M2: Lora Finetuning

Xueqing Xu (xx823)

Department of Physics, University of Cambridge

April 7, 2025

Introduction

We fine-tune the Qwen2.5-Instruct LLM for predator-prey population forecasting using Low-Rank Adaptation (LoRA). While LLMs can perform time series forecasting without explicit training[1], we investigate how targeted fine-tuning enhances these capabilities under computational constraints.

LoRA injects trainable low-rank matrices into existing weights without modifying original parameters, dramatically reducing trainable parameter count. This parameter-efficient approach is ideal for large models under limited compute budgets.

Methodology

Qwen2.5-Instruct model architecture

The Qwen2.5-0.5B-Instruct model implements a decoder-only transformer architecture with 494 million parameters (Table 1). With a hidden dimension of 896 across 24 transformer layers, the model balances depth and computational efficiency.

A key architectural feature is Grouped Query Attention (GQA), which employs 14 query heads but only 2 key-value heads—a 7:1 ratio that substantially reduces memory usage during inference while maintaining attention capabilities. For normalization, Qwen2.5 uses RMSNorm with $\epsilon = 10^{-6}$, which offers better training stability than traditional LayerNorm.

Key Components with Mathematical Definitions

RMSNorm (Root Mean Square Normalization): Unlike traditional LayerNorm, RMSNorm eliminates mean-centering and focuses only on variance normalization:

$$\text{RMSNorm}(x) = \frac{x}{\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2 + \epsilon}} \cdot \gamma \tag{1}$$

where $\epsilon = 10^{-6}$ is a small constant for numerical stability, and γ represents trainable scale parameters.

SiLU (Swish) Activation: The SiLU activation function used in the feed-forward networks is defined as:

$$\text{SiLU}(x) = x \cdot \sigma(x) = \frac{x}{1 + e^{-x}} \tag{2}$$

where $\sigma(x)$ is the sigmoid function. This activation combines properties of both ReLU and sigmoid functions.

Rotary Position Embeddings (RoPE): RoPE[3] encodes position information through rotation matrices in the complex plane:

Parameter	**Value**
Model Type	Decoder-only Transformer
Parameters	0.5 Billion
Hidden Size	896
Attention Heads	14
Head Dimension	64
Number of Layers	24
Intermediate Size (MLP)	4864
Vocabulary Size	151,936
Layer Structure (Repeated 24 times):	
Pre-Attention	RMSNorm
Attention	Multi-head attention with RoPE
	- Q, K, V projections
	- Rotary Position Embeddings
Post-Attention	Residual connection
Pre-MLP	RMSNorm
MLP	Gate and Up projections
	SwiGLU activation
	Down projection
Post-MLP	Residual connection
Final Output	RMSNorm + Linear (LM head)

Table 1: Qwen2.5-0.5B Model Architecture

$$\mathbf{q}^{\theta}_{m,i} = \begin{pmatrix} \cos(m\theta_i) & -\sin(m\theta_i) \\ \sin(m\theta_i) & \cos(m\theta_i) \end{pmatrix} \mathbf{q}_{m,i} \tag{3}$$

where $\mathbf{q}_{m,i}$ represents query vector components, m is the position index, and θ_i denotes frequencies constructed with base $\theta = 1,000,000.0$.

SwiGLU Feed-Forward Network: The FFN in each layer consists of three projections:

1. **Gate Projection:** $G(x) = W_g x$, where $W_g \in \mathbb{R}^{4864\times 896}$
2. **Up Projection:** $U(x) = W_u x$, where $W_u \in \mathbb{R}^{4864\times 896}$
3. **Down Projection:** $D(x) = W_d x$, where $W_d \in \mathbb{R}^{896\times 4864}$

The complete FFN operation is defined as:

$$\text{FFN}(x) = D(\text{SiLU}(G(x)) \odot U(x)) \tag{4}$$

where $\odot$ represents element-wise multiplication.

Property	**Value**
Vocabulary Size	151,936 tokens
BOS Token ID	151,643
EOS Token ID	151,645
Word Embeddings	Tied with output layer

Table 2: Qwen2.5-0.5B-Instruct Vocabulary Details

The model implements these components with a vocabulary of 151,936 tokens and tied word embeddings between input and output layers (Table 2). This architecture is particularly suitable for parameter-efficient fine-tuning methods like LoRA, as we can target high-leverage components (query and value projections) while keeping most parameters frozen.

Lora Finetuning

Low-Rank Adaptation (LoRA) [2] represents a parameter-efficient fine-tuning approach that substantially reduces the number of trainable parameters while maintaining model performance. The key innovation lies in decomposing weight updates into low-rank matrices instead of fine-tuning the entire weight matrices.

Mathematical Formulation

In standard fine-tuning of a pre-trained model, the weight matrix $W_0 \in \mathbb{R}^{d\times k}$ is updated to $W = W_0 + \Delta W$ during training. LoRA parameterizes the update ΔW using two low-rank matrices:

$$W = W_0 + \Delta W = W_0 + BA \tag{5}$$

where $B \in \mathbb{R}^{d\times r}$ and $A \in \mathbb{R}^{r\times k}$ with rank $r \ll \min(d, k)$. During the forward pass, for input x, the output is computed as:

$$h = W_0x + \Delta Wx = W_0x + BAx \tag{6}$$

The weight update is typically scaled during training by α/r, where α is a constant hyperparameter:

$$h = W_0x + \frac{\alpha}{r}BAx \tag{7}$$

Application to Qwen2.5

For the Qwen2.5-0.5B-Instruct model, we apply LoRA specifically to the query and value projection matrices in the multi-head attention mechanism. These matrices are chosen because:

- Query projections ($W_q \in \mathbb{R}^{896\times 896}$) directly influence the model's ability to focus on relevant context
- Value projections ($W_v \in \mathbb{R}^{896\times 128}$) affect how the model represents information for aggregation
- Modifying these components provides substantial adaptation power while minimizing parameter count

The LoRA matrices are injected into the model architecture, allowing for efficient adaptation without altering the original weights. The choice of low-rank matrices enables us to maintain a small number of trainable parameters while still achieving significant performance improvements.

Training Dynamics

During training, only the LoRA parameters (A and B) are updated while W_0 remains frozen. This approach offers several advantages:

- **Memory efficiency**: Only the gradients for LoRA parameters need to be stored
- **Composability**: Multiple task-specific LoRA modules can be trained and swapped without changing the base model
- **Preservation of general knowledge**: The original pre-trained weights remain intact

Additionally, we deviated from the standard LoRA approach by making the language model head trainable as well. This decision was motivated by the need for adaptation specifically at the vocabulary distribution level, which is crucial for accurate numerical predictions. The LM head contains $896 \times 151{,}936 = 136{,}134{,}656$ parameters, significantly increasing our trainable parameter count, but targeting this layer directly improves the model's ability to produce precise numerical outputs in text form.

The total trainable parameter count thus becomes:

$$\begin{aligned} \text{LoRA parameters} + \text{LM head parameters} &= 344{,}064 + 136{,}134{,}656 && (8) \\ &= 136{,}478{,}720 && (9) \end{aligned}$$

Predator-Prey Dynamics

The Lotka-Volterra model represents a classic mathematical framework for studying predator-prey interactions in ecological systems. It describes the oscillatory dynamics between two populations: a prey species (x) that has abundant food and can reproduce exponentially, and a predator species (y) that relies on consuming the prey for survival.

Data Analysis

Our exploratory analysis of the Lotka-Volterra dataset revealed important statistical properties across 1,000 trajectories. Each trajectory contained 100 time points with consistent sampling intervals. Data quality checks verified the absence of duplicates and negative values.

Using K-means clustering on trajectory features (including period, amplitude, and phase relationships), we identified four distinct dynamic patterns in the dataset as displayed in Figure 1:

- **Classic oscillatory dynamics (14.6%)**: Characterized by sustained oscillations with relatively balanced prey and predator populations, showing clear cyclic interactions.
- **Prey-dominant systems (69.3%)**: Systems where prey populations maintain higher relative values with predator populations showing dampened oscillations.
- **Equilibrium systems (14.9%)**: Trajectories that stabilize after initial oscillations, reaching a relatively steady state.
- **Predator collapse systems (1.2%)**: A small subset where predator populations drop dramatically, allowing prey populations to grow significantly without constraint.

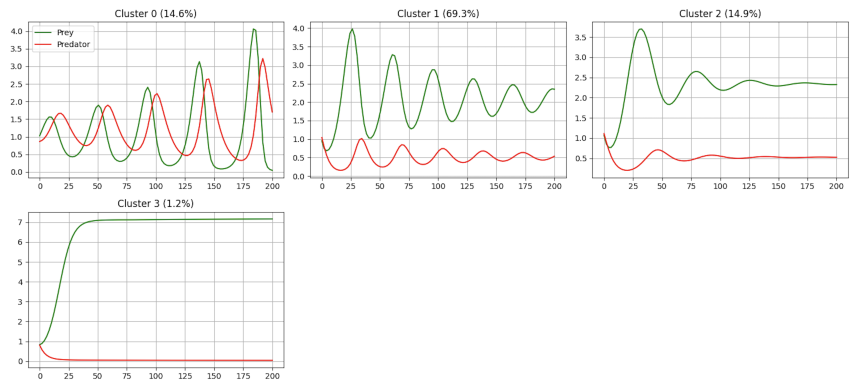

Figure 1: **Cluster Analysis**. Four distinct clusters identified in the Lotka-Volterra dataset through K-means clustering

The clustering revealed that the majority of systems (69.3%) fall into prey-dominant dynamics, suggesting parameter combinations that favor prey survival across much of the parameter space explored.

Figure 2 showcases individual trajectories representing diverse dynamic behaviors within the dataset. These examples highlight the variability in amplitudes, frequencies, and phase relationships that our forecasting model must learn to predict accurately.

Through peak detection analysis, we calculated oscillation periods for both prey and predator populations, finding average periods of approximately 20-25 time units for prey and 20-30 time units for predators, as shown in Figure 3.

Further analysis of the phase relationships between prey and predator oscillations showed that predator peaks typically lag behind prey peaks by about 0.2 time units (20% of the cycle period), as illustrated in Figure 3. This characteristic phase difference reflects the biological reality that predator populations grow in response to increasing prey availability, then decline as prey becomes scarce.

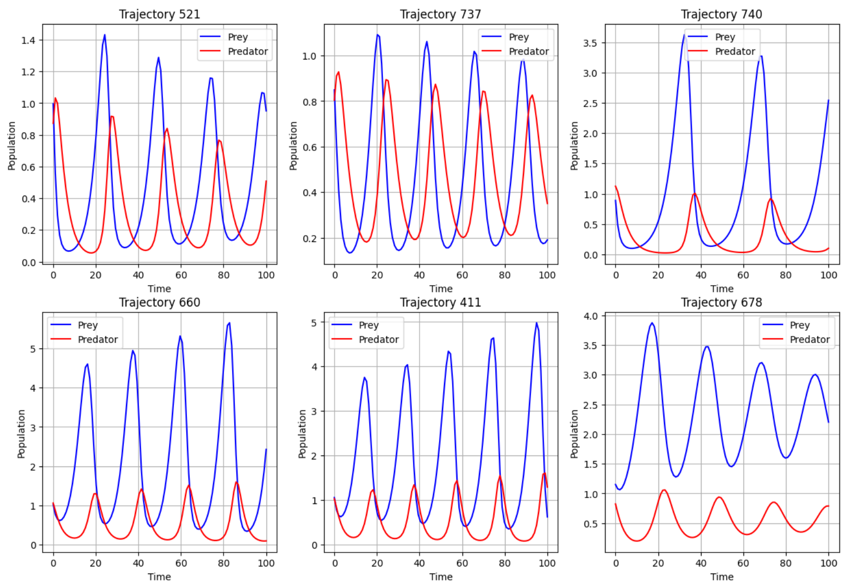

Figure 2: **Individual Trajectories**. Sample trajectories of prey and predator populations in the Lotka-Volterra dataset, illustrating the oscillatory dynamics characteristic of the model.

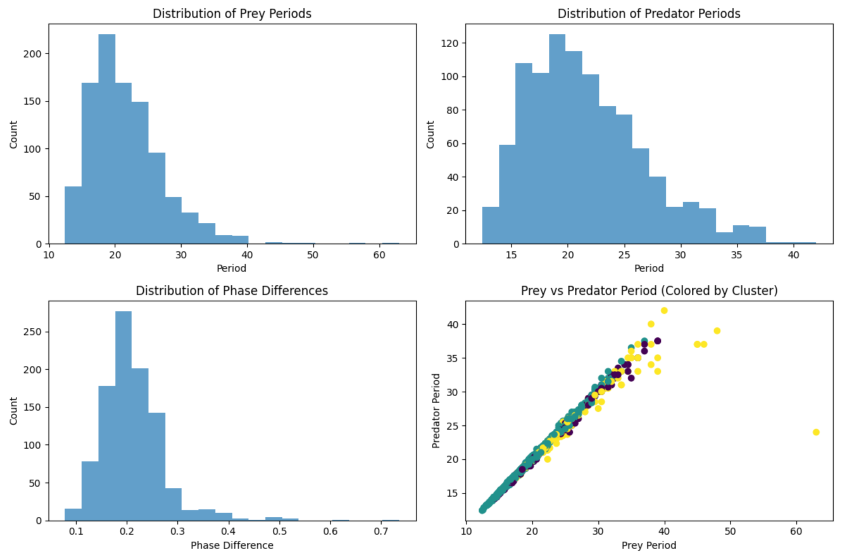

Figure 3: **Statistical Distributions**. Distribution of oscillation periods for prey and predator populations in the Lotka-Volterra dataset, highlighting the variability in dynamics across different parameter regimes.

Figure 4 reveals typical system behavior with prey populations maintaining higher values (averaging $\approx$ 1.7 units) compared to predator populations (averaging $\approx$ 0.6 units) after initial transient dynamics.

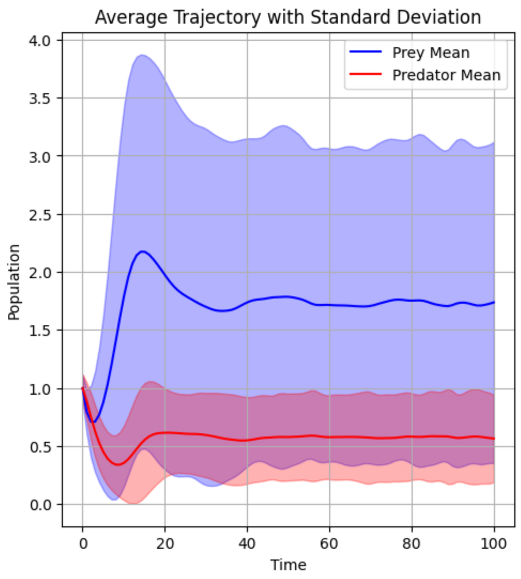

Figure 4: **Average Trajectories**. Average population trajectories with uncertainty: Mean prey (blue) and predator (red) population trajectories over time with shaded regions representing standard deviation.

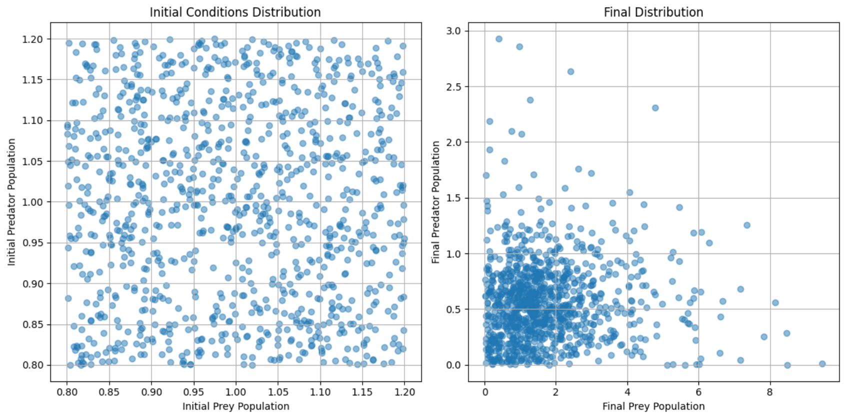

Figure 5: **Initial vs Final Distribution**. Comparison between initial conditions (left) and final states (right) of the predator-prey systems after simulation.

Figure 5 reveals the system's sensitivity to initial conditions, a hallmark of nonlinear dynamics. Despite starting in a narrow range (0.8-1.2 for both populations), final states diverge dramatically. The final distribution shows predator populations concentrated at lower values (0.1-1.0) while prey populations spread widely (0-8), indicating states where prey dynamics become less constrained by predator influence. This divergence highlights why long-term forecasting of such systems is challenging—small input variations amplify into significantly different outcomes over time.

For the current study with our limited dataset of 1,000 trajectories, we implemented a simple random split into training (70%), validation (15%), and testing (15%) sets. Given the time constraints and dataset size, this approach provided a practical balance between model training and evaluation needs. However, the observed dynamics suggest several potential improvements for future work with larger datasets.

Future Improvements for Data Handling

The observed sensitivity to initial conditions and the diverse range of dynamic behaviors have important implications for more sophisticated approaches to training, validation, and testing splits in future work. With a larger dataset, a stratified sampling approach based on the four identified clusters would better maintain the distribution of dynamic behaviors across all splits.

Such stratification would be particularly valuable because the complexity and diversity of the dynamics require the model to learn from examples across all behavioral regimes. For example, the rare "predator collapse" trajectories (1.2% of cases) should be proportionally represented in all splits to prevent the model from treating such cases as anomalies during evaluation. Similarly, the phase relationships and oscillation characteristics vary systematically across clusters, making it essential that the model train on and be evaluated against the full spectrum of possible dynamics.

LLMTIME Preprocessing Approach

To enable language models to process numerical time series data, we implement the LLMTIME preprocessing scheme introduced by Gruver[1]. This approach transforms multi-dimensional time series data into structured text representations that preserve temporal patterns while exploiting the language model's textual processing capabilities.

Text Representation Format

While the original LLMTIME approach [1] was designed for univariate time series, our Lotka-Volterra data requires representing multiple variables (prey and predator populations) across sequential timesteps. We therefore extended the LLMTIME encoding convention using a structured delimiter system:

- Comma (",") separates different variables at the same timestep
- Semicolon (";") separates different timesteps in the sequence

Our implementation supports different numerical precision formats. Below are examples showing the same trajectory segment at 3-decimal and 2-decimal precision:

```
2.458,10.249;1.916,7.677;1.503,5.985;...  # 3-decimal precision
2.46,10.25;1.92,7.68;1.50,5.99;...        # 2-decimal precision
```

Numerical Precision Considerations

A critical aspect of our preprocessing is determining the appropriate precision for numerical values. We implemented and tested both 2 and 3 decimal place precision options:

$$x_{\text{text}} = \text{round}(x, d) \tag{10}$$

where $d \in \{2, 3\}$ is the decimal precision. Higher precision provides more accurate population values but increases token consumption and potentially makes patterns more difficult for the model to recognize. Our hyperparameter search included precision as an experimental variable to determine its impact on forecasting performance.

Data Scaling

To ensure numerical stability and help the model work with a consistent range of values, we apply trajectory-specific scaling to the raw data before text conversion:

$$x_{\text{scaled}} = x \cdot \frac{\alpha}{P_{99}(x)} \tag{11}$$

where $P_{99}(x)$ is the 99th percentile of the specific time series being processed and $\alpha = 10.0$ is our target scale. This approach creates a unique scaling factor for each trajectory, ensuring that most values fall within the range $[0, \alpha]$ while preserving the relative dynamics and outliers within each sequence. By computing P_{99} independently for each time series, we normalize across trajectories with different magnitudes while maintaining their characteristic oscillatory patterns.

Our analysis showed that with $\alpha = 10.0$, only 1% of values exceed this threshold. This trajectory-specific scaling approach creates a more balanced distribution for the model to learn from while preserving the distinctive features of each predator-prey system.

Statistic	**Value** ($\alpha = 10.0$)
Token Shape	[1201]
Minimum Value	1.52
Maximum Value	10.29
Mean	5.15
Median	4.95
99th Percentile	10.00
95th Percentile	8.42
Values > 10.0	1.0%
Values > 20.0	0.0%

Table 3: Statistical properties of scaled trajectory data with $\alpha = 10.0$. Note how the 99th percentile matches the target scaling value, and only 1% of values exceed this threshold.

Tokenization Effects

The choice of text representation significantly impacts how the time series data is tokenized. For the Qwen2.5 tokenizer, numerical values are typically split into multiple tokens (e.g., 1.23 might tokenize as [1, ., 23]). Our analysis showed that each time step requires approximately 10-12 tokens, meaning that with a context length of 512 tokens, we can include roughly 40-50 time steps in the input context.

This tokenization behavior directly influenced our experimental design, particularly for evaluating optimal context length. By measuring the relationship between context length and prediction accuracy, we could determine how much historical information the model requires to make accurate forecasts.

Precision	**Text**	**Token Count**	**Complete Token Sequence**
2 decimal	10.02,10.03;7.81,7.51	21	[16, 15, 13, 15, 17, 11, 16, 15, 13, 15, 18, 26, 22, 13, 23, 16, 11, 22, 13, 20, 16]
	9.01,10.02;10.01,8.19	21	[24, 13, 15, 16, 11, 16, 15, 13, 15, 17, 26, 16, 15, 13, 15, 16, 11, 23, 13, 16, 24]
3 decimal	9.015,10.018;10.010,8.189	25	[24, 13, 15, 16, 20, 11, 16, 15, 13, 15, 16, 23, 26, 16, 15, 13, 15, 16, 15, 11, 23, 13, 16, 23, 24]
	10.022,10.025;7.813,7.510	25	[16, 15, 13, 15, 17, 17, 11, 16, 15, 13, 15, 17, 20, 26, 22, 13, 23, 16, 18, 11, 22, 13, 20, 16, 15]

Table 4: Complete tokenization comparison between 2-decimal and 3-decimal precision formats for predator-prey data. Note that 3-decimal precision requires approximately 19% more tokens (25 vs 21) for representing the same underlying information. The individual tokens show how numbers, decimal points, and delimiters are broken down by the tokenizer.

Text-to-Numeric Conversion

To convert the model's text output back into numerical values, we implemented a simple parsing function that reverses the tokenization process. This function takes the generated text and splits it based on the defined delimiters (commas and semicolons) to reconstruct the original numerical values. The conversion process also includes scaling back to the original range using the inverse of the scaling factor applied during preprocessing.

To handle potential generation errors, the implementation employs regex pattern matching to identify numerical values despite formatting inconsistencies, gracefully processes missing or malformed values, validates physical plausibility (enforcing non-negativity for population data), and verifies dimensional consistency between predictions and targets. These safeguards ensure that evaluation metrics reflect genuine forecasting errors rather than parsing artifacts, providing a fair assessment of the model's predictive capabilities.

Training Data Processing

For model training, we processed the data as follows:

1. Split the 1,000 trajectories into train (70%), validation (15%), and test (15%) sets
2. Convert numerical trajectories to text format with specified precision and scaling
3. For training data, create overlapping chunks with sequence length 512 and stride 256
4. For validation, create non-overlapping chunks to prevent information leakage
5. For testing, maintain complete trajectories without chunking

The chunking strategy helps to maximize the utility of training data while ensuring the model sees diverse sections of trajectories during training. The stride of 256 tokens provides a 50% overlap between consecutive chunks, helping the model learn to generate consistent predictions across different context windows.

FLOPs Calculation

To rigorously assess computational efficiency, we implemented a detailed floating-point operations (FLOPs) tracking mechanism for both training and inference. This approach provides a hardware-independent measure of computational cost, enabling fair comparison across different model configurations. During calculation, we ignore the effet of grouped query attention (GQA) and KV caching on FLOP counts, as these optimizations are not directly relevant to the core operations of the model. We also exclude the impact of tokenization and text-to-numeric conversion, as these processes are not part of the model's architecture.

For the standard transformer operations, we account for all major computational components:

- **Embedding layer**: Token embedding and positional encoding (RoPE)
- **Attention mechanism**: Query, key, value projections, attention score computation, softmax operations, and output projections
- **Feed-forward networks**: Gate projections, up projections, SwiGLU activations, and down projections
- **Normalization layers**: RMSNorm operations across all transformer layers
- **Language model head**: Matrix multiplication between hidden states and vocabulary embeddings

With LoRA, we modify weight matrices using low-rank decomposition: $W = W_0 + BA$ where $B \in \mathbb{R}^{d\times r}$ and $A \in \mathbb{R}^{r\times k}$ with rank $r \ll \min(d, k)$. The FLOPs for each LoRA-augmented projection are:

$$\begin{aligned}
\text{FLOPs}_{\text{LoRA}} &= \text{FLOPs}_{\text{original}} + \text{FLOPs}_{x\times A^T} + \text{FLOPs}_{(xA^T)\times B^T} && (12)\\
&= BS \cdot H \cdot (2H-1) + BS \cdot r \cdot (2H-1) + BS \cdot H \cdot (2r-1) && (13)
\end{aligned}$$

where B is batch size, S is sequence length, H is hidden dimension, and $r = 8$ is the LoRA rank.

The total training FLOPs include both forward and backward passes. For the backward pass, we use the standard approximation of doubling the forward pass FLOPs, resulting in a $3\times$ multiplier for training steps ($1\times$ forward + $2\times$ backward).

We established a total computational budget of 10^{17} FLOPs for our experiments, carefully tracking usage throughout the development process. This budget was allocated across hyperparameter search (50%), context length evaluation (15%), and final training (35%). A full breakdown of FLOP calculations for all operations is provided in the Appendix.

Implementation

Hardware

All experiments were conducted on NVIDIA A100 GPU instances with 40GB of RAM. All inferences of models were performed on Mac M1 Pro Chip with 16GB of RAM. Our implementation includes fallback mechanisms for different hardware accelerators, automatically selecting the optimal device (CUDA, MPS, or CPU) based on availability.

Model training setup

Optimization Approach

We employed the AdamW optimizer with weight decay of 0.01, which helps mitigate overfitting during fine-tuning. Gradient clipping at 1.0 was applied to prevent exploding gradients, which is particularly important when fine-tuning large language models. For each training step, we logged both pre-clipping and post-clipping gradient norms, maintaining a ratio typically between 0.8-1.0, indicating stable training dynamics.

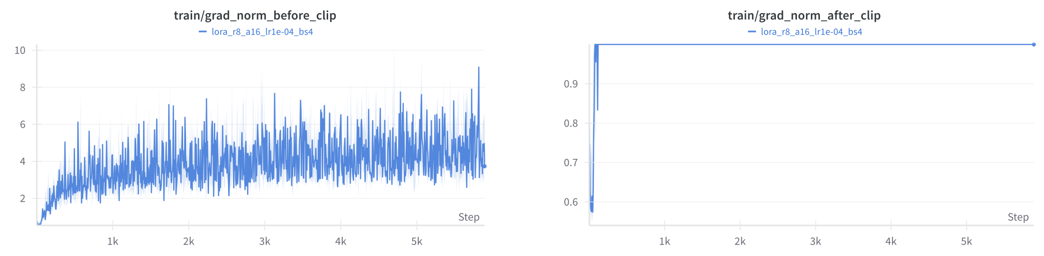

Figure 6: **Gradient norm comparison** Left: The plot shows increasing gradient magnitudes as training progresses, with values ranging from approximately 1 to 9. Right: The values are consistently maintained at or below 1.0 throughout training, demonstrating the effectiveness of gradient clipping.

We utilized the OneCycleLR scheduler which provides effective learning rate management through three phases:

- A warm-up phase (10% of total steps) where learning rate gradually increases
- A sustained high-learning-rate phase
- A cool-down phase with annealed learning rates

This approach allows the model to initially adapt quickly and then gradually refine its parameters.

Validation Set up

The validation step is an inference-based process that evaluates the model's performance on unseen data. We implemented a validation loop that runs every 500 steps during training, allowing us to monitor the model's performance. We used 30 samples from validation with random seed, perfomed mae and mse when predicting three time steps ahead. The validation set is used to track the model's performance and prevent overfitting. We also implemented a checkpointing mechanism to save the best-performing model based on validation metrics.

Evaluation Setup

Our evaluation pipeline implements a rigorous approach for assessing forecasting accuracy:

Forecasting Process

For each test trajectory, we:

1. Provide the first 50 timesteps as context
2. Generate the next 50 timesteps autoregressively
3. Apply post-processing to ensure valid numerical formatting
4. Compare predictions against ground truth values

To ensure robustness, we implemented pattern-matching techniques that correct common errors in the model's text generation, such as inconsistent decimal places or missing delimiters.

Evaluation Metrics

We calculated the following metrics to assess model performance:

- **Mean Absolute Error (MAE)**: Primary metric for overall accuracy assessment, measuring absolute difference between predicted and actual values.
- **Mean Squared Error (MSE)**: Secondary metric that penalizes larger errors more heavily.
- **Population-specific metrics**: Separate MAE and MSE values for prey and predator populations, capturing the model's ability to forecast different variables.
- **Success rate**: Percentage of trajectories where the model generated valid numerical predictions that could be parsed correctly.

Each test run involved evaluating 150 trajectories, with detailed performance tracking for both population variables across all generated timesteps.

Generation Settings

For model generation, we followed the same approach as the original LLMTIME paper [1], using stochastic sampling with temperature 0.9 and top-p 0.9:

```
output = model.generate(
    inputs["input_ids"],
    max_new_tokens=max_tokens,
    temperature=0.9,
    top_p=0.9,
    do_sample=True,
    renormalize_logits=True
)
```

This configuration provides a balance between diversity and coherence in generated sequences, which is particularly important for producing accurate numerical forecasts while maintaining the ability to capture the range of possible system behaviors.

LoRA Experiments

Hyperparameter search experiment

Context Length Exploration

Analysis of results

Model Performance Comparison

- Detailed analysis of the best model configuration
- Comparative evaluation against baseline
- Visualization of forecasting performance
- Error analysis for prey and predator populations

Discussion

- Analysis of the trade-offs between computational cost and accuracy
- Impact of different hyperparameters on performance
- Strengths and limitations of the approach
- Recommendations for time-series fine-tuning under tight compute budgets

Conclusion

- Summary of key findings
- Suggestions for future improvements
- Final FLOP accounting table

Appendix

Detailed FLOP Calculation

This section provides a comprehensive accounting of the floating-point operations (FLOPs) used in our experiments with the Qwen2.5-0.5B model, based on our custom FLOP tracking implementation.

Forward Pass Components

For a single forward pass with sequence length S, batch size B, hidden dimension $H = 896$, intermediate dimension $I = 4864$, and vocabulary size $V = 151,936$, we calculate FLOPs for each component as follows:

Embedding Layer The token embedding lookup is primarily a memory operation rather than computational, counted as:

$$\text{FLOPs}_{\text{embedding}} = B \times S \times H = 64 \times 512 \times 896 \approx 2.94 \times 10^7 \text{ FLOPs} \tag{14}$$

Attention Mechanism with LoRA For each attention layer:

1. **Query projection with LoRA**: For modules with LoRA enabled (rank $r = 8$):

$$\begin{aligned} \text{FLOPs}_{\text{q\_proj}} &= \text{FLOPs}_{\text{original}} + \text{FLOPs}_{x \times A^T} + \text{FLOPs}_{(xA^T) \times B^T} && (15) \\ &= B \times S \times H \times (2H - 1) && (16) \\ &+ B \times S \times r \times (2H - 1) + B \times S \times H \times (2r - 1) && (17) \end{aligned}$$

2. **Key/Value projections**: Similar to query.
3. **RoPE**: $B \times S \times 3 \times \text{num\_heads} \times \text{head\_dim} + 0.5 \times B \times S \times \text{num\_heads} \times \text{head\_dim}$, here we only negate half of the original length of x.
4. **Attention scores**: $B \times S \times \text{num\_heads} \times S \times (2 \times \text{head\_dim} - 1)$
5. **Attention mask**: $B \times \text{num\_heads} \times S \times S$
6. **Attention softmax**: $B \times \text{num\_heads} \times S \times (12S - 1)$
7. **Weighted sum**: $B \times \text{num\_heads} \times S \times \text{head\_dim} \times (2S - 1)$
8. **Output projection**: Similar structure to query projection if LoRA is applied

Feed-Forward Network

1. **Gate and Up projections**: $2 \times B \times S \times I \times (2H - 1)$
2. **SwiGLU activation**: $B \times S \times I \times 14$
3. **Element-wise multiplication**: $B \times S \times I$
4. **Down projection**: $B \times S \times H \times (2I - 1)$

Layer Normalization and Residuals

1. **RMSNorm (2 per layer)**: $2 \times B \times S \times (4H + 12)$
2. **Residual connections**: $2 \times B \times S \times H$

Output Layer

1. **Final RMSNorm**: $B \times S(4 \times H + 12)FLOPs$
2. **LM head**: $2 \times B \times S \times H \times V = 2 \times B \times S \times 896 \times 151936$ FLOPs

Total Forward Pass FLOPs

For our configuration with $H = 896$, $I = 4864$, and 24 layers, the total forward pass FLOPs for batch size $B = 4$ and sequence length $S = 512$ is approximately:

$$\text{FLOPs}_{\text{forward}} \approx 1.98 \times 10^{12} \text{ FLOPs} \tag{18}$$

Training Step FLOPs

For training, our implementation uses the standard approximation that backward pass requires twice the computation of the forward pass (in the same setting as in forward pass):

$$\text{FLOPs}_{\text{training\_step}} = \text{FLOPs}_{\text{forward}} + 2 \times \text{FLOPs}_{\text{forward}} \approx 5.94 \times 10^{12} \text{ FLOPs} \tag{19}$$

Budget Allocation and Usage

Our implementation tracked FLOPs throughout the experimental process, including:

Experiment	**Total FLOPs**	**Budget Percentage**
Hyperparameter Grid Search (18 configurations)		
Average per Hyperparameter Config	2.70×10^{15}	2.7%
Total Grid Search (18 Experiments)	4.86×10^{16}	48.6%
Context Length Evaluation		
Context Length 768	9.10×10^{15}	9.1%
Context Length 512	5.94×10^{15}	5.9%
Context Length 128	1.43×10^{15}	1.4%
Final Fine-tuning (LoRA)		
LoRA (r=8, α=16, lr=1e-04, bs=4)	3.50×10^{16}	35%
Overall FLOP Usage		
Total Experimental FLOPs	1.0×10^{17}	100.0%
Maximum Budget	1.0×10^{17}	100.0%

Table 5: FLOPs Usage Across Experiments

References

[1] 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.

[2] 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.

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