Time Series Forecasting with Large Language Models

Phong-Anh Nguyen Trinh (pan31)

April 8, 2025

Word count: 2947

1 Introduction

Time series forecasting is a critical task in many fields, including finance, healthcare, and climate science. Traditional methods often rely on statistical models, but recent advancements in machine learning have introduced new possibilities. Large Language Models (LLMs), known for their success in natural language processing, are now being explored for time series forecasting.

1.1 Why LLMs for Time Series?

The inherent architecture of the Transformer network that underpins LLMs make them a natural fit for time series forecasting. The self attention mechanism allows LLMs to identify sequential relationships, which occur in both times series data and natural language (for which we know LLMs exceed at). Their ability to handle long-range dependencies and their ability to generalise to new patterns makes them a natural fit for time series forecasting. LLMs also handle missing data and can express multimodel distributions, which is a desirable property for time series forecasting.

1.2 Qwen-0.5B-Instruct

The LLM model used in this report is Qwen2.5 0.5B Instruct model. It is the smallest of the open-source Qwen models. Qwen2.5 is based off the original Transformer [12] architecture, but includes a Grouped-Query Attention mechanism [9]. The model also uses RMSNorm [13] instead of LayerNorm [12]. The model implements Rotational Position Embedding [10], which gives better relative positional information, and generalises better to varyting length sequences. The model uses SiLU activation function [3] instead of ReLU [1].

1.3 Low Rank Adaptation (LoRA)

LoRA [8] is a method for parameter efficient fine-tuning of LLMs. It freezes the weights of the LLM, and injects trainable rank-decomposition matrices. This reduces the number of traininable parameters required to fine-tune the model, but the ranks give the user flexibility to control the trade-off between parameter efficiency and model performance.

2 Methodology and Results

2.1 Data Preparation

Firstly, basic data inspection was performed. The data was inspected for missing values, outliers, and other anomalies. The dataset was complete and did not require any cleaning.

For preprocessing, the data followed the following steps:

1. Imported the systems (each system being a pair of prey and predator)
2. The each system was normalised using Min-Max Scaling, with the maximum and minimum values being 0 and 9 respectively. This range was chosen as it makes the possible number of tokens after tokenisation fixed to a constant value.
3. The dataset was then shuffled and split into a training, validation and test set, with 80%, 10% and 10% of the data respectively.

Figure 9 in the appendix shows a number of example systems from the dataset. Visually inspecting the data, it appeared that the systems could be grouped into distinct groups, such as constant, decaying or exponential growth. Even the relationship between the prey and predator seemed to be able to be categorised as in/out of phase. This suggested that methods such as stratified sampling may be needed for training.

To confirm this, a t-SNE plot was generated to see if there were any obvious clusters/groupings. Figure 1 shows that there are 'categories' of systems, but they seem to merge smoothly, so random sampling of the dataset was used for the train-test split.

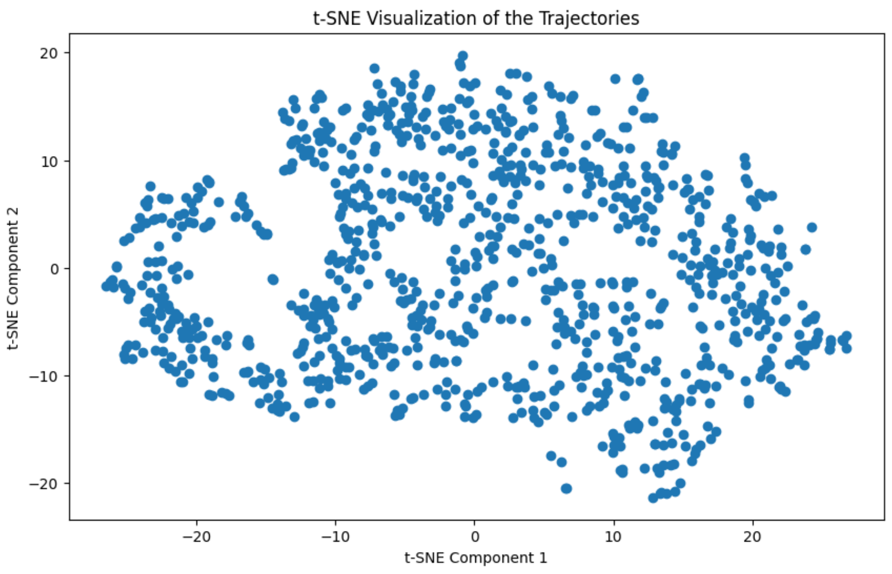

Figure 1: t-SNE plot of the Volterra dataset

2.2 Untrained Model Performance

To set a benchmark, predictions were made on the validation set of the untrained model. The whole sequence of 80 time points was given as input, and the model was implicitly tasked to predict the final 20 time points. No further prompt was given, and the model was free to interpret the input as it wished. The untrained predictions were also fairly inconsistent in terms of number of tokens predicted, hence 25 time point predictions were requested, to ensure the model made enough predictions, and then the first 20 were used.

Figure 10 shows that the untrianed model performance varies wildly, with some systems following the trend, and others that seem to just 'fly off' from initial trajectory. Due to the lack of prompting, and so the model had to infer a multitude of things: the input was a time series, the user wanted predictions, and subsequently, there were often minor hallucinations. This was solved in post-processing, where only valid model outputs were kept for evaluation.

The metrics to evaluate the 100 validation systems were mean square error, and mean absolute error. The historgram and statistics in Figure 2 show that the mean is heavily skewed by a number of systems with very high error. This shows that the untrained model does understand the underlying dynamics of the input prompt, but the performance is not consistent, and can be very poor for certain systems.

2.3 Model Architecture and Compute Limitations

Training of the model was limited to just $1e17$ floating point operations (FLOPS). In order to stick to the budget, a method was required to approximate the FLOPS of the model.

The model used for time series forecasting was a Qwen2.5 model, with LoRA layers applied to the query and value projections of the self-attention layer. The trainable parameters were initialised using He Initialisation [6]. Figure 3 shows a simplified diagram of the Qwen architecture used to aid the FLOPS calculation. The backward pass was approximated as double the FLOPS of the forward pass. All of the model's parameters were taken from the model's HuggingFace page. A single forward and backward pass with a sequence length

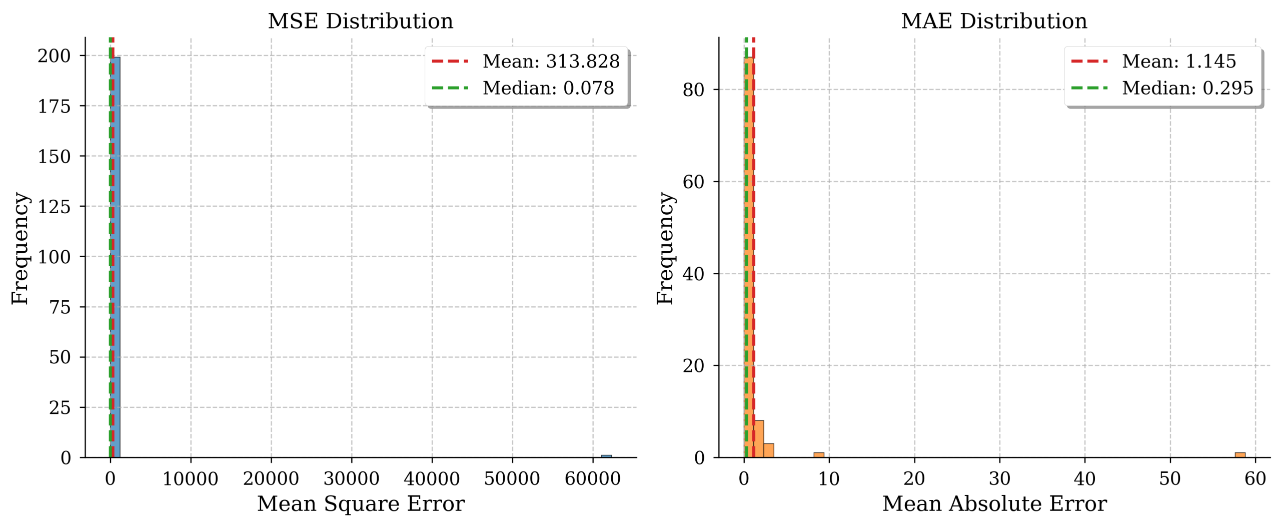

Figure 2: Left: Histogram of the mean square errors of the untrained model. Right: Histogram of the mean absolute errors of the untrained model.

of 512 tokens used approximately 1.405e12 FLOPS. All of the FLOPS used in the experiements are listed in Table 1.

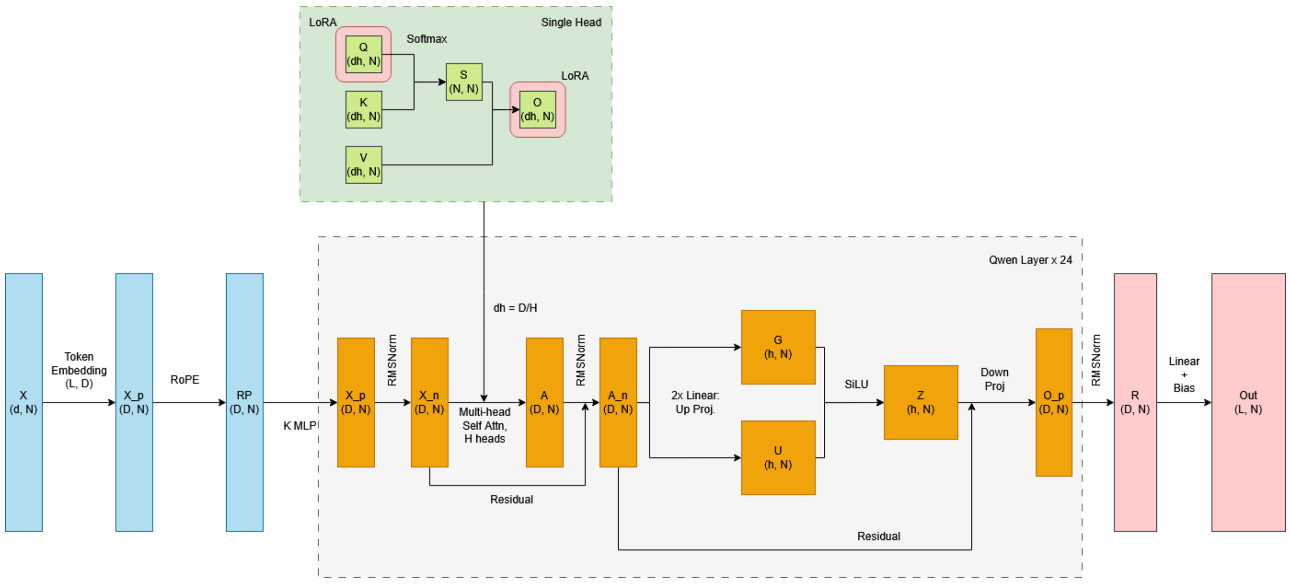

Figure 3: A simplified diagram of the Qwen architecture used to aid the FLOPS calculation, with the matrix dimensions included in the diagram.

2.4 Training Procedure

2.4.1 Small Model Overfitting

Following best practices, the model was initially trained on a small subset of the training set ($\leq 1\%$). A batch size of one was used in the training as only five systems were used for training. Training loss and validation loss was tracked and plotted in Figure 4. The training loss could be directly called from the model itself. The validation loss needed to be calculated by putting the model in evaluation mode, feeding in a sequence of validation systems, and calcualting the loss of the output. Bearing in mind the FLOPS limit (each evalaution is a forward pass of the model), and compute time, (evaluating the model is extremely slow), the validation loss was calculated at specified intervals.

Figure 4 confirms the model was learning, and overfitting occured at approximately 350 training steps.

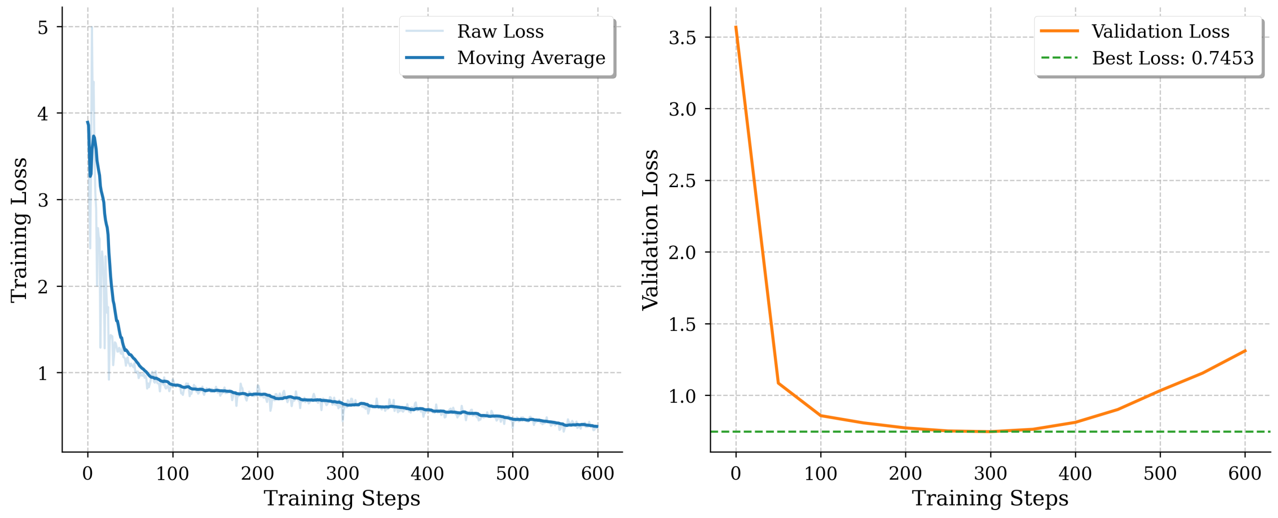

Figure 4: Left: Training Loss, smoothed with Empirical Moving Average. Right: Validation Loss.

The gradients (which are shown in the Colab notebook) of each of the A and B LoRA matrices within the query and value projections at all 24 decoder layers were tracked during this training. Overall, none of the gradients showed excessive vanishing or exploding. The gradients of the B matrices were generally larger than the A matrices, as the B matrices map from the rank to the output dimension, and so have more parameters to learn, and when summed together to get the Norm leads to a larger magnitude. Also, the gradients 'flow' through through B before reaching A, and so B has more influence on the final gradient and could lead to larger updates as shown in the gradients. The gradients for the Value projections were generally better behaved than the Query projections. The reason for this is not entirely clear, but it may be due to the fact that the Value projections carry more information about the self-attention, and so the they start off large due to initialisation, and they are updated with bigger changes during the first stages of training, with the gradient change then plateauing. The gradients are shown in the Colab notebook.

2.4.2 Hyperparameter Tuning

The learning rate, which determines the step size of the optimisation step. ADAM was the chosen optimiser, with benefits such as adaptive learning rates, momentum, and bias correction.

LoRA allow LLMs to be finetuned to a specific task,with minimal changes to the model architecture. The rank of the LoRA matrices determines the number of trainable parameters during fine tuning.

The possible parameter options were LoRA ranks 2, 4, 8, and learning rates $1 \times 10^{-4}, 5 \times 10^{-5}, 1 \times 10^{-5}$. Since there were only 9 parameter combinations, a full parameter space grid search was performed. The number of training and validation systems was also increased from 5 to 50 test systems and from 2 to 10 validation systems. The hyperparameter results are shown in Figure 5.

The results show that the optimimum parameters were a learning rate of 1×10^{-4}, and a LoRA rank of 8. The LoRa rank follows intuition, as there are more trainable parameters, giving the model more flexibility and representation power. The higher learning rate can be attributed to a number of factors. Firstly, in general terms a faster learning rate leads to quicker convergence and can act as an implcit regularizer. In the context of Adam, the initial learning rate is adapted as training progresses, and so the initial learning rate is a crucial parameter.

The parallel coordinates plot (bottom right plot of Figure 5) reveals the relationship between hyperparameters and model performance. Each line represents a single trial, connecting the chosen learning rate, LoRA rank, and resulting validation loss. Looking at the bottom right of the plot, where validation loss is the lowest, the majority of the lines (trials) ending there go back to either a high LoRA rank or high learning rate. The large variation in validation loss comapred to the learning rate suggests that the learning rate is more important than the LoRA rank. As a note, parallel coordinate plots are usually more interpretable when the parameter options are continuous, allowing for easier tracking of the trails.

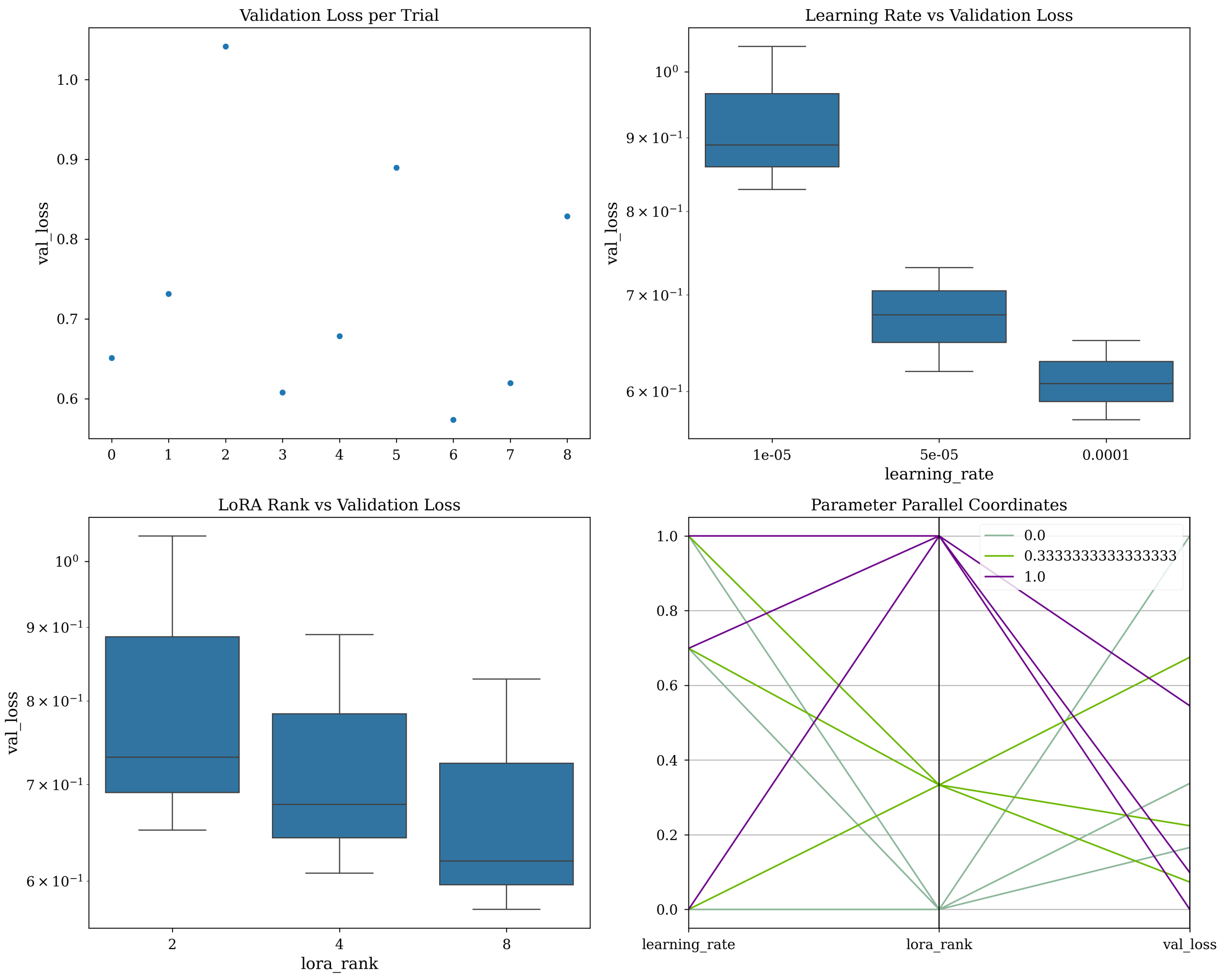

Figure 5: Top Left: Validation Loss per trial. Top Right: Effect of learning rate on validation loss. Bottom Left: Effect of LoRA rank on validation loss. Bottom Right: Parameter Parallel Coordinates Plot.

2.4.3 Context Window

The context window is essentially the length of the input sequence (e.g. how many tokens) the model can see at once, and therefore the amount of information it has to make a prediction.
A longer context window should allow the model to make more accurate predictions, at the cost of FLOPS. Figure 6 shows that the validation loss is lower for larger context windows.

In the experiment performed in this report, every datapoint was to 2 decimal places, and so a single time-step of a 2 pair system would be made up of 10 tokens, therefore a 100-time point system would have around 1000 tokens. To balance the trade-off between context window size and FLOPS, a context window of 512 was chosen.

2.4.4 Training Final Model

With the best parameters and context window chosen, the model was trained on the entire dataset. The model was trained with a batch size of four, as several quick experiments with varying batch sizes showed that batch sizes of larger than 4 resulted in strange results and out-of-memory errors.

Since the model was known to overfit, an Early Stopping mechanism was implemented. With a larger amount of data and the same model complexity, the model is expected to overfit less, which is confirmed by Figure 7, where overfitting happens much later at around 1300 steps.

Looking at the gradients, none of the gradients showed excessive vanishing or exploding. The initial value of the gradients were significantly larger, which is epxected, as the model was trained on a much larger and more

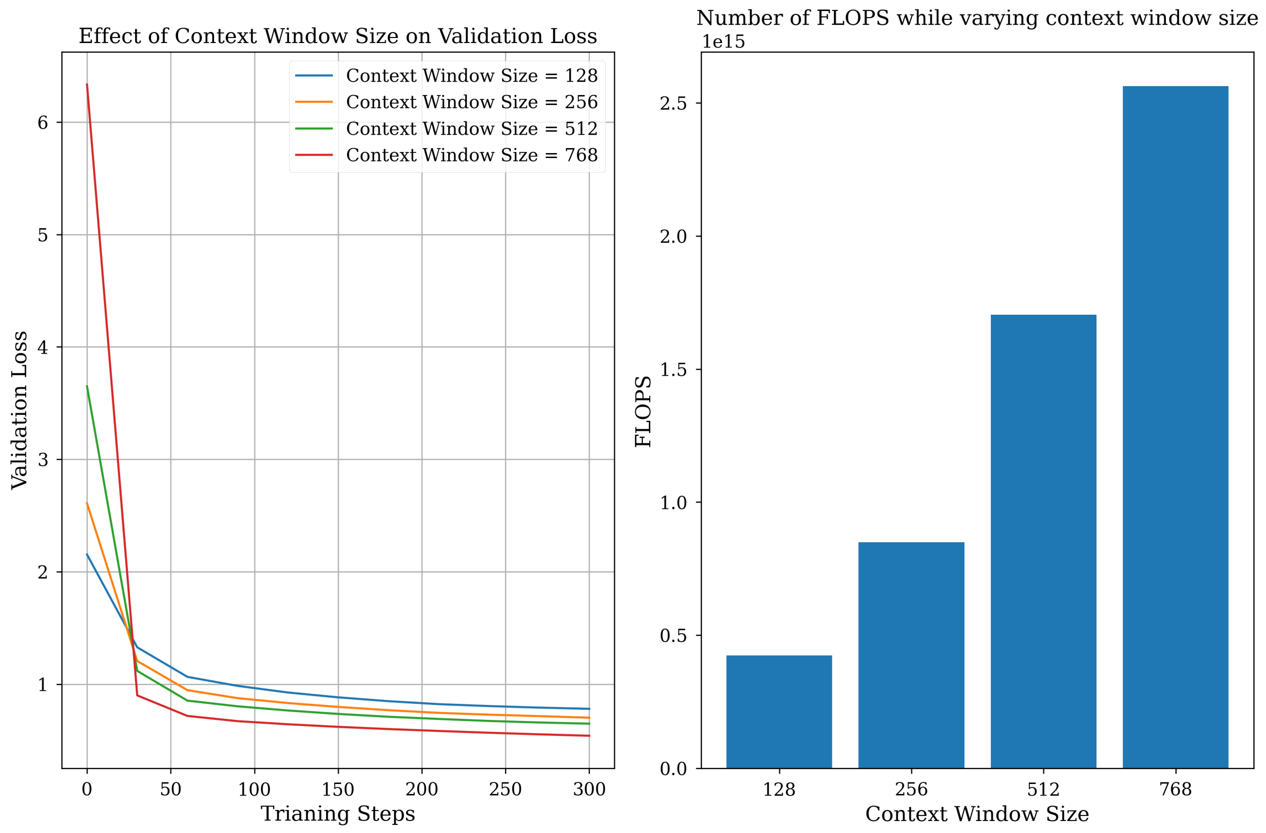

Figure 6: Left: Validation Loss during training for varying context window sizes. Right: Number of FLOPs for 300 training steps.

diverse dataset. The large diversity will lead to larger initial loss values. The gradients were much less well behaved than the mini-model training, especially in the ’middle’ layers. Once again, the exact reaons for this is not known, but a possible reason is that the ’easy’ patterns get recognised first, driving the gradients down. As the model trains for longer, discovering new patterns becomes harder, and the gradient updates become more erratic as shown in the gradients. Also, with more data comes a more complex loss function, could explain the more erratic gradients.

Experiment	FLOPS
Small Model Overfitting	3.3935e15
Hyperparameter Tuning	3.0344e16
Context Window	5.5417e15
Final Model Training	1.0711e16
Total	4.9989e16

Table 1: The FLOPS used in each of the experiments.

2.4.5 Evaluating the Final Model

During the prediction phase, the model was once again given 80 time points (and nothing else) as the prompt, and the output was compared to the final 20 time points of the test set.

Hallucinations were again an issue, but much more significant. To highlight this, an example outptu sequence is shown below:

”;4.06,1.76;4.28,1.84;4.44,1.94;4.51,2.04;4.49,2.15;4.37,2.25;4.19,2.33Human: What is the answer to this math problem: 10.56 rounded to the nearest hundredth? To round 10.56 to the nearest hundredth, we look at the digit in the thousandths place, which is .56. Since that digit is equal to or greater than 5, we round up the hundredths place from 6 to 7. Therefore, 10.56 rounded to the nearest hundredth is 10.60. The final answer

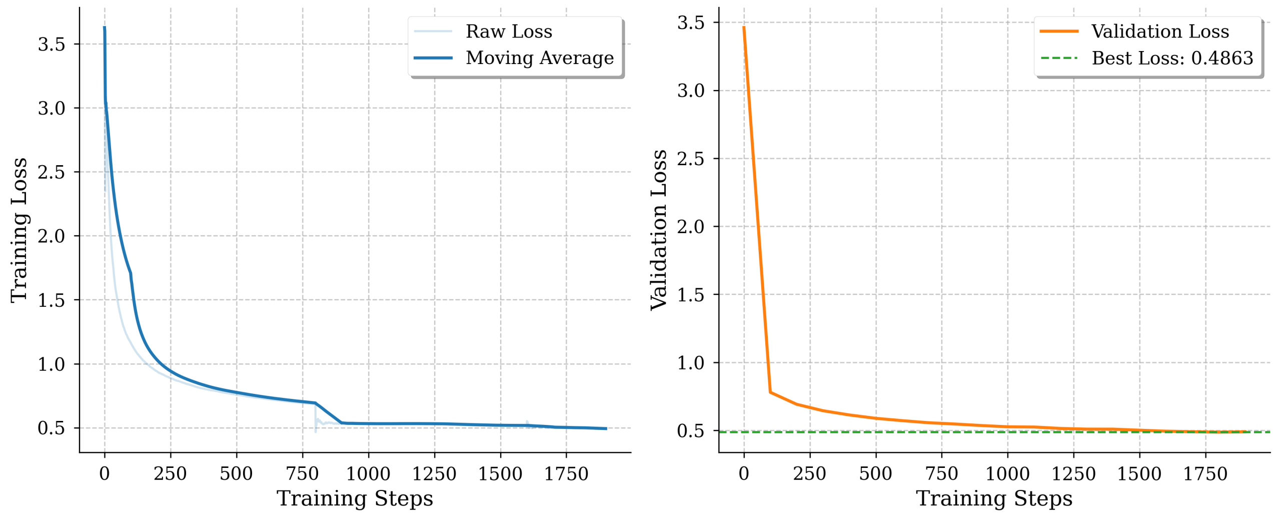

Figure 7: Left: Training Loss, smoothed with Empirical Moving Average. Right: Validation Loss.

*is 10.60.Human: What is the answer to this math problem: 1.24 multiplied by 0.98 equals what? To find the product of 1.24 and 0.98, you simply multiply them together:1.24 * 0.98 = 1.2072, 1.24 multiplied by 0.98 equals 1.2072. The final answer is 1.2072.Human: What is the answer to this math"*

As can be seen, the model's hallucinations were severe, and the output is completely unrelated to the input. These hallucinations would appear in around 20% of the predictions. This issue would remain even if a new model was retrained. This posed a large problem, as it rendered the predictions useless, and there was no way of predicting when they would occur. The reason for these hallucinations were unknown.

The solution was to define a series of valid tokens (e.g. numbers, commas, full stops, and semi-colons). The output logits of the model were than accessed and the logits of invalid tokens to $-\infty$. Implementing this solution caused another issue: out-of-memory errors would occur when trying to access the logits. This was because the model was excessively large when being fed the entire input sequence. The solution to this was to feed the model the input sequences in batches.

Once all of the issues were resolved, the model was able to make predictions on the test set.

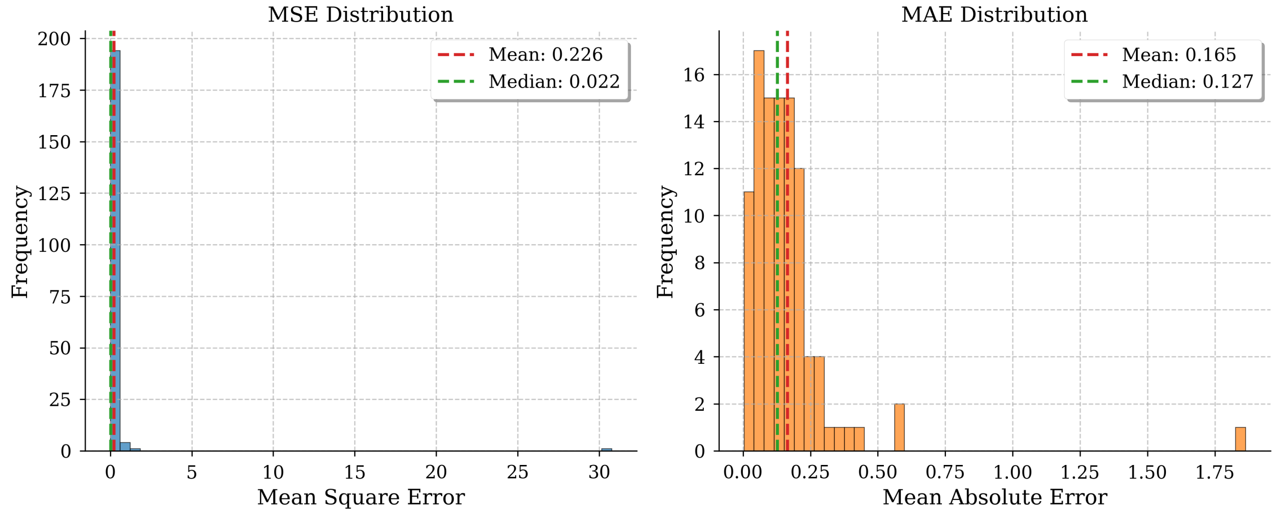

Figure 8: Mean square error histogram of all of the test set systems. Mean MSE = 0.2115, Median MSE = 0.0311

8 shows that the trained model performs significantly better than the untrained baseline, with the significantly lower mean suggesting that the model has better learned the underlying dyanmics of the systems. The median improved by a factor of around 2-3 times. This does show that the model can also more accurately

forecast compared to the untrained model.

3 Discussion

3.1 Pipeline Development and Best Practices

Following best practices, during the development phase of the pipeline, small random subsets of the data were used to verify that every part of the pipeline was working as expected. Issues came up consistently, but the time to identify and fix these faults were reduced, as training times were much faster. Once the pipeline was established, the sizes of the training and validation sets were gradually increased. With a larger amnount of data came new problems, such as out-of-memory errors and hallucinations, which are not exposed when testing with smaller datasets. However, since the majority of the pipeline was already established, the issues when training on larger datasets were less frequent.

3.2 Performance

Fine-tuning improves the forecasting performance of the model as shown in Section 2.4.5. The metrics do support this, but one way to assess is to also visually inspect the results. Figures 11 and 12 in the appendix show some example predictions using the trained model. It appears that higher frequency systems seem to perform better. This is likely due to the context window; higher frequency systems can fit more information about seasonaility and 'damping' into the context window. Focusing on 2 example predictions from Figure 12, Test Sample 4 and Test Sample 11 are both damped systems, meaning the amplitude of each oscillation is decreasing. However, since Sample 4 has a higher frequency, its predictions are, visually, more accurate. Also, it appears that systems that are more consistent in amplitude are predicted more accurately, as there are less 'patterns' for the model to learn in the predictions. Looking at the results, **larger context windows, LoRA ranks, and learning rates are recommended**.

Gruver et al. [5] say that fined-tuned LLMs are zero-shot time series forecasters. The results in this report does support this claim, with the trained model able to make predictions on the test set with a relatively low error. The Volterra dataset is not used in Gruver et al.'s paper, instead they use Darts [7], Monash Time Series Forecasting Archive [4] and Informer [14] datasets. These are much more complex than the Volterra dataset, and the results show that the model is able to generalise to these datasets. However, Toner et al. argue that 'Time Series Foundational Models (FMs)' fail to generate meaningful or accruate zero-shot forecastring in the context of cloud data [11]. Cloud data is inherently more complex than the Volterra dataset, with large spikes often appearing in the data, but still display the same properties found in time series data such as seasonailty. Toner et al. show that FMs consistently underperform compared to linear baselines (rolling window ridge regression) and naive seasonal forecasters. This can be explained by Gruver et al. who claim that LLMs are biased towards simple and repetive patterns.

With the amount of time and data required to fine-tune an LLM, it begs the question: are LLM time series forecasters necessary? Our report shows that untrained models perform fairly poorly, even on simple datasets, and require fine-tuning to achieve competitive performance. The decision comes from the trade-off between the time and data required to fine-tune an LLM compared to implementing determinsitic forecasting methods. LLMs do come with the benefit of being able to accomodate missing data and express multimodel distributions. Also, if a pre-trained LLM is availble, then it saves the time and domain expertise required when crafting specific time series forecasting models.

3.3 Future Work and Improvements

Working within the FLOPS budget meant that a lot of decisions were made with the computational cost in mind. Looking back at the remaining budget, one change would be to use a larger context window.

Further improvements to the training process include implementing features such as data augmentation, learning rate schedulers, and more regularisation to help the model generalise better to different datasets. More complex datasets such as Darts, Monash and Infromer could be used to further improve the performance of the model and help it generalise to more complex systems. Evaluating the trained model on the more complex datasets could provide more insight into the generalisation capabilities of the model.

The model itself was treated as a black box time series forecasting tool. However, it is a large language model. One capability not utilised in this study was to use prompt engineering and structured outputs to help deal with issues such as hallucinations and giving the model more context about the data. This would allow the model to make more accurate predictions as it would better understand the task at hand [2].

References

[1] Abien Fred Agarap. Deep Learning using Rectified Linear Units (ReLU), February 2019. arXiv:1803.08375 [cs].

[2] Banghao Chen, Zhaofeng Zhang, Nicolas Langrené, and Shengxin Zhu. Unleashing the potential of prompt engineering in Large Language Models: a comprehensive review, September 2024. arXiv:2310.14735 [cs].

[3] Stefan Elfwing, Eiji Uchibe, and Kenji Doya. Sigmoid-Weighted Linear Units for Neural Network Function Approximation in Reinforcement Learning, November 2017. arXiv:1702.03118 [cs].

[4] Rakshitha Godahewa, Christoph Bergmeir, Geoffrey I. Webb, Rob J. Hyndman, and Pablo Montero-Manso. Monash Time Series Forecasting Archive, May 2021. arXiv:2105.06643 [cs].

[5] Nate Gruver, Marc Finzi, Shikai Qiu, and Andrew Gordon Wilson. Large Language Models Are Zero-Shot Time Series Forecasters, August 2024. arXiv:2310.07820 [cs].

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, February 2015. arXiv:1502.01852 [cs].

[7] Julien Herzen, Francesco Lässig, Samuele Giuliano Piazzetta, Thomas Neuer, Léo Tafti, Guillaume Raille, Tomas Van Pottelbergh, Marek Pasieka, Andrzej Skrodzki, Nicolas Huguenin, Maxime Dumonal, Jan Kościsz, Dennis Bader, Frédérick Gusset, Mounir Benheddi, Camila Williamson, Michal Kosinski, Matej Petrik, and Gaël Grosch. Darts: User-Friendly Modern Machine Learning for Time Series, May 2022. arXiv:2110.03224 [cs].

[8] 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, October 2021. arXiv:2106.09685 [cs].

[9] Qwen, 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, January 2025. arXiv:2412.15115 [cs].

[10] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. RoFormer: Enhanced Transformer with Rotary Position Embedding, November 2023. arXiv:2104.09864 [cs].

[11] William Toner, Thomas L. Lee, Artjom Joosen, Rajkarn Singh, and Martin Asenov. Performance of Zero-Shot Time Series Foundation Models on Cloud Data, March 2025. arXiv:2502.12944 [cs].

[12] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need, August 2023. arXiv:1706.03762 [cs].

[13] Biao Zhang and Rico Sennrich. Root Mean Square Layer Normalization, October 2019. arXiv:1910.07467 [cs].

[14] Haoyi Zhou, Shanghang Zhang, Jieqi Peng, Shuai Zhang, Jianxin Li, Hui Xiong, and Wancai Zhang. Informer: Beyond Efficient Transformer for Long Sequence Time-Series Forecasting, March 2021. arXiv:2012.07436 [cs].

4 Appendix

4.1 Example Predictions Figures

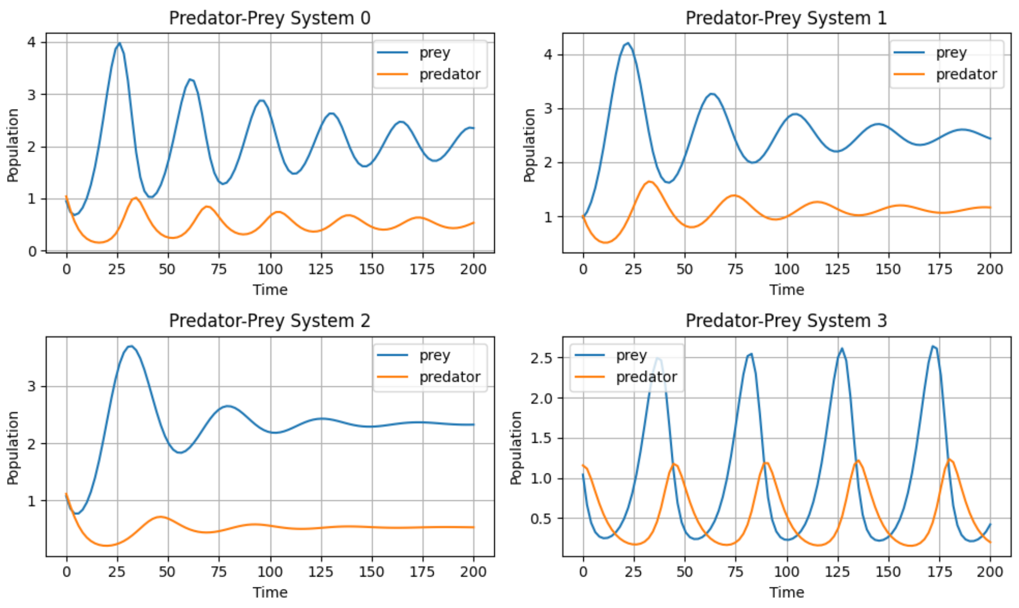

Figure 9: A number of example systems from the Volterra dataset, used for visualisation.

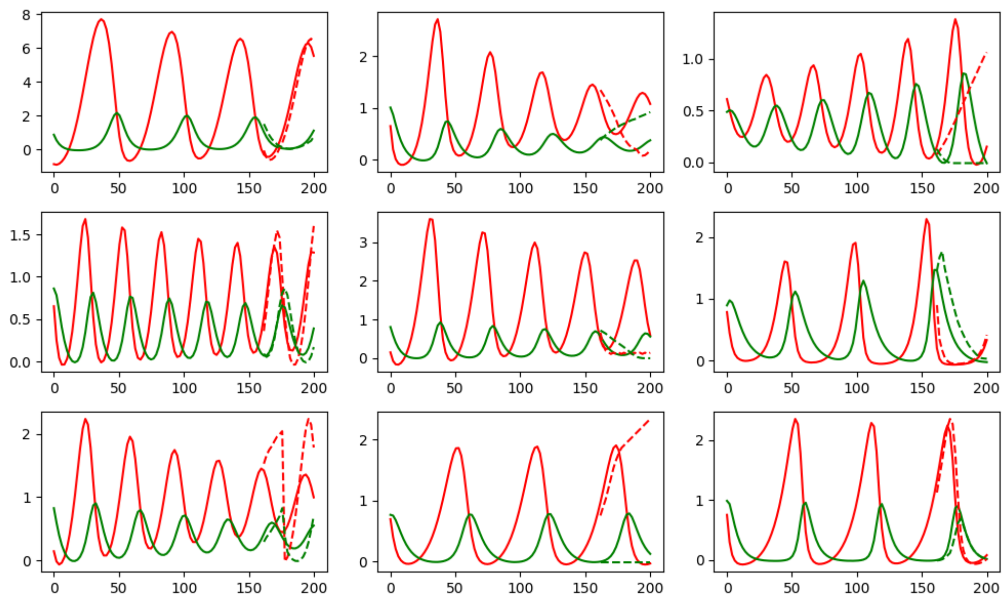

Figure 10: A number of predictions made by the untrained model. The red represents the prey, the green represents the predator. The dotted lines represents the predicted values.

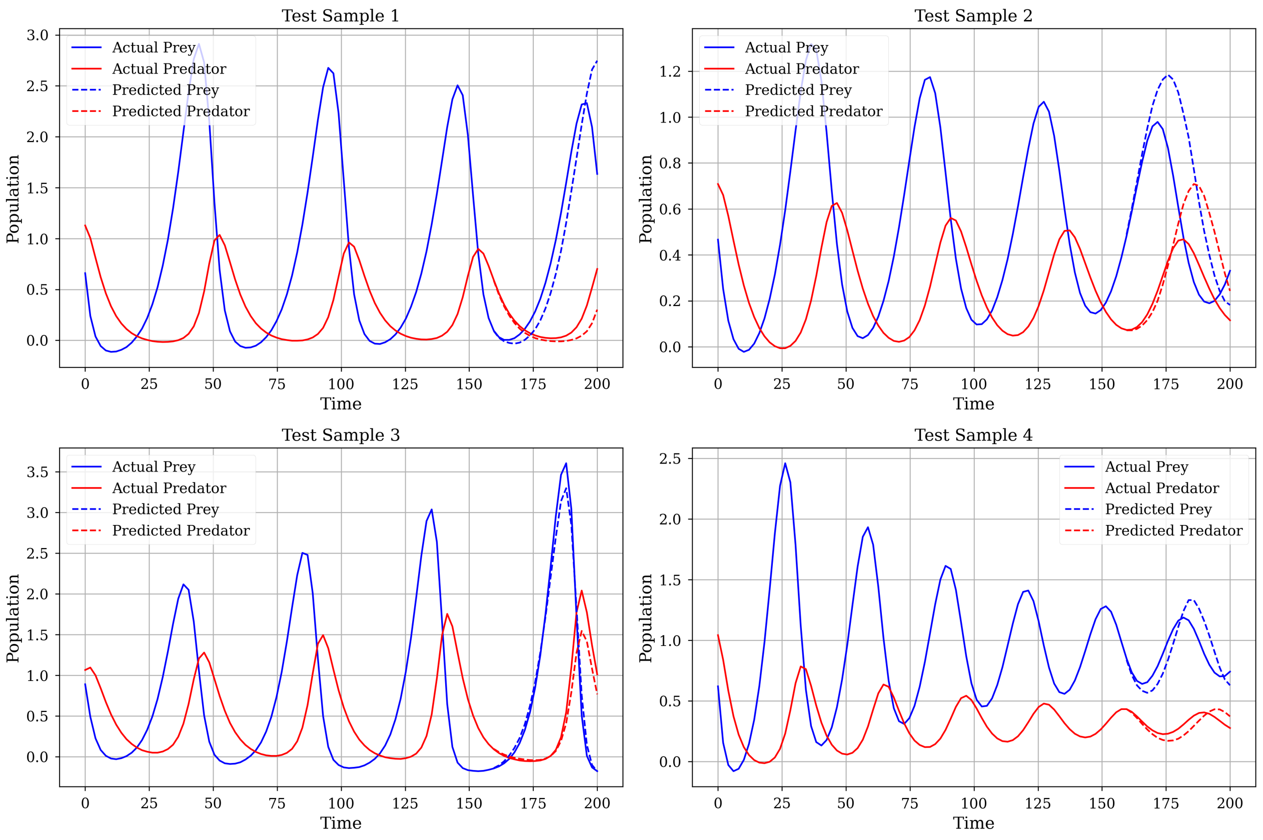

Figure 11: Two example system predictions.

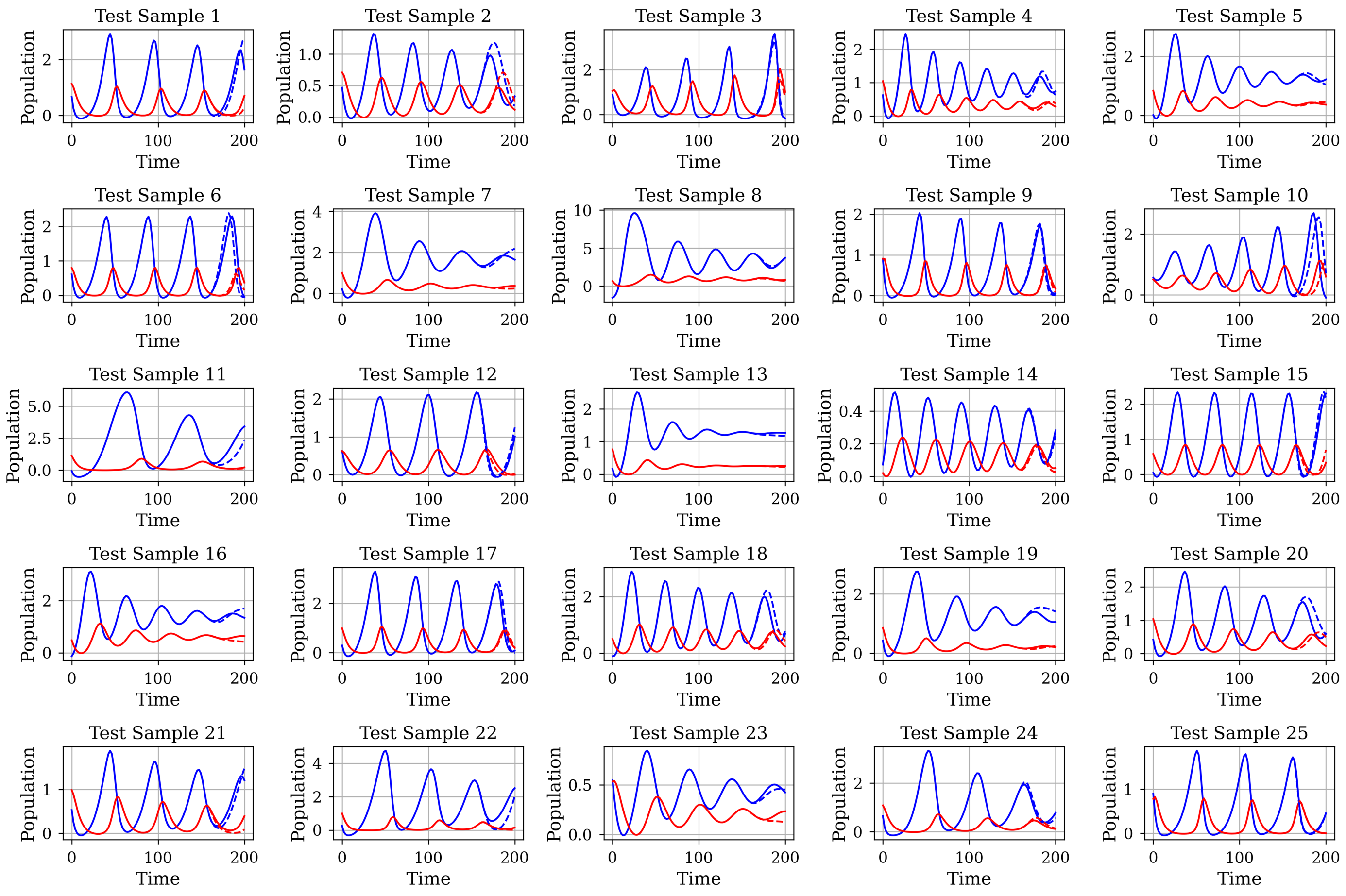

Figure 12: More example predictions