FLOPs by over **65%**, enabling significantly faster and cheaper model fine-tuning.

5 LoRA Hyperparameter Search

We conducted a two-stage hierarchical hyperparameter grid search using `src/hyper_search.py`. The first stage involved sweeping over LoRA matrix ranks {2, 4, 8} and learning rates {1e-4, 5e-5, 1e-5}, each configuration trained for 500 steps. The best configuration was selected based on validation loss capped at a predefined validation limit, with results stored in `hyperparameter_results.json`.

In the second stage, using the best-performing configuration from the first stage, we performed a context length ablation over {128, 512, 768}, again for 500 training steps (see Figure 6). This was done under both FLOPs constraints and practical limitations due to the unreliability of the CSD3 cluster, which could become unavailable at any moment.

Simultaneously, we evaluated forecasting performance using metrics such as MAE, MSE, and RMSE via the `src/eval.py` script to observe how performance scales with context length.

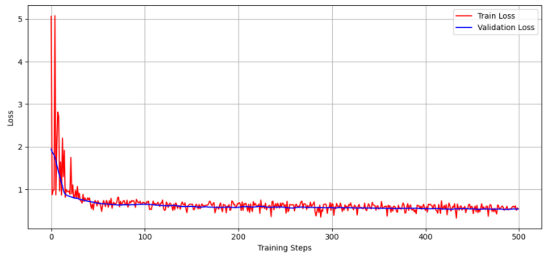
Best Configuration from First Grid Search

- LoRA Rank: 8
- Learning Rate: 1×10^{-4}

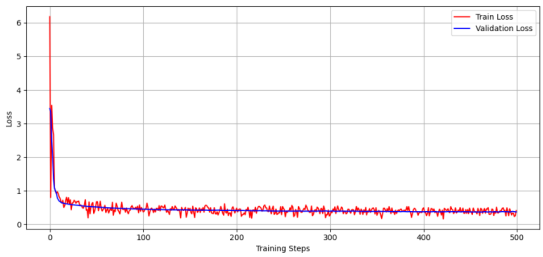
Best Configuration from Second Grid Search

- LoRA Rank: 8
- Learning Rate: 1×10^{-4}

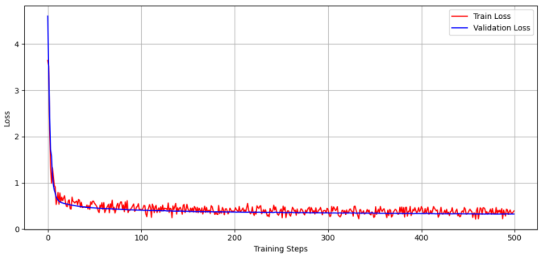
- Context Length: 768



(a) Context Length 128



(b) Context Length 512



(c) Context Length 768

Figure 6: Loss Curve Comparison for Context-Based Hyperparameter Search with Different Context Lengths.

5.1 Performance vs. Context Length

We evaluated the impact of input context length on forecasting performance using three configurations: 128, 512, and 768 tokens. The `src/eval.py` script was used to compute standard regression metrics—Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and Mean Absolute Error (MAE)—on a held-out validation set consisting of prey-predator population sequences.

Overall, we observed minimal differences in forecasting performance across the three context lengths. While increasing the context window from 128 to 512 tokens resulted in a slight improvement, the gains from 512 to 768 tokens were marginal. This suggests that a context length of 512 tokens is sufficient to capture the relevant temporal dependencies, offering a good trade-off between accuracy and computational efficiency.

Context Length	**MSE**	**RMSE**	**MAE**
128 tokens	0.81	0.75	0.65
512 tokens	0.79	0.68	0.62
768 tokens	0.79	0.67	0.62

Table 2: Forecasting performance across different context lengths

5.2 Final Model

We use a dual-stage hyperparameter search to determine the final model configuration, which is implemented in `src/final_model.py`. The final model is trained for 10,000 steps and evaluated on 100 validation samples, constrained by both computational resources and time limitations on the CSD3 system. We observed lower validation loss and more stable training dynamics, as

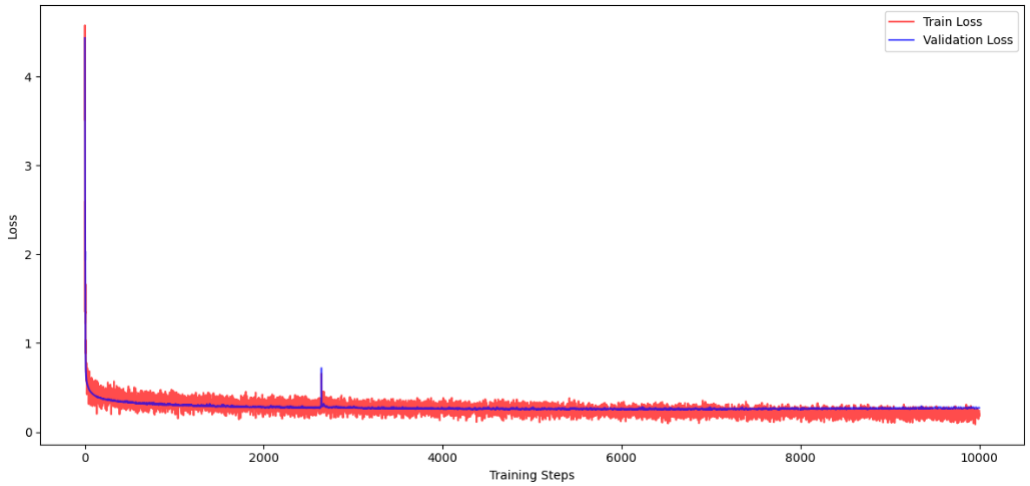
shown in Figure 7a but didn't much forecasting improvement and obtained the same rate of generation corruption with chinese or other foreign characters.

To assess the computational cost, we used a FLOPs counter logger, which recorded the total FLOPs used during training. These logs are stored in `logs/final_model_single.log`, consistent with the logging procedures from earlier runs. The final model demonstrated no significant improvement in forecasting, as visualized in Figure 7b.

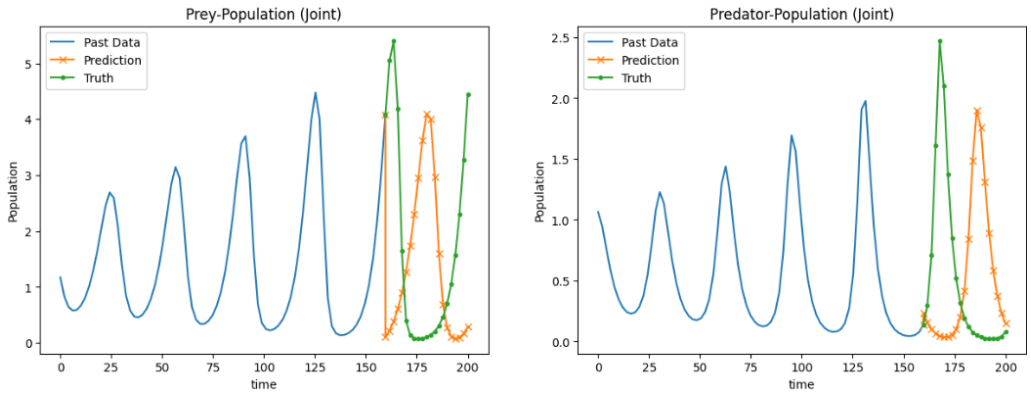
Similarly, to the default LoRA training we obtained almost 10

Final Model Configuration

- LoRA Rank: 8
- Learning Rate: 1×10^{-4}
- Context Length: 768
- Training Steps: 10,000



(a) Final loss curve



(b) Forecasting performance

Figure 7: Best model performance: Final loss curve and forecasting performance on held-out validation samples.

6 Performance

We evaluate model performance using a suite of standard forecasting metrics to assess the quality of predicted sequences against ground truth values (the remaining 20% of timesteps during temporal splitting). These include:

- **Mean Absolute Error (MAE)**: Average magnitude of the errors in a set of predictions, without considering their direction.
- **Mean Squared Error (MSE)**: Average squared difference between the predicted and actual values.
- **Root Mean Squared Error (RMSE)**: Square root of MSE, offering interpretability in the same unit as the target variable.

These metrics were computed using the `src/eval.py` script on a held-out validation set consisting of predator-prey population sequences. To evaluate the model's performance on the validation set after training, we run the following command in the terminal:

```
python src/eval.py --model_type=lora --lora_model_path=saves/lora_model.pt
--val_limit=10
```

We compare three different model configurations to assess forecasting performance:

- **Zero-shot Qwen2.5**: The pretrained language model baseline without any fine-tuning.
- **Default Fine-Tuned Model**: LoRA-adapted model trained with default hyperparameters (e.g., LoRA rank = 4, LR = 1×10^{-5}, context length = 512).
- **Best Fine-Tuned Model**: Final model obtained via two-stage hyperparameter search (e.g., LoRA rank = 8, LR = 1×10^{-4}, context length = 768).

Model	MAE	MSE	RMSE
Zero-shot Qwen2.5	0.66	0.73	0.72
Default Fine-Tuned	0.68	0.71	0.75
Best Fine-Tuned	0.68	0.73	0.76

Table 3: Comparison of forecasting performance across different model configurations

As we can see in Table 3, we don't observe significant improvements for either fine-tuned model, even though training proceeded as expected. This is due to non-trivial errors during training, leading to suboptimal generation. Both models perform similarly to the baseline when evaluated on average over the validation dataset, which was also severely limited for the LoRA models due to generation corruption.

7 FLOP Summary Calculation

7.0.1 FLOPs Calculation Across Experiments

For the different experiments, we calculate the total number of FLOPs involved for each training phase, excluding inference-related FLOPs.

7.0.2 2x Baseline Inference Run (Zero-shot, `ctx_len` = 512, Two Prompting Techniques)

As inference-related computation is excluded from the total FLOPs accounting, both zero-shot runs do not contribute to the final FLOPs total.

7.0.3 1x LoRA Fine-Tuning (batch size = 4, `ctx_len` = 512, LoRA rank = 4, 5000 steps)

In this run, the model is fine-tuned with: batch size = 4, context length = 512, LoRA rank = 4, for 5000 training steps.

The total FLOPs for a single training step are given by:

$$\text{FLOPs}_{\text{train}} = \text{FLOPs}_{\text{forward}} + 2 \times \text{FLOPs}_{\text{forward}} \times \frac{\text{lora\_params}}{\text{qv\_params}}$$

where:

$$\text{lora\_params} = 2 \times 896 \times 4 \times 48 = 344{,}064$$

$$\text{qv\_params} = 2 \times 896 \times 896 \times 24 = 38{,}486{,}016$$

So the backward FLOPs are scaled by a factor of approximately:

$$\frac{\text{lora\_params}}{\text{qv\_params}} \approx \frac{344{,}064}{38{,}535{,}168} \approx 0.0089$$

Thus:

$$\text{FLOPs}_{\text{train}} \approx \text{FLOPs}_{\text{forward}} \times (1 + 2 \times 0.0089) \approx 1.0178 \times \text{FLOPs}_{\text{forward}}$$

The total FLOPs for this full fine-tuning run (5000 steps) becomes:

$$\text{Total}_{\text{LoRA}} = 5000 \times \text{FLOPs}_{\text{train}}$$

7.0.4 1st Hyperparameter Search (9x) for LoRA Rank and Learning Rate

In the first hyperparameter search, we evaluate 9 configurations by varying:

- LoRA rank, $r \in \{2, 4, 8\}$, and
- Learning rate, lr $\in \{$1e-4, 5e-5, 1e-5$\}$.

For these runs, the context length is fixed (`ctx_len` = 512) so that the forward FLOPs, $\text{FLOPs}_{\text{forward}}(512)$, remain constant across configurations except for the influence of the LoRA parameters. In our configuration:

- hidden_size = 896,
- num_hidden_layers = 24, and

- the LoRA adapter is applied to both the Q and V projections, hence

$$\text{num\_lora\_modules} = 2 \times 24 = 48.$$

For a given LoRA rank r, the number of LoRA parameters is:

$$\text{lora\_params} = 2 \times 896 \times r \times 48,$$

and the total number of parameters in the Q and V projections is:

$$\text{qv\_params} = 2 \times 896 \times 896 \times 24.$$

Thus, for each configuration, the FLOPs for one run (500 training steps) are approximated by:

$$\text{FLOPs}_{\text{run}} = 500 \times \text{FLOPs}_{\text{forward}}(512) \times \left(1 + 2 \times \frac{\text{lora\_params}}{\text{qv\_params}}\right).$$

7.0.5 2nd Hyperparameter Search (3x) for Context Length with Best LoRA Rank and Learning Rate

In the second hyperparameter search, we fix the best-performing LoRA rank r_{best} and learning rate from the first search, and vary the context length over:

$$\texttt{ctx\_len} \in \{128, 512, 768\}.$$

Since the forward FLOPs scale linearly with the sequence length, for each context length the FLOPs for one run (500 training steps) are given by:

$$\text{FLOPs}_{\text{run}}(\texttt{ctx\_len}) = 500 \times \text{FLOPs}_{\text{forward}}(\texttt{ctx\_len}) \times \left(1 + 2 \times \frac{\text{lora\_params (with } r_{\text{best}})}{\text{qv\_params}}\right).$$

The total FLOPs for this second search is the sum over the three context lengths:

$$\text{Total}_{\text{HPO,2}} = \sum_{\texttt{ctx\_len} \in \{128,512,768\}} \text{FLOPs}_{\text{run}}(\texttt{ctx\_len}).$$

7.0.6 Final Run with Best Hyperparameters (10000 Steps)

The final run uses the best hyperparameters identified from the previous searches: the optimal LoRA rank r_{best}, best learning rate, and the best context length $\texttt{ctx\_len}_{\text{best}}$. The total FLOPs for the final run (10000 training steps) is calculated as:

$$\text{FLOPs}_{\text{final}} = 10000 \times \text{FLOPs}_{\text{forward}}(\texttt{ctx\_len}_{\text{best}}) \times \left(1 + 2 \times \frac{\text{lora\_params (with } r_{\text{best}})}{\text{qv\_params}}\right).$$

In these formulas, note that:

- $\text{FLOPs}_{\text{forward}}(\texttt{ctx\_len})$ scales linearly with `ctx_len`.
- The backward FLOPs are scaled by the ratio of LoRA parameters to the full Q/V parameters, which changes with r.

The extensive calculation table can be produced by running `src/flops_counter.py`, which displays the FLOPs for different experiments.

7.1 FLOP Summary Table

Component	FLOPs
LoRA Fine-tuning (Base)	7.9×10^{15}
LoRA Hyper - 1 Search (9 configs)	8.1×10^{15}
LoRA Hyper - 2 Search (3 configs)	2.2×10^{15}
Final Run (best hyperparams)	2.5×10^{16}
Total	4.5×10^{16}

Table 4: Estimated FLOP usage across training runs

8 Conclusion

In this work, we aimed to forecast predator-prey dynamics using a LoRA-tuned Qwen2.5 model while adhering to a stringent compute budget of up to 10^{17} FLOPs. Although fine-tuning with structured prompts allowed the LLM to capture complex temporal dependencies, the results were not as expected. Despite fine-tuning and hyperparameter optimization, the model failed to show significant improvements in forecasting. In fact, the results were underwhelming, as evidenced by the metrics, which did not show the expected gains. We were able to limit the total FLOPs usage within 10^{17}, achieving the target with only 4.5×10^{16} FLOPs. However, neither fine-tuning nor hyperparameter optimization provided meaningful performance improvements, indicating that the model's limitations could not be overcome within the given computational constraints.

8.1 Recommendations for Compute-Efficient Fine-Tuning

Given the challenges we encountered during this study, we recommend the following strategies for time-series fine-tuning under tight compute budgets:

- **Prioritize Data Quality Over Extensive Fine-Tuning**: Given the limited success of fine-tuning in this context, focusing on high-quality data preprocessing and feature engineering may provide better results. Enhancing data quality can lead to performance improvements without the need for substantial model updates.
- **Optimize Context Length with Caution**: While we found that context length plays a role in model performance, our results suggest diminishing returns beyond a certain point. Thus, selecting a context length that captures essential temporal dependencies, while considering the computational cost, is critical. However, context optimization alone may not yield significant improvements in all cases.

8.2 Future Work

Future research directions include:

- **Exploring Alternative Parameter-Efficient Methods**: Given the limited gains from LoRA-based fine-tuning, it would be valuable to explore alternative parameter-efficient fine-tuning methods such as prefix tuning or adapter modules. These techniques may offer better results in forecasting tasks, especially under stringent compute constraints.

- **Investigating Domain-Specific Architectural Innovations**: Designing model architectures tailored specifically for time-series forecasting, particularly for numerical prediction tasks, could improve performance. Incorporating mechanisms that better capture long-term dependencies and periodic patterns may enhance the model's ability to generalize and extrapolate complex time-series data.

References

[1] Gruver, J., et al. (2023). *LLMTIME: Language Models as Time Series Forecasters.*

[2] Hu, E. J., et al. (2021). *LoRA: Low-Rank Adaptation of Large Language Models.*

[3] Yang, H., et al. (2023). *Qwen Technical Report.*