Efficient Fine-Tuning of Large Language Models for Time-Series Forecasting via LoRA

Lewis McConkey
Cambridge University

Word count: 2351

Final Assessed Grade: 82% (Distinction)

Professor

Dr Miles Cranmer

Departments

Cambridge University, Cavendish Laboratory
Department of Applied Mathematics and Theoretical Physics

April 8, 2025

Abstract

This project investigates efficient fine-tuning strategies for large language models (LLMs) applied to time-series prediction tasks under constrained compute budgets. Leveraging Low-Rank Adaptation (LoRA), we systematically explore the impact of varying LoRA rank and learning rates on the performance of the Qwen2.5 model, using both training and validation loss as evaluation metrics. A grid search over varying LoRA ranks and learning rates identifies the optimal model as rank 4 with a learning rate of 4×10^{-5}. With this setup we show the performance of such a model over 30000 optimiser steps before giving practical recommendations for low-budget fine-tuning, including the use of small LoRA ranks, modest learning rates, and shorter context lengths.

1 Introduction

Large language models (LLMs) have demonstrated remarkable performance across a wide range of natural language processing tasks. However, fine-tuning these models for specific applications can be computationally expensive and parameter inefficient. Low-Rank Adaptation (LoRA) has emerged as a powerful solution to this challenge by enabling efficient fine-tuning through the insertion of trainable low-rank matrices into specific layers of a frozen pre-trained model.

In this report, we adapt the LoRA methodology to the Qwen2.5-0.5B-Instruct model, targeting the query and value projection layers of the transformer blocks. We evaluate the model's performance before and after fine-tuning using standard loss metrics, discuss the improvements observed on the validation set, and provide visualisations of the training dynamics

2 Baseline

2.1 LLMTIME Prepocessing

We start by implementing the LLMTIME preprocessing scheme described in the main.pdf file. We load in the Qwen tokeniser and the Lotka-Volterra dataset using the code in this same file. We create a preprocessing_time_series function that takes in alpha (our scaling factor), the data (the Lotka-Volterra dataset) and rounds the preprocessed sequence to 2 decimal places. In this function we scale the data by alpha which we have set as 10 in this case to standardise the numeric range and control token length. We then loop over all systems of independent predator-prey simulations governed by the Lotka-Volterra equations to return the preprocessed sequence. The function tokenize_sequence is then used to convert this preprocessed sequence to the tokenised one using Qwen. We have multivariate predator-prey data so we also separate different variables at the same time step with a comma and separate different time-steps with a semicolon in our loop. In this data we have the two variables predator and prey population so every two elements are separated by a semi colon for the next time-step and the two elements in between the semi colon's are separated by a comma to separate the prey and predator populations. We output two example sequences, giving both the preprocessed and tokenised sequences that have been created with our LLMTIME preprocessor and Qwen2.5 tokens. Finally, we reverse the encoding process by applying decode_sequence, which post-processes the tokenized sequence back into a numerical array. This allows us to recover the original prey and predator population values from the model's output.

2.2 Qwen2.5 Instruct Model

Staying with the best machine learning practices approach before we actually implement the model we have first provided some preliminary visualisation/analysis (in the pleliminary_visualisation Jupyter notebook) to be one with the data and ensure we properly understand the predator-prey data. The most significant plot is a plot of several predator and prey systems population plotted against time which is shown below:

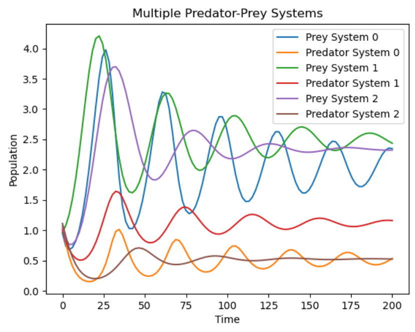

Figure 1: Three predator and prey systems population governed by the Lotka-Volterra equations vs time

There is not a lot in the preliminary visualisation but after visualising this it is quite clear to understand this time series data. We have predator and prey populations across time from different simulated systems with the populations generally being higher for prey. We can now move onto ML implementation for the rest of the report.

We proceed by evaluating the untrained Qwen2.5-Instruct model's forecasting ability on the tokenised dataset. We start by loading and setting up the Qwen2.5 model for evaluation and load the tokenised sequence from part a). We convert this to a PyTorch tensor so we can input it into the Qwen2.5 model to generate predictions. After this we convert the model-generated tokens back into a human-readable string before passing it through the decode_sequence function to get our meaningful predictions. We initially do this with one system to view quickly the performance, reduce computational power needed and save time. Here is the true vs predicted values for the first system and the absolute errors across timesteps:

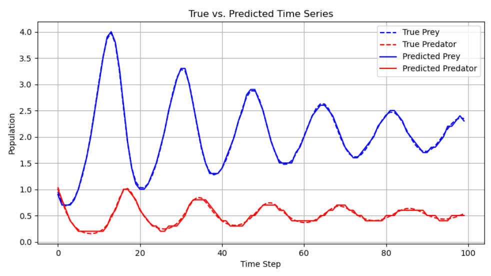

Figure 2: True vs predicted values from our Qwen2.5 model for both the prey and predator populations (first system).

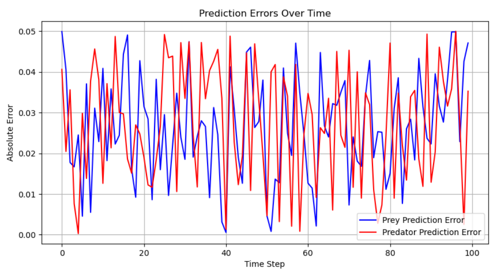

Figure 3: Absolute errors vs timesteps (first system).

The predicted vs. true values for the first system show an exceptionally close match and the absolute errors are close to 0 for every time step, indicating that the model has captured the underlying patterns well for this specific system. Since the model was only evaluated on this single system, there's a significant risk that it has overfitted to this specific structure rather than learning a more generalisable pattern. We should evaluate the model across all available systems and compare performance metrics to explore the model's performance further.

To evaluate the models performance on all systems we compute the average (across all systems) of the mean absolute error (MAE), the root mean squared error (RMSE) and the continuous ranked probability score (CRPS) like what was done in [1]. We get that the MAE and the CRPS equal 0.0258 and the RMSE is 0.0281. These are all quite low, which suggests that the untrained Qwen2.5 model is still producing relatively close predictions to the ground truth. MAE being equal to CRPS though suggests the model is not capturing uncertainty well.

2.3 FLOPS

We want to calculate the total number of FLOPS of the model following the table from the main.pdf file. We calculate the FLOPS for the forward pass first and then use the simplification that the backward pass is 2 times the forward pass. For each layer, we perform self-attention using Query, Key, and Value (Q, K, V) projections, attention scores, softmax, and weighted sum. We assume the FLOPS for the grouped query attention that Qwen2.5 instruct uses are the same as a standard multi-head attention and we proceed with this. First calculating the flops when computing the query, key and values. Each input token has dimension d and the Q,K and V matrices are of size d. For each input we have the batch size B and the token sequence length S along with the two dimensions from the last sentence. We therefore have $B \times S \times d \times d$, however there are 3 of these because we have query, keys and values so overall we have $3 \times B \times S \times d \times d$

We next have to compute the flops for the attention scores:

$$\frac{QK^T}{\sqrt{d_q}}$$

d_q is the dimension of the queries and keys. We have the batch size B, the dimension per attention head $d_h = \frac{d}{h}$, number of attention heads h and the token sequence length S (for both the queries and keys). This results in $B \times h \times S^2 \times d_h$ for the flops of the numerator. The two matrices are then scaled by the denominator which gives $B \times h \times S^2$. Overall the attention scores give a combined $(B \times h \times S^2 \times d_h) + (B \times h \times S^2) = B \times h \times S^2(d_h + 1)$ number of flops.

Now we compute the flops for the softmax. We have the exponential calculation on the denominator which is $10 \times B \times h \times S^2$ (using the flops table this is ten times more). On the denominator we have the summation over exponential so we need $(10 + 2) \times B \times h \times S^2$. In total this is now $22 \times B \times h \times S^2$.

We move onto calculating the flops for the weighted sum with V. This is just the same number of computations as multiplying the queries by the keys so we have $B \times h \times s^2 \times d_h$ again. Finally we have the flops for the merging of attention heads which is $B \times S \times d \times d$. Overall for the multi-head self attention we have: $B \times S \times (4dd + S(d_h + 1) + 22hS)$ number of flops

We now calculate all the flops for the multi-layer perceptron (MLP) with SwiGLU activation. We have 2 linear transformations that give $2 \times B \times S \times d \times d_{ff}$, where d_{ff} is the feedforward dimension in MLP. With the SwiGLU activation we have:

$$\begin{aligned}&\text{Swish}(X_1)\cdot(X_2)\\&=X_1\cdot\sigma(X_1)\cdot X_2\end{aligned}$$

where σ is the sigmoid function, X_1, X_2 are two linear transformations. For the Swish function alone we have here (per element) 10 flops for the exponential, 2 flops for the addition and division and 1 flop for the multiplication. For the Swish function we now have $(10+2+1)\times B\times S\times d_{ff}$. We multiply this function by X_2 though so we have an extra $B\times S\times d_{ff}$. Altogether for the SwiGLU activation we have: $24\times B\times S\times d_{ff}$

Lastly we have the normalisation, for the RMSNorm we have 1 flop per element for the summation, division and element-wise multiplication giving a total flops of $3\times B\times S\times d$. Putting all the flops together we have overall a total of $B\times S\times(4dd+S(d_h+1)+22hS+3d+24d_{ff})$ number of flops per transformer layer. Hence for the entire model with back propagation we have a total of $3\times B\times S\times(4dd+S(d_h+1)+22hS+3d+24d_{ff})$ number of flops. Implementing this in our code we get 1.62×10^{14} flops.

3 LoRA

3.1 Implementing the Model

We implement Low-Rank Adaptation (LoRA) on the Qwen2.5-Instruct model. We train the 0.5B parameter model using a single LoRA configuration with default hyperparameters, running 10,000 optimizer steps on the training set and evaluating performance on a validation set. We also compare the fine-tuned model's results against the untrained model. After building the end-to-end pipeline we can evaluate the untrained models performance by looking at the training and validation loss (we have an 80/20 split).

In LoRA, instead of updating the full weight matrices of the self-attention mechanism, we introduce two low-rank matrices and train only these while freezing the original model weights. We specifically apply LoRA to the query and value projection layer. Each of these layers is originally an nn.Linear layer, which we replace with a custom LoRALinear wrapper. The parameter blocks being tuned in this model are the A and B matrix which are of shape (r, d_{in}) and (d_{out}, r) respectively, where r is the LoRA rank and d_{in}, d_{out} are the input and output dimensions.

We first implement the model on a small batch and overfit on this. Here we give plots of the training and validation loss performance.

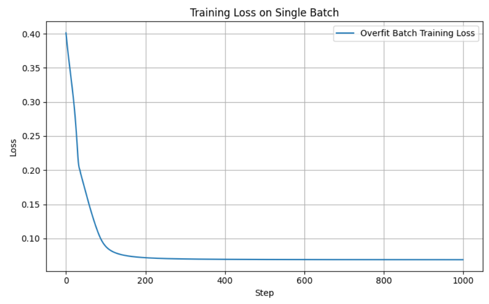

Figure 4: Training loss over steps for model overfitting on small batch

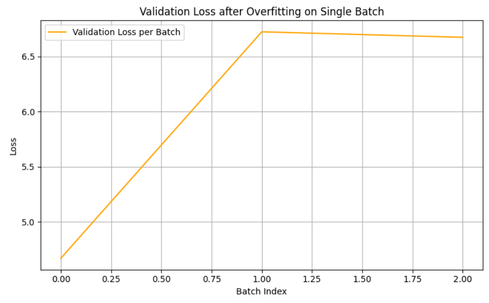

Figure 5: Validation loss over steps for model overfitting on small batch

We can see after overfitting the model on a small batch of the data that the training loss is incredibly low (ending around 0.0688). The validation loss increases and then converges just over 0.65. We can see that the pipelines fundamentals are good and that the model has the capacity to fit the data. We get what we would expect when overfitting as it memorises the training set really well but fails to generalise to the validation set.

After implementing the model on the whole training set, we make plots of the validation loss and training loss:

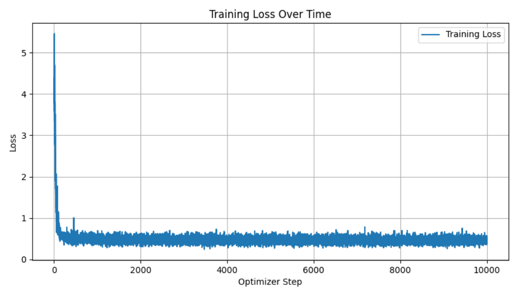

Figure 6: Training loss over steps for first implementation of the model

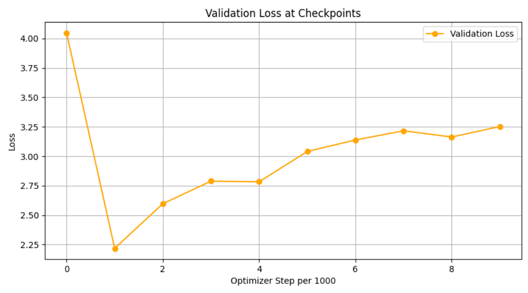

Figure 7: Validation loss over steps for first implementation of the model

The training loss performs how we would expect with it starting off at 3.94 and then decreasing significantly early on when the model learns to fit the training data. This then converges to around 0.4 after the 10000 optimiser steps. With the validation loss this improves initially but then gets progressively worse as it goes through 1000-10000 steps. This is a sign that the model might be starting to overfit (it's learning to fit the training data too well but is losing its ability to generalise to the validation data). We have yet to hyper-parameter tune so this makes sense, when we do this we would expect the validation loss to improve over the steps and not overfit anymore. We can also use methods like early stopping or regularisation as well to mitigate this behaviour.

When calculating the MAE and RMSE we got extremely high numbers compared to the baseline model which was suspect (We have left this out of the code) but we might be able to drive these down when tuning the hyper-parameters later on.

3.2 Hyper-parameter tuning

At this stage we could test out different regularisation methods like dropout, early stopping etc. but we will move on to the hyper-parameter tuning in part b). We initially test out 3 different LoRA ranks $(2, 4, 8)$ and learning rates $(10^{-5}, 5 \times 10^{-5}, 1 \times 10^{-4})$ with a grid search. Our best model comes out to be with rank 4 and learning rate 5×10^{-5}. See the Jupyter notebook for the visualisation of the performance of this model.

Using these hyper-parameters we now vary the context lengths between 128, 512, 768 and use 2000 optimiser steps instead of 10000 to limit the time and computational power needed. Once we run our grid search we see that with context length 768 the model performance the best, giving the final validation loss of 0.0277 and final training loss of 0.0183. From the visualisations in the jupyter notebook we see the training loss performs how we would expect by decreasing rapidly and then converging. The validation loss now does not increase after so many steps suggesting that the model is not overfitting anymore and this loss is also being driven down. We also notice the high initial loss though which is due to using high context lengths that contain more varied patterns. We now show the performance of this model over 30000 steps.

3.3 Best Performing Model

Our best performing model after all the experiments comes out to be rank 4, learning rate 5×10^{-5} and 768 context length. We now test out this on 30000 optimiser steps to showcase the full performance of our chosen model. We can visualise the performance through the training and validation losses below:

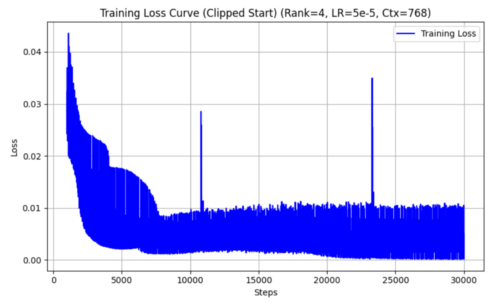

Figure 8: Final models training loss over 30000 steps

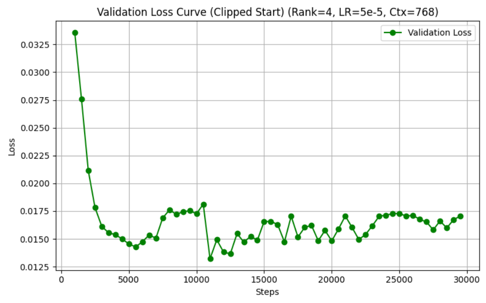

Figure 9: Final models validation loss over 30000 steps

We have clipped the first 1000 steps since the loss is so high in the beginning. The performance however is very good as the training loss converges to a low value after decreasing heavily initially and the validation loss no longer increase or shows signs of overfitting but instead also converges to a low value just like the training loss. The end values are 0.0030 for the training loss and 0.171 for the validation loss which shows the model has effectively fit the training data while maintaining reasonable generalisation performance on the validation set.

Under tight compute budgets, effective time-series fine-tuning with large models like Qwen can be achieved by using lightweight adaptation methods such as LoRA. Our experiments demonstrated that even with as few as 2,000 optimiser steps, it is possible to distinguish between models and choose an optimal model. In particular, a LoRA rank of 4 combined with a learning rate of 5×10^{-5} yielded the best trade-off between training and validation loss across different setups. We recommend limiting the number of training steps during hyperparameter tuning to reduce computational power needed, reserving longer training runs (e.g., 30,000 steps) only for the best-performing models.

For improved performance, future work could investigate scaling up context length more systematically. Additionally, leveraging techniques like early stopping, dropout, data augmentation etc. could improve performance. We could do further hyperparameter tuning with more parameters or with more varying learning rates etc. As an alternative we could also use QLoRA instead because it lets you scale to bigger models under tighter compute budgets, by combining the low-rank trick of LoRA with aggressive quantisation techniques [3].

Finally we provide the flops for each experiment of the context lengths. We wrote python code in the Jupyter notebook at the end for this, first calculating the parameters per layer and total parameters for the multi-layer perceptron and using this to compute the flops from multiplying paramaters. We calculate the self attention flops and add this to the flops from before. We summarise our findings in the below table (where CL is context length):

CL	flops (per step)
128	1.65×10^{12}
512	6.67×10^{12}
768	1.01×10^{13}

References

[1] Nate Gruver... (2024), Large Language Models Are Zero-Shot Time Series Forecasters, Cornell University

[2] An Yang... (2025) Qwen2.5 Technical Report, Qwen

[3] Tim Dettmers... (2023), QLORA: Efficient Finetuning of Quantized LLMs, University of Washington