M2 Deep Learning - Coursework Assignment

Raunaq Rai (rsr45@cam.ac.uk)
Data Intensive Science, Department of Physics, University of Cambridge

04 April, 2025

Word Count: XXX

Introduction

Large Language Models have revolutionised natural language processing and have recently shown promise in domains beyond text, including time-series forecasting [1]. In this coursework, we explore the application of Low-Rank Adaptation (LoRA) [2] to the Qwen2.5-Instruct model [3], a transformer-based LLM, for forecasting predator-prey population dynamics as described by the Lotka–Volterra equations [4].

Building on the preprocessing methodology in the LLMTIME framework [1], which adapts numeric sequences into a tokenizer-friendly format, we examine how Qwen2.5-Instruct can be repurposed for numerical prediction tasks. The objective is to fine-tune the model on a dataset of simulated predator-prey trajectories using LoRA, which allows for the adaptation of pretrained models without updating all weights.

Our experiments have a computational constraint: with a maximum allowable budget of 10^{17} floating point operations (FLOPS). The provided dataset consists of 1,000 synthetic time series representing prey and predator population dynamics, formatted as 100 time steps per sequence with two variables per step.

This report outlines the implementation of LLMTIME preprocessing, baseline evaluation of the untrained model, LoRA fine-tuning with hyperparameter sweeps, and analysis of forecasting performance under compute-efficient constraints. Our findings highlight the potential of adapting LLMs for scientific time-series forecasting.

1 Compute Constraint

This coursework promotes efficient experimentation under a strict compute budget of 10^{17} floating point operations (FLOPs) across all reported experiments. This constraint mirrors real-world limitations, and has encouraged us to be thoughtful and methodical with all experimentation.

FLOP Accounting Principles

FLOP costs are calculated using simplified, hardware-agnostic assumptions. Table 1 provides the per-operation FLOP estimates. For example, multiplying an $m \times n$ matrix by an $n \times p$ matrix requires:

$$\text{FLOPs}_{\text{matmul}} = m \times p \times (2n - 1) \tag{1}$$

We also assume that backpropagation incurs double the cost of the forward pass. This is because it is difficult to estimate the FLOPs for backpropagation accurately, as it depends on the implementation and optimisations used. However, a conservative estimate is to assume that the backward pass is approximately double the cost of the forward pass.

$$\text{FLOPs}_{\text{train}} = 3 \times \text{FLOPs}_{\text{forward}} \tag{2}$$

Table 1: FLOPs Accounting for Primitive Operations

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

FLOP Estimator Implementation

To estimate the computational cost of each experiment, we implemented a FLOP estimator in Python (`flops_model.py` and `compute_flops.py`). This estimator models a forward pass through the Qwen2.5-0.5B architecture at the operation level, breaking down FLOPs into different categories.

The estimator accounts for every trainable and non-trainable component of the model. For a given sequence length n, the estimator computes the total number of FLOPs for each of the following:

- **Token Embeddings**: Retrieved via indexing and incur no arithmetic FLOPs.
- **Positional Embeddings**: Sinusoidal encodings are added to token embeddings, requiring $n \times d_{\text{model}}$ additions.
- **Multi-head Attention (per layer)**:
  - Query, Key, and Value projections: Linear transformations to d_{head} per head; includes multiplications and bias additions.
  - Dot-product attention: Computation of $QK^{\top}$, scaling, softmax, and weighted sum with V.
  - Softmax operation: Includes exponentiations, summations, and divisions.
  - Output projection: Concatenation of heads followed by a linear transformation.

- **Feedforward MLP with SwiGLU**: As described by Shazeer [5], this block consists of:
  - Two parallel linear up-projections from the model dimension to the hidden dimension: one to produce the activation input, and one to produce the gating values.
  - A `SwiGLU` activation function, which applies the SiLU nonlinearity to one stream and multiplies it elementwise with the other (gating).
  - A final down-projection from the hidden dimension back to the model dimension.
- **RMSNorm Layers**: Following Zhang and Sennrich [6], these involve:
  - Elementwise squaring, summation, square root, division, and scaling by a learned parameter γ.
- **LoRA Projections (if used)**: As proposed by Hu et al. [2], LoRA introduces:
  - Low-rank down- and up-projection matrices added to the frozen base weights.
  - FLOPs from these include multiplications, additions, and a residual connection.
- **Final Projection to Vocabulary Logits**:
  - A linear layer projecting to the vocabulary space: $n \times d_{\text{model}} \times V$ multiplications and additions.

These per-layer estimates are summed across all transformer blocks, and added to the overhead of input/output layers. FLOPs for different sequence lengths, LoRA ranks, and model dimensions are configurable through the function arguments.

Estimating Backward Pass Cost

The exact computation of FLOPs for backpropagation is non-trivial and implementation-specific. In practice, the backward pass includes:

- Derivatives for every parameter (including gradients for each tensor)
- Multiplications and additions for each chain rule application
- Memory reuse optimisations depending on the framework

For the purposes of this coursework, we assume that the backward pass is approximately double the cost of the forward pass. This is a conservative estimate, as it does not account for optimisations such as gradient checkpointing or memory reuse.

Thus, the total training FLOPs are given by:

$$\text{FLOPs}_{\text{train}} \approx 3 \times \text{FLOPs}_{\text{forward}} \tag{3}$$

2 LLMTIME Preprocessing Implementation

We implemented the LLMTIME preprocessing scheme [1] to convert multivariate time-series data into a format suitable for Qwen2.5-Instruct. This was done by creating a dedicated Python module `src/preprocessor.py` containing a class `LLMTIMEPreprocessor`, which formats and tokenizes time-series data according to the scheme described in the coursework.

Scaling and Formatting

Each sample consists of a pair of sequences—prey and predator population values over time. These values can vary significantly in magnitude both within and across samples, which can lead to inconsistent token lengths after formatting and impair the model's ability to generalise.

To mitigate this, we compute a per-sample scaling factor α that normalises the numerical range before tokenisation:

$$\alpha = \frac{1}{10} \cdot \max\left(\text{percentile}_{95}(\text{prey}),\ \text{percentile}_{95}(\text{predator})\right) \tag{4}$$

We then divide each value in the prey and predator sequences by α, ensuring that the majority of values fall within the range $[0, 10]$.This sample-specific scaling avoids the limitations of global normalisation and makes the model more robust to differences in scale across the dataset.

Finally, all scaled values are rounded to two decimal places. This rounding step reduces the number of unique numeric tokens, improving tokenisation consistency and helping prevent overfitting to insignificant digit-level variation.

LLMTIME Encoding

Following the LLMTIME convention, each timestep is represented as a pair of variables (prey, predator), separated by a comma. Consecutive timesteps are separated by a semicolon. For example, a sequence of three timesteps would be formatted as:

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

While the original LLMTIME implementation by Gruver et al. [1] removes decimal points to reduce sequence length—especially for GPT-style models that tokenize numbers into subword units, we retain the decimal point in our implementation for several important reasons:

- Removing the decimal point would introduce additional digits and increase sequence length without benefit.
- Preserving the decimal enhances human interpretability and simplifies decoding during inference.
- The coursework specification explicitly recommends retaining the decimal point.

Tokenization

Once formatted, the numeric string is passed through Qwen2.5's tokenizer using Hugging Face's `AutoTokenizer` interface. Each digit and punctuation mark is tokenized into its own token. For example:

```
tokenizer("1.23", return_tensors="pt")["input_ids"].tolist()[0]
```

yields:

```
[16, 13, 17, 18]
```

Example 1.

- **Original Input:**

  ```
  Prey:     [2.9, 3.2, 3.8, 4.5, 5.1]
  Predator: [1.1, 0.9, 0.7, 0.6, 0.5]
  ```

- **Scale Factor α:** 0.498

- **Formatted Sequence:**

  ```
  5.82,2.21;6.43,1.81;7.63,1.41;9.04,1.20;10.24,1.00
  ```

- **Tokenized Output (Qwen2.5 input IDs):**

  ```
  [20, 13, 23, 17, 11, 17, 13, 17, 16, 26, 21, 13, 19, 18, 11,
  16, 13, 23, 16, 26, 22, 13, 21, 18, 11, 16, 13, 19, 16, 26,
  24, 13, 15, 19, 11, 16, 13, 17, 15, 26, 16, 15, 13, 17, 19,
  11, 16, 13, 15, 15]
  ```

Example 2.

- **Original Input:**

  ```
  Prey:     [1.5, 1.8, 2.1, 2.4, 2.7]
  Predator: [2.8, 2.5, 2.2, 1.9, 1.6]
  ```

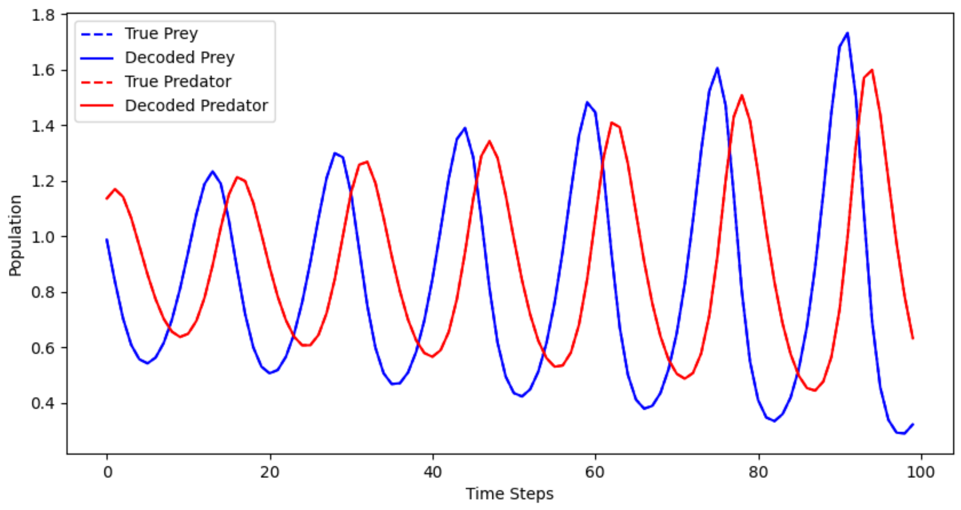
- **Scale Factor α:** 0.274

- **Formatted Sequence:**

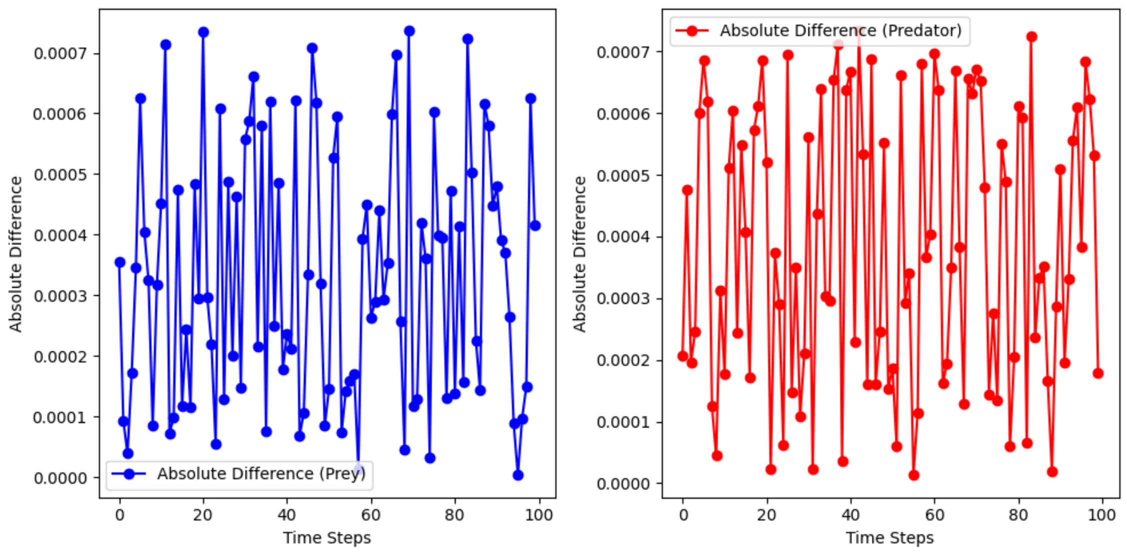
  ```
  5.47,10.22;6.57,9.12;7.66,8.03;8.76,6.93;9.85,5.84
  ```

- **Tokenized Output (Qwen2.5 input IDs):**

  ```
  [20, 13, 19, 22, 11, 16, 15, 13, 17, 17, 26, 21, 13, 20, 22,
  11, 24, 13, 16, 17, 26, 22, 13, 21, 21, 11, 23, 13, 15, 18,
  26, 23, 13, 22, 21, 11, 21, 13, 24, 18, 26, 24, 13, 23, 20,
  11, 20, 13, 23, 19]
  ```



(a) Decoded output vs. ground truth for the 972nd Lotka-Volterra trajectory in the test set. This sequence was passed through the LLMTIME preprocessing pipeline and fed into the untrained Qwen2.5-Instruct model. The resulting output was decoded and rescaled using the same scale factor α applied during encoding. The close alignment between original and decoded values confirms that the preprocessing, tokenization, and decoding pipeline is working as intended.



(b) Absolute differences between true and decoded prey (left) and predator (right) values for the 972nd test sequence. These errors quantify the deviation introduced by the LLMTIME encoding-decoding pipeline. The low magnitude of differences confirms the correctness of the numeric formatting, tokenizer compatibility, and inverse decoding process, independent of model training.

Figure 1: Encoding - Decoding check for Sample 972.

3 Baseline Evaluation

We assessed the forecasting ability of the untrained `Qwen2.5-0.5B-Instruct` model using Sample ID 972. The first 50 timesteps (prey–predator pairs) were provided as input in LLMTIME format, and the model was tasked with generating the remaining 50 steps.

Using the Hugging Face `generate()` API, the model predicted one token at a time, with each new token requiring a full forward pass. The 50-step input tokenized to **499 tokens**.

Forecasting Performance

We compare the predicted output (post-decoding and re-scaling) to the true population values for Sample ID 972 using standard metrics:

- **Prey:**
  - Mean Squared Error (MSE): 0.1770
  - Mean Absolute Error (MAE): 0.2485
  - R^2 Score: -0.2812
- **Predator:**
  - Mean Squared Error (MSE): 0.2337
  - Mean Absolute Error (MAE): 0.2793
  - R^2 Score: -1.6260

The prey trajectory shows moderate alignment, albeit with oversmoothing. The predator prediction, however, is unstable, with a negative R^2 indicating performance worse than simply predicting the mean. This reflects the model's lack of understanding of the underlying dynamics without domain-specific training.

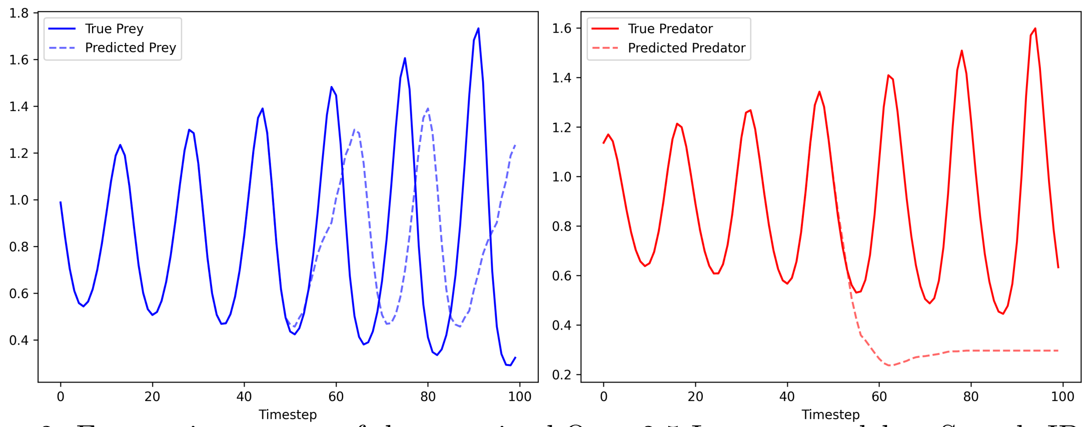


Figure 2: Forecasting output of the untrained Qwen2.5-Instruct model on Sample ID 972. The model is prompted with the first 50 timesteps and generates the next 50. Dashed lines show true values; solid lines show decoded predictions.

The untrained Qwen2.5-Instruct model exhibits some capacity to mimic the structure of the prey trajectory under LLMTIME formatting, likely due to its strong inductive bias and learned text patterns. However, it struggles with the more complex predator dynamics, which results in erratic predictions. This reinforces the need for fine-tuning or domain-specific adaptation when applying language models to structured forecasting problems.

4 FLOP Model

To estimate the total FLOPs used by the Qwen2.5-0.5B-Instruct model, we mapped every major component in the forward pass to arithmetic operations and computed their FLOP costs using Table 1. Bias additions are ignored throughout, as they are included in the matrix multiplication term. For inference, we assume reuse of Key/Value caches, meaning dot-product attention scores are only computed for new tokens, not the full input sequence.

Token and Positional Embeddings

- **Token Embeddings:** Retrieved via table lookup — **0 FLOPs**.
- **Positional Embedding Addition:** $n \times d_{\text{model}}$ additions.

RMSNorm (applied before attention, after attention, and before MLP)

The Root Mean Square Layer Normalisation (RMSNorm) [6] normalises each input token $x \in \mathbb{R}^d$ using the root mean square of its elements, without subtracting the mean. The operation is defined as:

$$\text{RMSNorm}(x) = \frac{x}{\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2 + \epsilon}} \cdot \gamma \tag{5}$$

where $\gamma \in \mathbb{R}^d$ is a learned scaling parameter and ϵ is a small constant for numerical stability.

For a batch of n tokens (each of dimension d), the FLOPs are as follows:

- Square each element: $n \times d$ multiplications.
- Sum across dimensions: $n \times (d-1)$ additions.
- Compute the square root of the mean: n square root operations.
- Divide each element by the norm: $n \times d$ divisions.
- Scale with learned weight γ: $n \times d$ multiplications.

Total: $2nd$ multiplications, $n(d-1)$ additions, nd divisions, n square roots.

Multi-Head Attention (per layer)

Let h be the number of heads and $d_h = d/h$.

- **Q/K/V Projections:** $3 \times n \times d \times d_h$ multiplications and $3 \times n \times (d-1) \times d_h$ additions.

- **Attention Scores:** $n \times n \times d_h$ multiplications and $n \times n \times (d_h - 1)$ additions.
- **Softmax:** For n^2 elements:
  - Exponentiation: n^2
  - Summation: $n \times (n - 1)$ additions
  - Normalisation: n^2 divisions
- **Softmax-Value Multiplication:** $n \times n \times d_h$ multiplications and $n \times d_h \times (n - 1)$ additions.
- **Concatenation:** Memory operation — **0 FLOPs**.
- **Final Output Projection:** $n \times d \times d$ multiplications and $n \times (d - 1) \times d$ additions.

Note: For inference, we reuse cached Key and Value projections from previous tokens. Only new tokens incur the full attention computation.

MLP Block with SwiGLU (per layer)

Let d be the model embedding dimension, d_{ff} the hidden (feedforward) dimension (e.g., 4864), and n the number of tokens in the sequence. The MLP block consists of two parallel up-projections (one gated) and a down-projection, followed by a SwiGLU activation [5].

- **Up-Projections and Gating:** Two parallel linear layers (one for gate, one for activation) each compute:

  $$\text{Multiplications: } n \times d \times d_{\text{ff}}, \quad \text{Additions: } n \times (d - 1) \times d_{\text{ff}}$$

  Multiplied by 2 for both paths:

  $$\Rightarrow 2 \times n \times d \times d_{\text{ff}} \text{ multiplications, and } 2 \times n \times (d - 1) \times d_{\text{ff}} \text{ additions.}$$

- **SwiGLU Activation:** The gated activation combines SiLU and elementwise multiplication:
  - Each unit requires: 1 exponentiation, 1 division, 2 multiplications, and 1 addition.
  - Total per token: $n \times d_{\text{ff}}$ of each (add, div, exp) and $2 \times n \times d_{\text{ff}}$ multiplications.
- **Down-Projection:** A single linear layer reduces dimensionality:

  $$\text{Multiplications: } n \times d_{\text{ff}} \times d, \quad \text{Additions: } n \times (d_{\text{ff}} - 1) \times d$$

Final Projection to Vocabulary Logits

- **Linear Projection:** $n \times d \times V$ multiplications and $n \times (d - 1) \times V$ additions.

LoRA Projections (if enabled)

Low-Rank Adaptation (LoRA) replaces a standard linear layer $W \in \mathbb{R}^{d \times d}$ with a low-rank approximation consisting of two smaller matrices:

- A **down-projection** matrix $A \in \mathbb{R}^{r \times d}$ (reducing dimensionality),
- An **up-projection** matrix $B \in \mathbb{R}^{d \times r}$ (projecting back to the original space).

Let:

- n be the number of tokens in the input sequence,
- d be the model's hidden dimension,
- r be the LoRA rank.
- **Down-Projection (Ax):** Projects from $\mathbb{R}^d$ to $\mathbb{R}^r$:
  - Multiplications: $n \times d \times r$
  - Additions: $n \times (d-1) \times r$
- **Up-Projection ($B(Ax)$):** Projects from $\mathbb{R}^r$ back to $\mathbb{R}^d$:
  - Multiplications: $n \times r \times d$
  - Additions: $n \times (r-1) \times d$
- **Scaling and Residual Addition:** The result is scaled (typically by α/r) and added to the original output:
  - Multiplications: $n \times d$
  - Additions: $n \times d$

Total FLOPs for one LoRA adapter:

- **Multiplications:** $2 \times n \times d \times r + n \times d$
- **Additions:** $2 \times n \times d \times r + n \times d$ (approximately)

This total assumes that both the down and up projections are active and used once per token per transformer block.

Summary

The total FLOPs are computed as:

$$\text{Total FLOPs} = \sum_{i=1}^{n} (a_i + m_i + d_i + 10e_i + 10s_i)$$

where a_i, m_i, d_i, e_i, and s_i represent the counts of additions, multiplications, divisions, exponentiations, and square roots respectively.

All computations are implemented in the Python function `forwards_pass_flops()`, which returns detailed per-operation and total FLOPs based on sequence length, LoRA rank, and model configuration.

5 LoRA Adaptation and Fine-Tuning Procedure

To enable parameter-efficient fine-tuning of the Qwen2.5-0.5B-Instruct model, we applied **Low-Rank Adaptation (LoRA)** to the **query (Q)** and **value (V)** projection layers in each transformer block. Specifically, we wrapped each of these layers with a custom `LoRALinear` module that augments the frozen base projection with a trainable low-rank update.

For each modified projection layer, we introduced two trainable matrices:

- A down-projection matrix $A \in \mathbb{R}^{r \times d}$
- An up-projection matrix $B \in \mathbb{R}^{d \times r}$

The output of a LoRA-adapted linear layer becomes:

$$\text{output} = Wx + \frac{\alpha}{r} BAx$$

where W is the original frozen weight matrix and α is a scaling factor (typically set equal to r) [2].

We fine-tuned only the LoRA parameters (A, B) while keeping the base model parameters frozen. The adapted layers were implemented by replacing `q_proj` and `v_proj` in each transformer block:

Training was performed for 600 steps using default hyperparameters:

- `lora_rank` = 4
- `learning_rate` = 1e-5
- `batch_size` = 4
- `context_length` = 512

We compared the trained model against an untrained baseline (LoRA-enabled but with zero optimisation steps). Both models were evaluated on a held-out validation set of 200 trajectories, using cross-entropy loss as the evaluation metric.

Results

- **Untrained model (LoRA only, 0 steps):** Validation loss = **1.1926**
- **LoRA-trained model (600 steps):** Validation loss = **0.9555**

The performance improvement demonstrates that even a modest number of training steps is sufficient to adapt the model to the domain of Lotka–Volterra trajectories.

6 LoRA Hyperparameter Search

To understand how key hyperparameters affect forecasting performance, we carried out a targeted grid search over:

- **LoRA Rank** $\in \{2, 4, 8\}$

- **Learning Rate** $\in \{10^{-5}, 5 \times 10^{-5}, 10^{-4}\}$

This search was designed to explore the trade-off between model expressivity and training dynamics. Increasing the LoRA rank introduces more trainable parameters per adapted layer, improving flexibility but increasing FLOP usage. Conversely, the learning rate controls the speed of adaptation: values that are too high may destabilise training, while those too low may cause underfitting or slow convergence.

Each configuration was trained for 600 optimiser steps and evaluated on a held-out validation set. Due to FLOP constraints and clear trends in performance, not all combinations were run. Specifically, validation loss consistently decreased as we moved **downward** (higher LoRA rank) and **leftward** (higher learning rate) in the grid. These trends allowed us to avoid redundant runs while still identifying the most promising setup.

LoRA Rank	**LR** = 10^{-4}	**LR** = 5×10^{-5}	**LR** = 10^{-5}
2	0 steps 0.0000% n/a	600 steps 4.2212% 0.7899	0 steps 0.0000% n/a
4	600 steps 4.22252% 0.6551	600 steps 4.22252% 0.7444	600 steps 4.22252% 0.9555
8	600 steps 4.22516% 0.6147	600 steps 4.22516% 0.6788	0 steps 0.0000% n/a

Table 2: LoRA Hyperparameter Search Results. Each cell contains: training steps (top), percentage of FLOPs used (middle), and validation loss (bottom). Results for LoRA rank=4 and LR=1e-5 were reused from the previous section

As shown in Table 2, the best-performing configuration used **LoRA rank 8** and a **learning rate of** 10^{-4}, achieving a validation loss of **0.6147** with only **4.23%** of the total FLOP budget. This combination strikes an effective balance between adaptation capacity and compute efficiency.

Effect of Context Length on Forecasting Performance

Having identified an effective LoRA configuration, we next explored how the model's performance is influenced by **context length**—the number of input tokens the model sees at once. Using the same LoRA rank (8) and learning rate (10^{-4}), we trained models using context lengths of {128, 512, 768}:

Context Length	**Validation Loss**	**Optimiser Steps**	**FLOPs Used (%)**
128	0.7088	600	1.02421
512	0.6147	600	*(from previous section)* 4.22516
768	0.4207	600	6.46607

Table 3: Effect of context length on validation loss using LoRA rank 8 and learning rate 10^{-4}.

As expected, validation loss improves with increasing context length. This trend is intuitive: longer contexts provide the model with more of the input sequence, enabling it to learn richer temporal patterns—especially important in cyclical systems like the Lotka–Volterra dynamics.

Given that Qwen2.5-Instruct is an autoregressive decoder-only model, its predictions are entirely conditioned on previous tokens. Extending the context window increases the number of prey–predator pairs the model can attend to during forecasting, resulting in more accurate extrapolations.

However, this benefit comes at a significant computational cost. Moving from 128 to 768 tokens increases the proportion of total FLOPs used from approximately 1% to over 6%, even with the same number of training steps.

7 Final Model Training and Evaluation

Based on insights from the previous experiments, we selected the following hyperparameters for our final model configuration:

- **LoRA Rank:** 8
- **Learning Rate:** 10^{-4}
- **Context Length:** 768

This configuration was chosen as it consistently outperformed others during the hyperparameter and context length sweeps, offering a strong balance between predictive accuracy and FLOP efficiency.

We trained the model for 6,000 steps using this setup, which approximately 62% of the total FLOP budget. Evaluation on a held-out validation set yielded a final loss of 0.2878 and a perplexity of 1.33. Figure 3 shows the full training curve, while Figure 4 presents a zoomed-in view of the final 3,000 steps.

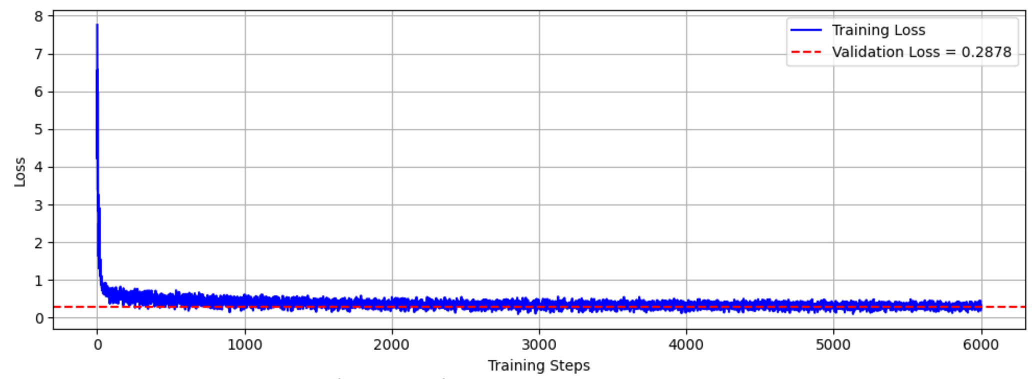


Figure 3: Training loss curve for the full 6,000-step run. The dashed red line indicates final validation loss.

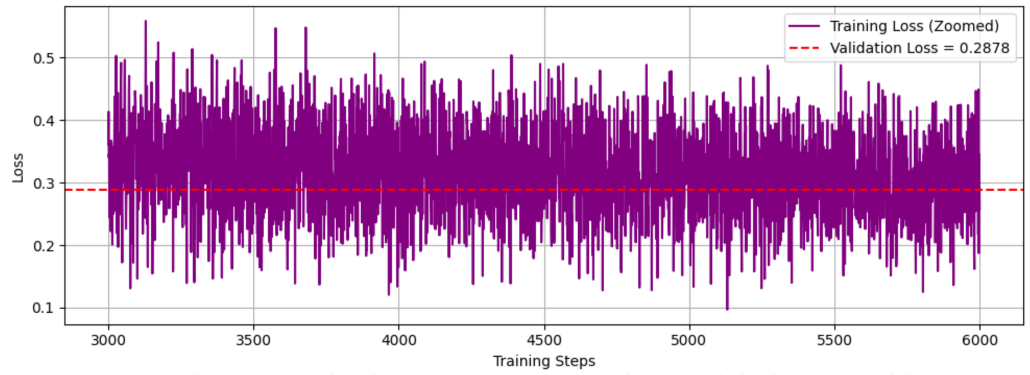


Figure 4: Training loss over the last 3,000 steps. The model shows stable convergence.

The validation loss is comparable to the training loss throughout, suggesting that the model generalises well to unseen data and is not overfitting.

Evaluation

To assess forecasting performance, we used the trained model to generate predictions on a representative test sequence (Sample ID 972). As shown in Figure 5, the model closely tracks the ground-truth oscillatory behaviour of both prey and predator populations across the full 100-timestep horizon.

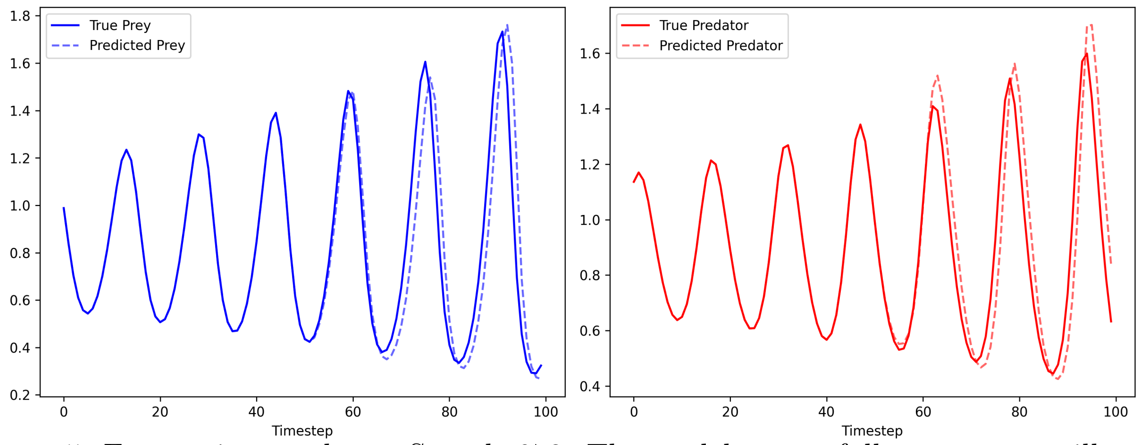


Figure 5: Forecasting results on Sample 972. The model successfully captures oscillatory dynamics in both prey (left) and predator (right) populations.

Species	**MSE**	**MAE**	**R^2**
Prey	0.0170	0.0671	0.8772
Predator	0.0125	0.0608	0.8600

Table 4: Evaluation metrics for Sample 972

8 Discussion + Assumptions

In this coursework, we assume a compute efficiency of 1, meaning all estimated FLOPs are fully utilised during inference without memory or runtime optimisations. Some large-scale language models, such as GPT-4, employ Mixture-of-Experts (MoE) architectures to reduce active parameters per forward pass—this lowers compute cost by activating only a subset of layers during token generation. However, `Qwen2.5-0.5B-Instruct` does not employ MoE or dynamic routing, meaning that **all model parameters are active for every forward pass**.

Recent innovations such as SkipDecode [7] propose early-exit strategies that skip computation in lower layers during autoregressive decoding. This approach can significantly reduce inference FLOPs while preserving performance, and is compatible with batching and KV cache reuse. Nonetheless, `Qwen2.5-0.5B-Instruct` does not implement such mechanisms, and so the full model is evaluated on every token without layer skipping or selective computation.

It is important to note that generation is inherently non-deterministic unless a seed is set. This means different runs on the same input can yield very different outputs—some closer to ground truth, others worse. This behaviour arises from sampling strategies like temperature scaling or nucleus sampling used during decoding.

9 FLOPs Usage

insert table for all flops used

References

[1] Nate Gruver, Marc Finzi, Shikai Qiu, and Andrew Gordon Wilson. Large language models are zero-shot time series forecasters. *arXiv preprint arXiv:2310.07820*, 2023. URL `https://doi.org/10.48550/arXiv.2310.07820`. NeurIPS 2023.

[2] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021. URL `https://arxiv.org/abs/2106.09685`. Draft v2 includes better baselines, experiments on GLUE, and more on adapter latency.

[3] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024. URL `https://doi.org/10.48550/arXiv.2412.15115`. Additional authors not shown.

[4] Y. Takeuchi, N.H. Du, N.T. Hieu, and K. Sato. Evolution of predator–prey systems described by a lotka–volterra equation under random environment. *Journal of Mathematical Analysis and Applications*, 316(2):291–314, 2006. URL `https://doi.org/10.1016/j.jmaa.2005.11.009`.

[5] Noam Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020. URL `https://doi.org/10.48550/arXiv.2002.05202`.

[6] Biao Zhang and Rico Sennrich. Root mean square layer normalization. *arXiv preprint arXiv:1910.07467*, 2019. URL `https://doi.org/10.48550/arXiv.1910.07467`. NeurIPS 2019.

[7] Luciano Del Corro, Allie Del Giorno, Sahaj Agarwal, Bin Yu, Ahmed Awadallah, and Subhabrata Mukherjee. Skipdecode: Autoregressive skip decoding with batching and caching for efficient llm inference. *arXiv preprint arXiv:2307.02628*, 2023. URL `https://doi.org/10.48550/arXiv.2307.02628`.