LoRA Fine-Tuning for Time Series with Qwen2.5-0.5B

Deep Learning for Data Intensive Science

mem97

Lent Term 2025

Contents

Acknowledgment

I would like to thank my brother Riccardo for running my codes on his computer. I found myself in a tricky situation, this project required a lot of computing power and memory. The machine I use is a Lenovo Legion 5 15ACH56H, RAM 16GB (of which 13.9GB usable), Processor AMD Ryzen 7 58000H with Radeon Graphics 3.20 GHz, NVIDIA GeForce RTX 3070 Laptop GPU: 8GB GDDR6, boost clock 1560 MHz, maximum graphics power 130W. I have quite a remarkable machine, but unfortunately with not enough memory available to withstand tasks execution of this project. In comparison Riccardo's PC specs are, CPU: AMD Ryzen 9 7900X3D, RAM 32GB, GPU: GeForce RTX 4080 Super 16GB. The two machines are quite far in terms of memory and computational power. I have tried to train Qwen2.5-0.5B injected with LoRA layers for 5000 steps, this procedure took approximately 8.5 hours, the same experiment was conducted with Riccardo's PC and completed in approximately 20 minutes. The reader is welcome to clone the repository and play with the codes, but beware, you need a high-level performance machine, with a minimum of RAM 32GB. GPUs may help speeding up the training process, however between experiments it is recommended to empty GPUs cache.

I would like to acknowledge the invaluable assistance of ChatGPT-4 and Grammarly in the preparation of this project. The tools played a significant role in:

- Debugging and resolving coding challenges encountered during the development of the Python package.
- Providing insights and suggestions for improving the structure and functionality of the code.
- Generating concise and accurate summaries of complex texts to enhance understanding and clarity.

While ChatGPT-4 contributed significantly to streamlining the development process and improving the quality of outputs, all results were rigorously reviewed, tested, and refined to ensure their accuracy, relevance, and alignment with project objectives.

I would like to state that all results are set to be reproducible, using `torch.manual_seed()`, however these results are machine dependent, thus if any user run my notebooks or Python scripts, results are not guaranteed to be the same as my findings (see PyTorch Reproducibility Documentation)
Word Count: ~ 2800

1 Introduction

This project aims to investigate the forecasting ability of large language models (LLMs)[3] on time series data. What takes us here, is the fact that traditional time series forecasting methods often struggle to capture long-term dependencies or patterns in time series data. The approach we just suggested, i.e. using LLMs to model time series data is commonly referred to as LLMTIME[2], where time series are treated as textual sequences for language modelling tasks. This is exactly what we have adopted, allowing seamless integration with language models. Using the pre-trained Qwen2.5-0.5B-Instruct model from Hugging Face[4], we fine-tune it for forecasting task via parameter-efficient Low Rank Adaptation (LoRA), enabling better predictions compared to the pure forecasting ability of Qwen2.5-0.5B.
Furthermore, for the nature of this project, i.e. heavy machinery computation, we are instructed to track floating point operations per seconds (FLOPs) during training up to order 10^{17}. The reminder of this report is structured as follows. Section 2 presents a brief overview of the data structure. Section 3 introduces LLMTIME preprocessing format and presents baseline evaluation using untrained model, and a detailed explanation for flop count procedure. Section 4 provides a discussion of LoRA fine-tuning and training experiments. Section 5 a collection of results, i.e. forecasting performance and metric comparison of all the experiments conducted. Conclusions are provided in Section 6.

2 Understanding Data

The time series dataset used in this project simulates Lotka-Volterra dynamics[1], representing predator-prey interactions over time across multiple systems. The data is stored in the directory data, in a h5 file format (Hierarchical Data Format version 5), it is format and data model commonly used for strong and organised large amount of numerical data, to read the data in Python we are going to use the package h5py. Each system consists of a sequence of population values prey and predator species, recorded at regular time intervals. Its raw structure is a 3D array of shape (N, T, 2), where N (=1000) is the number of systems, T (=100) is the number of timesteps, and the last dimension corresponds to the prey and predator populations, respectively.

3 LLMTIME Preprocessing & Baseline

This section covers the details of preprocessing the time series dataset we are working on, lotka_volterra_data.h5, tokenisation and how the untrained Qwen2.5 model predictions ability is evaluated. Moreover, a well detailed explanation of how FLOPs are estimated for during training procedure.

3.1 Split Data

As good machine learning practices, we want to split the data into train and validation sets. For this case, we split the data into train set, with 900 elements, and validation sets with 100 elements. More precisely, when the data are split, we create two copies of the validation sets, both with 100 different system IDs, but one of them, within each system, only contains the first 70 pair points. This is because after training and validating the model, we also want to test its prediction ability with the test set (validation copy but with less points within each system), and measure metrics (as we will see metric used are MSE and error between observed and predicted data in each system). To split the data, we use both functions `data_scale_split()` and , `load_and_preprocess()` (each with different purpose) both in src/preprocess.py (both functions input are lotka_volterra_data.h5 see documentation).

3.2 Preprocess & Tokenisation

Qwen2.5-0.5 processes text like data, hence we want to convert our timeseries dataset into string format. There is a function in `src/preprocess.py`, `load_and_preprocess()`. As the name suggests, this function is designed to load the data, using h5py, divide the data into `trajectories` and `time_points` (this is not relevant to us). The next step is to scale the data by a factor α. Why scaling? Scaling is used to normalise the range of prey and predator values, ensuring numerical stability during tokenisation and model training. It prevents large magnitudes from dominating learning, improves generalisation, and

helps the language model interpret population values consistent across different systems. How have we implemented this in Python? To standardise the numeric range, we used the quantiles, Quantiles are values that divide the dataset into equal-sized intervals. To apply this in Python we have used the module `numpy.quantile()` in NumPy and divided by the scale factor $\alpha = 10$, (we set the quantile value to be 0.9, so that 90% of the data will fall in values range (1,10)). After scaling we also rounded all values to 3 decimal places (as we will see this helps to shorten the size of tokens generated, which contributes to speed up prediction and training process). And the last step is to convert the scaled time series into string format. Here is an example:

```
3.757,4.116;2.929,3.083;2.698,2.232;2.835,1.612;3.261,1.188;3.973,0.909
```

Where each number represents values for prey and predator coordinates, the comma separates prey and predators, while semicolon represents time-steps. Thus, in the above example, there are six pair points divided by five time points. (Notice, there is not spacing between observed and time points, as spacing will results in more tokens to process).

Now the data are ready for tokenisation. To tokenise the scaled and string converted time series data, there are two designed functions, `process_sequences()` and `tokenize_time_series_np()` in `src/ste_up_lora.py` and src/qwen.py respectively (both functions use the tokenizer from load_qwen()).

How does tokenisation work?
Tokenisation with Qwen2.5 uses byte-level tokeniser (compatible with Hugging Face's AutoTokenizer) to split input text into subword tokens. Each string is broken down into smaller token units. Those tokens are then mapped to unique IDs from model's vocabulary, which are passed as input to the language model. Padding and truncation are applied as needed to fit context length.

Example of tokenised dataset:

```
Two examples of tokens from tokenised_data:

Preprocessed data:
4.020,4.367;2.069,4.548;1.067,4.153;0.616,3.546;0.409,2.936;0.310,2.397;...

After tokenisation:
[19, 13, 15, 17, 15, 11, 19, 13, 18, 21, 22, 26, 17, 13, 15, ...
Length of the above token: 1203


Preprocessed data:
3.536,4.214;2.483,3.330;1.996,2.488;1.812,1.815;1.809,1.315;1.937,0.958;...

After tokenisation:
[18, 13, 20, 18, 21, 11, 19, 13, 17, 16, 19, 26, 17, 13, 19, 23, 18,...
Length of the above token: 1227
```

Example from `Baseline.ipynb` (we followed the LLMTIME Preprocessing Scheme).

3.3 Evaluate Untrained Qwen2.5

To evaluate the untrained Qwen2.5 model's forecasting ability we used the function `model.generate()`.

How does `model.generate()` work?
The `model.generate()` method uses autoregressive decoding to generate tokens one at the time. Staring from an input prompt (in our case the tokenised LLMTIME strings), the model predicts the most likely next token based on its internal attention layers and previously generated tokens. This process is repeated iteratively until a specific `max_length` is reached or a stop condition (e.g. an end-of-sequence token) is met.

In notebook `Baseline.ipynb` we implemented `model.generate()` with the whole test set `tok_val_texts_70` in a for loop, to predict the remaining 30 pair-points, then we decoded the output and evaluated metrics (see Section 5).

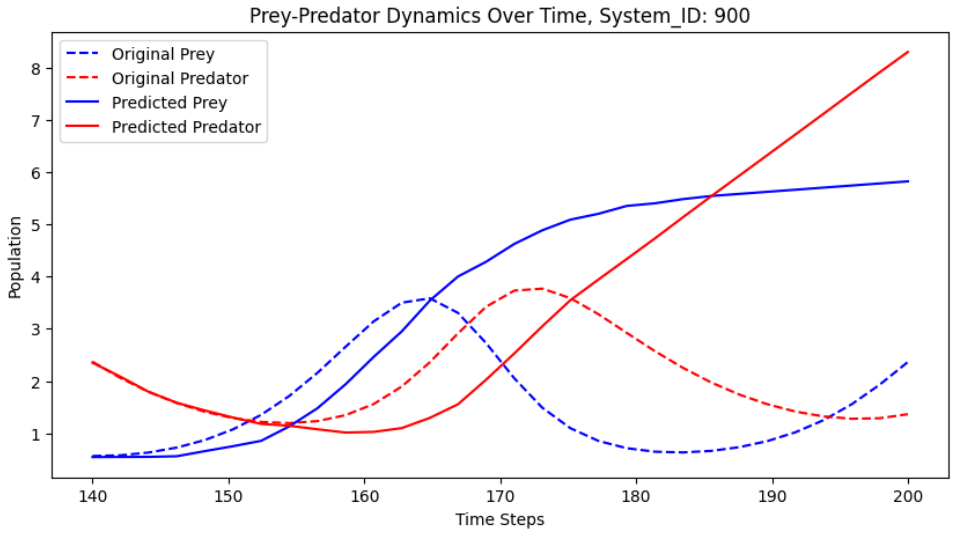

Figure 1: Results of untrained Qwen2.5 model's forecasting ability. This is a prediction vs true values comparison for system ID 900, as we can see the model was not able to capture the periodicity of the system. The point where the curves start to separate, is where token prediction starts to occur.

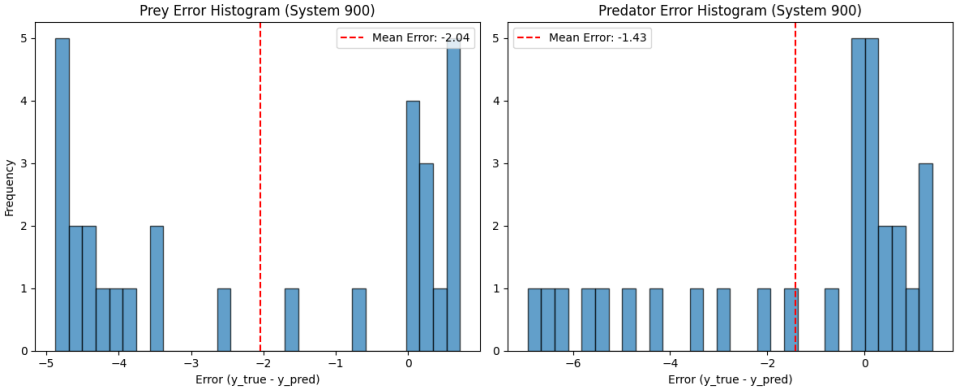

Figure 2: This figure, is a representation of errors between each point with in system ID 900, for the untrained Qwen2.5 model. As we can observe the error is unsurprisingly large!, with mean error -2.04 and -1.43 for prey and predator observations respectively. We calculated the error as $e_i = y_i^{pred} - y_i^{true}$ for $i \in [0, 99]$.

3.4 FLOPs Account Procedure

To estimate the total computational cost of training our model, we calculate the number of Floating Point Operations (FLOPs) required over the course of training. The computation considers all major components of a Transformer[1] architecture, this includes self-attention[2], feedforward layers, LayerNorm, and optionally, LoRA modules.

Formula Structure

The total FLOPs over training are given by:

$$\text{Total FLOPS} = \text{num\_steps} \times \text{batch\_size} \times \text{num\_layers} \times \text{FLOPs per layer} \times 3 \tag{1}$$

Here we are assuming that a forward pass = 2× backward pass!

Components of FLOPs Per Layer

1. Self-attention:
   - Q, K, V projections: $3 \cdot d_{\text{model}}^2 \cdot L$, where L is the number of layers.
   - QK^T attention score & attention$\times V$ application: $L \cdot d_k \cdot L$, where $d_k = \frac{d_{\text{model}}}{\text{num\_heads}}$.
   - Output projections: $d_{\text{model}}^2 \cdot L$.
   - Thus, total Self-Attention FLOPs: $\text{FLOPs}_{\text{Attention}} = \left(3d^2 + 2Ld_k^2 + d^2\right) \cdot L$.
2. LoRA (when used):For each injected LoRA module (Q and V), two low-rank projections: $\text{FLOPs}_{\text{LoRA}} = 4 \cdot r \cdot d \cdot L$, where r is the rank, d is the dimension of both matrices and L is the number of hidden layers.
3. Feedforward (SwiGLU activation function):
   - Two linear projections: $2 \cdot d \cdot d_{ff} \cdot L$, where d_{ff} is the feedforward dimensions.
   - Swish activation: $d_{ff} \cdot L$
   - Gating multiplication: $d_{ff} \cdot L$
   - Output projections: $d \cdot d_{ff} \cdot L$
   - Hence the total feedforward FLOPs are $\text{FLOPs}_{ff} = (2dd_{ff} + 2d_{ff} + d_{ff}d) \cdot L$
4. Layer normalisation, two operations per norm layer: $\text{FLOPs}_{norm} = 4 \cdot d \cdot L$

Therefore, after gartering all the above considerations, the total number of estimate FLOPs expected during training are:

$$\text{Total FLOPs} = \text{num\_steps} \times \text{FLOPs}_{\text{steps}} \times 3 \tag{2}$$

where $\text{FLOPs}_{\text{steps}}$ is:

$$\text{FLOPs}_{\text{steps}} = \text{batch\_size} \times \text{num\_layers} \times (\text{FLOPs}_{\text{Attention}} + \text{FLOPs}_{ff} + \text{FLOPs}_{norm} + \text{FLOPs}_{\text{LoRA}}) \tag{3}$$

This procedure is implemented in the function `total_transformer_training_flops` (see `src/flops.py`).

4 LoRA Fine-Tuning & Training Experiments

This section explores the process and results of fine-tuning the Qwen2.5 model using the Low-Rank Adaptation (LoRA) method. We describe how LoRA is integrated into the model, evaluate its performance after training, analyse computational cost via FLOPs, and investigate the impact of key hyperparameters on forecasting accuracy and efficiency.

[1] Transformer models use attention to process sequences in parallel for efficiency

[2] Self-attention lets models weigh relationships between all input tokens to capture context effectively.

4.1 LoRA + Qwen

We adapt the Qwen2.5-0.5-Instruct model for efficient fine-tuning[3] using LoRA by wrapping the query (`q_proj`) and values (`v_proj`) projection layers within each self-attention block using custom `LoRALinear` modules. This approach injects two trainable low-rank matrices `A` and `B` into each targeted layer.

4.1.1 Untrained Qwen with LoRA layers

When LoRA is added to Qwen2.5 model but no fine-tuning has occurred yet, the injected LoRA layers (matrices `A` and `B`) are randomly initialised, with Kaiming and zero initialisation for `A` and `B` respectively. Furthermore, the based model weights, `q_proj` and `v_proj`, are frozen, hence the model's behaviour is effectively alerted by this random LoRA injection.

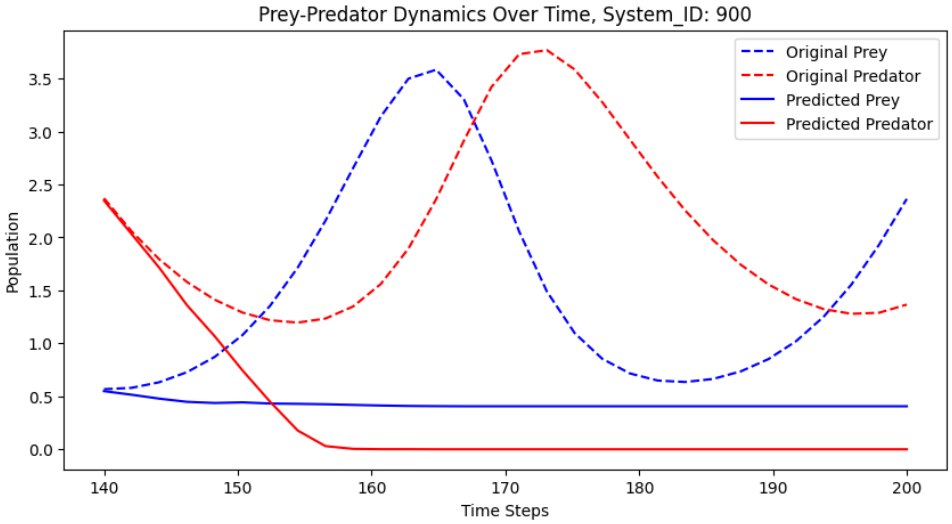

Figure 3: Result comparison between untrained Qwen2.5 with LoRA layers forecasting ability and true data for system ID 900. As we can see there is no major difference to untrained Qwen2.5, the prediction of tokens is still totally random, no signs of periodicity. Compare with Figure 1, untrained Qwen2.5.

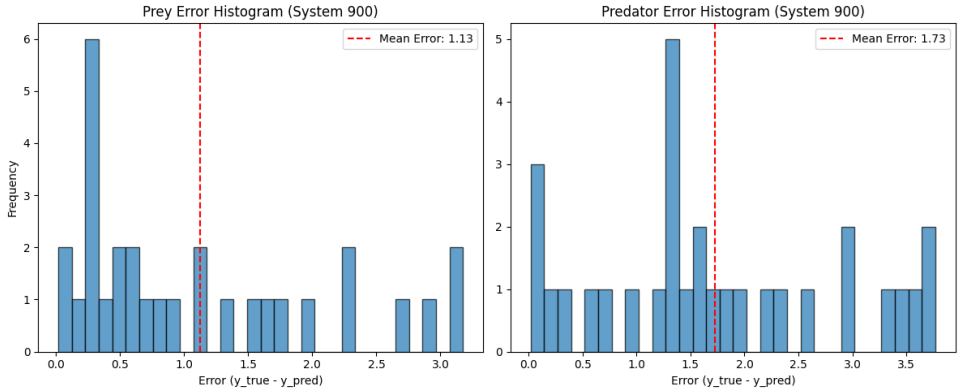

Figure 4: Histogram plot of errors between each point within system ID 900, for the untrained Qwen2.5 with LoRA layers, compare to Figure 2.

4.1.2 Fine-Tuning Performance

If we compare Figures 1 and 3, it is clear that there is no difference in terms of predicting performance between, at least for system ID 900 (this will be further discussed in section 5, where we compare

[3]Fine-tuning adjusts a pre-trained model's weights on a specific task or dataset.

generalised performance across all 100 systems), Qwen2.5 and Qwen2.5 with LoRA layers (no fine-tuning) models. The predicted tokens seems to be completely random and generated based on no prior knowledge.

What happens to the forecasting model's ability after fine-tuning?

Our goal here is to train a forecasting model that can accurately predict future dynamics of prey-predator systems using a limited set of tunable parameters. By leveraging the structure of LoRA, we aim to adapt the pre-trained Qwen2.5 model to our specific time series task without updating the full model. This raises the questions: how does the model's ability change after applying LoRA-based fine-tuning? What happens under the hood during training?

During training, the original projection weight matrices, namely `q_proj.weight, v_proj.weight`, are frozen, meaning their gradients are disabled, and only LoRA parameters are updated. As a result, the number of trainable parameters is significantly reduced, focusing on just small set of task-specific linear paths, while preserving the general knowledge of the pre-trained backbone, enabling parameter-efficient fine-tuning even on large models.

There is a designed function in the `src/set_up_lora.py`, `train_lora_model` to train models, this function is also used for hyper parameter search, discussed in section 4.2. texttttrain_lora_model fine-tunes a Qwen2.5 model using LoRA on time series data by applying low-rank adapters to attention layers, then training with next-token prediction for a fixed number of steps using Hugging Face's Accelerator for efficient optimisation, furthermore, we adopt gradient clipping using `accelerator.clip_grad_norm_(model.parameters(), max_norm=2.0)`,this will avoid exploding gradients with an early-stop. We did not implement any stopping criteria for vanishing gradients, in our case they are a minor concern, as we saw during training all L2 gradients norm are never vanishing. Transformers have residual connections, LayerNorm, and ReLU, GELU or,like in our case, SwiGLU activations, all of which naturally mitigate vanishing gradients, moreover, we are not training over the whole model, but rather LoRA adapters and bias term, which is a very shallow and stable set up. However, it is good practice to keep track of the norm gradients while training, so we designed the code to report loss and norm gradient values approximately every 1100 steps (reporting gradients is dependent on the chosen context lengths). The function returns the trained model and final loss. For all of our training procedure we trained for 5000 steps.

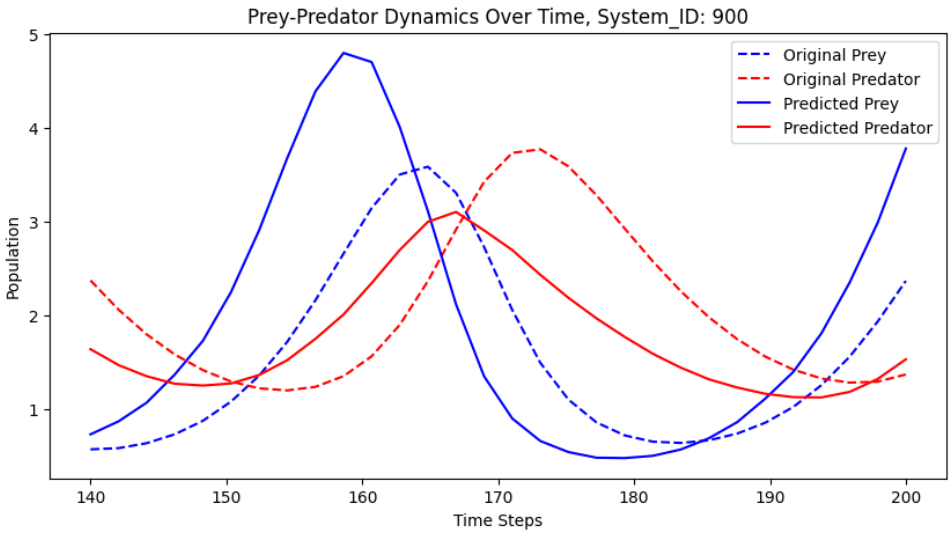

Figure 5: Result comparison between Qwen2.5 with LoRA after training (5000 steps, learning rate = 10^{-5}, rank = 4, batch size = 4, context length = 512, these are the default parameter values in `train_lora_model()`). If we compare this plot to Figures 1 and 5, one can observe a major improvement in terms of forecasting results, we see that the predicted trajectories are somewhat resembling the periodicity of the true trajectories. I cannot stress enough that observing a single system out of 100, is not enough to evidence to suggest general improvement, metrics will be compared Section 5.

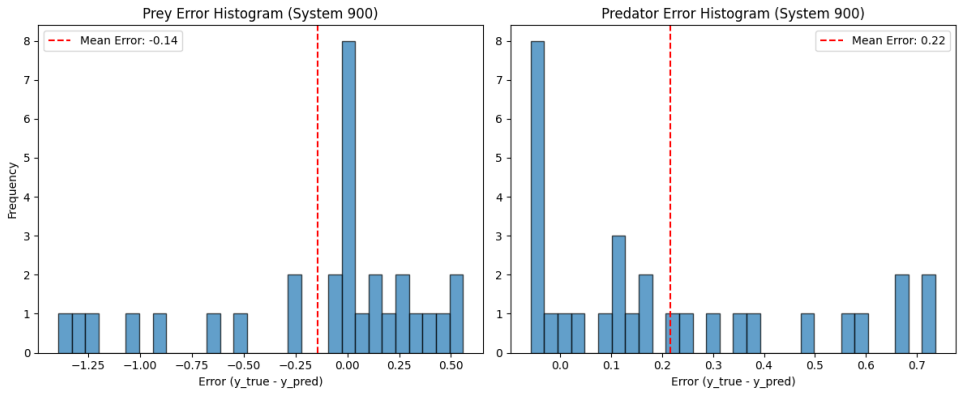

Figure 6: Histogram plot of errors measured in Figure 5. If we compare this to Figures 2 and 4, judging on the error spread, we can see that for system ID 900, the predicted and true trajectories are much closer.

4.2 Hyper-Parameter Tuning

Hyper-parameter (HP) tuning is the process of optimising the settings (like learning rate, batch size, or model depth) that control how a machine learning model learns. Unlike model parameters, these are set before training and significantly impact performance.

4.2.1 HP tuning for Qwen2.5-0.5B with LoRA layers

Our goal is to perform HP search as follows:

1. HP search for LoRA rank $\in [2, 4, 8]$, learning rate $\in [10^{-5}, 5 \times 10^{-5}, 10^{-4}]$ and fixing context length to 512.

2. Once we determined the optimal values for rank and learning rate, we fix those values and search for the optimal context length $\in [128, 512, 768]$.

HP Search Execution

We coded a nested loop to execute point 1. and a for loop for point 2. The maximum number of steps used for each search is 1000 steps, as briefly mentioned, we used the function `train_lora_model()` using all the possible HP combinations, and then picked the optimal HP based on the L2 gradient norm behaviour, training and validation loss (look for results that provided the least difference between the two, to avoid overfitting). To ensure that we were not using the "already trained model" at the end of each loop, we refreshed model and tokenizer, using `del` and `torch.cuda.empty_cache()`, see notebook `train_tune_LoRA.ipynb` part 3 (b) Hyper Parameter search.

Rank	Learning Rate	Train Loss	Validation Loss
2	1e-05	0.963	0.890
2	5e-05	0.795	0.726
2	0.0001	0.901	0.689
4	1e-05	0.919	0.828
4	5e-05	0.774	0.701
4	0.0001	0.611	0.667
8	1e-05	0.921	0.784
8	5e-05	0.780	0.676
8	0.0001	0.668	0.655

Table 1: Training and validation performance across different LoRA ranks and learning rates, with fixed context length = 512

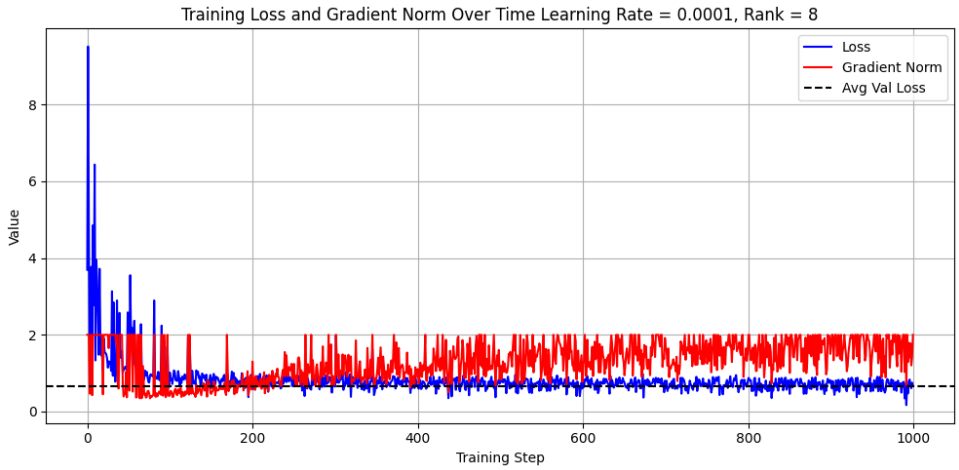

Figure 7: Gradient norm L2 and Loss values over number of steps, this graph is the output obtained during HP search, see last row of Table 1. As we can see there is no apparent overfitting, loss values are oscillating around the average validation loss (black dotted line), and the gradient norm values are acceptable (there is some clipping).

Context Length	Train Loss	Validation Loss
128	0.990	0.912
512	0.834	0.652
768	0.604	0.634

Table 2: Training and validation loss across different context lengths. With rank=8 and learning rate=0.0001.

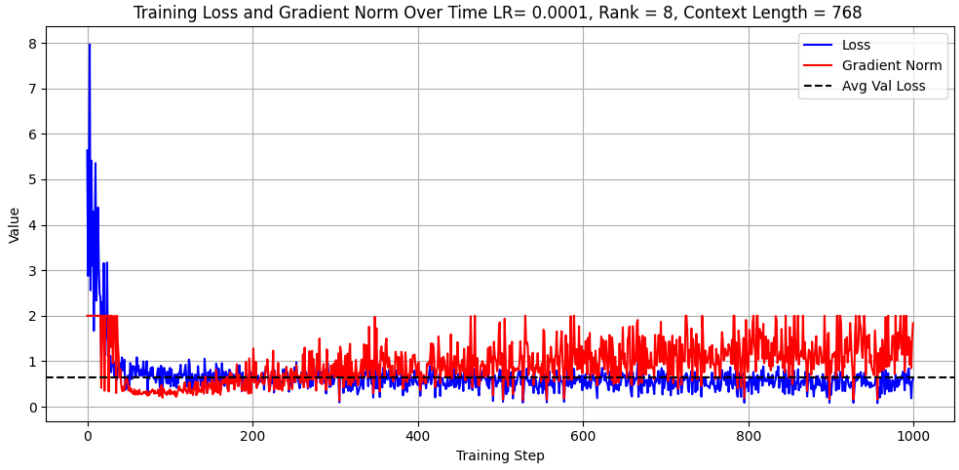

Figure 8: From Table 2, third row. Training and validation loss are close, and gradient norm is well behaved.

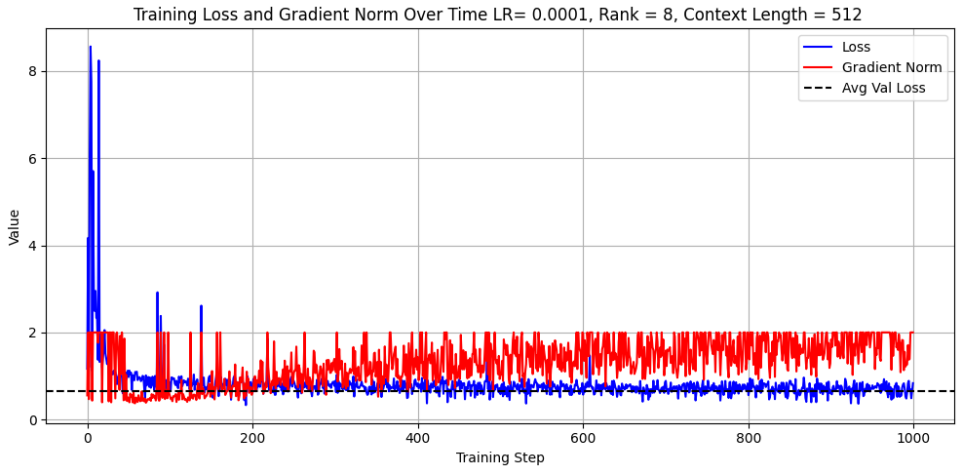

Figure 9: From Table 2, second row. Results are very similar to Figure 8, however, gradient norm has required more clipping

4.3 Tuned Model

From now on we will refer to the fine-tuning model as Model 1 (trained with 5000 steps, rank = 4, learning rate = 10^{-5}, context length = 512). Based on the HP search results, we trained two more models:

- Model 2 with 5000 steps, rank = 8, learning rate = 10^{-4}, context length = 768
- Model 3 with 5000 steps, rank = 8, learning rate = 10^{-4}, context length = 512

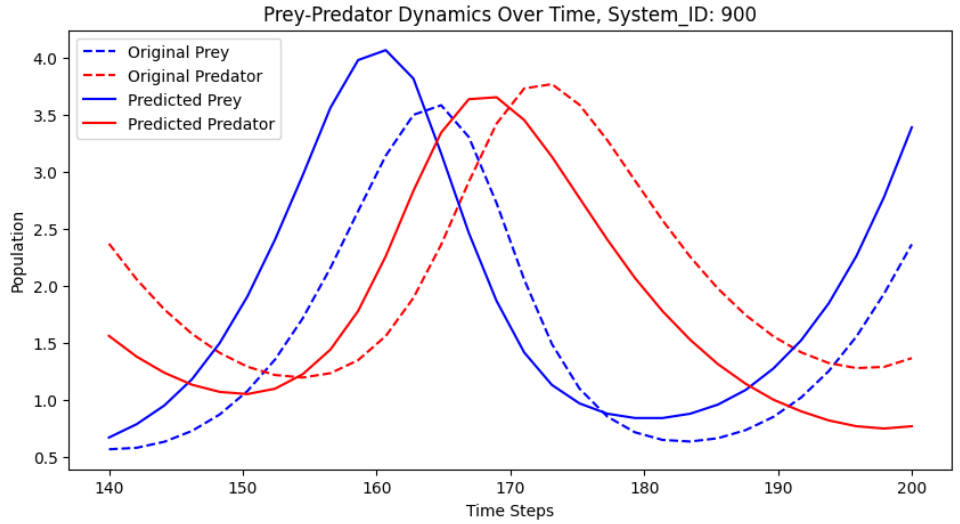

Figure 10: Model 2: Result comparison between predicted and true trajectories for system ID 900.

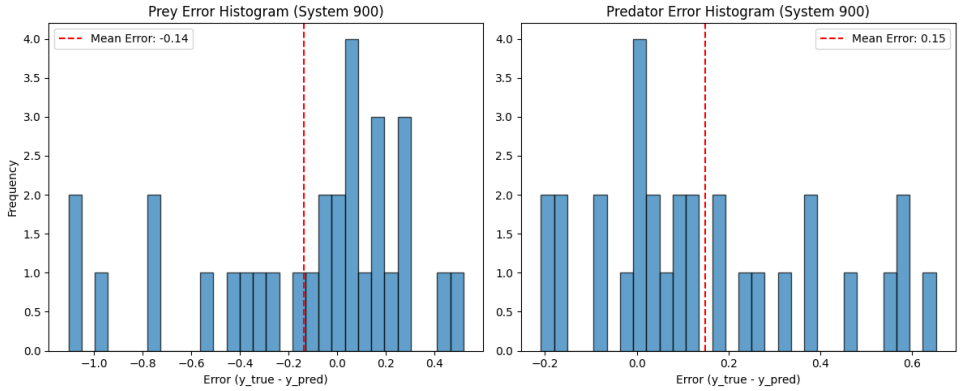

Figure 11: Measured errors in Figure 10, Model 2.

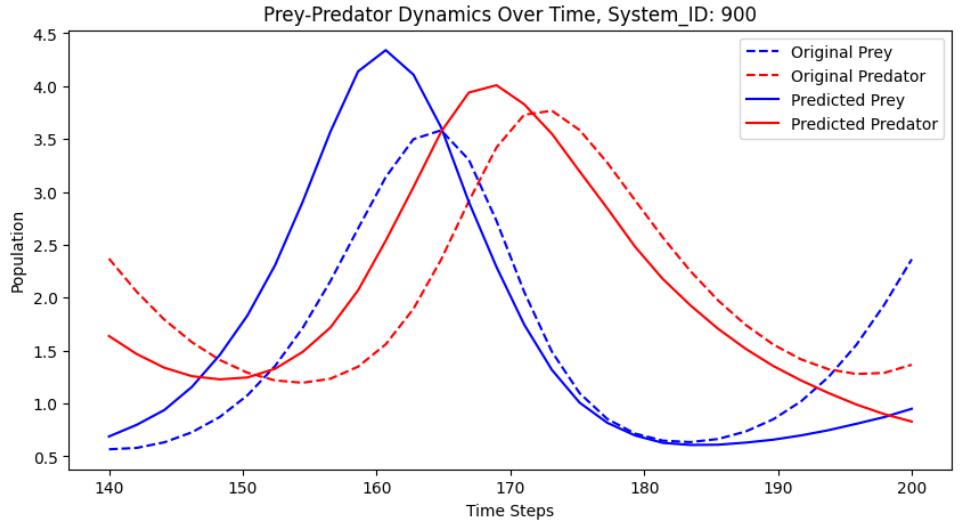

Figure 12: Model 3: Result comparison between predicted and true trajectories for system ID 900.

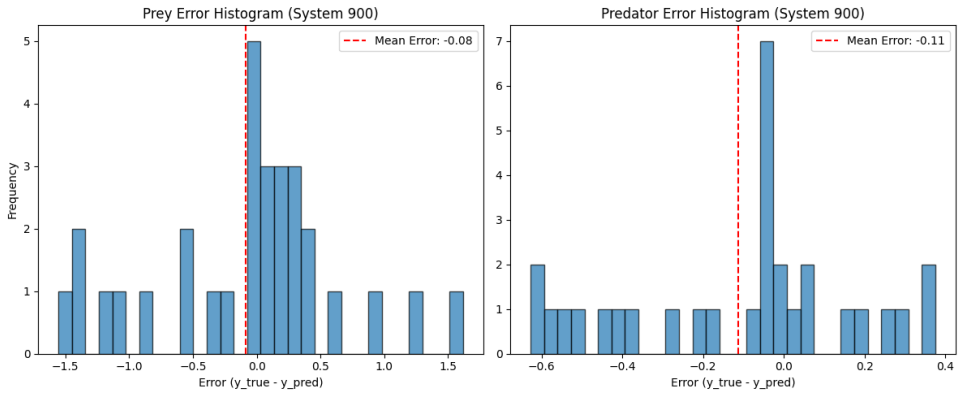

Figure 13: Measured errors in Figure 12, Model 3.

4.4 Total FOLPs

Here we are reporting the total number of estimated FLOPs used in this project, for all training procedures and hyper-parameter searches. See `flops.py` and Section 3.

Model/HP Search	**FLOPs**
Model 1	5,983,174,656,000,000
Model 2	9,026,855,239,680,000
Model 3	5,993,744,302,080,000
HP Search for Fixed Context Length (=512)	
rank=2, lr=10^{-5}	1,195,577,966,592,000
rank=2, lr=5×10^{-5}	1,195,577,966,592,000
rank=2, lr=10^{-4}	1,195,577,966,592,000
rank=4, lr=10^{-5}	1,196,634,931,200,000
rank=4, lr=5×10^{-5}	1,196,634,931,200,000
rank=4, lr=10^{-4}	1,196,634,931,200,000
rank=8, lr=10^{-5}	1,198,748,860,416,000
rank=8, lr=5×10^{-5}	1,198,748,860,416,000
rank=8, lr=10^{-4}	1,198,748,860,416,000
HP Search for Fixed Rank & Learning Rate (r=8, lr = 0.0001)	
cl = 128	297,875,275,776,000
cl = 512	1,198,748,860,416,000
cl = 768	1,805,371,047,936,000
Total FLOPs count	35,078,654,656,512,000
Allowed FLOPs Budget	100,000,000,000,000,000
FLOPs Left	64,921,345,343,488,000

Table 3: FLOPs table, displaying all the FLOPs used for training and HP search.

5 Forecasting Performance & Metrics

In this section, we are going to provide concrete results on which model was the most successful in terms of forecasting ability on the test set `val_text_70`.

5.0.1 Models Gradient Norms & Loss

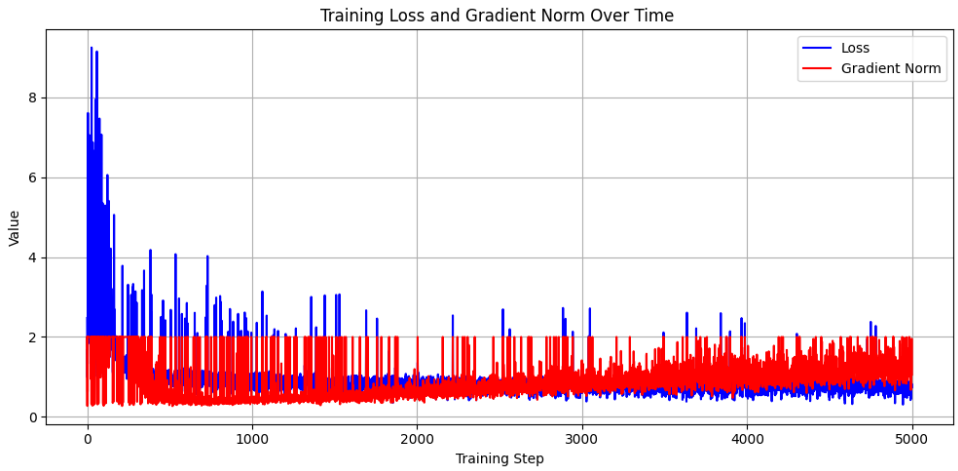

Figure 14: This is a plot of loss and gradient norm L2 behaviour during Model 1 training.

Figure 15: This is a plot of loss and gradient norm L2 behaviour during Model 2 training.

As we can see in Figures 14, 15 and 16, the loss drastically drops for all models after the first few hundreds steps, in particular for Model 2 the drop is almost immediate and the loss does not have any "crazy" spiking behaviour throughout training, while the other two models although overall well behaved, the loss is still exploding in many points. There are signs of gradient clipping in all models, the most affected is Model 2, and as mention before, there are no signs of vanishing gradients for all the models during training, hence early stop due to vanishing gradients is not required. If we had to guess which of these three models would be the best at predicting time series, we can confidently choose Model 2, as the loss values are well behaved compared to the other two models.

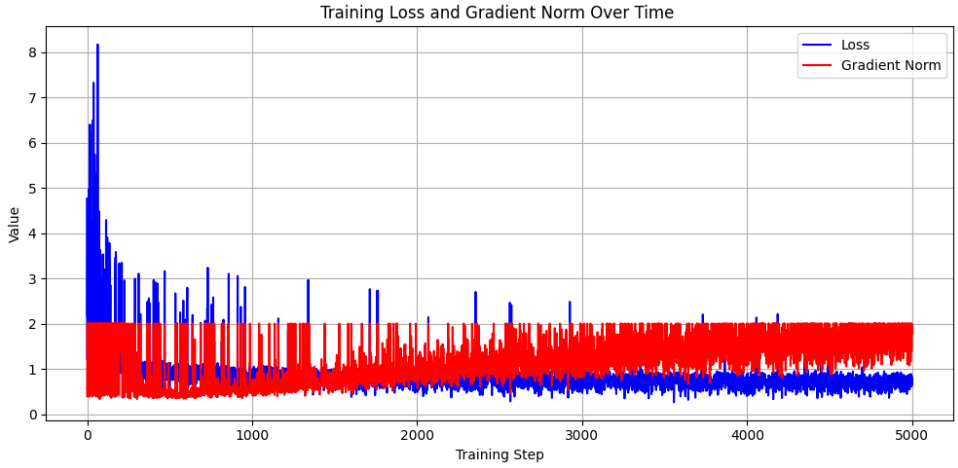

Figure 16: This is a plot of loss and gradient norm L2 behaviour during Model 3 training.

5.1 Metric Comparison

The metric that we are going to compare in this section, are training and validation loss, and mean square error (MSE) observed in each of the predicted 100 system IDs (only MSE for untrained models).

Mean Square Error (MSE)

This is computed by determining the square difference between $\hat{y}_i$ (predicted values), and y_i (true values), and take the average. For n points:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{4}$$

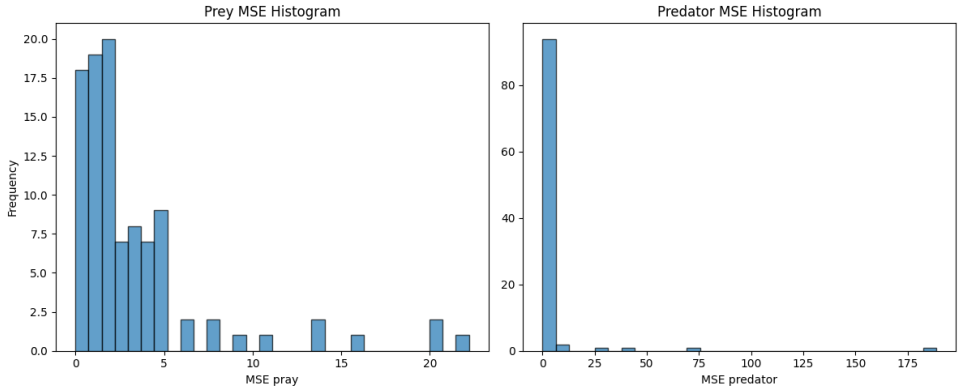

Figure 17: MSE distribution for Qwen2.5-0.5B untrained model's forecasting test on test-set

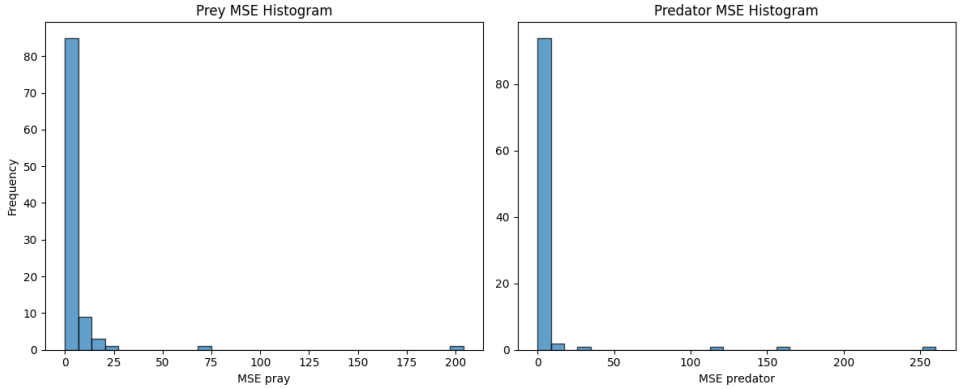

Figure 18: MSE distribution for Qwen2.5-0.5B with LoRA layers, untrained model's forecasting test on test-set

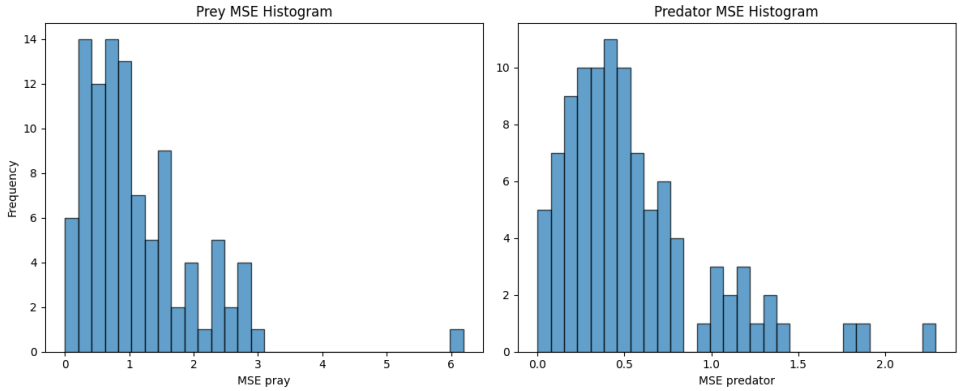

Figure 19: MSE distribution measured for Model 1

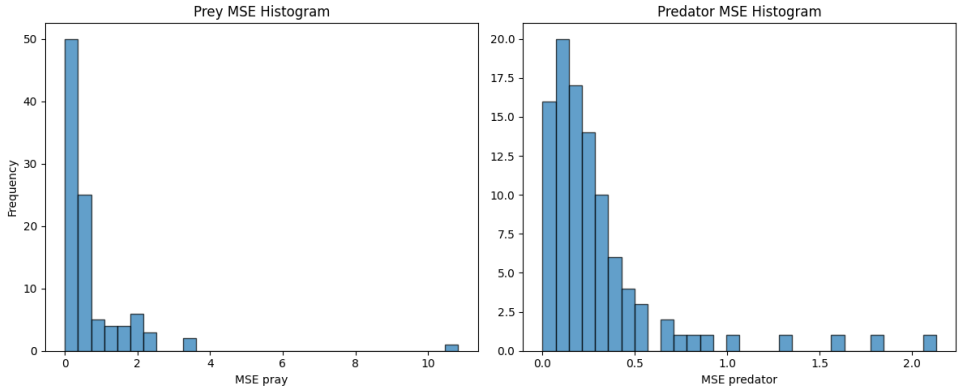

Figure 20: MSE distribution measured for Model 2

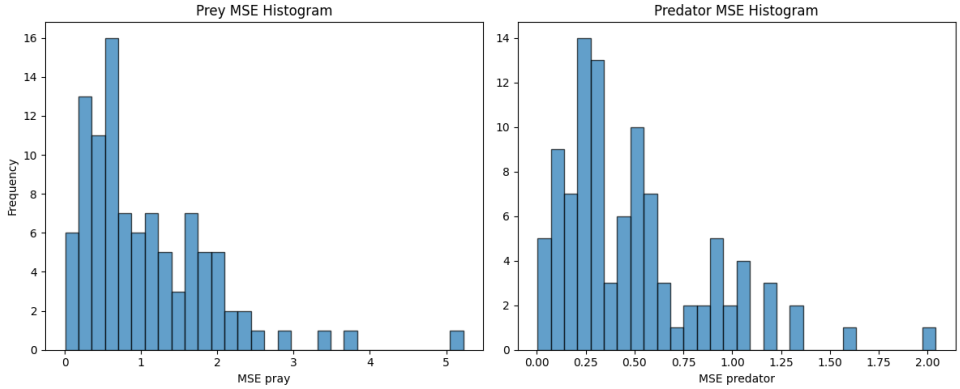

Figure 21: MSE distribution measured for Model 3

Without a doubt, all the trained models outperform the untrained models in forecasting ability. What we want to focus on is to determine which of the three trained model performed the best. We previously assumed that Model 2 would be the favourite one based on the controlled gradient norm and loss behaviour during training, is this claim going to stand after metric analysis?

Model	Average MSE	Train Loss	Validation Loss	Loss Ratio
Model 1	0.8332	0.8073	0.6911	0.8561
Model 2	0.5290	0.3874	0.5615	1.449
Model 3	0.7742	0.6568	0.6703	1.021
Qwen2.5 untrained	4.0419	N/A	N/A	N/A
Qwen + LoRA untrained	6.5425	N/A	N/A	N/A

Table 4: Table display MSE results, along with train and validation loss with their respective ratio

Comparing the MSE distributions in Figures 19, 20 and 21 and looking at Table 4, we can conclude that Model 2 outperformed the others. However, based on the validation and training loss ratio, Model 2 is the most likely to overfit the data, Model 1 is underfitting and Model 3 has a good ration ≈ 1.0. Despite the MSE results, Model 3 is the most balanced of the three.

Key Findings

Increasing LoRA rank from 4 to 8 significantly improves forecasting ability by enabling richer adaptation capacity. When setting a higher learning rate (1e-4) slightly degrades accuracy compared to 1e-5, but yields faster convergence. Longer context length (768) enables more accurate temporal pattern modelling but at a higher computational cost. And Model 2 outperforms the others in raw accuracy but may overfit Model 3 strikes a practical balance.

6 Conclusion

In this project, we explored the application of LLMs, specifically of Qwen2.5-0.5B, to forecast time series. We evaluated the string-based formatting scheme LLMTIME that allows time series data to be tokenised using natural language processing tools. Our results confirm that, even without task-specific training, LLMs possess surprising baseline forecasting abilities (untained models).

With a series of LoRA-based fine-tuning (Model 1,2 &3) experiments, we demonstrated that injecting low-rank trainable adaptations into Qwen2.5-0.5B model's attention layers leads to substantial improvements in prediction accuracy. Of all tested configuration, Model 2 (rank=8, learning rate = 10^{-5} and context length = 768) achieved the lowest MSE across both prey and predators species, however it showed signs of overfitting. Model 3, while sightly less accurate (higher MSE average), showed a more balanced loss profile, making it a better candidate in real-word deployment scenarios.

We also accounted for training compute via FLOPs estimation, staying under the budget, at approximately 3.5×10^{16} FLOPs used.

Out of all the hyper-parameters, we found that the LoRA rank and context length can significantly affect both model accuracy and computational cost. The ability to fine-tune only a small number of parameters while preserving performance highlights LoRA's potential as a lightweight adaptation strategy to use LLMs in low-resource settings.

Based on our overall results, we might wonder if 5000 steps lead to overfitting for Model 1 and 2, thus, one of the future implementation can be early stop when loss profile gives signs of under/overfitting.

Going forward, this approach could be extended to multi-variate, real-world datasets, and scaled to larger base models or alternative architectures.

References

[1] Qamar Din. "Dynamics of a discrete Lotka-Volterra model". In: *Advances in Difference Equations* 2013 (2013), pp. 1–13.

[2] Nate Gruver et al. "Large language models are zero-shot time series forecasters". In: *Advances in Neural Information Processing Systems* 36 (2023), pp. 19622–19635.

[3] Simon JD Prince. *Understanding deep learning*. MIT press, 2023.

[4] An Yang et al. "Qwen2. 5 technical report". In: *arXiv preprint arXiv:2412.15115* (2024).