M2 Report

Chris Miller, clm96

University of Cambridge, Department of Physics

Word Count: 2500

Contents

1 Introduction

1.1 The Task

Our dataset consists of 1000 predator-prey time series, each containing 100 sequential measurements of the predator population size and the prey population size at that time. Δt is assumed constant between time steps. We aim to tune an LLM to forecast these time series. To do this we separate the data randomly into 900 training series, 50 validation series, and 50 testing series, we keep the entirety of the validation and test sets separate from the train set and each other in order to prevent leakage and to truly evaluate the models ability to generalise. Since our model is an LLM we are in a regime where training loss and model use are not the same (unlike for instance MNIST digit classification). We commit to training the LLM by minimising training loss, feeding the model sequences of predetermined length chosen from the training set and maximising the probability of predicting the next token correctly at every step along the sequence, as is the convention for training language models. When performing inference we shall give the model a sequence of tokens ending at the 80th time step (sequence chosen from the validation or test set depending on the context) and then forecast the next 20 time steps (approximately 220 tokens in our chosen tokenisation scheme) with the model in generate mode. In (Gruver et al., 2024[3]) a distinction is made between metrics to evaluate the quality of a forecast. While we agree that CRPS (continuous ranked probability score) is preferred over mean absolute error, the computational cost of sampling multiple estimates per time step and constructing a distribution of possible forecasts is far too expensive to consider feasible under the given 10^{17} FLOPs budget. Thus, we settle for mean absolute error between the forecast and the ground-truth as our key metric for forecast performance.

1.2 The Model

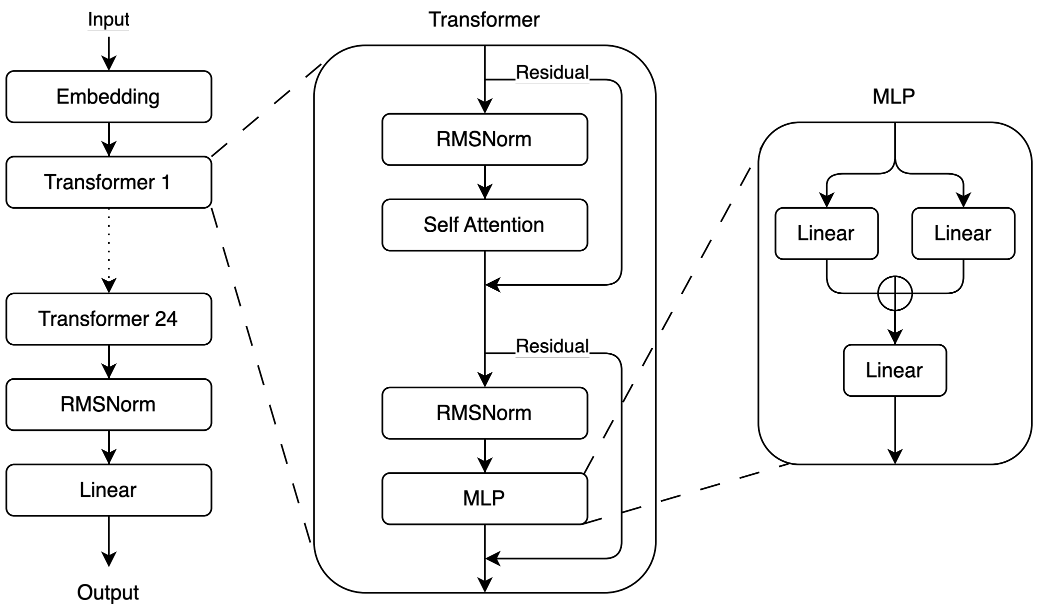

Figure 1: Left: Diagram of full model. Middle: Diagram of one transformer block. Right: Diagram of the MLP inside the transformer block.

As per the technical report[6], the model (seen in figure 1) that we train is a decoder-only LLM with 24 transformer[8] layers. The self-attention layers use grouped-query-attention[2] with 14 query heads, 2 key heads, and 2 value heads. As per instruction, we pretend that the model uses multi-head-attention with 14 heads in our FLOP calculations. For the entirety of the project this model's parameters are frozen, with the exception of the bias on the final linear layer (the language model (LM) head). The only other trainable parameters are included in the Low Rank Adaptation (LoRA)[4] layers that we inject into each self-attention block. Inside the self-attention block there are linear layers which calculate the query, key, and value vectors for each embedding vector passed through self-attention. To the query and value linear layers we add trainable LoRA matrices (or unbiased linear layers) as seen in figure 2. These LoRA layers are the parameters which we learn in hopes of tuning the LLM to be well suited to time series forecasting.

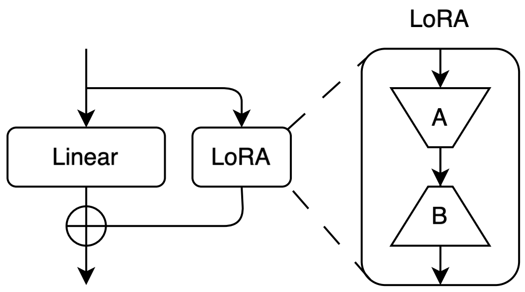

Figure 2: Diagram of Low Rank Adaptation where A is a down-projection (unbiased linear layer / matrix) to a space of dimension equal to the lora rank, B is an up-projection back to the original input dimension.

Another alteration that we make to the model is to reduce its vocabulary from the default 151,936 unique tokens to 13 in order to reduce the cost of the embedding (which could be interpreted as a free memory operation or a matrix multiplication) and the LM Head linear layer. We can do this since the forecasting model only has a few valid outputs, so we remove the model's ability to produce anything but these desired outputs. The thirteen new tokens correspond to a comma ",", a semicolon ";" and a padding token as well as a token for each digit 0,...,9. This reduces the number of parameters in the LM head from 136 million to 11,661. To do this, we replace the embedding layer and head of the original model with downsized versions, we also create a way to map from the Qwen token scheme to our new one before passing a sequence to the model and a way to map the output of the model to the Qwen scheme before passing it through the token decoder provided. Our implementation in the project code works without issue but is sub-optimal. Given more time, it would be better practice to create a new instance of the tokenizer class as opposed to a middleman converter class to place between the tokenizer and the model.

2 Baseline

2.1 Preprocessing

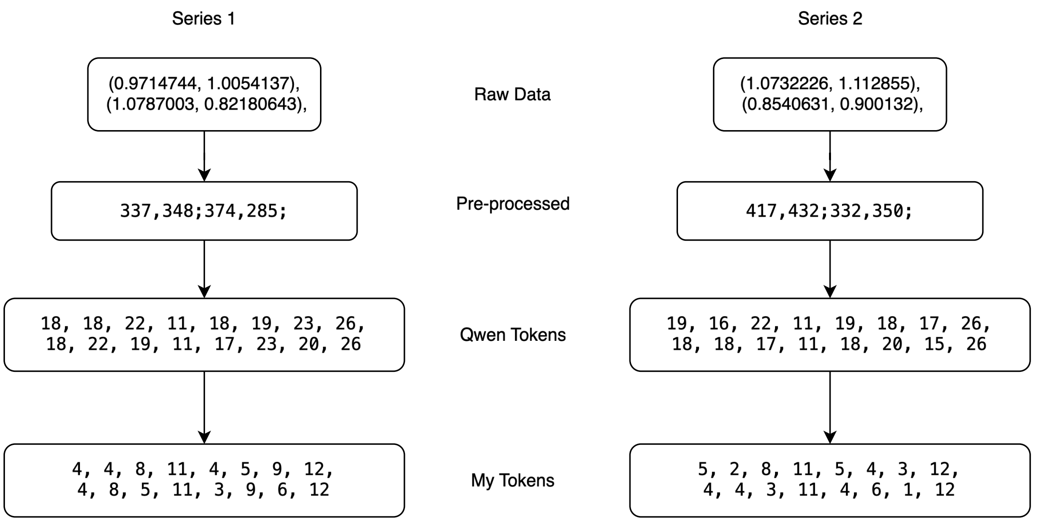

Figure 3: Pre-processing scheme applied to the first two time steps of series 1 (left) and series 2 (right).

As is demonstrated in the LLMTime[3] paper, for each series we scale the values so that the 90th percentile is 10, we then round each value to two decimal places and convert to strings, removing the decimal point since it is a waste of tokens. Under this interpretation, each series is scaled by a different value so that they all have the same 90th percentile. For this reason we perform all evaluation in scaled space so as to keep the relative error of each forecast comparable across series as well as to avoid having to rescale forecasts. See figure 3 for an example. We see that each time step translates to 8 tokens. During inference we choose to allow breathing room for the model to predict larger values at each time step, thus arriving at the 220 tokens permitted to ensure a forecast of 20 time steps is generated. In hindsight, this is an unnecessarily large buffer and we could have saved tokens by reducing from 11 down to 10 or even 9 tokens per step.

2.2 Performance

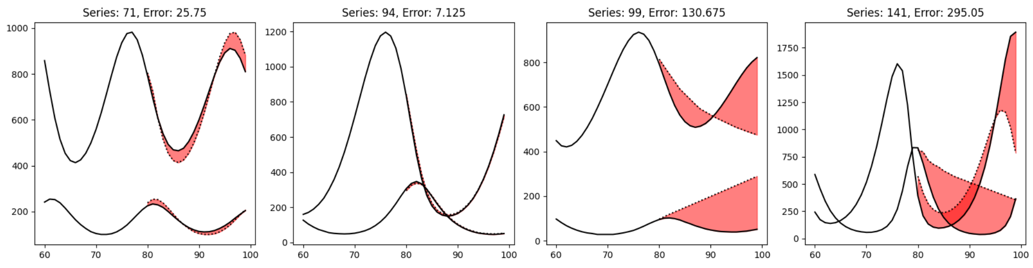

Figure 4: Forecasts of untrained Qwen model. Ground-truth in solid black, forecast in dotted black and difference between the two in red.

Evaluating the untrained Qwen model on the entire validation set of 50 series, we get an average MAE of 123.5. Even with no tuning, the model was able to forecast more than half of the series very well. In addition to showing the distribution of all MAEs, figure 5 shows the MAE of the best 30 series and the worst 20. In order to save on compute during the LoRA experiments, we choose a subset of 25 validation data series consisting of the worst 20 and 5 randomly chosen from the best 30. It is unlikely that the model will forget how to forecast series which it already understands well without tuning, so this shrinking of the validation set is without risk. As seen from figure 4, the variance in how well the model can forecast is high. In series 94 the forecast is near perfect, whereas in 99 and 141 the model is very far from fitting a sensible forecast. Since we are only using the MAE metric and thus are only generating one forecast per series, we generate all forecasts with sampling disabled so as to make the results deterministic. If using CRPS this would obviously be changed and we would vary hyper-parameters such as temperature when sampling for the distribution of possible curves.

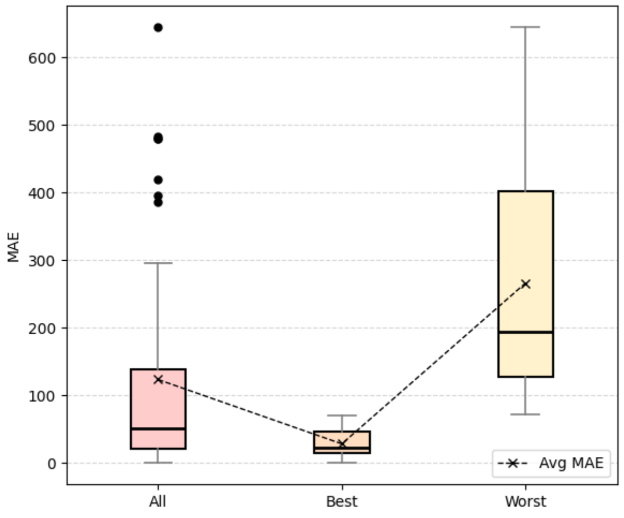

Figure 5: Boxplot of the MAEs of the training set. Median MAE of ≈ 50. The best and worst boxes represent the best 30 and worst 20 resp. in the validation set.

2.3 FLOPs

The total FLOPs used for establishing this baseline was 3.89×10^{15}, 3.89% of the budget. This value can be broken down as follows. Each forecast cost a 7.78×10^{13} FLOPs and consists of one full pass of the model followed by 219 passes where KV-caching saves on the recomputation of all but the N+1-th key, query and value vector. This computational saving is complicated to implement but more details can be found in the code. Claims for how much computational cost is saved with KV-Caching vary, one in particular[1] states a $3\times$ speed up which we could assume to mean a third of the FLOPs are used. Attempting to establish KV-Cache savings in our estimation of FLOPs yeilds far lower gains, and we choose to assume our calculated values are correct. We estimate that generating 220 tokens with KV-Caching enabled costs the equivalent of 210 full forward passes, meaning we estimate 3.7×10^{11} FLOPs per token generated. This may have resulted in an artificial inflation of the compute cost for inference. The following describes one full pass through the model, keeping track of the FLOPs used.

The input is passed through the embedding layer which we treat as a matrix multiplication. Then we pass through 24 transformer layers as shown in figure 1. Inside each transformer we pass through an RMSNorm[5], $x \leftarrow x/\sqrt{\frac{||x||_2^2}{d} + \epsilon}$. Calculating $||x||^2$ costs $2D - 1$ for D the dimension of the embedding vector since we have D squares and $D - 1$ additions. Then $\div d$ and $+\epsilon$ cost an extra 2. The square root costs 10 and then we perform D more divisions to complete the operation on one vector. We then repeat this for each token in the input (N) to get $(3D + 11) \times N$ flops per RMSNorm. After this we pass to the self-attention layer. Through self-attention we have 14 heads each containing three linear layers to compute each of the keys, queries and values. After calculating these, we add the rotary positional embeddings[7] to the queries and keys. Then we perform the matrix multiplication of $K^T Q$ followed by addition of the causal attention mask, adding a large negative number to each element in the upper right half of the attention matrix. We then divide by $\sqrt{D}$ and softmax the attention matrix along the columns. For each column of softmax we exponentiate every attention value costing us $10N$ FLOPs, then add them together, and divide each individual value by this sum. Repeating this calculation for each column gives $N \times (10N + N - 1 + N) = N \times (12N - 1)$ FLOPs to softmax the matrix. We then multiply the value and attention matrices. To recombine the 14 attention head results they are truncated and passed through a final linear layer. We then add the residual and pass through another RMSNorm. The MLP has two up-projection layers from dimension D to H (hidden dimension = 4864). The results of each up-projection are added and then projected back down to dimension D in a final linear layer followed by SiLU activation costing $13 \times DN$ as $\text{SiLU}(x) = \sigma(x)x$. Then we add another residual and we are finished with the transformer block. After 24 repetitions we pass our $D \times N$ matrix on to another RMSNorm followed by a linear layer projecting from the embedding space back to the vocabulary. Since we are not sampling, we skip the softmax here and just pick the token corresponding to the largest value.

3 LoRA

3.1 Default Configuration

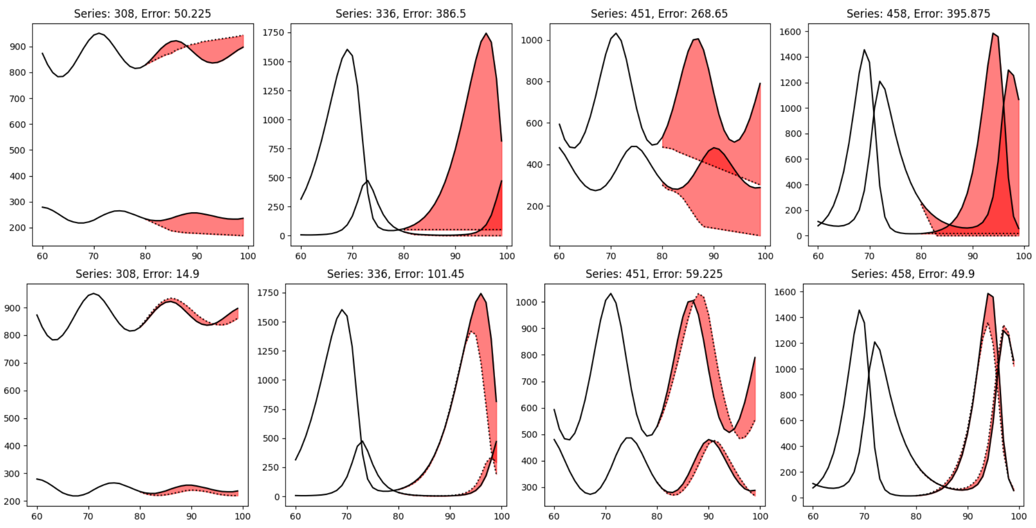

Figure 6: Comparison of forecasts generated by the base Qwen model (top) and the one tuned on default hyper-parameters (bottom).

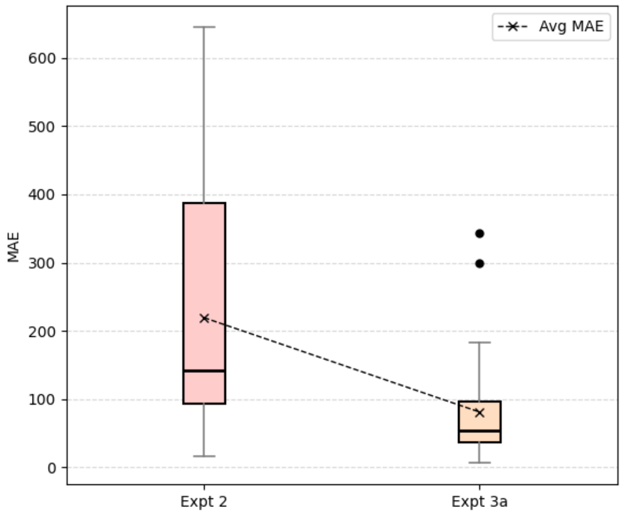

Figure 7: Distribution of MAEs on the reduced validation set for the base Qwen model (left) and the tuned model (right).

Due to the high cost per training run, we reduce the batch size from 4 to 2 in order to prioritize more optimizer steps. For the default run we train for 5000 steps with sequence length of 512, LoRA rank 4, and learning rate 1×10^{-5}. As seen in figure 6, LoRA is capable of improving the performance considerably. In figure 7 this improvement is clear. We more than halve the median MAE, taking the average MAE from 219.4 to 81.2. The training cost 7.39×10^{15} flops and the inference cost 1.95×10^{15}. In total, this stage used 9.34% of the budget.

3.2 Hyper-parameter Search

Name	Learning Rate	LoRA Rank
expt_0	1×10^{-05}	2
expt_1	1×10^{-05}	4
expt_2	1×10^{-05}	8
expt_3	5×10^{-05}	2
expt_4	5×10^{-05}	4
expt_5	5×10^{-05}	8
expt_6	1×10^{-04}	2
expt_7	1×10^{-04}	4
expt_8	1×10^{-04}	8

Table 1: Hyper-parameter configurations for the grid search.

To ensure security in our choice of which hyper-parameters to continue with, we perform a shallow grid search over all configurations shown in table 1, each model is trained to 3150 optimizer steps. Evaluation of all models is shown in figure 8. To cheaply infer which models to focus on, we initially evaluate each of the 9 models on the 5 worst performing series from the default parameter experiment. Seeing that experiment 6, 7, and 8 performed the best, we evaluate these three on 15 more series from the reduced validation set to explore their performance further. Model 6 and 7 have near identical MAEs of 69.3 and 69.0 respectively, we choose to continue with the parameters from experiment 6 as it has a lower median MAE. We note two promising takeaways from the search. Firstly, we achieved a lower MAE over the default run, and secondly, this result was obtained over fewer optimizer steps. Each of which are desired outcomes which support further experimentation to find the limits of LoRA's ability to improve forecasting accuracy.

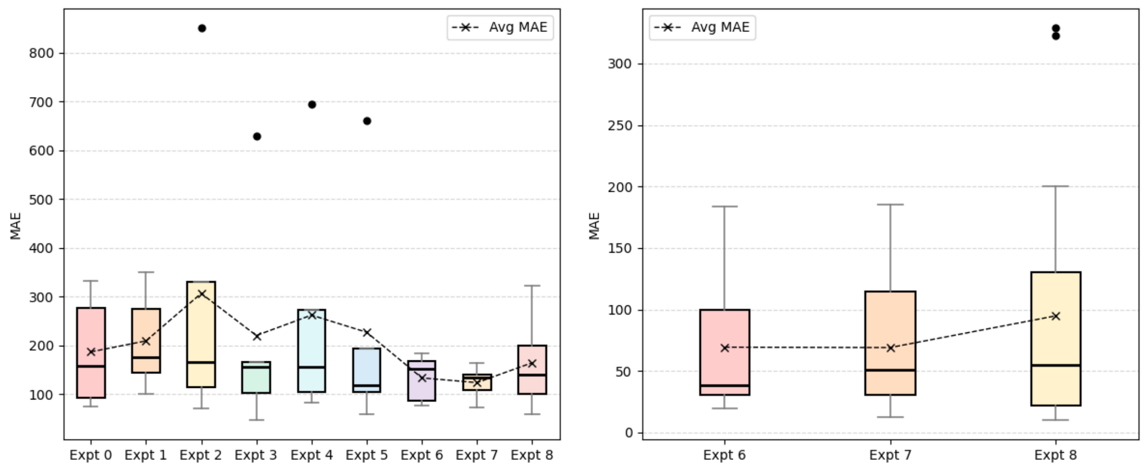

Figure 8: Left: MAE distributions of all 9 experiments in the grid search evaluated on the worst 5 series from the default parameter run. Right: MAE distributions of the best three experiments on the next 15 worst performing series in the validation subset of size 25.

The training for this search cost 4.19×10^{16} and inference cost 7.01×10^{15}, in total this set of experiments used 48.9% of the budget.

We subsequently train three models to 1500 optimizer steps using learning rate 1×10^{-4} and LoRA rank 2 to explore the effect of context length on model performance. Initial results are shown in table 2. Under the strict compute budget we find it more appropriate to train the final model with 512 sequence length since 768 doesn't provide a sufficient improvement to performance to warrant the 50% cost increase for computing each optimizer step or forecast. This second search cost in total 7.25×10^{15}.

Context Length	Avg MAE	Median MAE
128	2203.86	359.85
512	363.69	224.75
768	273.26	231.35

Table 2: Metrics from the experiments on context length.

3.3 Final Model

Up to this point we have used 69.43% of the budget. This lets us train our final model for 15400 optimizer steps and to evaluate this final model and the untuned Qwen on the test set to compare performance. See figure 9 for results. The average MAE has improved from 247.9 to 37.6.

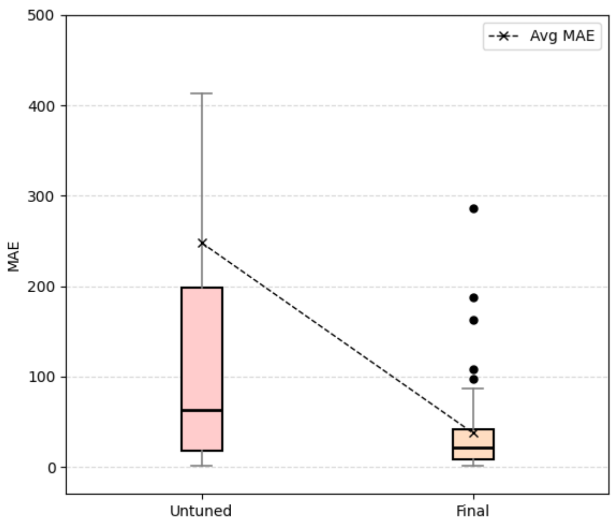

Figure 9: MAEs of the Untrained and Final model evaluated on the test set of 50 series. Cropped to exclude outliers of untrained model.

4 FLOP Review

Experiment	Training FLOPS	Inference FLOPS	Total FLOPS
Baseline	0	3.89×10^{15}	3.89×10^{15}
Default	7.39×10^{15}	1.95×10^{15}	9.34×10^{15}
First Parameter Search	4.19×10^{16}	7.01×10^{15}	4.89×10^{16}
Second Parameter Search	6.17×10^{15}	1.08×10^{15}	7.25×10^{15}
Final Model	2.28×10^{16}	7.78×10^{15}	3.05×10^{16}
All Experiments			9.99×10^{16}

Table 3: Table showing all FLOPs used.

References

[1] Transformers key-value caching explained. https://neptune.ai/blog/transformers-key-value-caching. Accessed: 2025-03-30.

[2] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023.

[3] Nate Gruver, Marc Finzi, Shikai Qiu, and Andrew Gordon Wilson. Large language models are zero-shot time series forecasters, 2024.

[4] 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, 2021.

[5] Zixuan Jiang, Jiaqi Gu, Hanqing Zhu, and David Z. Pan. Pre-rmsnorm and pre-crmsnorm transformers: Equivalent and efficient pre-ln transformers, 2023.

[6] 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, 2025.

[7] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2023.

[8] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.