M2 Deep Learning - Coursework Assignment

Raunaq Rai (rsr45@cam.ac.uk)
Data Intensive Science, Department of Physics, University of Cambridge

04 April, 2025

Word Count: XXX

Introduction

Large Language Models (LLMs) have revolutionised natural language processing and have recently shown promise in domains beyond text, including time-series forecasting [1]. In this coursework, we explore the application of LoRA (Low-Rank Adaptation) [2] to the Qwen2.5-Instruct model [3], a transformer-based LLM, for forecasting predator-prey population dynamics as described by the Lotka–Volterra equations.

Building on the methodology proposed in the LLMTIME framework [1], which adapts numeric sequences into a tokenizer-friendly format, we examine how Qwen2.5-Instruct can be repurposed for numerical prediction tasks. The objective is to fine-tune the model on a dataset of simulated predator-prey trajectories using LoRA, a parameter-efficient method that allows adaptation of pretrained models without updating all weights.

Our experiments are conducted under strict computational constraints, with a maximum allowable budget of 10^{17} floating point operations (FLOPS), encouraging efficient training practices. The provided dataset consists of 1,000 synthetic time series representing prey and predator population dynamics, formatted as 100 time steps per sequence with two variables per step.

This report outlines the implementation of LLMTIME preprocessing, baseline evaluation of the untrained model, LoRA fine-tuning with hyperparameter sweeps, and analysis of forecasting performance under compute-efficient constraints. Our findings highlight the potential of adapting LLMs for scientific time-series forecasting and provide recommendations for future work in compute-constrained settings.

1 Compute Constraint

This coursework promotes efficient experimentation under a strict compute budget of 10^{17} floating point operations (FLOPs) across all reported experiments. This constraint mirrors real-world limitations and encourages careful model design and evaluation, particularly in environments where access to large-scale compute is restricted. Under ideal utilisation, this budget equates to approximately five hours of GPU processing time on a MacBook M1 Pro.

FLOP Accounting Principles

FLOP costs are calculated using simplified, hardware-agnostic assumptions. Table 1 provides the per-operation FLOP estimates. For instance, multiplying an $m \times n$ matrix by an $n \times p$ matrix requires:

$$\text{FLOPs}_{\text{matmul}} = m \times p \times (2n - 1) \tag{1}$$

We assume backpropagation incurs double the cost of the forward pass, yielding:

$$\text{FLOPs}_{\text{train}} = 3 \times \text{FLOPs}_{\text{forward}} \tag{2}$$

Table 1: FLOPs Accounting for Primitive Operations

Operation	FLOPs
Addition / Subtraction / Negation (float or int)	1
Multiplication / Division / Inverse (float or int)	1
ReLU / Absolute Value	1
Exponentiation / Logarithm	10
Sine / Cosine / Square Root	10

FLOP Estimator Implementation

To systematically estimate the computational cost of each experiment, we implemented a modular FLOP estimator in Python. This estimator models a forward pass through the Qwen2.5-0.5B architecture at the operation level, breaking down FLOPs into five categories: additions, multiplications, divisions, exponentiations, and square roots.

The estimator accounts for every trainable and non-trainable component of the model. For a given sequence length n, the estimator computes the total number of FLOPs for each of the following:

- **Token Embeddings:** These are retrieved via indexing and incur no arithmetic FLOPs.
- **Positional Embeddings:** Sinusoidal encodings are added to token embeddings: $n \times d_{\text{model}}$ additions.
- **Multi-head Attention (per layer):**
  - *Query, Key, Value projections:* Linear transformations to d_{head} per head; includes multiplications and bias additions.
  - *Dot-product attention:* Computation of QK^T, scaling, softmax, and weighted sum with V.
  - *Softmax:* Includes exponentiations, summation, and division.
  - *Output projection:* Concatenation of heads followed by linear transformation.

- **Feedforward MLP (with SwiGLU) [4]:**
  - Two parallel up-projections (gated MLP) and a down-projection.
  - SwiGLU activation, combining SiLU and elementwise multiplication, is approximated as 22 FLOPs per hidden unit.
- **RMSNorm Layers [5]:**
  - Elementwise squaring, summation, root, division, and scaling by learned γ.
- **LoRA Projections (if used) [2]:**
  - Low-rank down and up-projections added to the frozen weights.
  - Includes multiplications, additions, and an elementwise residual connection.
- **Final Projection:**
  - Linear layer projecting to vocabulary logits: $n \times d_{\text{model}} \times V$ multiplications and additions.

These per-layer estimates are summed across all 24 transformer blocks, and added to the overhead of input/output layers. FLOPs for different sequence lengths, LoRA ranks, and model dimensions are configurable through function arguments, enabling detailed accounting for any experimental configuration.

Estimating Backward Pass Cost

The exact computation of FLOPs for backpropagation is non-trivial and implementation-specific. In practice, the backward pass includes:

- Derivatives for every parameter (including gradients for each tensor)
- Multiplications and additions for each chain rule application
- Memory reuse optimisations depending on the framework

For the purposes of this coursework, we assume that the backward pass is approximately double the cost of the forward pass. This is a conservative estimate, as it does not account for optimisations such as gradient checkpointing or memory reuse.

- Each forward operation has a corresponding gradient computation (typically with similar or higher cost).
- Additional operations such as gradient accumulation, weight updates (e.g. for LoRA), and some recomputation of activations.

Thus, the total training FLOPs are given by:

$$\text{FLOPs}_{\text{train}} = \text{FLOPs}_{\text{forward}} + \text{FLOPs}_{\text{backward}} + \text{FLOPs}_{\text{update}} \approx 3 \times \text{FLOPs}_{\text{forward}} \quad (3)$$

2 Baseline: LLMTIME Preprocessing Implementation

The first task was to implement the LLMTIME preprocessing scheme [1] to convert multivariate time-series data into a format suitable for Qwen2.5-Instruct. This was done by creating a dedicated Python module `src/preprocessor.py` containing a class `LLMTIMEPreprocessor`, which formats and tokenizes time-series data according to the scheme described in the coursework.

Scaling and Formatting

Each sample consists of a pair of sequences: prey and predator population values over time. These values often differ significantly in scale, both between and within samples, which can lead to inconsistencies in token length and hinder model generalisation if left unprocessed.

To address this, a per-sample scaling factor α is computed to normalise the numerical range. Specifically, α is chosen as:

$$\alpha = \max\left(\text{percentile}_{95}(\text{prey}),\ \text{percentile}_{95}(\text{predator})\right)/10 \tag{4}$$

This dynamic, data-dependent approach ensures that most values fall within a range of approximately $[0, 10]$, regardless of the absolute scale of the original input. By avoiding fixed global scaling, the model remains robust to distributional shifts across samples.

Each value is then scaled by α and rounded to a fixed number of decimal places (default: two). This rounding ensures token consistency and limits the number of unique tokens generated by the tokenizer, improving model efficiency and reducing overfitting to numeric noise.

LLMTIME Encoding

Following the LLMTIME convention, each timestep is represented as a pair of variables (prey, predator), separated by a comma. Consecutive timesteps are separated by a semicolon. This ensures clarity for tokenization and decoding. For example, a sequence of three timesteps would be formatted as:

```
0.25,1.50;0.27,1.47;0.31,1.42
```

While the original LLMTIME implementation by Gruver et al. [1] removes decimal points to reduce sequence length—especially for GPT-style models that tokenize numbers into subword units—we retain the decimal point in our implementation for several important reasons:

- Qwen2.5's tokenizer already tokenizes numeric strings into digit-level tokens without requiring digit spacing or decimal removal.
- Removing the decimal point would introduce additional digits and increase sequence length without benefit.
- Preserving the decimal enhances human interpretability and simplifies decoding during inference.
- The coursework specification explicitly recommends retaining the decimal point and avoiding space-separated digit encoding.

Tokenization

Once formatted, the numeric string is passed through Qwen2.5's tokenizer using Hugging Face's `AutoTokenizer` interface. Each digit and punctuation mark is tokenized into its own token. For example:

```
tokenizer("1.23", return_tensors="pt")["input_ids"].tolist()[0]
```

yields:

```
[16, 13, 17, 18]
```

confirming tokenization into digit and punctuation tokens.

Example Sequences and Tokenization

We illustrate the LLMTIME preprocessing pipeline with two example predator-prey sequences. For each, we show:

- the original prey and predator population values (first 5 timesteps),
- the scaled and formatted text string,
- the tokenized output (Qwen2.5 input IDs),
- the computed scale factor α.

Example 1.

- **Original Input:**

  ```
  Prey:     [2.9, 3.2, 3.8, 4.5, 5.1]
  Predator: [1.1, 0.9, 0.7, 0.6, 0.5]
  ```

- **Scale Factor α:**

  ```
  0.498
  ```

- **Formatted Sequence:**

  ```
  5.82,2.21;6.43,1.81;7.63,1.41;9.04,1.20;10.24,1.00
  ```

- **Tokenized Output (Qwen2.5 input IDs):**

  ```
  [20, 13, 23, 17, 11, 17, 13, 17, 16, 26, 21, 13, 19, 18, 11,
  16, 13, 23, 16, 26, 22, 13, 21, 18, 11, 16, 13, 19, 16, 26,
  24, 13, 15, 19, 11, 16, 13, 17, 15, 26, 16, 15, 13, 17, 19,
  11, 16, 13, 15, 15]
  ```

Example 2.

- **Original Input:**

```
Prey:      [1.5, 1.8, 2.1, 2.4, 2.7]
Predator: [2.8, 2.5, 2.2, 1.9, 1.6]
```

- **Scale Factor α:**

```
0.274
```

- **Formatted Sequence:**

```
5.47,10.22;6.57,9.12;7.66,8.03;8.76,6.93;9.85,5.84
```

- **Tokenized Output (Qwen2.5 input IDs):**

```
[20, 13, 19, 22, 11, 16, 15, 13, 17, 17, 26, 21, 13, 20, 22,
11, 24, 13, 16, 17, 26, 22, 13, 21, 21, 11, 23, 13, 15, 18,
26, 23, 13, 22, 21, 11, 21, 13, 24, 18, 26, 24, 13, 23, 20,
11, 20, 13, 23, 19]
```

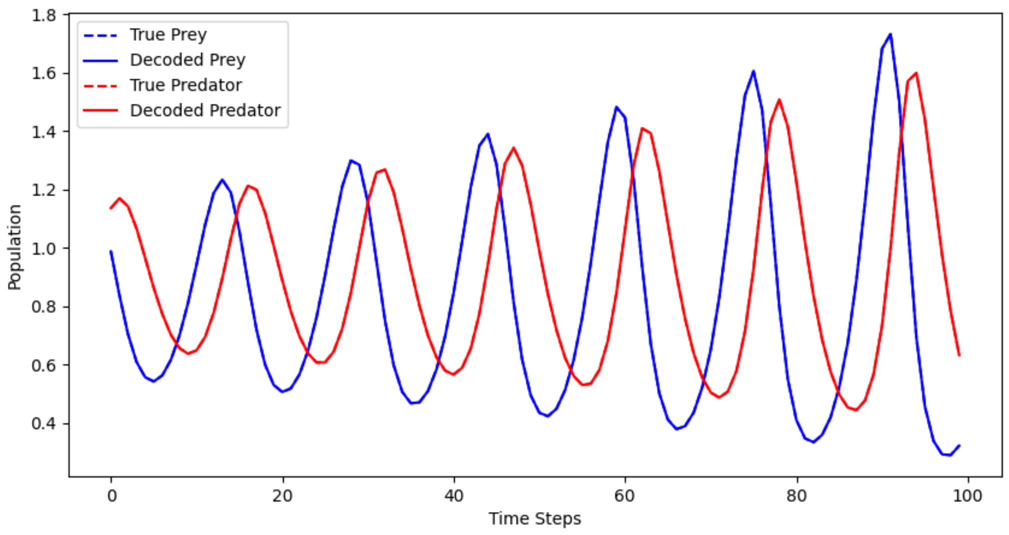

Figure 1: Decoded output vs. ground truth for the 972nd Lotka-Volterra trajectory in the test set. This sequence was passed through the LLMTIME preprocessing pipeline and fed into the untrained Qwen2.5-Instruct model. The resulting output was decoded and rescaled using the same scale factor α applied during encoding. The close alignment between original and decoded values confirms that the preprocessing, tokenization, and decoding pipeline is working as intended.

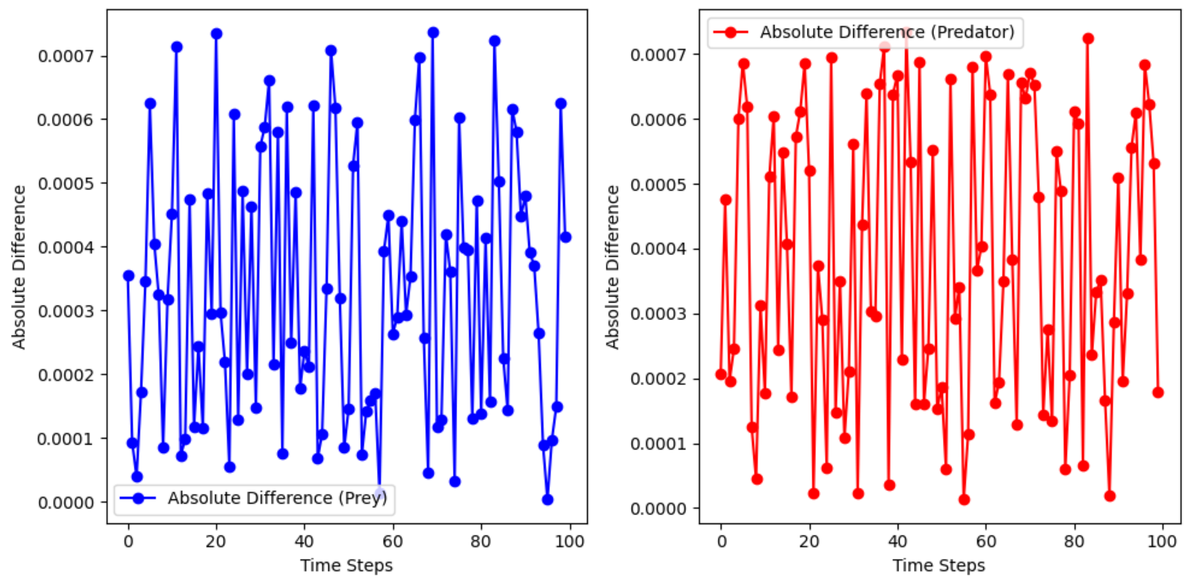

Figure 2: Absolute differences between true and decoded prey (left) and predator (right) values for the 972nd test sequence. These errors quantify the deviation introduced by the LLMTIME encoding-decoding pipeline. The low magnitude of differences confirms the correctness of the numeric formatting, tokenizer compatibility, and inverse decoding process, independent of model training.

3 Baseline Evaluation: Forecasting with Untrained Qwen2.5-Instruct

We evaluate the zero-shot forecasting capability of the untrained `Qwen2.5-0.5B-Instruct` model using one example from the dataset (Sample ID 972). The first 50 timesteps are provided as input, and the model is tasked with generating predictions for the remaining 50 steps using the LLMTIME token format.

Language Model Generation and Forward Pass Estimation

Generation is performed using the Hugging Face `generate()` API, which operates autoregressively—predicting one token at a time, where each new token requires a full forward pass over the model. The input sequence is constructed from 50 timesteps (each a prey-predator pair), formatted as `5.38,1.44;...`. In our run, this input tokenized to **499 tokens**.

To generate 50 new timesteps, we first call `generate(..., max_new_tokens=500)`. Because each timestep typically requires around 10 tokens, this initial attempt may not reach the desired 50-step output. To monitor progress, we count the number of semicolon-separated pairs in the output after each block of generation. If fewer than 100 semicolons (original + generated) are found, we call `generate(..., max_new_tokens=20)` repeatedly until the desired length is reached or we hit a limit.

In this particular run, the model generated a total of **520 new tokens** to complete 100 timesteps. Since each new token corresponds to one forward pass (under causal decoding), this implies **520 forward passes** were performed on top of the original 499-token input context.

FLOPs Estimation: To estimate computational cost, we use the Qwen2.5 architecture's FLOP profile via the `forwards_pass_flops()` utility. Based on our input length of 499 tokens, the model uses approximately **5.48e11 FLOPs per token** during inference. Therefore, generating 520 tokens requires:

$$\text{FLOPs} = 520 \times 5.48 \times 10^{11} = \mathbf{2.85} \times 10^{14}$$

This estimate assumes a compute efficiency of 1 and does not account for memory bandwidth, activation caching, or I/O overhead.

Forecasting Performance

We compare the predicted output (post-decoding and re-scaling) to the true population values for Sample ID 972 using standard metrics:

- **Prey:**
  - Mean Squared Error (MSE): 0.1770
  - Mean Absolute Error (MAE): 0.2485
  - R^2 Score: -0.2812
- **Predator:**
  - Mean Squared Error (MSE): 0.2337
  - Mean Absolute Error (MAE): 0.2793
  - R^2 Score: -1.6260

The prey trajectory shows moderate alignment, albeit with oversmoothing. The predator prediction, however, is unstable, with a negative R^2 indicating performance worse than simply predicting the mean. This reflects the model's lack of understanding of the underlying dynamics without domain-specific training.

Non-Determinism in Generation

It's important to note that generation is inherently non-deterministic unless a seed is set. This means different runs on the same input can yield very different outputs—some closer to ground truth, others worse. This behaviour arises from sampling strategies like temperature scaling or nucleus sampling used during decoding.

Compute Assumptions and Efficiency

In this coursework, we assume a compute efficiency of 1, meaning all estimated FLOPs are fully utilised during inference without memory or runtime optimisations. Some large-scale language models, such as GPT-4, employ Mixture-of-Experts (MoE) architectures to reduce active parameters per forward pass—this lowers compute cost by activating only a subset of layers during token generation. However, `Qwen2.5-0.5B-Instruct` does not employ MoE or dynamic routing, meaning that **all model parameters are active for every forward pass**.

Recent innovations such as SkipDecode [6] propose early-exit strategies that skip computation in lower layers during autoregressive decoding. This approach can significantly reduce inference FLOPs while preserving performance, and is compatible with batching and KV cache reuse. Nonetheless, `Qwen2.5-0.5B-Instruct` does not implement such mechanisms, and so the full model is evaluated on every token without layer skipping or selective computation.

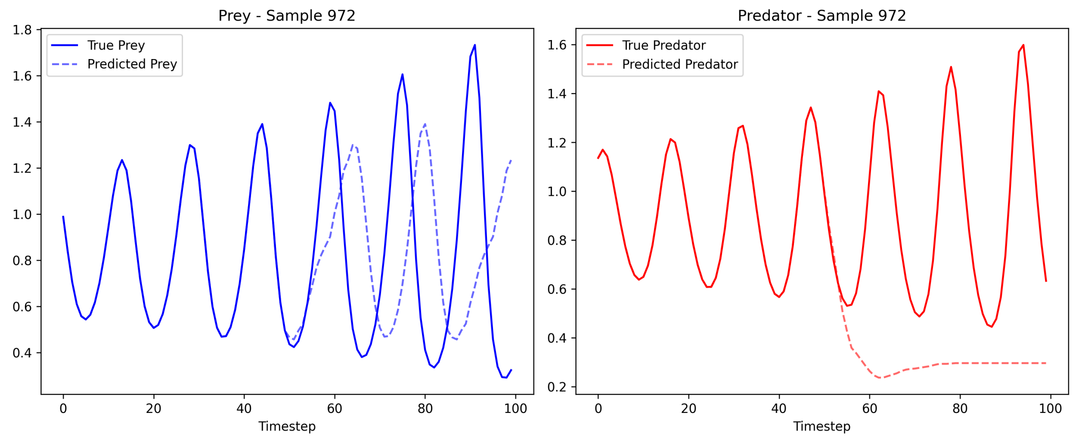

Figure 3: Forecasting output of the untrained Qwen2.5-Instruct model on Sample ID 972. The model is prompted with the first 50 timesteps and generates the next 50. Dashed lines show true values; solid lines show decoded predictions.

The untrained Qwen2.5-Instruct model exhibits some capacity to mimic the structure of the prey trajectory under LLMTIME formatting, likely due to its strong inductive bias and learned text patterns. However, it struggles with the more complex predator dynamics, which results in erratic predictions. This reinforces the need for fine-tuning or domain-specific adaptation when applying language models to structured forecasting problems.

4 FLOP Accounting for Transformer Components

To estimate the total FLOPs consumed by the Qwen2.5-0.5B-Instruct model, we explicitly mapped every major component in the forward pass to primitive arithmetic operations and computed their FLOP costs using the provided FLOPs table (Table 1). Below we summarise the decomposition of each operation.

Token and Positional Embeddings

- **Token Embeddings:** Retrieved via table lookup — **0 FLOPs**.
- **Positional Embedding Addition:** $n \times d_{\text{model}}$ additions.

RMSNorm (applied before attention, after attention, and before MLP)

For each token:

- Square each element: $n \times d$ multiplications.

- Sum across dimension: $n \times (d-1)$ additions.
- Square root: n square root operations.
- Divide each element by norm: $n \times d$ divisions.
- Scale with learned weight: $n \times d$ multiplications.

Total: $n \times (2d)$ multiplications, $n \times (d-1)$ additions, $n + n \times d$ divisions, n square roots.

Multi-Head Attention (per layer)

Let h be the number of heads and $d_h = d/h$.

- **Q/K/V Projections:** $3 \times n \times d \times d_h$ multiplications and $(3 \times n \times (d-1) \times d_h)$ additions.
- **Attention Scores:** $n \times n \times d_h$ multiplications and $n \times n \times (d_h - 1)$ additions.
- **Softmax:** For n^2 elements:
  - Exponentiation: n^2
  - Summation: $n \times (n-1)$ additions
  - Normalisation: n^2 divisions
- **Softmax-Value Multiplication:** $n \times n \times d_h$ multiplications and $n \times d_h \times (n-1)$ additions.
- **Concatenation:** Memory operation — **0 FLOPs**.
- **Final Output Projection:** $n \times d \times d$ multiplications and $n \times (d-1) \times d$ additions.

MLP Block with SwiGLU (per layer)

Let d_{ff} be the hidden dimension (e.g., 4864).

- **Up-Projection and Gating:** $2 \times n \times d \times d_{\text{ff}}$ multiplications and $2 \times n \times (d-1) \times d_{\text{ff}}$ additions.
- **Down-Projection:** $n \times d_{\text{ff}} \times d$ multiplications and $n \times (d_{\text{ff}} - 1) \times d$ additions.
- **SwiGLU Activation:**
  - SiLU requires: 1 exp, 1 div, 2 mul, 1 add per unit.
  - Total: $n \times d_{\text{ff}}$ exponentiations, divisions, additions, and $2 \times n \times d_{\text{ff}}$ multiplications.

Final Projection to Vocabulary Logits

- **Linear Projection:** $n \times d \times V$ multiplications and $n \times (d-1) \times V$ additions.

LoRA (if enabled)

For each adapted projection (Q and V):

- **Down-projection:** $n \times d \times r$ multiplications and $n \times (d-1) \times r$ additions.
- **Up-projection:** $n \times r \times d$ multiplications and $n \times (r-1) \times d$ additions.
- **Scaling + Residual Addition:** $n \times d$ multiplications and additions.

Total for one LoRA adapter: $2\times$ (down + up) + residual = $(2 \times n \times d \times r + n \times d)$ mults and similar additions.

Summary

Each primitive operation has the following FLOP cost:

- Addition / Subtraction / Negation: **1**
- Multiplication / Division / Inverse: **1**
- ReLU / Absolute Value: **1**
- Exponentiation / Logarithm: **10**
- Square Root / Sine / Cosine: **10**

Using these constants, the total FLOPs can be calculated with:

$$\text{Total FLOPs} = \sum_{i=1}^{n} (a_i \cdot 1 + m_i \cdot 1 + d_i \cdot 1 + e_i \cdot 10 + s_i \cdot 10) \tag{5}$$

where a_i, m_i, d_i, e_i, and s_i represent the counts of additions, multiplications, divisions, exponentiations, and square roots respectively for each model component.

The complete implementation of this accounting is automated in our Python function `forwards_pass_flops()`, which accepts parameters such as number of tokens, LoRA rank, and returns per-operation and total FLOPs.

References

[1] Nate Gruver, Marc Finzi, Shikai Qiu, and Andrew Gordon Wilson. Large language models are zero-shot time series forecasters. *arXiv preprint arXiv:2310.07820*, 2023. URL `https://doi.org/10.48550/arXiv.2310.07820`. NeurIPS 2023.

[2] 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. *arXiv preprint arXiv:2106.09685*, 2021. URL `https://arxiv.org/abs/2106.09685`. Draft v2 includes better baselines, experiments on GLUE, and more on adapter latency.

[3] 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. *arXiv preprint arXiv:2412.15115*, 2024. URL `https://doi.org/10.48550/arXiv.2412.15115`. Additional authors not shown.

[4] Noam Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020. URL `https://doi.org/10.48550/arXiv.2002.05202`.

[5] Biao Zhang and Rico Sennrich. Root mean square layer normalization. *arXiv preprint arXiv:1910.07467*, 2019. URL `https://doi.org/10.48550/arXiv.1910.07467`. NeurIPS 2019.

[6] Luciano Del Corro, Allie Del Giorno, Sahaj Agarwal, Bin Yu, Ahmed Awadallah, and Subhabrata Mukherjee. Skipdecode: Autoregressive skip decoding with batching and caching for efficient llm inference. *arXiv preprint arXiv:2307.02628*, 2023. URL `https://doi.org/10.48550/arXiv.2307.02628`.