Deep Learning Coursework Assignment

Low-Rank Adaptation of Large Language Models for Time Series Forecasting

Laura Just Fung (lj441)

April 7, 2025

Word count: 2968

1 Introduction

As shown by Gruver et al., by encoding time series data as a string of digits, time series forecasting can be directly translated to next-token prediction in text. Thus, large language models (LLMs) can be zero-shot repurposed to this task fairly naturally, helping tackle this challenging machine learning problem. In this report, the Qwen2.5-0.5B-Instruct model [4] along with Low-Rank Adaptation (LoRA) of the q and v projection matrices are used to explore this concept further and demonstrate performance of a LLM that has been fine-tuned towards this task.

2 Compute constraints

A budget of 10^{17} floating point operations (FLOPS) was applied to this coursework. The FLOPS accounting for primitives are defined in Table 1.

Operations	FLOPS
Addition/Subtraction/Negation	1
Multiplication/Division/Inverse	1
ReLU/Absolute value	1
Exponentiation/Logarithm	10
Sine/Cosine/Square root	10

Table 1: Standardised FLOPS for common primitive operations as defined for the rest of this report.

2.1 FLOPS calculations

While Qwen2.5-0.5B-Instruct uses Grouped Query Attention, the FLOPS for standard multi-headed attention have been used to simplify the calculations. For similar reasons, it has been assumed that the FLOPS of backpropagation are exactly twice that of the forward pass. For all reported FLOPS, any operations performed outside of the model are not included. Finally, the cost of computing the rotary positional encodings (RoPE) have been ignored.

2.1.1 RoPE

As previously stated in Section 2.1, the cost of calculating RoPE is not included but the cost of adding them to the input token embedded matrix $T_{D\times N}$ is. $D = 896$ is the dimension of embedding and $V = 151,936$ is the vocabulary size. The cost of adding the RoPE is thus DN FLOPS.

2.1.2 Key, query, and value projections

As the model uses multi-headed attention, the keys, queries, and values matrices of dimension $\frac{D}{H} \times N$ are calculated by combining the biases $\beta_{\frac{D}{H}\times 1}$ with the projection matrices $\Omega_{\frac{D}{H}\times\frac{D}{H}}$:

$$V_h = \beta_{vh}1^T + \Omega_{vh}X \tag{1}$$

$$Q_h = \beta_{qh}1^T + \Omega_{qh}X \tag{2}$$

$$K_h = \beta_{kh}1^T + \Omega_{kh}X \tag{3}$$

Calculating for one head, $3\frac{D}{H}N(2\frac{D}{H}+1)$ FLOPS are required. This then results in a total of $3H\frac{D}{H}N(2\frac{D}{H}+1)$ FLOPS.

2.1.3 RMS Norm

The model uses RMS Norm layers to re-centre and re-scale the embeddings, applying it column-wise. It is defined as follows in Eq. 4 [5]:

$$x'_i = \frac{x_i}{\sqrt{\varepsilon + \frac{1}{D}\sum_{j=1}^{D} x_j^2}}\gamma_i, \tag{4}$$

where x_i is the i^{th} column of the matrix, ε is a small constant to prevent division by zero, and γ_i is the i^{th} column of the scaling matrix. The calculation of the RMS requires $2DN + 10D$ FLOPS for the whole matrix, including squaring each element, summing them, multiplication by $\frac{1}{D}$, and finally the square root operation. Dividing the matrix by this as well as scaling and shifting by γ_i and ε respectively costs an additional $3DN$ FLOPS, summing to cost $D(5N+10)$ FLOPS in total.

2.1.4 Self-attention

The self-attention for each head is calculated as in Eq. 5:

$$\text{Sa}_h[X] = V_h \cdot \text{Softmax}\Big[\frac{K_h^T Q_h}{\sqrt{D/H}}\Big], \tag{5}$$

where softmax acts on the matrix columns:

$$\text{Softmax}\left[\begin{pmatrix} z1 \\ \vdots \\ zN \end{pmatrix}\right] = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}}. \tag{6}$$

Assuming that $\sum_{j=1}^{N} e^{z_j}$ is calculated once per column, the cost of calculating softmax on an input matrix $X_{N\times N}$ is $N(12N-1)$ FLOPS, as it requires NN exponentiation operations, $N(N-1)$ summation operations, and NN division operations. The self-attention FLOPS per head is then $\frac{D}{H}N(N-1)+N(12N-1)$.

2.1.5 Multi-head self-attention output

The outputs from each head are concatenated and a final linear transformation is applied as seen in Eq. 7:

$$\mathrm{MhSa}(X) = \Omega_c[\mathrm{Sa}_1[X]^T,\ldots,\mathrm{Sa}_H[X]^T]^T, \tag{7}$$

requiring $DN(D-1)$ FLOPS.

2.1.6 SwiGLU

The model also uses a SwiGLU activation layer, which consists of two matrix projections and a SiLU activation [3]. With an input matrix $X_{D\times N}$, there are two independent projections: the gate projection matrix $G_{M\times D}$ and the up projection matrix $U_{M\times D}$, where $M = 4864$ and is the intermmediate dimension of the model.

$$G = W_g X \tag{8}$$

$$U = W_u X \tag{9}$$

The projections cost $2ND(2N-1)$ FLOPS.

The SiLU activation is then applied to G elementwise, as defined in Eq. 10:

$$\mathrm{SiLU}(x_i) = x_i \cdot \sigma(x_i) = \frac{x_i}{1+e^{-x_i}}, \tag{10}$$

where x_i is the i^{th} element of the input vector. The cost of calculating the SiLU activation function is $13NM$ FLOPS, as it requires NM multiplication operations, N exponentiation operations, and $2N$ addition operations.

Then, element-wise multiplication of $L = \mathrm{SiLU}(G)\odot U$ as well as a down projection $O_{D\times N}$:

$$O = W_d A \tag{11}$$

where W_d is the down projection matrix. The down projection costs $2ND(2N-1)$ FLOPS, and the element-wise multiplication costs $2NM$ FLOPS.

In total, SwiGLU costs $2ND(2M+2N-2)+15NM$ FLOPS.

2.1.7 LoRA linear layer

When applying LoRA, the forward pass of the LoRALinear layer combines the original layer's output with a scaled low-rank output produced by the A and B matrices, as shown in Eq. 12:

$$\text{output} = \text{original}(x) + \frac{\alpha}{r} \cdot (x \cdot A^T \cdot B^T), \tag{12}$$

where r is the rank of the LoRA matrices, and A and B are the low-rank matrices added to the query and value projections, respectively. α is a scaling factor that controls the magnitude of the LoRA output's contribution to the update and is set to be equal to r.

Thus, the cost of computing and adding the output of the LoRA linear layer to the original matrix is $2(\frac{H}{D})^2 r$ per head.

3 Preprocessing

A text-based numeric encoding method adapted from the time series data preprocesing scheme described by Gruver et al. was implemented for all data given to Qwen2.5-0.5B-Instruct. This preprocessing ensures that the numeric sequences are appropriately formatted for the best performance of the model.

First, as the numeric values in the dataset may vary, to standardise the numeric range of the data and control the token length, a simple scaling was applied, as shown in Eq. 13

$$x'_t = \frac{x_t}{\alpha}. \tag{13}$$

The scaled numeric values were rounded to a fixed number of decimal places to ensure uniformity when tokenized.

The data used for this coursework involves multivariate Lotka-Volterra time series data. Thus, the following encoding is used, as shown in Eq. 14:

$$\begin{pmatrix} P_0 & Q_0 \\ P_1 & Q_1 \\ \vdots & \vdots \\ P_t & Q_t \end{pmatrix} \rightarrow p_0, q_0; p_1, q_1; \ldots; p_t, q_t, \tag{14}$$

where P_t refers to the original prey value at time t and Q_t refers to the original predator value at time t. The lowercase p_t and q_t refer to the scaled and rounded values of the prey and predator series. The comma is used to separate the predator and prey values, while the semicolon is used to separate the time steps.

The code for this preprocessing is shown in Listing 1.

Throughout, $\alpha = 5$ and the values were rounded to 3 decimal places. Two examples are shown below in Tables 2 and 3.

Stage	Value
Raw data	$\begin{pmatrix} 1.1335121 & 1.1031258 \\ 0.55542254 & 1.2579137 \end{pmatrix}$
Scaled data	$\begin{pmatrix} 2.267 & 2.206 \\ 1.111 & 2.516 \end{pmatrix}$
Encoded data	2.267,2.206;1.111,2.516
Tokenized data	[17,13,17,21,22,11,17,13,17,15,21,26,16,13,16,16,16,11,17,13,20,16]

Table 2: Example of the preprocessing method for two steps in the time series data. The raw data is scaled, encoded, and then tokenized.

Stage	Value
Raw data	$\begin{pmatrix} 0.8521567 & 0.9479834 \\ 0.6482769 & 0.94272685 \end{pmatrix}$
Scaled data	$\begin{pmatrix} 2.267 & 2.206 \\ 1.111 & 2.516 \end{pmatrix}$
Encoded data	1.704,1.896;1.297,1.885
Tokenized data	[16,13,22,15,19,11,16,13,23,24,21,26,16,13,17,24,22,11,16,13,23,23,20]

Table 3: Another example of the preprocessing method for two steps in the time series data.

```python
def scale_and_encode(prey, predator, alpha, decimals):
    """
    Scale and encode the prey and predator data.

    Args:
        prey (np.ndarray): The prey values.
        predator (np.ndarray): The predator values.
        alpha (float): Scaling factor.
        decimals (int): Number of decimal places for scaling.
    Returns:
        encoded (str): The encoded data string.
    """
    prey = np.array(prey)
    predator = np.array(predator)
    data = np.stack([prey, predator], axis=-1)
    rescaled = data/alpha * 10
    rescaled = np.round(rescaled, decimals=decimals)
    series = np.column_stack((rescaled[:, 0], rescaled[:, 1]))
    encoded = ';'.join([','.join(map(str, row)) for row in series])
    return encoded
```

Listing 1: The function takes in the prey and predator values, the scaling factor α, and the number of decimal places to round to. It returns the encoded string representation of the time series data.

4 Baseline

Using the preprocessing method described in Section 3, the untrained model was evaluated on the tokenized Lotka-Volterra time series data. The following plots and metrics calculations are done using the scaled and rounded values of the time series data to compare to the predictions. This is the format that the model is trained upon and rescaling the data back to the original values would not be useful for evaluating the model's performance as it would introduce a source of error that the model cannot account for.

To evaluate the performance of the untrained model, stratified sampling was used to select four systems from the given dataset. Broadly speaking, there are four types of system behaviour present in the dataset: oscillatory, decaying, growing, and stable.

The model was given the first 80 points in the time-series data and asked to predict the next 20 points, as shown in Fig. 1. The model was evaluated using mean absolute error (MAE), the mean absolute percentage error (MAPE), the mean squared error (MSE), the root mean squared error (RMSE), and running mean squared error (RMSE) for both time series. The metrics for an example of each of these types are shown in Table 4 and in Fig. 2.

	506		20		654		906	
Metric	Prey	Predator	Prey	Predator	Prey	Predator	Prey	Predator
MSE	0.217	0.341	12.207	10.331	0.001	0.000	0.023	0.003
RMSE	0.465	0.583	3.494	3.214	0.030	0.001	0.151	0.050
MAE	0.372	0.471	2.385	2.312	0.028	0.001	0.103	0.032
MAPE	0.301	0.365	1.332	0.760	0.004	0.001	0.031	0.056

Table 4: Metrics for the untrained model performance on a subset of the test set.

As can be seen in Fig. 1, the untrained model is fairly capable at predicting the oscillatory and stable systems 906 and 654 respectively, but is unable to predict decaying and growing systems 506 and 20 very well. This is reflected in the metrics shown in Table 4, where the error metrics for systems 654 and 906 are all very small, but the metrics for systems 506 and 20 are significantly greater. The model is especially terrible at predicting system 20, the growing system.

What also seems important is where the model begins to forecast. This is demonstrated especially by system 654, which settled into a fairly stable and predictable state before the cut-off point, and so the model is able to predict the next 20 points very well. This is also shown by sytem 506, where the model was able to predict the rough shape of the curve, but not the exact values for the prey time series. The cut-off point occurred just after the peak of the prey time series but before the peak of the predator time series.

Without fine-tuning, it is clear that the model is unable to learn the more subtle underlying patterns in the time series data. This is probably becaues it's not been previously trained on time series forecasting tasks.

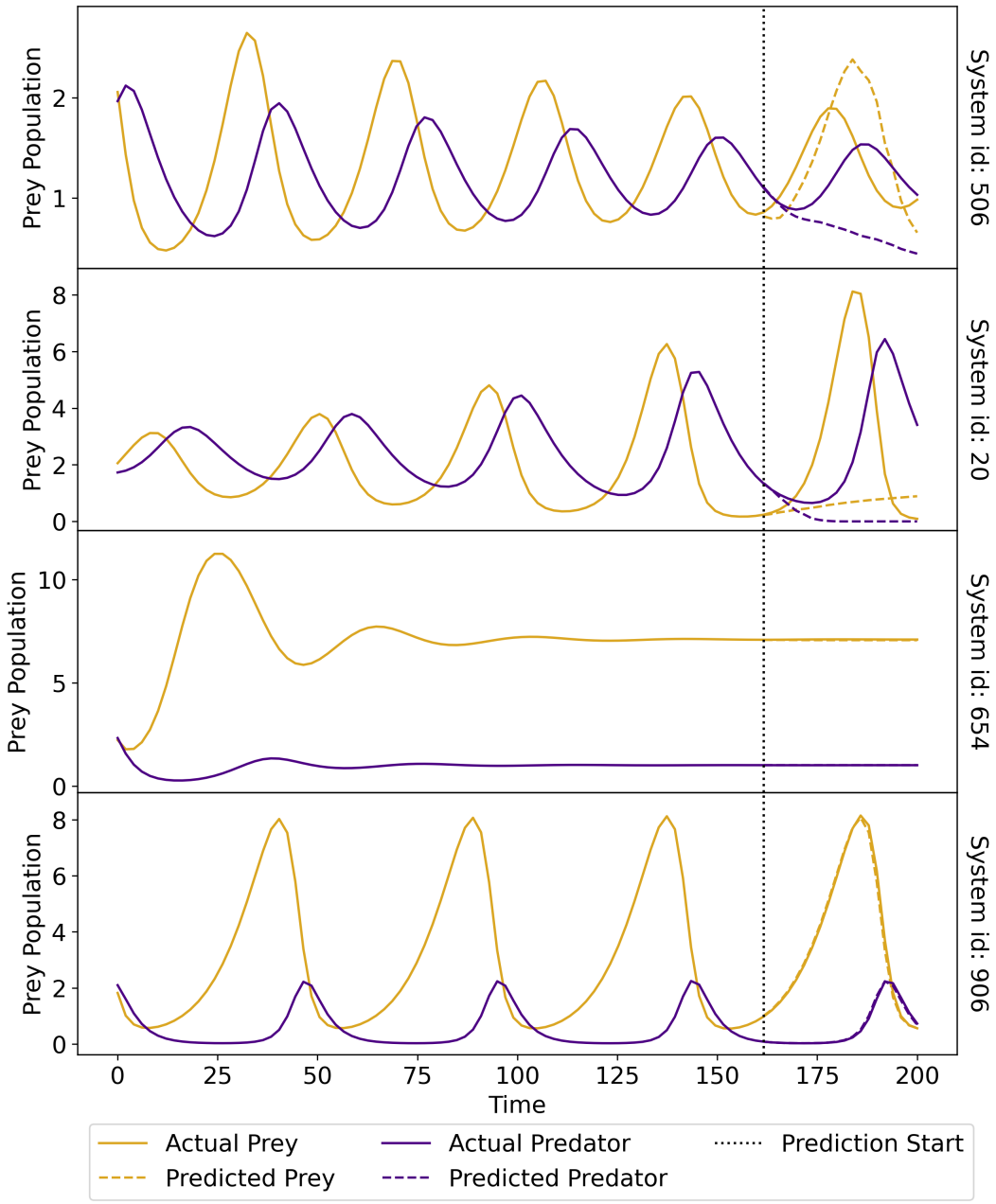

Figure 1: Predictions (dashed) of the untrained model compared with the true (solid) values on a subset of the dataset for both predator (purple) and prey (gold).

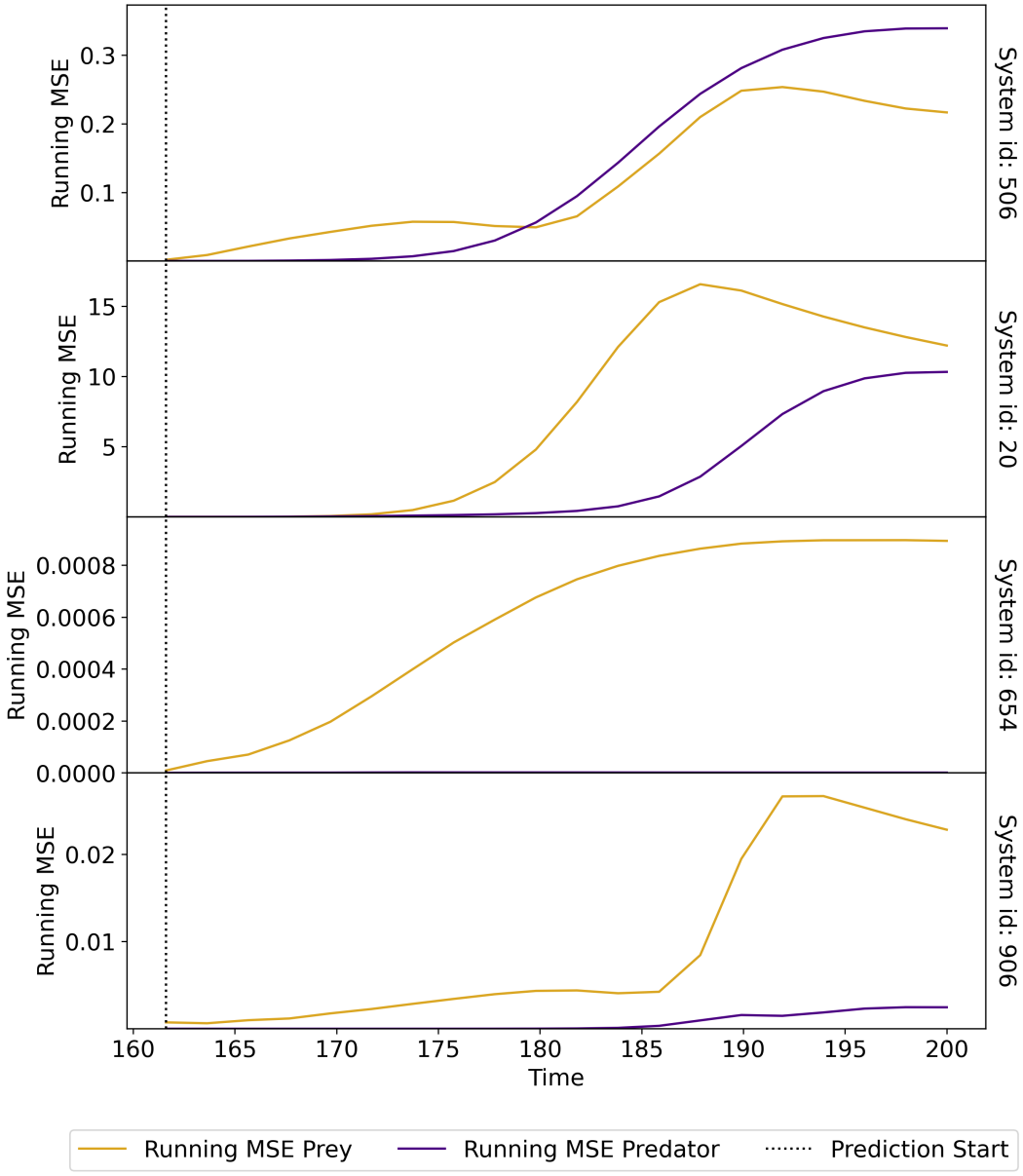

Figure 2: Running mean squared error (RMSE) for the model on a subset of the test set over the out-of-distribution data for the predator (purple) and prey (gold) time series.

5 LoRA

The model was fine-tuned using the LoRA [2] implementation to wrap the query and value projection layers with LoRALinear layers. This adds low-rank matrices to the model's weights for the query and value matrices, which allows for efficient training and adaptation to new tasks. Then, when training the model, only the LoRA weights are updated whilst the rest of the model's weights are frozen. This allows for efficient training and adaptation to new tasks, as this approach allows for the model to be fine-tuned on a specific task without the need for retraining the entire model, which can be computationally expensive and time-consuming.

The specific parameter blocks that are being tuned when using LoRA are the A and B matrices added to the query and value projections in each transformer layer's self-attention block.

5.1 Training with LoRA

To demonstrate the baseline improvements when adding LoRA, the model was fine-tuned using the default hyperparameters shown in Table 5 for 3,000 time steps. The model was evaluated on the validation set every 50 steps. The training and validation loss are shown in Fig. 3, and the model's performance on stratified examples from the test set is shown in Fig. 4. The metrics for the model's performance on the test set are shown in Table 6.

Hyperparameter	Value
Learning rate	10^{-5}
Batch size	4
LoRA rank	4
LoRA scaling factor α	4
Maximum context length	512

Table 5: Default hyperparameters used for training the model with LoRA on the dataset.

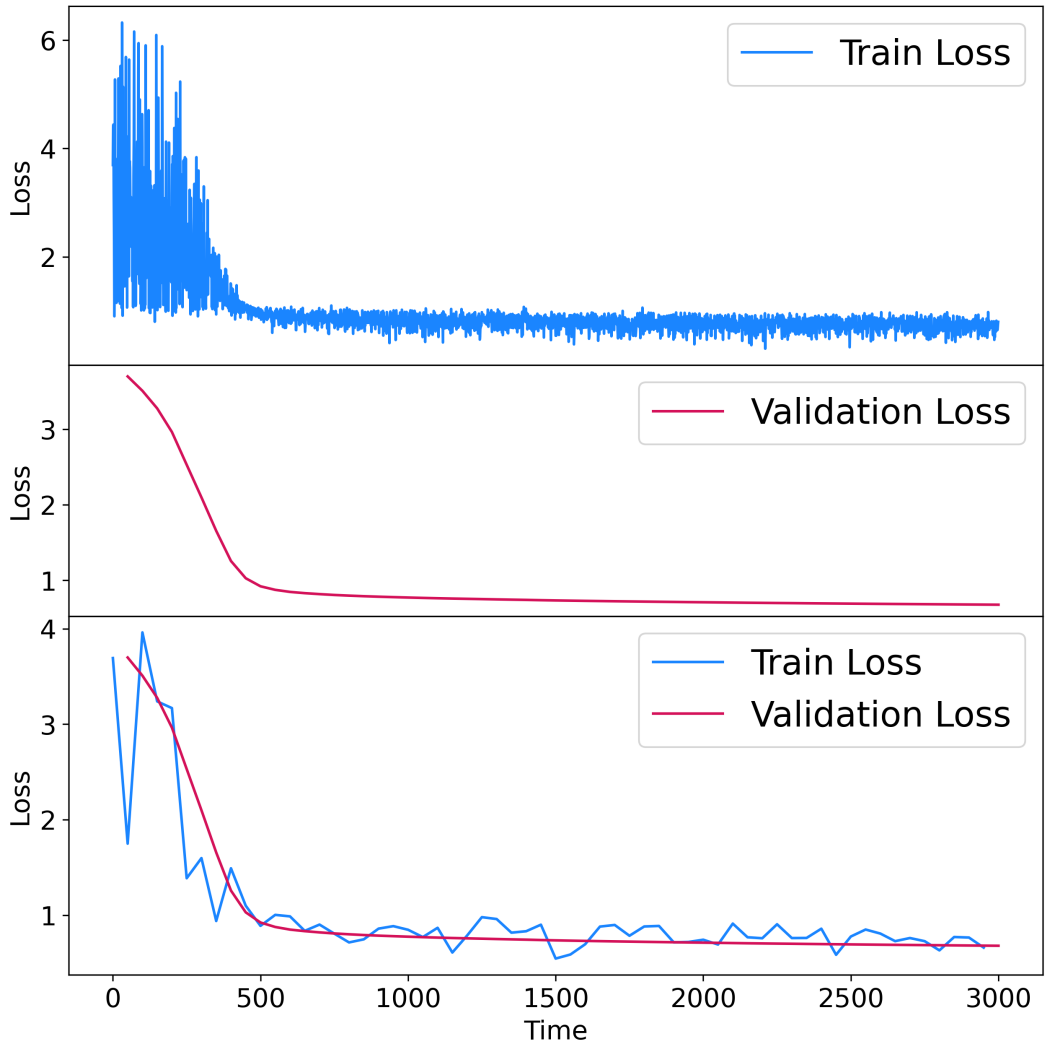

Figure 3: Training and validation loss for the model fine-tuned with LoRA on the dataset using default hyperparameters. The bottom figure compares the training (blue) and validation (red) loss using the same resolution of every 50 steps.

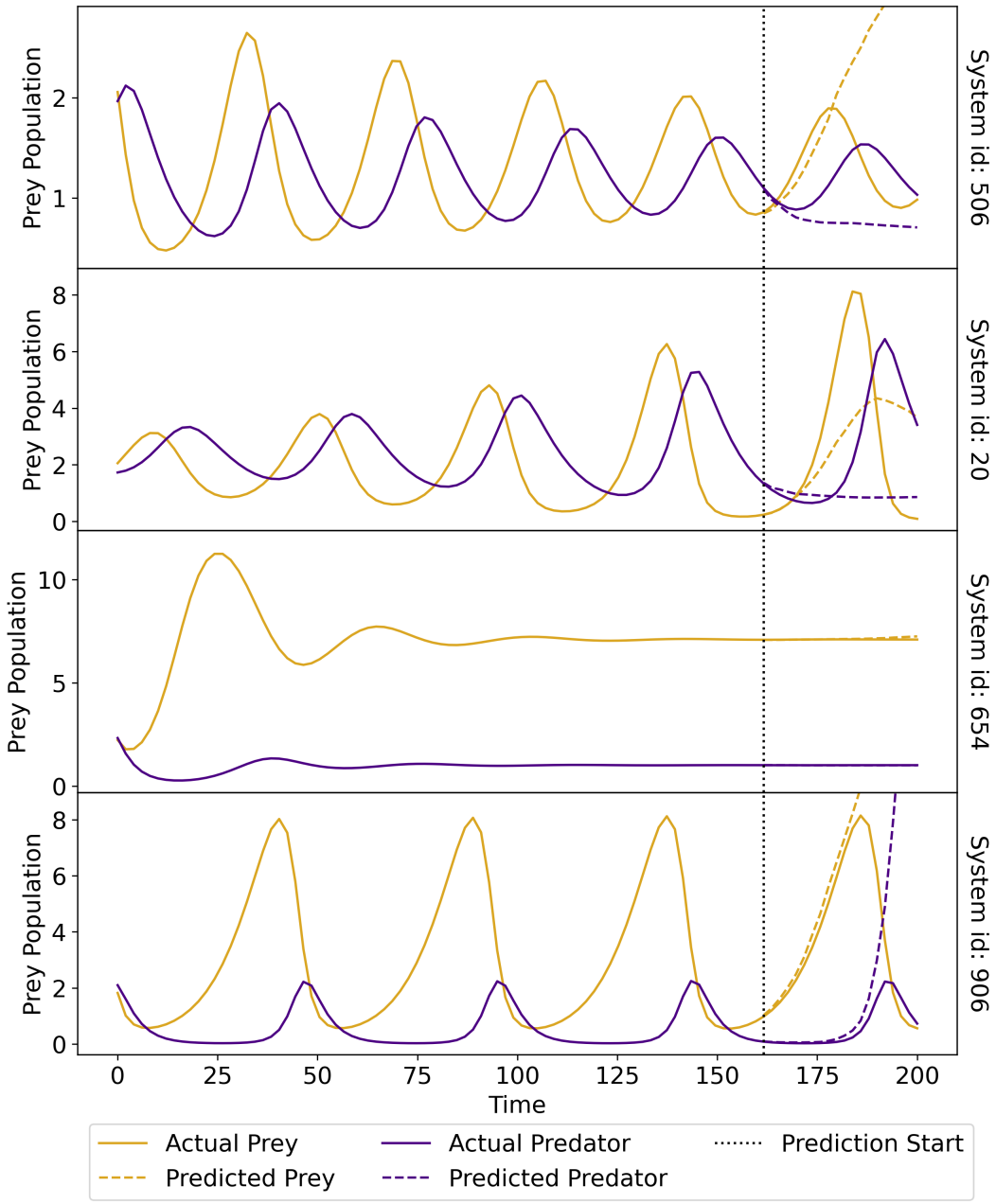

Figure 4: Predictions (dashed) of the model fine-tuned with LoRA using default hyperparameters compared with the true (solid) values on a subset of the test set for both predator (purple) and prey (gold).

	506		20		654		906	
Metric	Prey	Predator	Prey	Predator	Prey	Predator	Prey	Predator
MSE	1.561	0.227	6.618	6.994	0.004	0.000	37.543	24.459
RMSE	1.249	0.477	2.573	2.645	0.061	0.005	6.127	4.946
MAE	0.880	0.388	1.962	1.775	0.038	0.005	3.545	2.393
MAPE	0.837	0.300	4.530	0.493	0.005	0.005	3.397	2.592

Table 6: Metrics for the model fine-tuned with LoRA using default hyperparameters on a subset of the test set.

Hyperparameter	Values
Learning rate	10^{-5}, 5×10^{-5}, 10^{-5}
LoRA rank	2, 4, 8
Maximum context length	128, 512, 768

Table 7: Hyperparameters tuned for the model fine-tuned with LoRA on the dataset.

As can be seen in Fig. 3, there is a gradual plateau in the training and validation loss starting around 1,000 steps, indicating that the model is converging to a local minimum. The validation loss is comparable to the training loss but generally higher, which indicates that the model is not overfitting to the training data.

As shown by Fig. 4, the model performs somewhat better on predicting the growing and decaying systems, as it seems to understand that there is an underlying pattern of growing or decaying oscillations. However, it now struggles with the oscillatory system 906, which now has fairly large errors in both the prey and predator time series predictions. This is very different to before, where it was able to predict it almost perfectly, as seen in Figures 1 and 2. This is likely because very few oscillatory systems are present in the entire Lotka-Volterra dataset, with most of the systems being some form of decaying or growing.

The gradients of the A and B matrices were tracked throughout the training process and it was found that the gradients of the A and B matrices neither exploded nor vanished. Thus, gradient clipping was not applied to the model. As the figure is quite large, comprising of 48 plots, it is not shown in this report and is instead included as supplementary material in the `addendum` folder.

5.2 Hyperparameter tuning

To further improve the performance of the model fine-tuned with LoRA, hyperparameter tuning was performed. The hyperparameters that were tuned are shown in Table 7. They were tuned using a grid search approach, and the best combination of hyperparameters was then selected based on the final validation loss.

Learning rate	LoRA Rank		
	2	4	8
1×10^{-5}	2.932	2.154	1.046
5×10^{-5}	0.770	0.736	0.706
1×10^{-4}	0.713	0.686	0.661

Table 8: Final validation loss for the model fine-tuned with LoRA on the dataset using different combinations of η and r.

5.2.1 Learning rate and LoRA rank

The learning rate η and LoRA rank r were tuned first, with the maximum context length set to 512. The model was trained for 300 steps with each combination of hyperparameters. The training and validation loss are shown in Fig. 5.

As can be seen in Fig. 5, η is has a significant impact on the model's training, with the largest learning rate $\eta = 0.0001$ causing significant decreases in both training and validation loss in in the first 100 steps regardless of r. This is likely because with a larger η, the model is able to make large initial updates to the weights, allowing it to learn quickly from the data. However, this can also lead to instability in the training process, as the model may overshoot. Fig. 5 also shows that r also has a significant impact. As the LoRA weights are the only weights being updated, the larger the contribution the LoRA matrices are allowed to make to the model's output, the better the model is able to learn to learn from the data.

The final validation loss for each of the combinations of hyperparameters is shown in Table 8. From this it can be seen that the best combination of hyperparameters is be $\eta = 10^{-4}$ and $r = 8$.

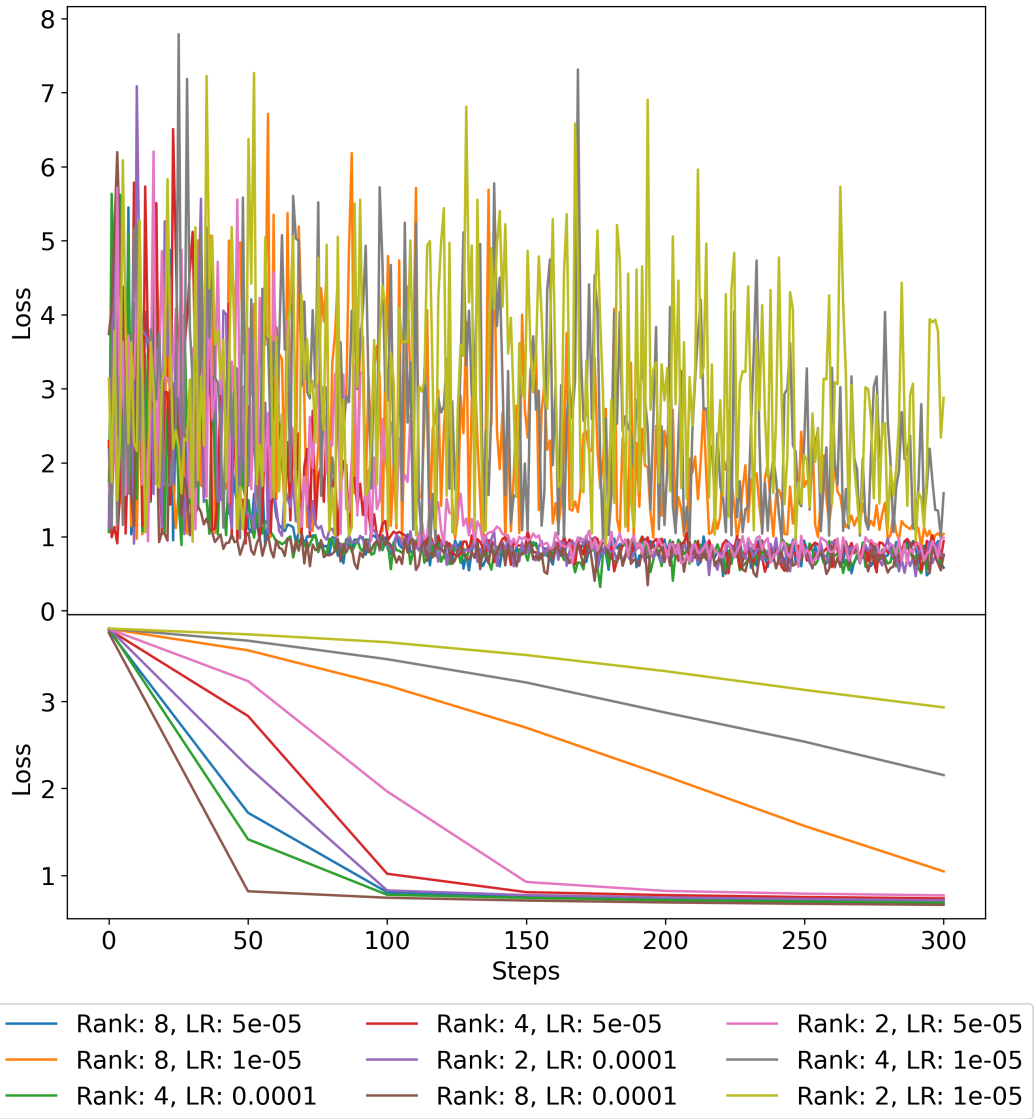

Figure 5: Training (top) and validation (bottom) loss for the model fine-tuned with LoRA on the dataset using different η and r.

5.2.2 Maximum context length

The maximum context length was then tuned, setting $\eta = 10^{-4}$ and $r = 8$. The model was trained for 300 steps with each context length. The training and validation loss are shown in Fig. 6.

As can be seen in Fig. 6, the context length has an impact on the model's performance up to a point. With a too-small context length, the model is unable to effectively learn from the data, as it is only able to see a small portion of the time series data at any one time. However, the validation loss of the model using a length of 512 is very similar to that of a length of 768, indicating that the model is able to learn from the data effectively with either maximum context length. This is also shown in Table 9: the longest maximum context length of 768 results in the lowest final validation loss, but not significantly lower than using 512. The training loss shown in Fig. 6 also shows this.

The experiments were only run for a maximum of 300 steps as the model's training and validation loss tends to plateau at around 50–100 steps.

	Maximum Context Length		
	128	512	768
Validation loss	0.895	0.664	0.661

Table 9: Final validation loss for the model fine-tuned with LoRA on the dataset using different maximum context lengths with $\eta = 0.0001$ and $r = 8$.

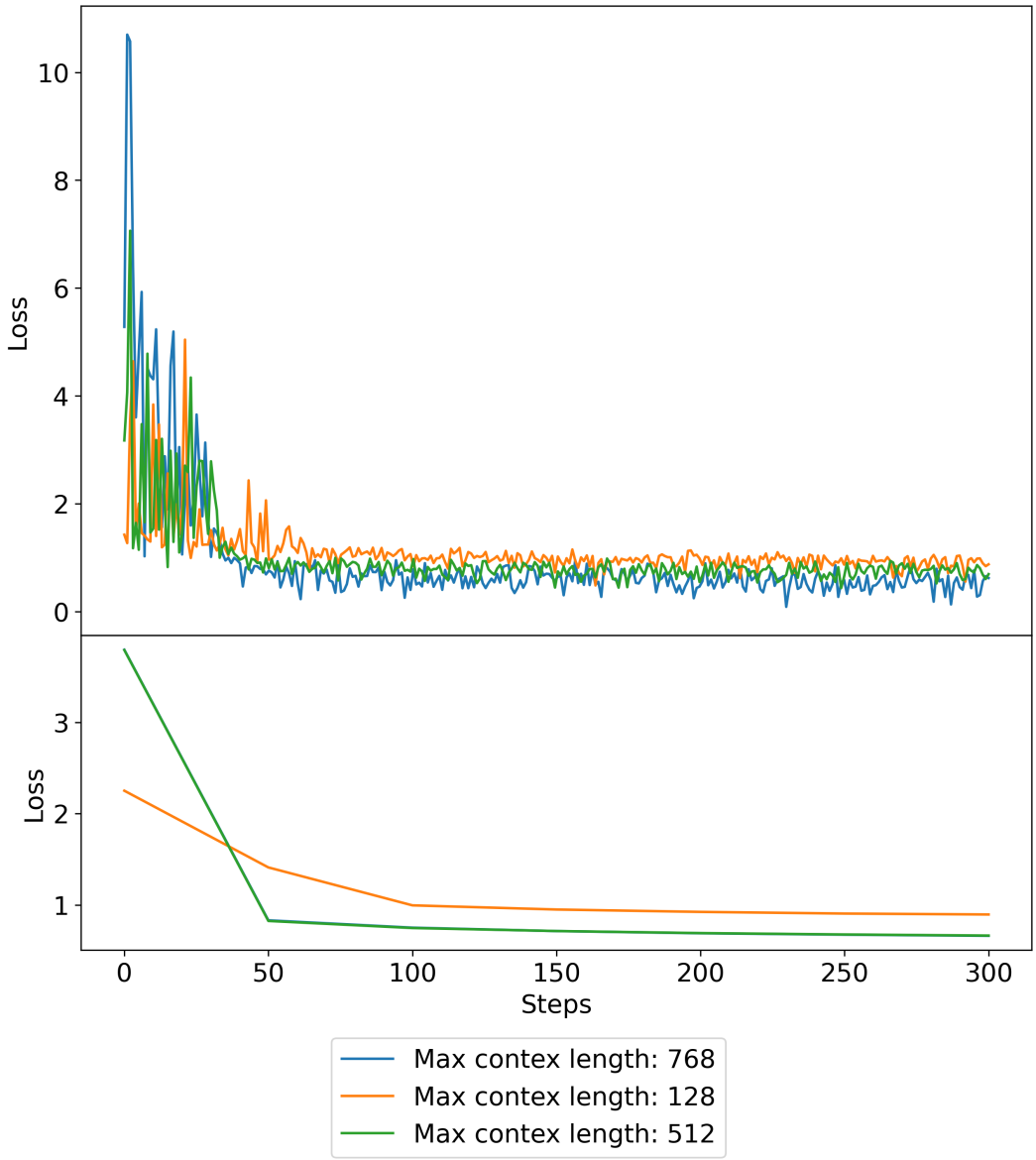

Figure 6: Training (top) and validation (bottom) loss for the model fine-tuned with LoRA on the dataset using different maximum context lengths with $\eta = 0.0001$ and $r = 8$.

5.3 Final model

Using the final hyperparameters, shown in Table 10, the model was fine-tuned on the dataset for 15,000 steps. The training and validation loss are shown in Fig. 7, and the model's performance on stratified examples from the test set is shown in Fig. 8. The metrics for the model's performance on a selected stratified subset of test set are shown in Table 11. The running MSE for the subset is shown in Fig. 9.

Hyperparameter	Value
Learning rate	10^{-4}
LoRA rank	8
Maximum context length	768
Batch size	4
LoRA scaling factor α	8

Table 10: Final best hyperparameters for the model fine-tuned with LoRA on the dataset.

As can be seen in Fig. 7, the validation loss has not yet plateaued by 15,000 steps, indicating that the model was still learning from the data when training was stopped. However, with deference to the computational constraint of this coursework, the model was not trained for longer.

Comparing Figures 1 and 8, the model's performance on the test set is better than that of the untrained model or the LoRA adapted model trained with the default hyperparameters. This is especially shown by the growing and decaying systems 506 and 20, where the model is now able to predict at least the rough shape of the curve, though not the exact values. The model is also able to predict the overall shape of the oscillatory system 906 fairly well, though not as well as the untrained model. And as the stable system 654 has settled into constant values before the cut-point, the model is able to predict the next 20 points with very small error.

	506		20		654		906	
Metric	Prey	Predator	Prey	Predator	Prey	Predator	Prey	Predator
MSE	0.004	0.001	2.013	2.121	0.000	0.000	6.177	0.739
RMSE	0.064	0.024	1.419	1.456	0.012	0.001	2.485	0.859
MAE	0.053	0.018	0.779	0.881	0.012	0.001	1.488	0.602
MAPE	0.041	0.015	0.232	0.231	0.002	0.001	0.325	1.887

Table 11: Metrics for the model fine-tuned with LoRA using the final best hyperparameters on a subset from the test set.

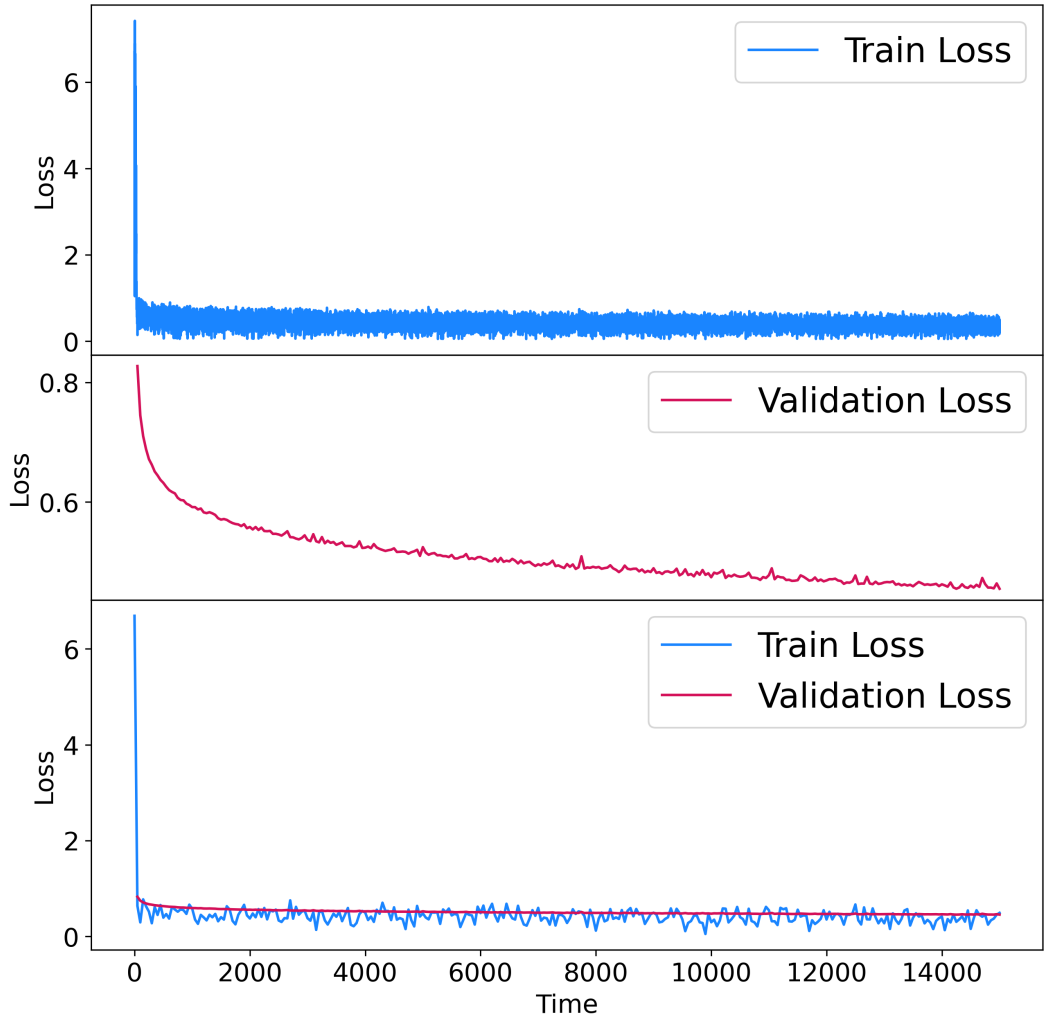

Figure 7: Training (top) and validation (middle) loss for the model fine-tuned with LoRA on the dataset using the final best hyperparameters. The bottom figure directly compares the training (blue) and validation (red) loss using the same resolution of every 50 steps.

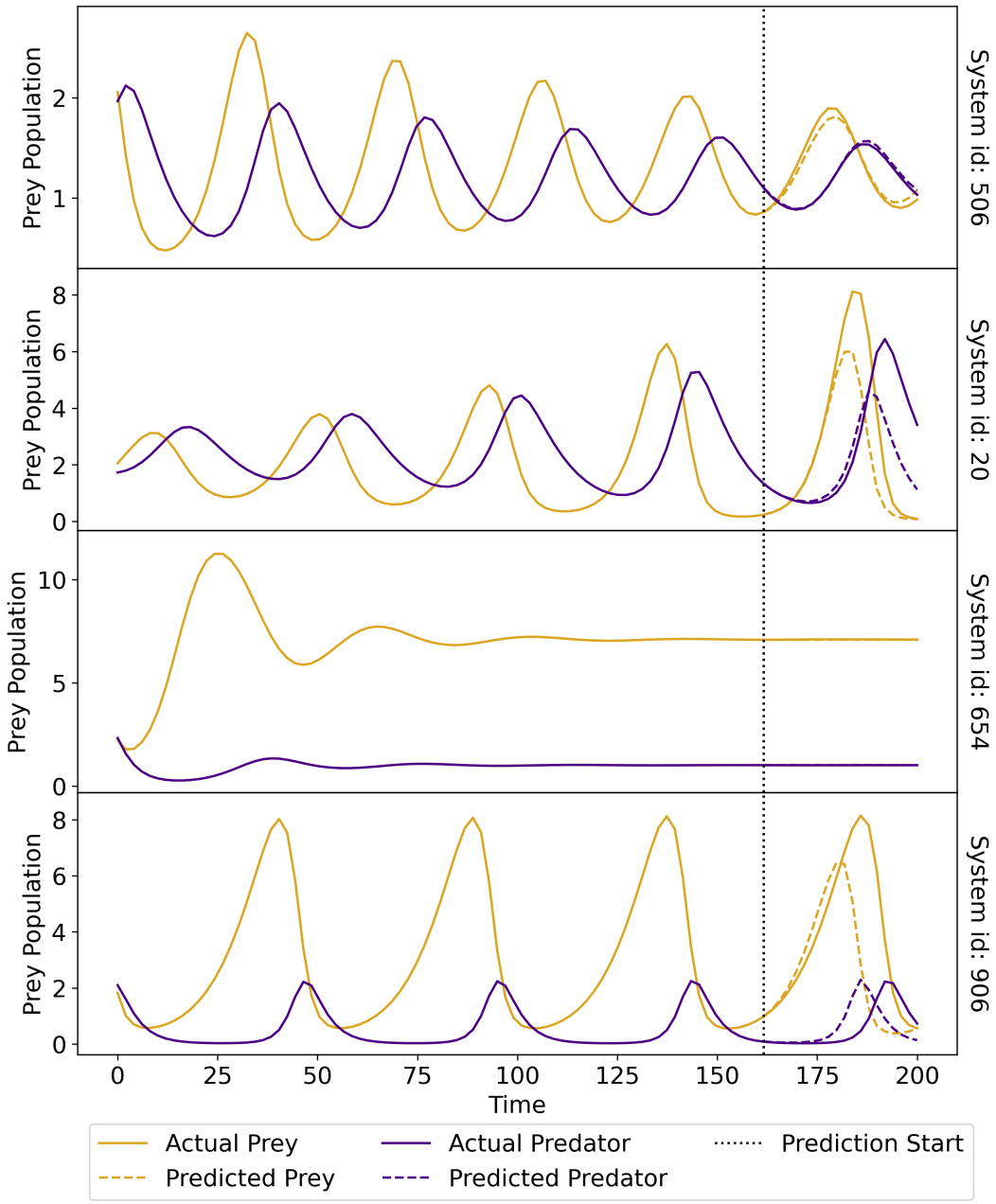

Figure 8: Predictions (dashed) of the model fine-tuned with LoRA using the final best hyperparameters compared with the true (solid) values on a subset of the test set for both predator (purple) and prey (gold).

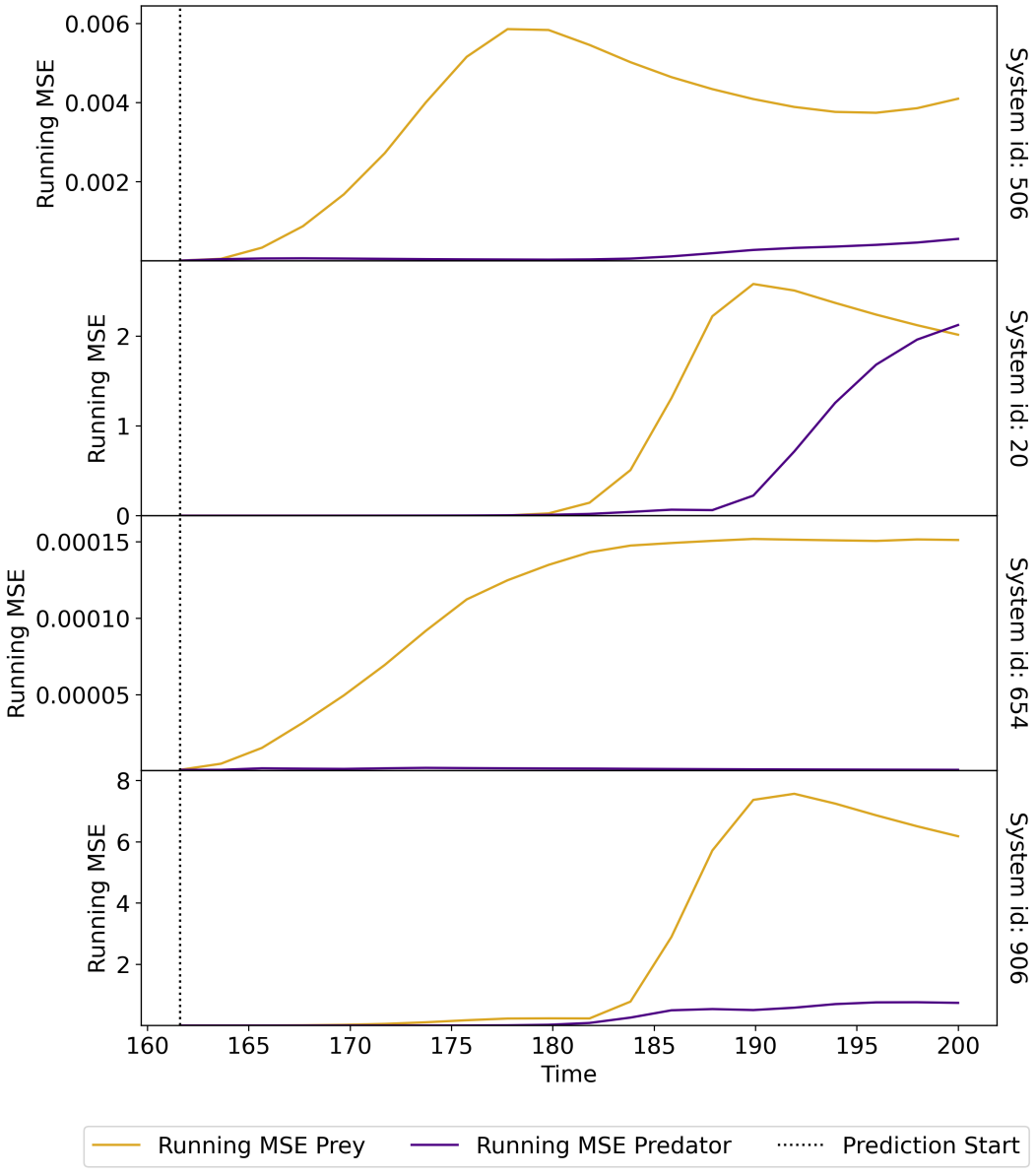

Figure 9: Running mean squared error (RMSE) for the model fine-tuned with LoRA using the final best hyperparameters on a subset of the test set over the out-of-distribution data for the predator (purple) and prey (gold) time series.

This is also reflected in the metrics shown in Table 11, where the error metrics for all the systems except system 906 have decreased significantly compared to the metrics from Tables 4 and 6. In contrast, the errors recorded for system 906 increased significantly once the model was fine-tuned with LoRA, as seen in Table 6. This was improved with hyperparameter tuning and a longer training time, as the model learns to predict Lotka-Volterra time series data.

6 FLOPS usage

The FLOPS used for every experiment are shown in Table 12. As can be seen, the coursework is well within the stated budget of 1×10^{17} FLOPS.

	FLOPS			
Stage	Training	Validation	Inference	Total
Untrained	–	–	5.416e+12	5.416e+12
Default hyperparameters	9.812e+15	1.308e+14	5.416e+12	9.948e+15
η & r tuning	8.831e+15	1.177e+14	–	8.948e+15
Context length tuning	2.723e+15	3.631e+13	–	2.759e+15
Final run	9.812e+15	1.308e+14	5.416e+12	9.948e+15
Total	9.659e+16	1.288e+15	1.625e+13	9.790e+16

Table 12: FLOPS usage for each stage of the coursework including inference by the untrained model; training and inference of the LoRA adapted model with default hyperparameters; grid search of η, LoRA rank, and context length; and the training and inference of the final model.

When fine-tuning models for time-series forecasting under tight budgets, it is recommended to limit the length of training steps. This is because as seen in Sections 5.2.1 and 5.2.2, it became clear fairly quickly which hyperparameters were best, sometimes being obvious at even just 50 steps. For this coursework, the hyperparameter search runs were done for 300 steps to make sure the inital best hyperparameter stayed as such.

For this coursework, a maximum context length of 512 could have been chosen over that of 768 if the FLOPS budget was becoming a serious concern. This is because as seen in Section 5.2.2, it performed fairly similarly whilst commanding a lower FLOPS cost.

Another recommendation is to limit the amount of times the validation loss calculation is performed whilst training. It not only takes up more of the FLOPS budget, but also increases the compute time per training run. During this coursework, the validation was computed every 50 steps. This saved FLOPS as well as compute, but also provided enough information to see how the validation loss changed over time.

7 Further improvements

As seen in Fig. 8 in Section 5.3, the final model still does not predict Lotka-Volterra time series data to a high degree of accuracy for all types of systems. This may be improved by simply increasing the compute by allowing it to train for longer, but there are other optimisations possible.

One way to do this is by considering α as a tunable hyperparameter. According to Hu et al., it is recommended that α should be around twice the LoRA rank. Throughout this coursework, α was fixed at the LoRA rank, limiting the impact the LoRA weights had on the model's output and how effectively the model learns from the training data, affecting its performance.

Another way to further improve the performance of the model is to include a scheduler. This is because of the problem mentioned in Section 5.2.1, where a large learning rate showed larger dividends compared to the smaller learning rates. However, with a larger learning rate, it might only show such benefits in only the initial few training steps of the model and in the later stages, a smaller learning rate might prove more beneficial. This is especially obvious in Figures 3 and 7, where the training and validation loss curves begin to plateau at around 1,000 steps. This is where a scheduler comes in, where η of the optimizer is allowed to change over the course of the training run. Some tuning would be required to find the optimal scheduler and parameters for that scheduler, again impacting the FLOPS budget, but that would further improve the performance of the model.

8 Summary

The Qwen2.5-0.5B-Instruct LLM was fine-tuned using LoRA matrices and the optimised hyperparameters η, r, and maximum context length. This improved the performance of the model on predicting Lotka-Volterra time series data especially from its default state, but there are several improvements yet to be made.

LLMs may be somewhat competent at time series forecasting, but for more complex systems, it takes a large training dataset and lots of fine-tuning to reach acceptable levels of accuracy.

References

[1] Gruver N, Finzi M, Qiu S, Wilson AG, Large Language Models Are Zero-Shot Time Series Forecasters; 2024. https://arxiv.org/abs/2310.07820.

[2] Hu EJ, Shen Y, Wallis P, Allen-Zhu Z, Li Y, Wang S, et al., LoRA: Low-Rank Adaptation of Large Language Models; 2021. https://arxiv.org/abs/2106.09685.

[3] Shazeer N, GLU Variants Improve Transformer; 2020. https://arxiv.org/abs/2002.05202.

[4] Yang A, Yang B, Hui B, Zheng B, Yu B, Zhou C, et al., Qwen2 Technical Report; 2024. https://arxiv.org/abs/2407.10671.

[5] Zhang B, Sennrich R, Root Mean Square Layer Normalization; 2019. https://arxiv.org/abs/1910.07467.

A Use of auto-generation tools

Auto-generation tools were used to help parse error messages throughout the project, and to help format this LaTeX report. Auto-generation tools were also used in helping to create a `move_to_cpu` function to move tensors on the GPU to the CPU. Auto-generation tools were also used to help with creating a `decoding` function that stripped out LLM injected errors in the model's output to transform it into time series arrays.

Auto-generation tools were not used elsewhere, for code generation, writing, or otherwise.