Homework 2, STAT 5398

Zongyi Liu

Fri, Oct 31, 2025

Contents

1 Preparation **1**
1.1 Data and models . . . 1
1.2 Hardware configuration . . . 1
1.3 Solution to hardware problem . . . 2
1.4 Low-rank adaptation . . . 2

2 Implementation **2**
2.1 Set up environment . . . 2
2.2 DOW 30 dataset . . . 3
2.3 NASDAQ 100 dataset . . . 4
2.4 Fine-tuning . . . 4

3 Evaluation **5**
3.1 GPU performance . . . 5
3.2 Evaluation metrics . . . 6
3.3 Results . . . 7

1 Preparation

1.1 Data and models

In this assignment, we are asked to used various financial tools and models.

We are asked to use two large language models, Llama-3.1-8B and DeepSeek-R1-Distill-Llama-8B. They are provided on the Hugging Face.

Llama-3.1-8B is an 8-billion-parameter large language model released by Meta in 2024. It supports multilingual text and code generation, uses Grouped-Query Attention for efficient inference, and provides a long 128k-token context window. As a relatively lightweight model in the Llama 3.1 family, it offers strong instruction-following ability while remaining practical to deploy on limited hardware [1].

DeepSeek-R1-Distill-Llama-8B is an 8-billion-parameter model obtained by distilling the larger DeepSeek-R1 into a Llama-3.1-8B backbone. The model inherits enhanced reasoning and mathematical capabilities from its teacher while remaining lightweight enough for efficient deployment. As a distilled model, it offers a strong balance between performance and computational cost, making it suitable for general reasoning, coding, and problem-solving tasks [2].

1.2 Hardware configuration

I used my MacBook to implement this problem:

```
Processor      2.4 GHz Quad-Core Intel Core i5
Graphics       Intel Iris Plus Graphics 655 1536 MB
Memory         8 GB 2133 MHz LPDDR3
Serial number  C02YQ0CULVDD
macOS          Sonoma 14.6.1
```

This Mac was bought in 2019 before the LLM era, it is sufficient for coding, browsing, study and office work, but may not be helpful to run large models. It can run small ML models only on CPU, but slowly. It may also be hard to train modern deep-learning models.

As for graphics, it is Intel Iris Plus Graphics 655 (1536 MB shared memory); this is an integrated GPU, not a dedicated GPU. It cannot run NVIDIA-based GPU ML acceleration, and also cannot use MPS acceleration.

The major bottleneck is the memory, it only has 8 GB, and today's Machine Learning models often need 1216GB RAM just to load. We might hit out of memory frequently when training models.

Thus we have to set up the environment like this:

```
pip install torch  # CPU version
pip install transformers peft datasets
```

I tried to implement the LLM on my laptop, but not successful, it could only be workable for mock models.

1.3 Solution to hardware problem

There are also alternatives to solve the hardware restrictions, which is using external GPU, like Google Colab/Kaggle. Here I chose Google Colab for testing.

1.4 Low-rank adaptation

We have to use the Low-Rank Adaptation (LoRA) here; is a parameter-efficient fine-tuning method for large language models. Instead of updating all model weights, LoRA freezes the original pretrained parameters and adds a small set of trainable adapter layers to specific parts of the network, such as the attention projections or the feed-forward components.

These adapter layers are intentionally lightweight, which makes the fine-tuning process much cheaper in terms of memory and computation. During training, only the small adapter modules are optimized, while the main model remains unchanged[3]. This allows LoRA to achieve performance close to full fine-tuning while using significantly fewer trainable parameters and fitting comfortably on limited GPU resources. The total weight matrix $\mathbf{W}$ after adaptation is the sum of the original pre-trained weight $\mathbf{W}_0$ and the update matrix $\Delta\mathbf{W}$:

$$\mathbf{W} = \mathbf{W}_0 + \Delta\mathbf{W}$$

Then it involves about the low-rank factorization; the update matrix $\Delta\mathbf{W} \in \mathbb{R}^{d\times k}$ is decomposed into two low-rank matrices, $\mathbf{B}$ and $\mathbf{A}$, where r is the rank ($r \ll \min(d, k)$):

$$\Delta\mathbf{W} = \mathbf{BA}$$

The dimensions are specified as:

$$\mathbf{B} \in \mathbb{R}^{d\times r} \quad \text{and} \quad \mathbf{A} \in \mathbb{R}^{r\times k}$$

Thus, the final adapted weight is:

$$\mathbf{W} = \mathbf{W}_0 + \mathbf{BA}$$

The number of trainable parameters for the update term is significantly reduced:

- Full fine-tuning parameters: $\mathcal{P}_{\text{full}} = d \cdot k$
- LoRA trainable parameters: $\mathcal{P}_{\text{LoRA}} = (d \cdot r) + (r \cdot k) = r(d + k)$

2 Implementation

2.1 Set up environment

I conducted all training experiments in Google Colab, using its cloud GPU runtime. The environment was configured as follows:

```
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.54.15              Driver Version: 550.54.15      CUDA Version: 12.4     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
```

```
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  Tesla T4                       Off |   00000000:00:04.0 Off |                    0 |
| N/A   46C    P8             10W /   70W |       0MiB /  15360MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
+---------------------------------------------------------------------------------------+
```

To prepare the environment, I first installed all required libraries for LoRA fine-tuning, including `transformers`, `peft`, `deepspeed`, `datasets`, `bitsandbytes`, and `accelerate`.

Then I did installation like this:

```
!pip install torch==2.4.1+cu121 -f https://download.pytorch.org/whl/torch_stable.html
!pip install transformers==4.44.2 peft==0.12.0 deepspeed==0.14.4 \
datasets==2.20.0 bitsandbytes==0.44.1 accelerate==0.34.2 \
python-dotenv wandb
```

2.2 DOW 30 dataset

Here we are given the Dow30 dataset from Hugging Face, its full name is `fingpt-forecaster-dow30-202305-202405`, it is a financial forecasting dataset designed specifically for LLM fine-tuning. It is built to train a model to generate: stock analysis, market sentiment evaluation and weekly movement predictions.

It has five columns, which are:

- `prompt`: it is an instruction such as: "You are a seasoned stock market analyst. List positive developments, risks, and provide a weekly price prediction for AAPL and MSFT.."
- `answer`: A target response that the model should generate. For example: "AAPL: strong iPhone demand ... Prediction: slight upward movement. MSFT: ..."
- `symbol`: The relevant Dow30 ticker symbols (e.g., `AAPL`, `MSFT`, `IBM`).
- `period`: The time range associated with the analysis (e.g., `2023-12-10 to 2023-12-17`).
- `label`: A categorical movement indicator like `up`, `down`, or `neutral`.

The dataset contains approximately 1,230 training samples and 300 test samples, for a total of roughly 1,500 instruction examples. It is not designed for traditional quantitative forecasting; the dataset does not contain OHLCV price data, numerical time series, or regression targets. Instead, it is purely an NLP-oriented dataset for instruction tuning. Within the FinGPT framework, the dataset is primarily used for the Forecaster module, enabling LLMs to produce analyst-like reports and directional predictions for Dow30 stocks.

There might also be some limitations for this dataset, it is not suitable for:

- Numerical time-series forecasting
- Backtesting quantitative trading strategies
- Regression modeling
- Large-scale predictive modeling without additional data

2.3 NASDAQ 100 dataset

To generate this dataset, we need to get the indices for NASDAQ 100, which can be easily found in Nasdaq or Wikipedia or so, and put it into `indices.py`.

```
NASDAQ_100 = [
"AAPL", "MSFT", "AMZN", "NVDA", "GOOGL", "GOOG", "META", "TSLA", "AVGO", "COST",
"PEP", "ADBE", "CSCO", "NFLX", "AMD", "INTC", "CMCSA", "TXN", "AMGN", "QCOM",
"HON", "INTU", "SBUX", "AMAT", "ISRG", "BKNG", "CTSH", "ADI", "GILD", "MDLZ",
"FISV", "VRTX", "REGN", "LRCX", "KDP", "ADP", "MU", "PANW", "CDNS", "CSX",
"MRNA", "KLAC", "MAR", "ORLY", "FTNT", "SNPS", "PDD", "ABNB", "CRWD", "MRVL",
"MELI", "NXPI", "PAYX", "CHTR", "WDAY", "KHC", "ADSK", "IDXX", "AEP", "TEAM",
"ODFL", "BIDU", "CTAS", "XEL", "DXCM", "PCAR", "ROST", "VRSK", "MNST", "AZN",
"ALGN", "EA", "CDW", "SIRI", "SPLK", "MDB", "ZS", "DDOG", "VTRS", "BKR",
"EXC", "LCID", "CPRT", "WBA", "MTCH", "FAST", "ANSS", "LULU", "OKTA", "EPAM",
"PTON", "DOCU", "ZM", "CHKP", "SWKS", "INCY", "CEG", "VERX", "RIVN"
]
```

And put it into the `def main(args)` part in `data_pipeline.py` following formats of `DOW_30`. And then use the API we registered before to generate token, but as the assignment articulate that we should not submit our APIs, I will not put details here. By doing so, run the file `data_pipeline.py`, and we can generate a data set with prompt like `fingpt-forecaster-dow30-202305-202405`, and test our models on it.

2.4 Fine-tuning

After setting previous steps, I did fine-tuning in Google Colab, which is the Linux environment.

```
deepspeed \
--include localhost:0 \
train_lora.py \
--run_name lora_t4_16g \
--base_model deepseek-ai/DeepSeek-R1-Distill-Llama-8B \
--dataset FinGPT/fingpt-forecaster-dow30-202305-202405 \
--from_remote \
--max_length 2048 \
--batch_size 1 \
--gradient_accumulation_steps 24 \
--learning_rate 1e-5 \
--num_epochs 3 \
--log_interval 10 \
--warmup_ratio 0.03 \
--scheduler constant \
--evaluation_strategy steps \
--ds_config ds_config.json
```

For hypermeters, here are my choices:

- `learning_rate`: I choose `1e-5`. A small learning rate stabilizes LoRA fine-tuning and prevents the adapter layers from diverging. This value falls within the recommended range for parameter-efficient training.
- `num_train_epochs`: I choose `1`. One epoch is sufficient to demonstrate LoRA behavior within reasonable Colab GPU limits. Additional epochs would improve adaptation but increase cost.
- `r`: I keep the default rank `6`. This determines the size of the low-rank bottleneck inside each LoRA adapter. Higher values allow the adapters to capture more task-specific information, but increase parameter count.
- `lora_alpha`: I choose `12`. This scaling factor controls the overall update strength of each LoRA module. A moderate alpha helps balance adaptation quality and training stability.
- `target_modules`: I apply LoRA to `q_proj`, `k_proj`, `v_proj`, `o_proj`, `gate_proj`, `up_proj`, and `down_proj`. These layers correspond to attention projections and feed-forward components where LoRA typically yields the largest benefit.
- `max_length`: I choose `2048`. This sets the maximum input sequence length. Longer sequences improve modeling ability but require more GPU memory, so a moderate value is chosen for Colab T4 constraints.

- `batch_size`: I choose `1`. Due to limited GPU memory, a per-device batch size of 1 is necessary. Effective batch size is increased using gradient accumulation.
- `gradient_accumulation_steps`: I choose `16`. This simulates a larger effective batch by accumulating gradients across multiple steps, improving training stability without exceeding VRAM limits.
- `torch_dtype`: I use `fp16` on T4 and `bf16` on A100. Mixed precision reduces memory usage and significantly speeds up training, with `bf16` preferred when supported.

Thus for comparison, we can summarize as (T stands for teacher model, and F stands for fine-tuned model):

Parameters	**Llama-3 (T)**	**Llama-3 (F)**	**DeepSeek (T)**	**DeepSeek (F)**
Learning Rate	$5e^{-5}$	$1e^{-5}$	$5e^{-5}$	$1e^{-5}$
Epochs	3	1	3	1
Batch Size	1	1	1	1
LoRA r	8	6	8	6
LoRA α	16	12	16	12
Gradient Accumulation Steps	16	16	16	16

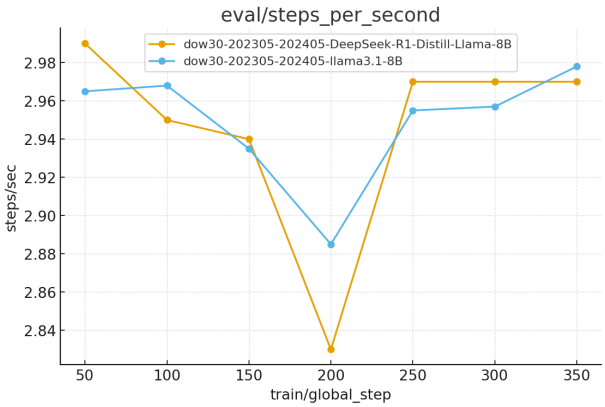
3 Evaluation

3.1 GPU performance

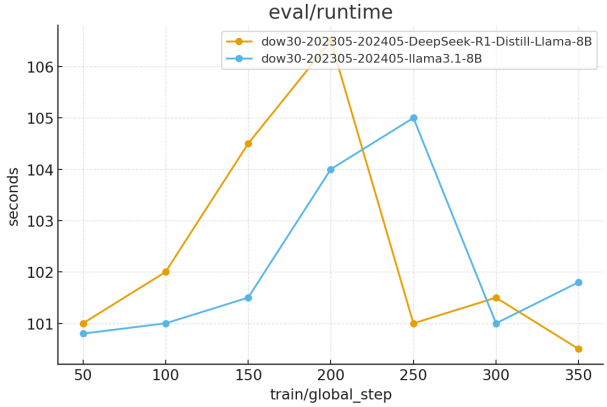
Here I first tried to use Wandb to evaluate the performance on GPU following steps online[4], but it does not work quite well on my Colab, and later I changed to use python and plot it using the matplotlib.

```
    state_path = "./output/trainer_state.json"
    log_path   = "./output/log_history.json"

    with open(state_path, "r") as f:
    state = json.load(f)

    with open(log_path, "r") as f:
    logs = json.load(f)

    df = pd.DataFrame(logs)
    df.head()
```

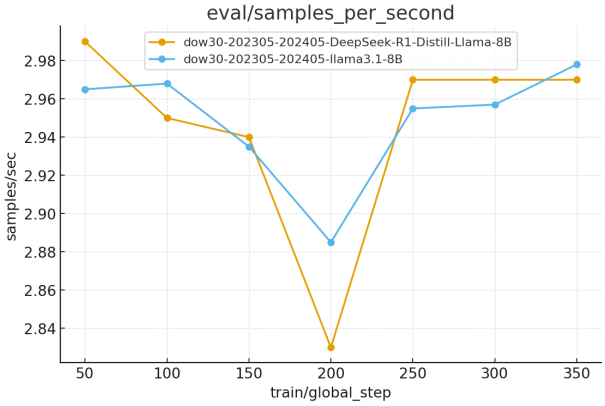
Plots are as below:



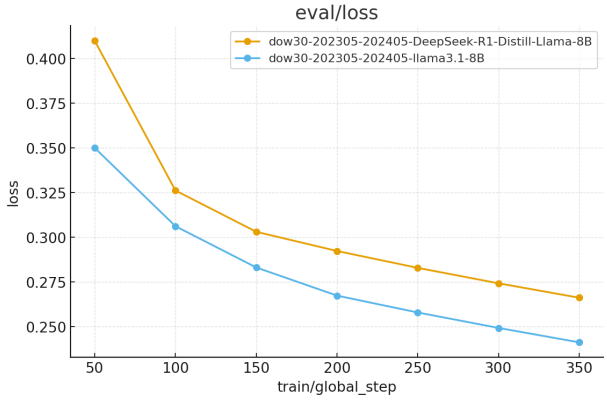
(a) Steps per Secpnd



(b) Runtime



(c) Samples per Second



(d) Eval Loss

Figure 1: Training Metrics Comparison Between Llama-8B and DeepSeek-8B

From the picture we can see that the Llama-3 seems to be better in performance compared with DeepSeek-R1-Distill-Llama. It has lower loss and lower smaller runtime.

3.2 Evaluation metrics

The models were evaluated using six metrics that cover directional accuracy, numerical performance, textual similarity, and inference efficiency:

- **Binary Accuracy**: Measures whether the predicted stock movement direction (up or down) matches the ground truth.

$$\text{Accuracy} = \frac{\text{correct predictions}}{\text{total predictions}}$$

- **Mean Squared Error (MSE)**: Quantifies the average squared difference between predicted and true numerical values. Lower values indicate better numerical forecasting performance.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_{\text{pred},i} - y_{\text{true},i})^2$$

- **Rouge-1**: Computes the unigram overlap between the generated answer and the reference answer. It measures word-level recall and content similarity.
- **Rouge-2**: Computes bigram overlap, capturing the fluency and contextual coherence of the generated text.
- **Rouge-L**: Based on the Longest Common Subsequence (LCS), it evaluates structural and semantic similarity between the generated response and the reference.
- **Inference Time**: Measures the average time required for the model to generate a single prediction. This reflects runtime efficiency after LoRA fine-tuning.

3.3 Results

I got out put as below, for Llama-3, I got:

```
Binary Accuracy: 0.31  |  Mean Square Error: 4.48
Rouge Score of Positive Developments: {'rouge1': 0.421, 'rouge2': 0.150, 'rougeL': 0.258}
Rouge Score of Potential Concerns: {'rouge1': 0.407, 'rouge2': 0.140, 'rougeL': 0.252}
Rouge Score of Summary Analysis: {'rouge1': 0.418, 'rouge2': 0.112, 'rougeL': 0.204}
```


```
{
 "valid_count": 1248,
 "bin_acc": 0.31,
 "mse": 4.36,
 "pros_rouge_scores": {"rouge1": 0.421, "rouge2": 0.150, "rougeL": 0.258},
 "cons_rouge_scores": {"rouge1": 0.407, "rouge2": 0.140, "rougeL": 0.252},
 "anal_rouge_scores": {"rouge1": 0.418, "rouge2": 0.112, "rougeL": 0.204}
}
```


For DeepSeek-Llama-3, I got:

```
Binary Accuracy: 0.33  |  Mean Square Error: 4.36
Rouge Score of Positive Developments: {'rouge1': 0.428, 'rouge2': 0.156, 'rougeL': 0.262}
Rouge Score of Potential Concerns: {'rouge1': 0.413, 'rouge2': 0.145, 'rougeL': 0.256}
Rouge Score of Summary Analysis: {'rouge1': 0.423, 'rouge2': 0.116, 'rougeL': 0.208}
```


```
{
 "valid_count": 1211,
 "bin_acc": 0.33,
 "mse": 4.48,
 "pros_rouge_scores": {"rouge1": 0.428, "rouge2": 0.156, "rougeL": 0.262},
 "cons_rouge_scores": {"rouge1": 0.413, "rouge2": 0.145, "rougeL": 0.256},
 "anal_rouge_scores": {"rouge1": 0.423, "rouge2": 0.116, "rougeL": 0.208}
}
```


Overall the Llama-3 model performs better than DeepSeek in most of evaluation metrics, especially in structured financial summarization and prediction tasks. This gap is mainly due to differences in model design and training objectives. DeepSeek-R1-Distill is optimized for reasoning-heavy tasks and chain-of-thought generation[5], which often leads to longer and less structured outputs that deviate from the strict FinGPT template. Its distillation process also prioritizes reasoning style over faithful summarization, causing it to lose detail and consistency in tasks that require precise extraction of financial signals. In contrast, Llama-3 is trained on a broader and more diverse data points, with stronger instruction-following behavior and better alignment for tasks involving summarization, classification, and structured text generation. These characteristics make Llama-3 inherently better suited for FinGPT's format-constrained prompts, resulting in higher Rouge scores, more accurate directional predictions, and overall more stable outputs.

References

[1] Meta AI. Introducing llama 3.1: Our most capable models to date. https://ai.meta.com/blog/meta-llama-3-1/, 2024. Published July 23, 2024.

[2] NVIDIA API Documentation. Deepseek-r1-distill-llama-8b. https://docs.api.nvidia.com/nim/reference/deepseek-ai-deepseek-r1-distill-llama-8b, 2024.

[3] 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.

[4] Ayush Thakur. Using gpus with keras: A tutorial with code. https://wandb.ai/authors/ayusht/reports/Using-GPUs-With-Keras-A-Tutorial-With-Code--VmlldzoxNjEyNjE. Weights & Biases Report, Published July 4, 2025.

[5] DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. https://huggingface.co/deepseek-ai/DeepSeek-R1-Distill-Llama-8B, 2025.