M2 Coursework Report

Zhimei Liu (zl474)

March 27, 2025

Word count: 2794

Declaration: I, Zhimei Liu, hereby declare that the work presented in this report and the GitLab repository is entirely my own. I have only used ChatGPT for some debugging to speed up the process.

Contents

0 Introduction

The goal of this coursework is to investiagte the ability of large language models (LLMs) to forecast the behaviour of complex dynamical systems, specifically a predator-prey system governed by the Lotka-Volterra equations:

$$\begin{aligned} \frac{dx}{dt} &= \alpha x - \beta xy \\ \frac{dy}{dt} &= -\gamma y + \delta xy \end{aligned} \tag{1}$$

where here x is the population density of prey and y is the population density of predator. Accurate forecasting of such systems is crucial for understanding

population dynamics, ecological stability and broader applications in mathematically biology.

Recent advancements in LLMs, such as Qwen2.5-Instruct, have demonstrated remarkable capability in handling structured data and generating complex sequences. This coursework explores whether such models can be adapted to forecast time-series data from a predator-prey system. However, fine-tuning large-scale models from scratch is computationally expensive and memory-intensive. To address this challenge, we employ **Low-Rank Adaptation (LoRA)**, a technique that reduces the number of trainable parameters by introducing low-rank matrices into specific projection layers. This allows for more efficient fine-tuning while preserving the generalization capacity of the pre-trained model.

The coursework is structured into three main tasks:

1. **Preprocessing and Tokenization:** We preprocess the time-series data using the LLMTIME scheme and tokenize the sequences using the Qwen2.5-Instruct tokenizer.
2. **LoRA-Based Fine-Tuning:** We adapt the query and value projection layers of the model using LoRA, ensuring that only the low-rank matrices are trainable while freezing the base model.
3. **Model Evaluation:** We evaluate the model's forecasting performance on the validation set using metrics such as Mean Squared Error (MSE) and Mean Absolute Error (MAE). The fine-tuned model's performance is compared with the untrained model to assess the impact of LoRA-based adaptation.

1 Baseline

In the notebook `baseline.ipynb`, we first load the dataset `data/lotka_volterra_data.h5` and view the first 5 systems using the function `plot_predator_prey`. Figure (1) shows the time-series of the first 5 preditor-prey systems.

1.1 LLMTIME Preprocessing Scheme

To use Qwen2.5-Instruct for forecasting numerical time series data, we implement a text-based numeric encoding method adapted from the **LLMTIME** preprocessing scheme [1]. This preprocessing ensures the numeric sequences are suitably formatted for Qwen's tokenizer and forecasting capabilities.

The LLMTIME scheme is a method for using LLMs for *zero-shot time series forecasting* by encoding numerical time series daya as text sequences. It enables

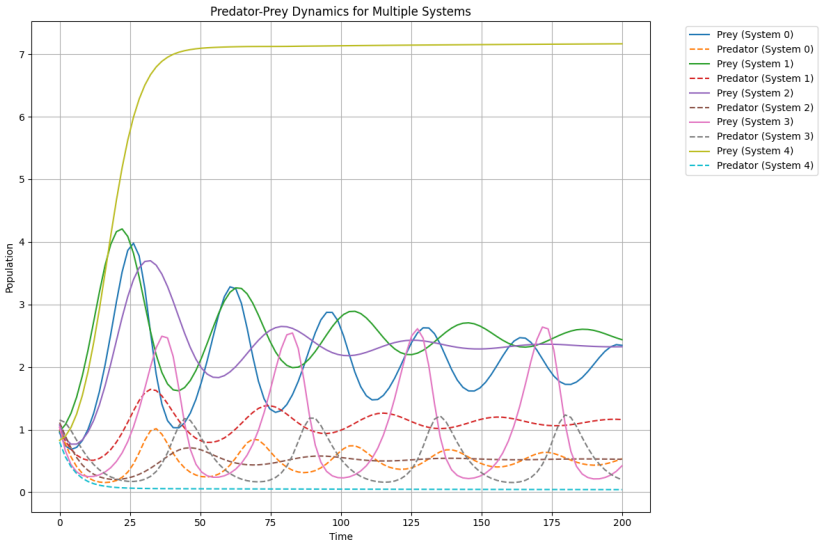

Figure 1: First 5 predator-prey systems in the dataset.

pretrained LLMs such as Qwen2.5 to predict future time series values by treating forecasting as a text generation problem.

The functions for implementing LLMTIME preprocessing scheme are saved in the file `src/preprocessor.py`.

The function `preprocess_time_series` performs two steps. It first rescales the input data from a particular system by choosing an α for each system such that most of the data are within the range $[0, 10]$:

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

Note that this function picks different α values for each system. Next, it rounds values to a fixed decimal precision, e.g. 3 decimal places by default in this function. But this number will become a hyperparameter to be tuned later.

The second function `preprocess_all_time_series` is used for LoRA in Section 2. It works the same as `preprocess_time_series` except that it processes all the systems and convert the data from all systems into text strings, and lastly performs a train-test split to divide the 1000 systems into a training set of 800 systems and a test set of 200 systems.

The last function `tokenize_sequence` tokenize the preprocessed sequence using Qwen's tokenizer.

Let's show some examples. Running the function `test`, an example of a preprocessed sequence can be:

```
1.900,2.081;1.481,1.559;1.364,1.129;1.433,0.815;1.649,0.601;2.009,
0.460;2.526,0.372;3.211,0.324;4.062,0.309;5.046,0.324;6.082,......
```

with the corresponding tokenized sequence using Qwen2.5:

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

where I have truncated the remaining sequences in .....

1.2 Evaluation of Untrained Model

The function `plot_prediction` in `baseline.ipynb` evaluates the untrained Qwen2.5-Instruct model on time-series data. The model hasn't been fine-tuned yet, so this establishes a baseline for performance.

The function `plot_prediction` works as follows. For the 100 total time steps of each of the 6 systems, it splits the first 80 time steps as the input, and then pass them to the trained model. Next, the model would generate the subsequent 20 time steps and compare these predicted values with the true values. For each of the time steps, it generates 10 time-series samples and then compute the mean and standard deviation. In Figure (2), each of the solid line represents the true population curve for predator and prey, while the dashed line means the mean of the predicted values. The shaded areas are the ± 1 standard deviation width. Since Qwen is an autoregressive model, we expect the width of the shaded area to increase as time, as the error will accumulate. The title for each subplot also shows the mean square error (MSE) for prediction. The MSE is calculated with the following formula (you can find the following code in the function `plot_prediction`)

```
prey_mse = np.mean((mean_pred[:, 0] - prey[80:]) ** 2)
predator_mse = np.mean((mean_pred[:, 1] - predator[80:]) ** 2)
total_mse = prey_mse + predator_mse
```

1.3 FLOPS

This subsection calculates the **FLOPS** (Floating Point Operations) used by the forward pass of the Qwen2.5-Instruct model. There are different operations to keep track of, including matrix multiplications, activations, and normalization.

A table of FLOPS per operation is given as in Table (1). The key FLOPS-heavy operations in a transformer model are:

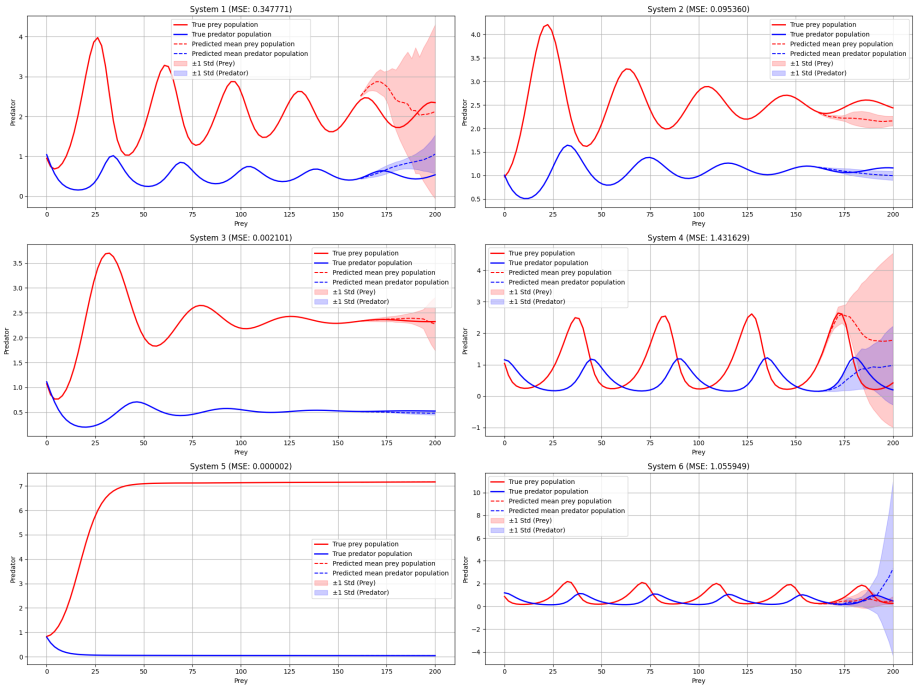

Figure 2: The prediction plots for the first 6 systems in the dataset, with MSE values showing next to the subtitles.

1. **Embedding Lookup:** FLOPS for embeddings are negligible because they are just indexing. Hence we ignore for FLOPS calculation.

2. **Multi-Head Attention (Grouped Query Attention):** Let's consider multi-head attention with the number of attention heads being H, hidden size (embedding) being D and sequence length being N.

   - **QKV projection:** We first divide the input into H different sets of values, keys and queries by multiplying corresponding projection matrix $\boldsymbol{\Omega}_{vh}$, $\boldsymbol{\Omega}_{qh}$, $\boldsymbol{\Omega}_{kh}$ [4]:

$$\begin{aligned} \boldsymbol{V}_h &= \boldsymbol{\beta}_{vh}\mathbf{1}^{\mathrm{T}} + \boldsymbol{\Omega}_{vh}\boldsymbol{X} \\ \boldsymbol{Q}_h &= \boldsymbol{\beta}_{qh}\mathbf{1}^{\mathrm{T}} + \boldsymbol{\Omega}_{qh}\boldsymbol{X} \\ \boldsymbol{K}_h &= \boldsymbol{\beta}_{kh}\mathbf{1}^{\mathrm{T}} + \boldsymbol{\Omega}_{kh}\boldsymbol{X} \end{aligned} \tag{2}$$

     As shown in Figure (3), the input matrix $\boldsymbol{X}$ has a shape of $D \times N$, while $\boldsymbol{Q}$, $\boldsymbol{K}$ and $\boldsymbol{V}$ all have a shape of $\frac{D}{H} \times N$. So the projection matrix $\boldsymbol{\Omega}$ has a shape of $\frac{D}{H} \times D$.

     Then a matrix multiplication such as $\boldsymbol{\Omega}\boldsymbol{X}$ gives a total FLOPS of $\frac{D}{H} \times N \times (2D-1)$, and the total FLOPS for QKV projection is (adding

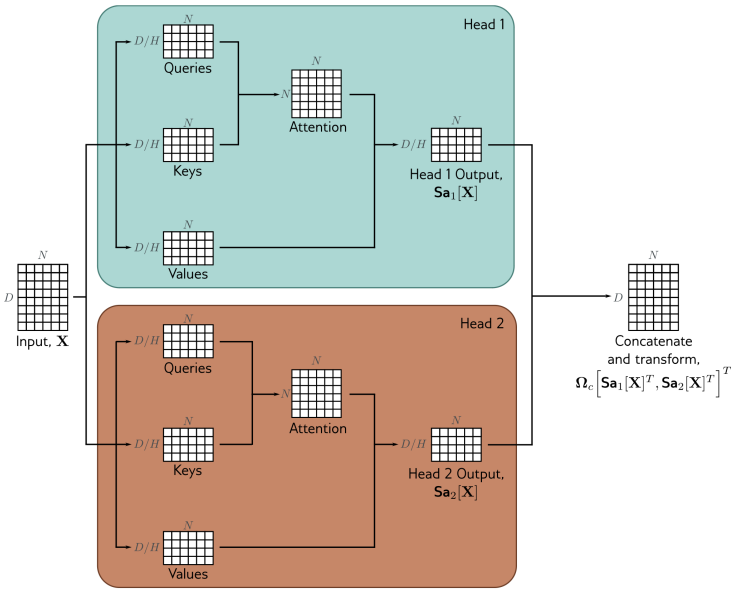

Figure 3: Multi-head self-attention.

Operation	FLOPS
Addition/Subtraction/Negation (float OR integer)	1
Multiplication/Division/Inverse (float OR integer)	1
ReLU/Absolute Value	1
Exponentiation/Logarithm	10
Sine/Cosine/Square Root	10

Table 1: FLOPS accounting per operation.

the contribution from biases):

$$3 \times H \times \big(\frac{D}{H} \times N \times (2D-1) + \frac{D}{H} \times N\big) = 6D^2N.$$

- **Attention weights calculation:** performing matrix product $\boldsymbol{K}_h^{\mathrm{T}}\boldsymbol{Q}_h$ gives a cost of (for a total of H heads)

$$H \times N \times N \times (2\frac{D}{H} - 1) = N^2(2D - H).$$

- **Softmax calculations:** Softmax involves exponentiation and division:

$$\text{Softmax}\Big[\frac{\boldsymbol{K}_h^T\boldsymbol{Q}_h}{\sqrt{D_q}}\Big]$$

so the total FLOPS is

$$H \times \big[11N^2 + N(10N + (N-1)) + \underbrace{N^2 + 10}_{\text{division by } \sqrt{D_q}}\big].$$

- **Attention output caluclation:** Multiplying attention weights with V gives a total FLOPS of

$$H \times N \times \frac{D}{H} \times (2N-1) = DN(2N-1).$$

- **Final projection:** Multiplying output of shape $N \times D$ by final projection matrix of shape $D \times D$ gives a total FLOPS of

$$N \times D \times (2D-1).$$

3. **Feedforward Network (MLP):** The feedforward network with a SwiGLU as an activation function is [6]:

$$\text{FFN}_{\text{SwiGLU}}(\boldsymbol{X}; \boldsymbol{W}_1, \boldsymbol{W}_2, \boldsymbol{W}_3, \boldsymbol{b}_1, \boldsymbol{b}_2, \boldsymbol{b}_3) = \boldsymbol{W}_3\big(\underbrace{\text{Swish}_1(\boldsymbol{W}_1\boldsymbol{X} + \boldsymbol{b}_1) \odot (\boldsymbol{W}_2\boldsymbol{X} + \boldsymbol{b}_2)}_{=\boldsymbol{X}'}\big) + \boldsymbol{b}_3$$

$$\text{Swish}_1(x) = x\sigma(x) = x \cdot \frac{1}{1+e^{-x}},$$

which combines the Swish and GELU, and $\odot$ denotes element-wise multiplication of two gates.

- **First linear layer:** Hidden size is usually 4 times the model size, so $\boldsymbol{W}_1$ has a shape of $D_{FFN} \times D$. Also, $\boldsymbol{X}$ has a shape of $D \times 1$, and we are applying N FFN for each token in parallel. So $\boldsymbol{W_1X}$ has a shape of $D_{FFN} \times 1$, and same for $\boldsymbol{b}_1$. Then the FLOPS is

$$\underbrace{D_{FFN} \times 1 \times (2D-1)}_{\text{from } \boldsymbol{W}_1\boldsymbol{X}} + \underbrace{D_{FFN} \times 1}_{\text{from } \boldsymbol{b}_1} = 2DD_{FFN}.$$

  Similarly, the FLOPS for calculating $\boldsymbol{W}_2\boldsymbol{X} + \boldsymbol{b}_2$ is also

$$2DD_{FFN}.$$

- **SwiGLU activation:** The FLOPS for calculating $\text{Swish}_1(x)$ is

$$10 \text{ (exponentiation)} + 1 \text{ (addition)} + 1 \text{ (division)} = 12.$$

  Therefore, the FLOPS from Swish activation and element-wise multiplication is

$$\underbrace{12 \times D_{FFN} \times 1}_{\text{from Swish}} + \underbrace{D_{FFN} \times 1}_{\text{from } \odot} = 13D_{FFN}.$$

- **Second linear layer:** It maps back to the original hidden size by multiplying $\boldsymbol{W}_3$ of shape $D \times D_{FFN}$ with $\boldsymbol{X}'$ of shape $D_{FFN} \times 1$ with bias $\boldsymbol{b}_3$. So the FLOPS is

$$\underbrace{D \times 1 \times (2D_{FFN} - 1)}_{\text{from } \boldsymbol{W}_3 \boldsymbol{X}'} + \underbrace{D \times 1}_{\text{from } \boldsymbol{b}_3} = 2DD_{FFN}.$$

Then the total feedforward FLOPS is

$$N \times (2DD_{FFN} + 2DD_{FFN} + 13D_{FFN} + 2DD_{FFN}) = N(6DD_{FFN} + 13D_{FFN}).$$

4. **RMSNorm:** involves element-wise multiplication and division: for each token $\boldsymbol{x} \in \mathbb{R}^D$ (N tokens in total),

$$\text{RMSNorm}(\boldsymbol{x}) = \frac{\boldsymbol{x}}{\sqrt{\frac{1}{D}\sum_{i=1}^{D} \boldsymbol{x}_i^2}} \odot \boldsymbol{g}$$

where $\boldsymbol{g} \in \mathbb{R}^D$ is the gain parameter used to re-scale the standardized summed inputs, and is set to 1 at the beginning.

$$(D + (D+1) + 1 + 10 + D) \times N = (3D + 12)N.$$

5. **Residual connections:** Addition after attention and feedforward [2]:

$$2 \times N \times D.$$

6. **Position embedding:** For Qwen model, it used Rotary Position Embedding [5]. According to Section 3.2.2 of the paper [7], it involves multiplying a rotational matrix $\boldsymbol{R} \in \mathbb{R}^{D/H \times D/H}$ to query, key and value. Therefore, the FLOPS is

$$3 \times H \times \frac{D}{H} \times N \times (2\frac{D}{H} - 1) = 3DN(2\frac{D}{H} - 1).$$

Combining everything above together, and assuming the FLOPS of back-propagation are exactly equal to 2 times the FLOPS for the forward pass, I calculated the order of magnitude for the FLOPS to be 10^{12}, as shown in the notebook `flops.ipynb`.

2 LoRA

Fine-tuning LLMs is a problem because LLMs tend to have billions of parameters. Therefore, full fine-tuning requires huge memory, high computational cost and may have the risk of overfitting. To overcome this problem, a method called **Low-Rank Adaptation** (LoRA) was introduced [3]. LoRA reduces the number of trainable parameters by keeping the base model frozen and can therefore improve training efficiency while preserving model performance.

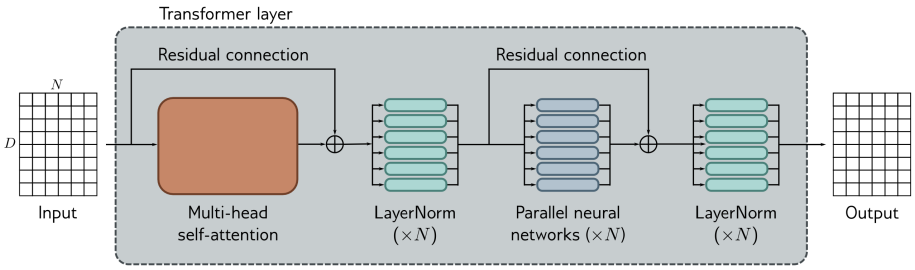

Figure 4: The transformer layer.

2.1 How Does LoRA Work?

In a transformer model, attention weights are computed as (ignoring biases)

$$\boldsymbol{Q}_h = \boldsymbol{\Omega}_{qh}\boldsymbol{X}, \quad \boldsymbol{K}_h = \boldsymbol{\Omega}_{kh}\boldsymbol{X}, \quad \boldsymbol{V}_h = \boldsymbol{\Omega}_{vh}\boldsymbol{X}$$

where $\boldsymbol{X} \in \mathbb{R}^{D\times N}$, $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V} \in \mathbb{R}^{\frac{D}{H}\times N}$ and $\boldsymbol{\Omega} \in \mathbb{R}^{\frac{D}{H}\times D}$ are trainable projection matrices.

To fine-tune the model, one would normally update the full projection matrices as:

$$\boldsymbol{\Omega}_{qh} \to \boldsymbol{\Omega}_{qh} + \Delta\boldsymbol{\Omega}_{qh} \tag{3}$$

where $\Delta\boldsymbol{\Omega}_{qh}$ has the same size as $\boldsymbol{\Omega}_{qh}$.

Instead of training a full-rank matrix $\Delta\boldsymbol{\Omega}_{qh}$, LoRA constrains it to a **low-rank decomposition**:

$$\Delta\boldsymbol{\Omega}_{qh} \approx \boldsymbol{A}\boldsymbol{B}$$

where $\boldsymbol{A} \in \mathbb{R}^{\frac{D}{H}\times r}$ and $\boldsymbol{B} \in \mathbb{R}^{r\times D}$, with $r \ll D$. This reduces the number of trainable parameters significantly from

$$\frac{D}{H} \times D \quad \to \quad r \times (\frac{D}{H} + D).$$

Hence, the updated query becomes

$$\boldsymbol{Q}_h = \boldsymbol{\Omega}_{qh}\boldsymbol{X} + \boldsymbol{A}\boldsymbol{B}\boldsymbol{X}. \tag{4}$$

To control the strength of the low-rank update, one can add a weight as:

$$\boldsymbol{Q}_h = \boldsymbol{\Omega}_{qh}\boldsymbol{X} + \frac{\alpha}{r}(\boldsymbol{A}\boldsymbol{B}\boldsymbol{X}). \tag{5}$$

where α is a scaling factor that controls the contribution of LoRA, and r is the rank of low-rank matrices.

2.1.1 FLOPS Calculation for LoRA

To track the FLOPS used when applying LoRA linear layers, the following steps are executed:

1. Multiplying lower rank matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ gives a FLOPS of
$$\frac{D}{H} \times D \times (2r - 1).$$
2. Adding the low-rank decomposition to previous weight matrix ($\boldsymbol{\Omega} \to \boldsymbol{\Omega} + \boldsymbol{AB}$) gives extra
$$\frac{D}{H} \times D$$
number of FLOPS.
3. Multiplying the sum of above two by 2 (LoRA updates for both query and key), by H (there are H heads in multi-head self attention) and by T (the total number of transformer layers), we get the final extra FLOPS due to LoRA for the forward pass:
$$2 \times H \times T \times \left[\frac{D}{H} \times D \times (2r - 1) + \frac{D}{H} \times D\right] = 4rTD^2.$$

2.2 Training with LoRA

I run the Jupyter notebook `lora.ipynb` on Google Colab, with GPUs to speed up the training process. The hyperparameters for training are chosen to be:

```
lora_rank = 4
max_ctx_length = 512 # maximum context length
batch_size = 4
learning_rate = 1e-5
epochs = 10
```

I divided the first 100 systems in the dataset `trajectories` to be the test set, and then for the remaining 900 systems, 80% of them are taken as the training set, while the remaining 20% are the validation set.

We are asked to train the model with no more than 10,000 optimizer steps on the training set. With a batch size of 4, there are 903 steps, so I trained the model with LoRA implementation for 10 epochs, which took a total of 9030 optimizer steps. The `Accelerator` library will put everything (e.g. model, optimizer, etc) on device (GPU if available; otherwise CPU).

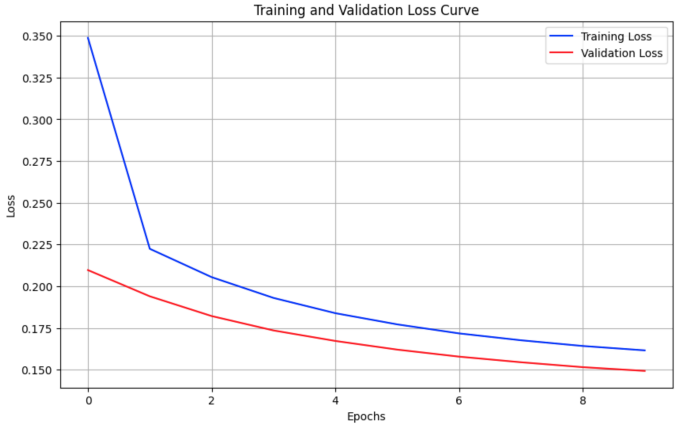

Figure 5: Training and Validation loss curves as a function of trained epochs during model training.

The training and validation loss curves as a function of the number of epochs are showing in Figure (5). We can see a decreasing trend as expected, and the model is not overfitting. The loss curves are becoming more steady, which suggests that the losses are converging, and hence no more training steps are needed.

2.3 Evaluation of Model with LoRA

To view the performance of the model with LoRA linear layers, I look at the performance of the predictions for each of the first 6 systems. The function `plot_predictions` in `lora.ipynb` is performing this task. It works the same as the function `plot_predictions` in `baseline.ipynb`.

To make better comparison, I plot the performance of predictions of the model with LoRA linear layers, as shown in Figure (6). Comparing it with Figure (2), we can clearly see that the trained model with LoRA gives more accurate prediction, in a sense that the mean prediction curves are closer to the actual curves, and the widths of the error bars are narrower. Moreover, the MSE values for each of the 6 systems are lower. For example, for System 1, the untrained Qwen model makes predictions with MSE = 0.3478, while trained model makes predictions with MSE = 0.01599.

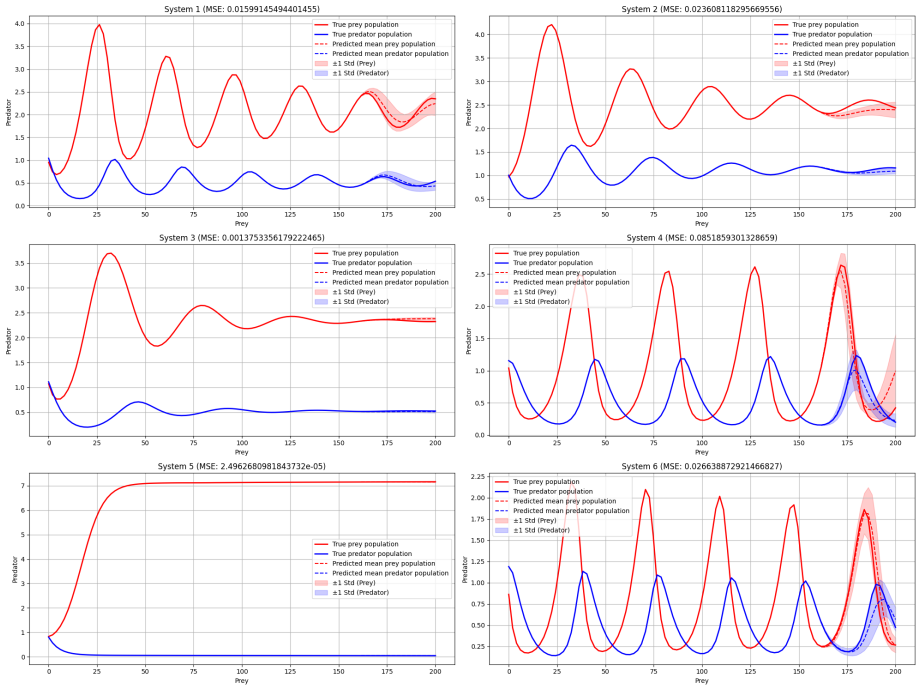

Figure 6: Model prediction for the first 6 predator-prey systems. The red solid lines denote the true prey populations. The blue solid lines denote the predator populations. The dashed lines represent predictions from the trained model with LoRA linear layers being added. The shaded regions show the ± 1 standard deviation for each prediction. The MSEs are shown next to the subtitles.

3 Hyperparameter Search

3.1 Searching Learning Rate and LoRA Rank

To find the best hyperparameters of the LoRA configuration, I explored different learning rates: $\{10^{-5}, 5 \times 10^{-5}, 1 \times 10^{-4}\}$ and different LoRA rank: $\{2, 4, 8\}$ with default maximum context length of 512. The codes are in the Jupyter notebook named `param_search.ipynb`. Due to computational limitations, I couldn't train for 10,000 optimizer steps for each configuration, but rather I trained for 5 epochs for each, which corresponds to about 5,000 optimizer steps.

I plot the training and validation loss curves for each configuration to explore the best configuration. They are shown in Figure (7) and (8).

I have also summarized the final training and validation loss for each con-

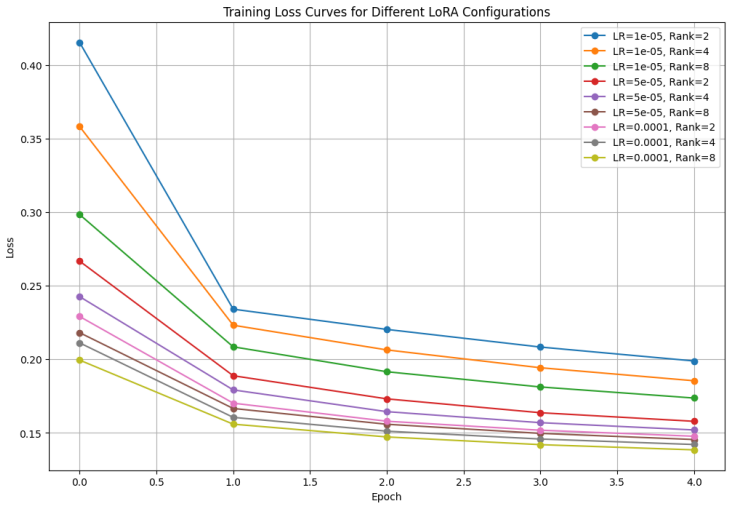

Figure 7: Training loss curves for 9 different LoRA configurations.

figuration in Table (2).

Index	Learning Rate	LoRA Rank	Final Training Loss	Final Validation Loss
8	0.00010	8	0.138472	0.128060
7	0.00010	4	0.142115	0.131203
5	0.00005	8	0.145497	0.135002
6	0.00010	2	0.147744	0.136313
4	0.00005	4	0.152022	0.139927
3	0.00005	2	0.157881	0.144813
2	0.00001	8	0.173696	0.157625
1	0.00001	4	0.185437	0.167862
0	0.00001	2	0.198879	0.178819

Table 2: Results from hyperparameter search over learning rate and LoRA rank after training for 5 epochs.

Therefore, it is obvious that the configuration with learning rate $= 10^{-4}$ and LoRA rank $= 8$ is the best configuration, since both the training and validation loss are the lowest.

To further evaluate different configurations and make comparison on the validation set, I also plot the prediction and measure the MSE for the last 20 time steps, just like in Figure (2) and (6). Again, due to computational limitation, I only predict the first two systems for each configurations. They are listed from Figure (9) to (17).

It is as expected that the predictions for System 1 and System 2 with con-

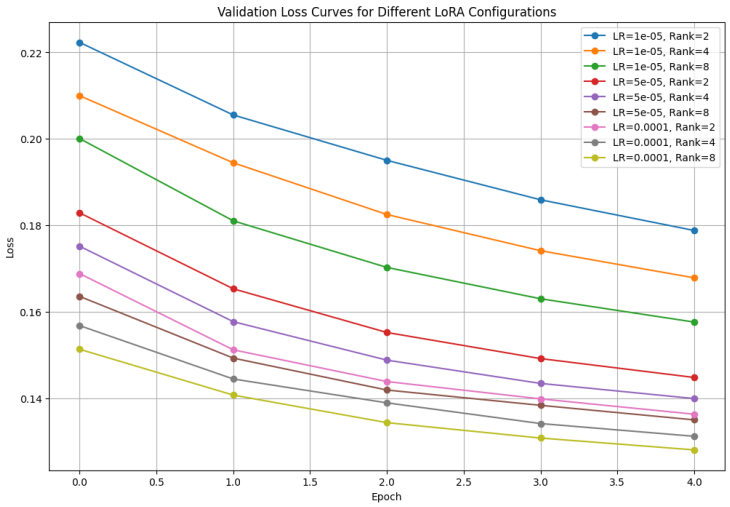

Figure 8: Validation loss curves for 9 different LoRA configurations.

figuration of learning rate = 10^{-4} and LoRA rank = 8 are close to the actual curves, with low enough MSE values, compared with other configurations. This observation suggests that such configuration is indeed the best configuration. However, we can also notice that the MSE for some other configurations are even lower. This cannot be the evidence that other configurations are better, as we still need to look at the average MSE over the whole test set (which contains 100 systems). However, due to computational limitations again, I did not measure the average MSE over the test set.

3.2 Searching for Maximum Context Length

Having obtained the best learning rate and LoRA rank, we now explore the maximum context length that gives us the best model. In the Jupyter notebook `param_search.ipynb`, I searched for different maximum context lengths of 128, 512 and 768. The final training and validation losses after training are listed in Table (3).

Index	Max Context Length	Final Training Loss	Final Validation Loss
1	512	0.195878	0.150171
2	768	0.169085	0.155861
0	128	0.212036	0.187947

Table 3: Final training and validation losses for different context lengths (sorted by final validation loss).

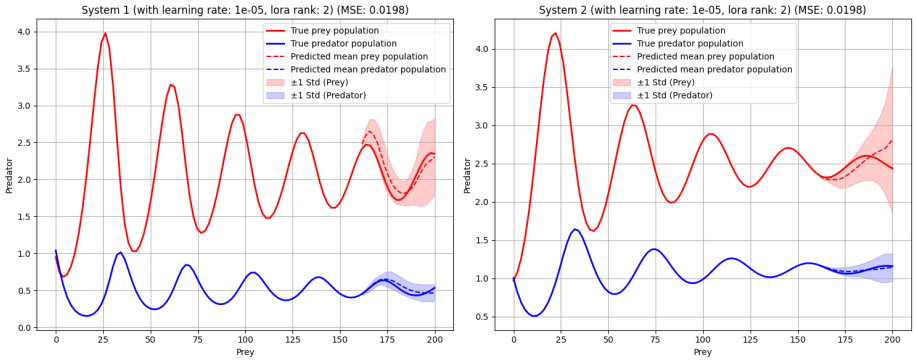

Figure 9: Prediction with learning rate = 10^{-5}, LoRA rank = 2.

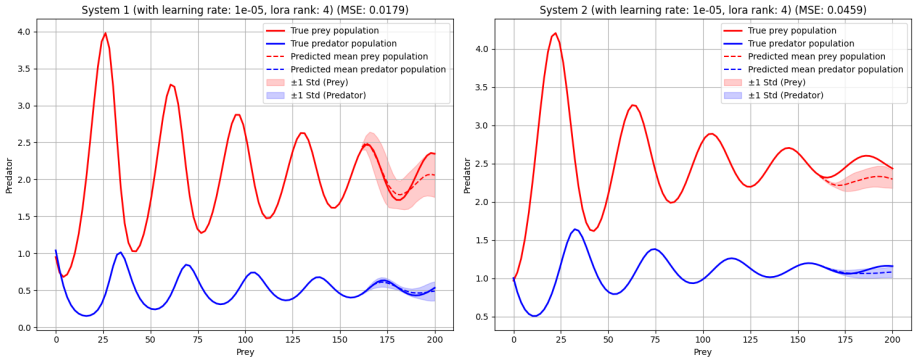

Figure 10: Prediction with learning rate = 10^{-5}, LoRA rank = 4.

We can conclude that 512 is the best maximum context length, as it gives the lowest final validation loss of 0.150171. Also, it provides the best balance between training efficiency and generalization, without model overfitting.

We observed that increasing the context length to 768 resulted in lower training loss but higher validation loss, suggesting possible overfitting due to increased model capacity. Moreover, doubling the context length increased FLOPS quadratically, as shown in Section 2.1.1, but the gain in validation performance was marginal, suggesting diminishing returns from larger sequence sizes.

To explore the prediction effects of the model with different context length, I plotted Figure (18), (19) and (20) for System 1 and 2. We can see that with context length = 512, the MSE for both System 1 and 2 are the lowest, with values of 0.0091 and 0.0317 respectively. Moreover, the mean predicted dashed lines are closer to the actual curve with this context length. These observations support the previous result that with 512 context length, the model would give

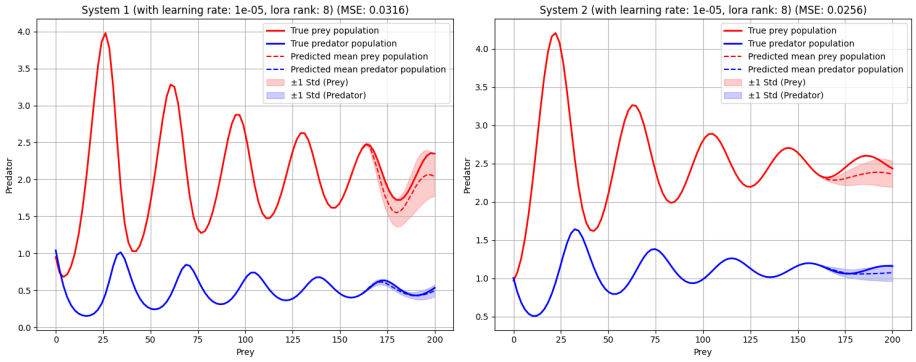

Figure 11: Prediction with learning rate = 10^{-5}, LoRA rank = 8.

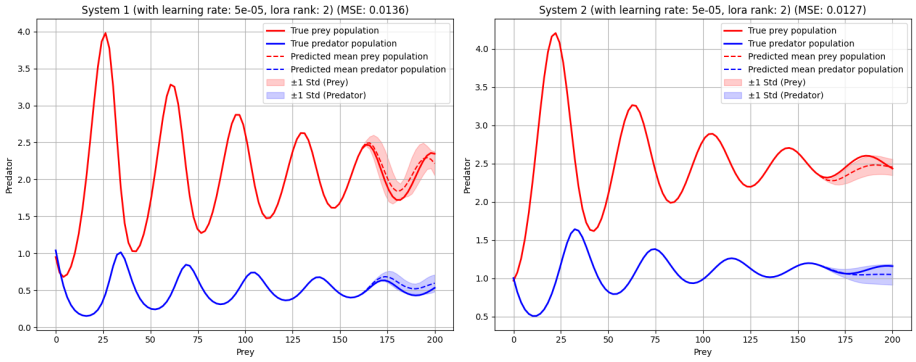

Figure 12: Prediction with learning rate = 5×10^{-5}, LoRA rank = 2.

better performance to forecast future time-series data.

The FLOPS for different experiments are summarized in Table (4).

Experiment	Total FLOPS
LoRA = 2, context length = 128	1.54×10^{15}
LoRA = 4, context length = 512	6.45×10^{15}
LoRA = 8, context length = 768	9.95×10^{15}

Table 4: Total FLOPS for various experiments.

3.3 Performance of the Final Best Model

Having found the best hyperparameters (learning rate = 10^{-4}, LoRA rank = 8, context length = 512), we are ready to train the model with best hyperparameters for 20 epochs (which corresponds to 18060 optimizer steps) for around

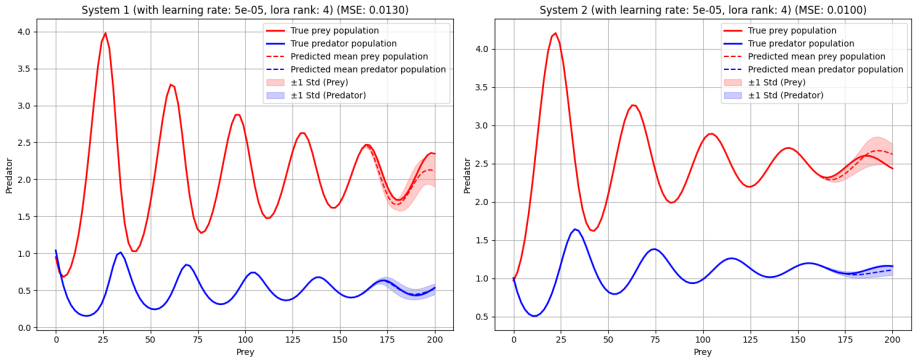

Figure 13: Prediction with learning rate $= 5 \times 10^{-5}$, LoRA rank $= 4$.

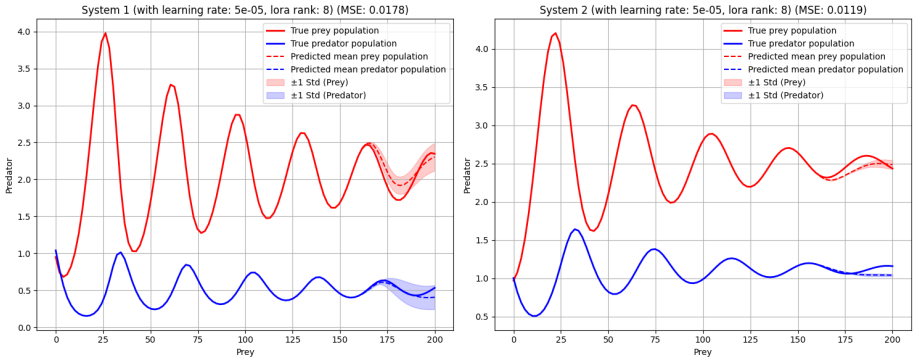

Figure 14: Prediction with learning rate $= 5 \times 10^{-5}$, LoRA rank $= 8$.

1 hour on Colab A100 GPU. The final training loss is 0.1156, while the final validation loss is 0.1121. These loss values are lower than our previous experiments, which support our finding that this configuration gives better model performance.

Figure (21) shows how training and validation loss curves change with number of epochs. It is clear that this final model is not overfitting. There is a steady decreasing trend for both of the loss curves, which suggests that the model can be improved further if longer training is performed. However, because of limited GPU compute units, I only trained for 20 epochs.

Figure (22) demonstrates the performance of predictions of the final model. Comparing it to previous prediction plots, Figure (22) gives more accurate predictions, with narrower error bar and lower MSE values. These observations further back up our experiment result that the final model has better performance with those hyperparameters.

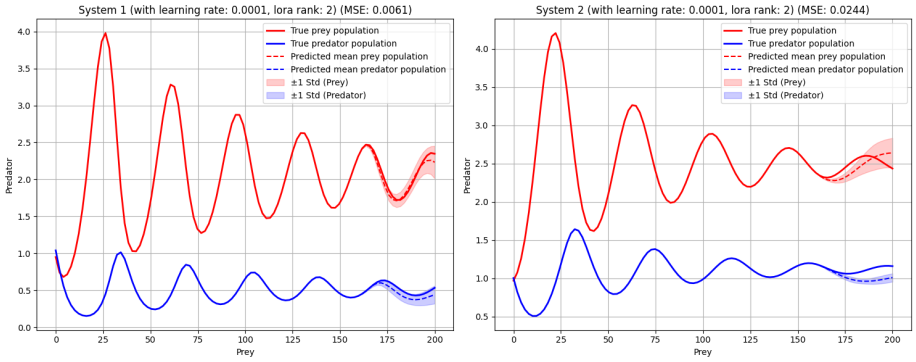

Figure 15: Prediction with learning rate = 10^{-4}, LoRA rank = 2.

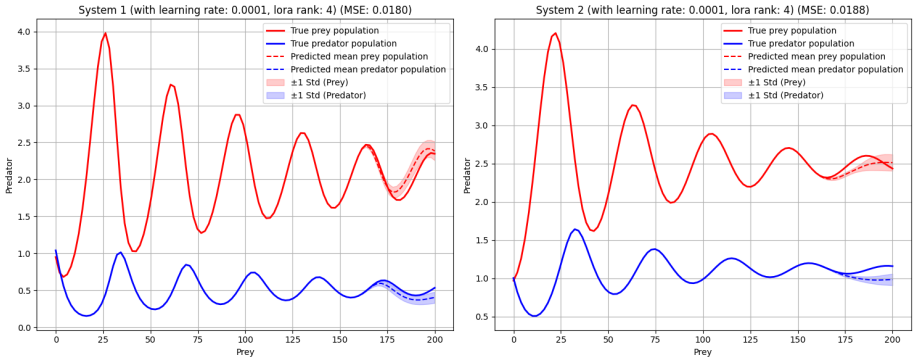

Figure 16: Prediction with learning rate = 10^{-4}, LoRA rank = 4.

4 Conclusion and Future Work

Our experiments highlight following key trade-offs and insights for fine-tuning time-series models under tight compute budgets:

1. **Optimal context length:** A maximum context length of **512** provided the best balance between computational cost and model performance. Increasing to 768 led to slightly better training loss but higher validation loss, suggesting overfitting and diminishing returns due to higher FLOPS. For time-series data, longer context lengths may not necessarily translate to better generalization due to the repetitive nature of patterns.

2. **Learning rate and convergence:** A learning rate of $\mathbf{10^{-4}}$ led to stable convergence with a good trade-off between training speed and final performance. Higher learning rates may cause training instability, while lower rates led to slow convergence.

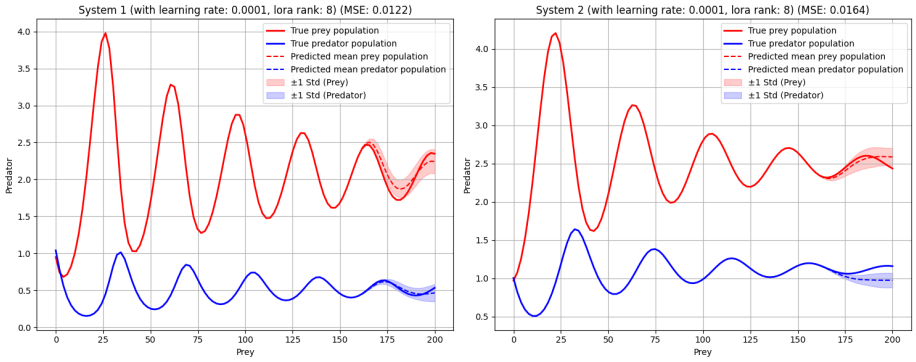

Figure 17: Prediction with learning rate = 10^{-4}, LoRA rank = 8.

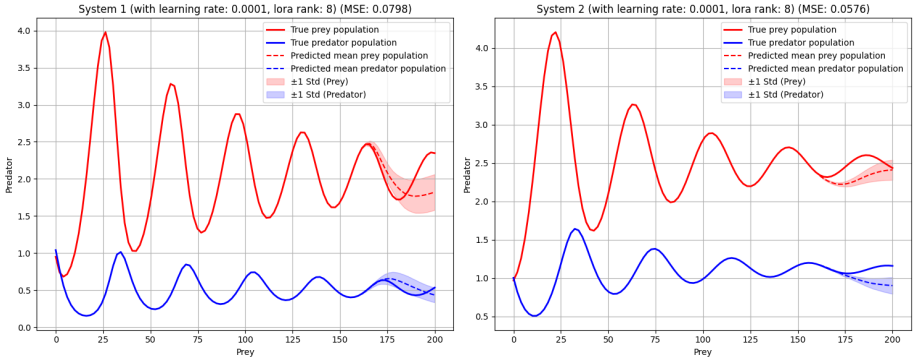

Figure 18: Prediction with context length = 128.

3. **LoRA rank efficiency:** A LoRA rank of **8** achieved the best performance. For compute-limited scenarios, using a lower LoRA rank combined with a modest context length (e.g. 512) could reduce training time while preserving reasonable accuracy.

4. **Computational trade-offs:** FLOPS increased quadratically with context length, leading to rapidly increasing compute costs. For practical use cases, a context length of 512 with LoRA rank 4 could be a good compromise between compute and performance.

If computational power are enough, there are a few suggestions for future improvements:

1. **Regularization:** Use techniques like weight decay and dropout to improve generalization and reduce overfitting at hight context lengths.

2. **Context length:** Explore intermediate context lengths between 512 and

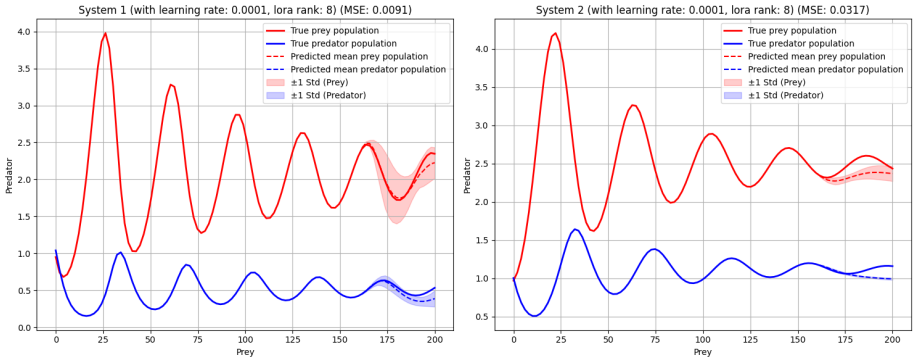

Figure 19: Prediction with context length = 512.

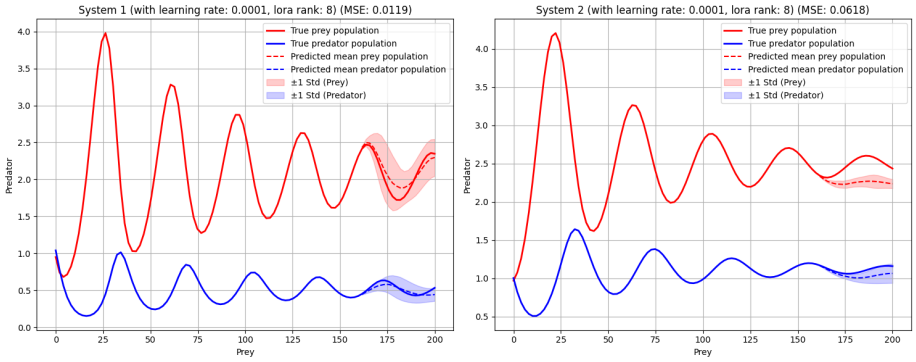

Figure 20: Prediction with context length = 768.

768 to fine-tune the balance between generalization and computational cost.

3. **Dynamic context length:** Explore adaptive context lengths where the model adjusts context size based on the complexity of the time-series data.

4. **Efficient attention:** Test alternative attention mechanisms (e.g. sliding window or sparse attention) to handle long sequences more efficiently.

References

[1] Nate Gruver et al. *Large Language Models Are Zero-Shot Time Series Forecasters*. 2024. arXiv: 2310.07820 [cs.LG]. URL: https://arxiv.org/abs/2310.07820.

[2] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV]. URL: https://arxiv.org/abs/1512.03385.

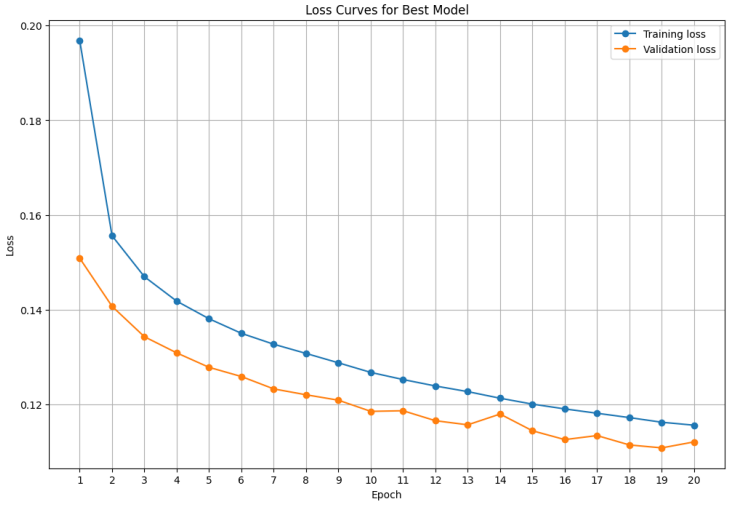

Figure 21: Training and validation loss curves for the final best model for 20 epochs.

[3] Edward J. Hu et al. *LoRA: Low-Rank Adaptation of Large Language Models*. 2021. arXiv: 2106.09685 [cs.CL]. URL: https://arxiv.org/abs/2106.09685.

[4] Simon J.D. Prince. *Understanding Deep Learning*. The MIT Press, 2023. URL: http://udlbook.com.

[5] Qwen et al. *Qwen2.5 Technical Report*. 2025. arXiv: 2412.15115 [cs.CL]. URL: https://arxiv.org/abs/2412.15115.

[6] Noam Shazeer. *GLU Variants Improve Transformer*. 2020. arXiv: 2002.05202 [cs.LG]. URL: https://arxiv.org/abs/2002.05202.

[7] Jianlin Su et al. *RoFormer: Enhanced Transformer with Rotary Position Embedding*. 2023. arXiv: 2104.09864 [cs.CL]. URL: https://arxiv.org/abs/2104.09864.

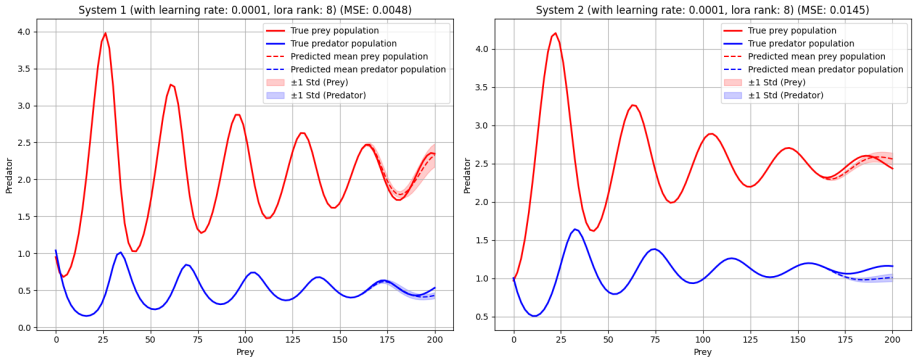

Figure 22: Prediction plots for the final best model for the first 2 systems.