Adapting Language Models

CSCI 601-471/671 (NLP: Self-Supervised Models)

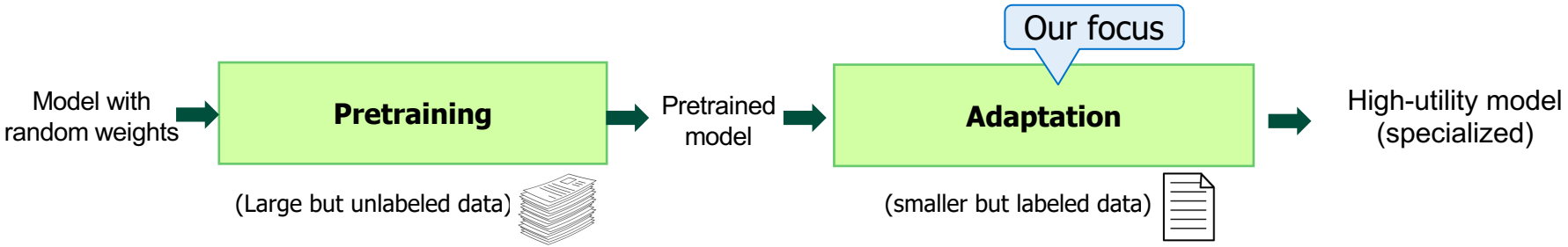
https://self-supervised.cs.jhu.edu/sp2026/

Adapting Language Models: Chapter Plan

- You have a pre-trained language model that is pre-trained on massive amounts of data.
- The pretraining objective does not directly optimize your downstream task!

Now how to you ”adapt” them for your use-case?

- **Prompting:** adapting model inputs (language statements)
- **Tuning:** adapting (modifying) model parameters



Adaptation: Problem Setup

- Given a pre-trained LM, we want to adapt it to our input-output pairs.

task $\mathcal{T} : \mathcal{X} \rightarrow \mathcal{Y}$

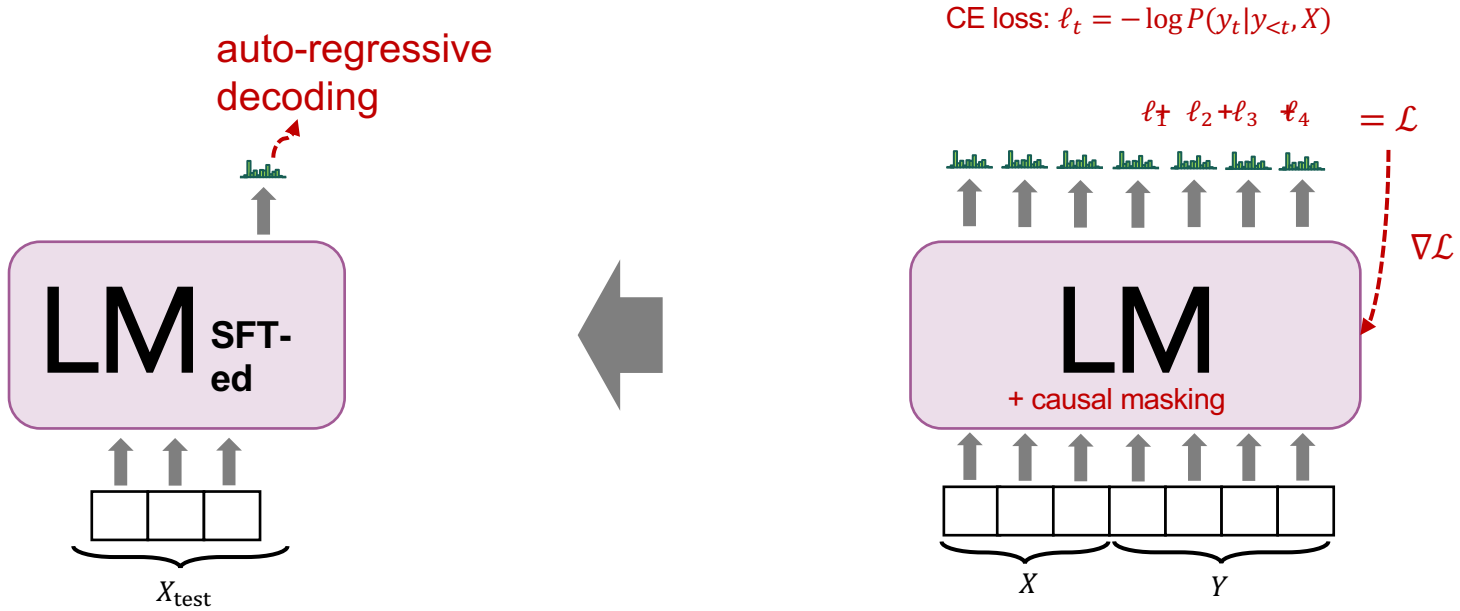
N labeled data

$$\begin{cases} X_1 \rightarrow Y_1 \\ \vdots \\ X_N \rightarrow Y_N \end{cases}$$

- **Task:** Convert a customer message into a support ticket.
- **Example input:** “My headphones won’t charge even after being plugged in overnight.”
- **Expected output:**

```
{"issue": "charging_failure",
"product": "headphones"}
```

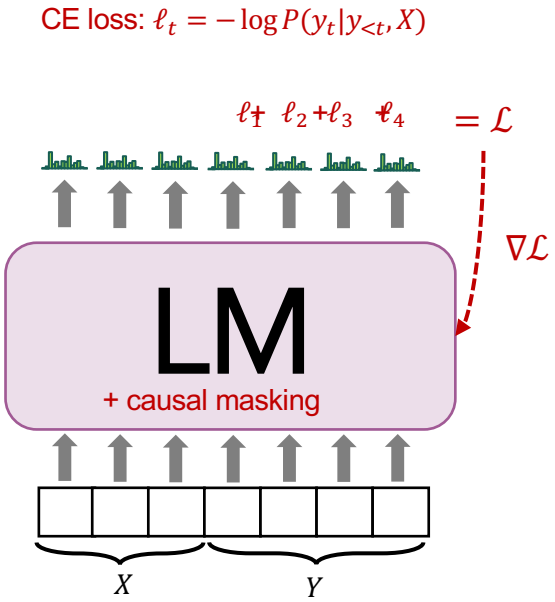
Supervised Fine-tuning (SFT)



SFT on Full model vs Head

- **Whole model tuning:**
  - Your training (optimization) updates **all** model parameters

- **Head-tuning:**
  - "head" := the vocabulary projection matrix for generation
  - Your training (optimization) updates the parameters of the model "head" (the unembed matrix)



Training time

SFT on Full model vs Head: Pulse check

- **Q1:** Which changes more parameters?
- **A1:** Full-model SFT updates more parameters.

- **Q2:** Which takes up less FLOPS?
- **A2:** Head tuning uses fewer training FLOPs.T
  - The forward computation is the same for both.
  - Head tuning only backpropagates through the unembedding layer; no backward pass through the frozen backbone is needed.

- **Q3:** Which takes up less GPU memory?
- **A3:** Head tuning uses less GPU memory.
  - It stores gradients and optimizer states only for the head.
  - It also avoids saving backbone activations for backward.

SFT on Full model vs Head:
Pros and Cons

- **Checkpoint storage space:**
  - Full-model SFT is done on **all params**, after which we store **a copy of all params.**
  - With large models, storage management becomes difficult
    - E.g., A model of size 170B parameters requires ~340Gb of storage
    - If you fine-tune a separate model for 100 tasks: 340 x 100 = 34 TB of storage!
    - Head-tuning requires both less storage

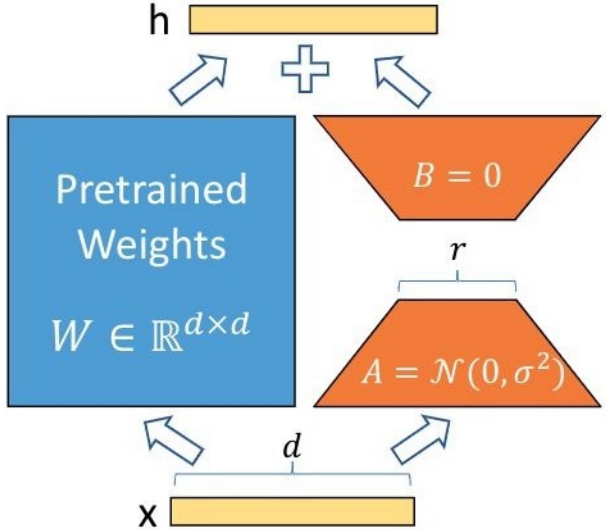
- **Training memory:** The more *trainable* params you have, the more GPU memory need.
  - Full-model training needs more GPU memory for training.
  - Head-tuning requires both less storage and less GPU memory. (even though you effectively keep all model parameters in GPU in both cases).

- **Quality/capacity tradeoff:** head tuning is cheaper but restricts adaptation to the output mapping.

LoRA: Low-Rank Adaptation

- **Hypothesis:** weight update matrices are low-ranked.
- Parameter update for a weight matrix is decomposed into a product of two low-rank matrices.

$$W \leftarrow W + \Delta W$$
$$\Delta W = BA$$
$$B \in \mathbb{R}^{d \times r}, A \in \mathbb{R}^{r \times d}, \qquad r \ll d$$



- A is initialized with random Gaussian Initialization, B is initialized to zero.
  - **Why init with zero?**
  - The adapted model initially reproduces the base model.

"LoRA: Low-Rank Adaptation of Large Language Models" Hu et al., 2021.

LoRA: Where you place them?

- LoRA used to be applied to W_q and W_v matrices in SA.
- But since then, it's been common to target all matrices in SA and FFN of all layers.

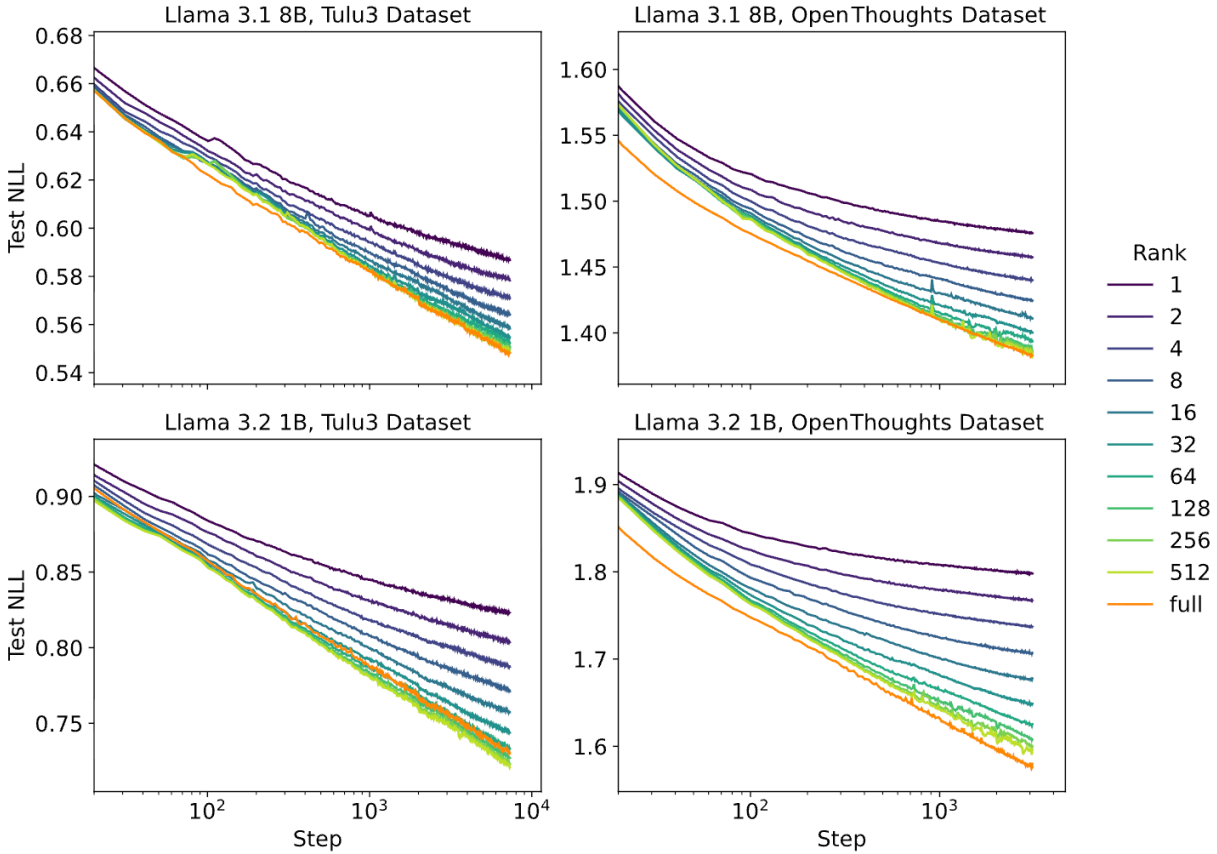
$$\{W_q^{(l)}, W_k^{(l)}, W_v^{(l)}, W_o^{(l)}\}_{l=1}^{L} \quad \{W_{\text{gate}}^{(l)}, W_{\text{up}}^{(l)}, W_{\text{down}}^{(l)}\}_{l=1}^{L}$$

- You can view these as different "knobs" for update expressivity:
  - **Update locations:** adapting more projections increases coverage and trainable parameters;
  - **Update ranks:** raising rank increases capacity within each targeted projection.

This blog post shows that LoRA on all linear projections is more effective:
LoRA Learns Less and Forgets Less, 2024

LoRA, empirically

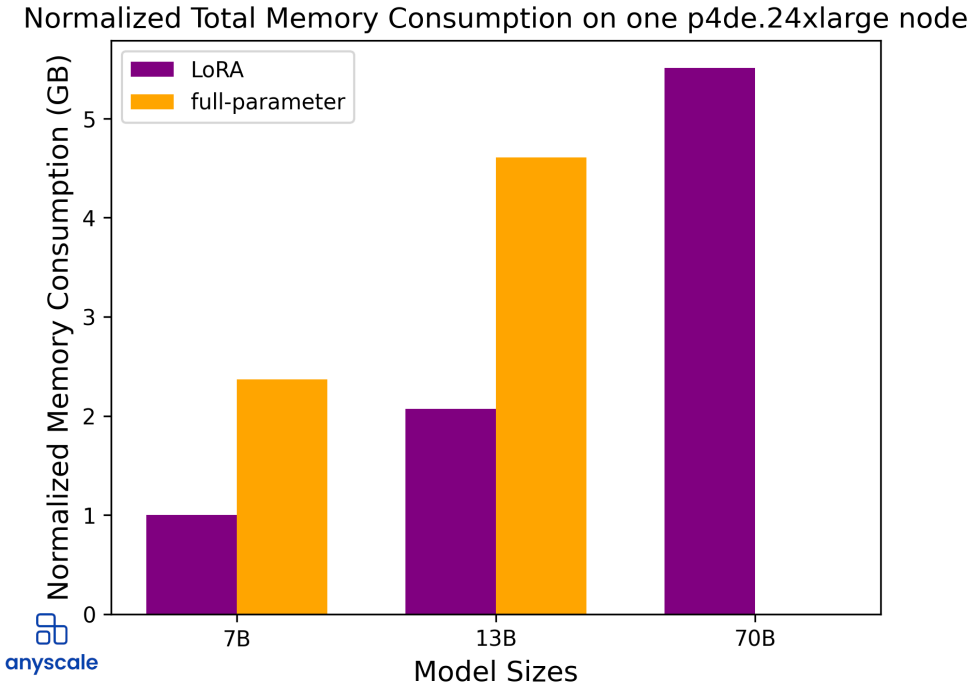
- LoRA at **low** ranks underperforms full finetuning.
- **High**-rank LoRA can perform (almost) as good as full SFT



LoRA Learns Less and Forgets Less, 2024

LoRA memory footprint

- LoRA reduces memory requirements during fine-tuning (up to x3 for LoRA).
- Why?

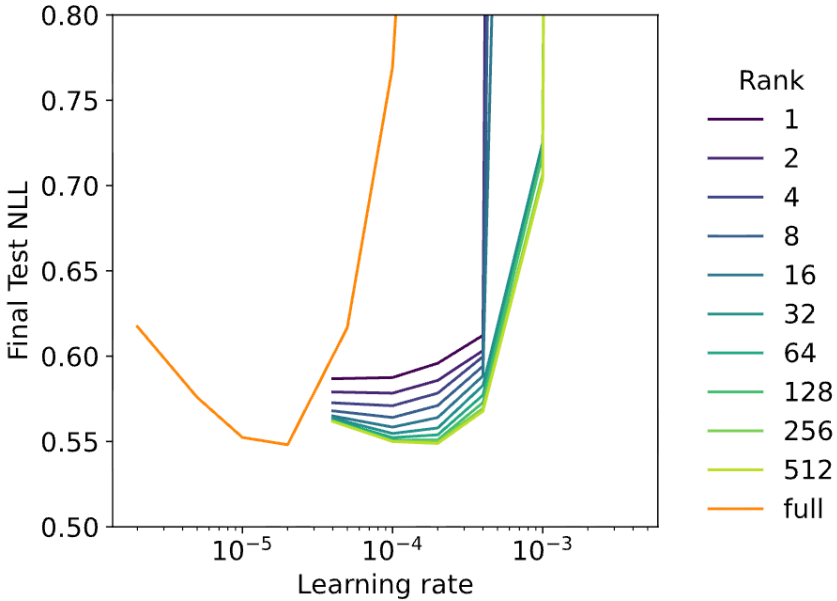


https://www.anyscale.com/blog/fine-tuning-llms-lora-or-full-parameter-an-in-depth-analysis-with-llama-2

LoRA, empirically: Selecting Learning Rate

- LoRA often benefits from a **larger** learning rate tune it **separately** from full SFT.
  - In the following example, the optimal learning rate for LoRA is 10x larger than for full model SFT.

- *Typically (but not always),* the larger your trainable parameters are, the lower learning rate you need (with all else equal).



LoRA Learns Less and Forgets Less, 2024

LoRA vs full-model SFT: Compute Cost

- Does LoRA lead to computational speedup during training?

- **No, in terms of FLOPs:** LoRA does not reduce the number of floating-point operations during the forward or backward pass.
  - The full weight matrix $W + BA$ is still computed, so per-step compute is unchanged (or marginally higher).

- **Yes, in terms of memory efficiency:** LoRA dramatically reduces the number of *trainable* parameters.
  - Since optimizer states (e.g., Adam's first and second moments) are only maintained for *trainable* parameters, GPU memory consumption drops significantly.
  - This allows larger batch sizes, which allows more tokens processed per second, even though per-step latency is not reduced.
  - **Note you still want to keep all the parameters (including non-LoRA ones) in GPU!!**
  -

LoRA vs Head-Tuning: Which is better?

- Which takes up more training GPU memory?
- This is a non-trivial question. Depends on a couple of things.

1. Activations stored during backpropagation
   - **Head-tuning:** Need not much activations saved for backward.
   - **LoRA:** Backprop through the backbone is needed to train adapters across layers, so activation memory can be substantial.

2. Gradients and optimizer states
   - Scales with the number of trainable params. **Which has more?**

LoRA vs Head-Tuning: # of trainable params

- Which has more trainable parameters?
  - Let's do the math for Llama-2 7B; $d = 4096$, $\mathrm{L} = 32$ layers, $V = 32k$

- Assuming we're doing head-tuning with text generation, we train the unembed matrix.
  - $d \times V$ params = 131M parameters

- For LoRA (applied to Q, K, V, O) matrices, let's consider various ranks:
  - If $r = 1 \Rightarrow 1 \times 4 \times 2 \times Lrd = 1 \times 4 \times 2 \times 4096 \times 32 = 1.05M$ parameters
  - If $r = 4 \Rightarrow 4 \times 4 \times 2 \times Lrd = 4 \times 4 \times 2 \times 4096 \times 32 = 4.2M$ parameters
  - If $r = 16 \Rightarrow 8 \times 4 \times 2 \times Lrd = 8 \times 4 \times 2 \times 4096 \times 32 = 16.8M$ parameters

- For low-ranks, LoRA has less # of trainable parameters (and hence, smaller optimizer states).

LoRA vs Head-Tuning: Overall perf

- So, which takes up less GPU memory depends on your setup.

- Which is more performant on downstream tasks?
  - The evidence is [also] mixed on this.
  - Depends on various parameters:
    - task type and data distribution, LoRA rank, amount of data, etc.

- In practice, you need to try it for your setup to find out which works better.

- See other adaptation algorithms: https://huggingface.co/docs/peft/

Recap: Model Adaptation

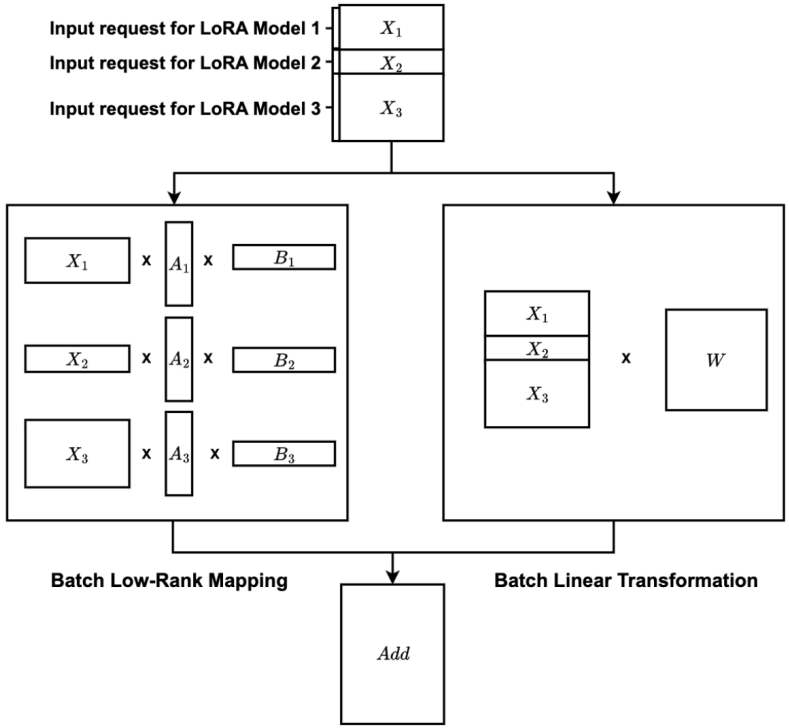
- SFT: A gradient-based way of adapting models for for a given training objective.
- LoRA: Low-rank adaptation to Transformer weight matrices.
- Memory footprint: LoRa reduces memory requirements during fine-tuning (up to x3 for LoRA).
- The main speedup is coming from the fast that you can pack more things in a batch.

Bonus

Other benefits of LoRA

- **Multi-tenant serving.**
  - A single inference server can keep many adapters (different model versions) in memory and sample from them simultaneously in a batched way.
  - Modern inference engines e.g., vLLM and SGLang implement this feature.

- This will be an effective tool later when we use do RL (switching among policy variants).



Punica: Multi-Tenant LoRA Serving, 2023
MixLoRA: An Efficient Multi-Tenant Framework, 2025