THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY | DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

RELAXED SYSTEM LAB

Parameter Efficient Finetuning

COMP4901Y

Binhang Yuan

Fine-Tuning

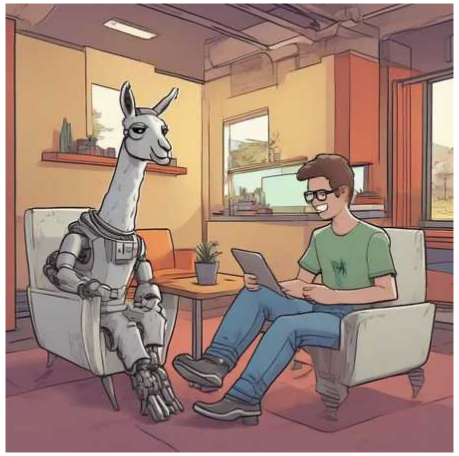
How LLM Are Usually Deployed?

- **Pre-training** is the initial phase of training an LLM, during which it learns from a large, diverse dataset, often consisting of trillions of tokens.
  - The goal is to develop a broad understanding of language, context, and various types of knowledge for the model.
  - Pre-training is usually computationally intensive (thousands of GPUs for weeks) and requires huge amounts of data (trillions of tokens).
- **Fine-tuning** is where you take an already pre-trained model and further train it on a more specific dataset.
  - This dataset is typically smaller and focused on a particular domain or task.
  - The purpose of fine-tuning is to adapt the model to perform better in specific scenarios or on tasks that were not well covered during pre-training.
  - The new knowledge added during fine-tuning enhances the model's performance in specific contexts rather than broadly expanding its general knowledge.

How LLM Are Usually Deployed?

Stage 1: Pretraining

1. Prepare 10TB of text as the training corpus.
2. Use a cluster of thousands of GPUs to train a neural network with the corpus from scratch.
3. Obtain the ***base model***.



Stage 2: Fine-tuning

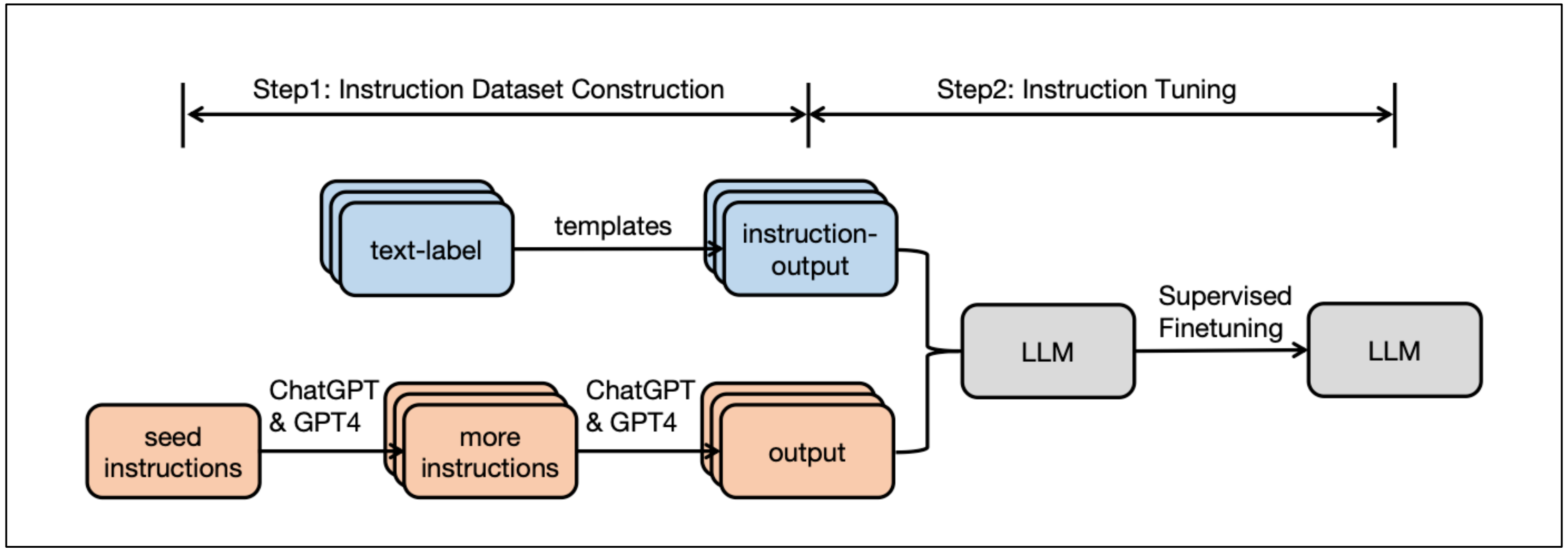
1. Prepare 100K high-quality text as the task-specific training dataset.
2. Use a small-scale GPU cluster to fine-tune the base model with the training dataset.
3. Obtain the ***domain-specific model***.
4. Run the domain-specific model for evaluation.
5. Repeat until we are happy with the results.

Instruction Tuning

- Fine-tune the LLM using *(Instruction, output)* pairs:
  - *Instruction*: denotes the human instruction for the model;
  - *Output*: denotes the desired output that follows the instruction.
- Finetuning an LLM on the instruction dataset bridges the gap between the next-word prediction objective of LLMs and the users' objective of instruction following;
- Instruction tuning allows for a more controllable and predictable model behaviour compared to standard LLMs.
  - The instructions serve to constrain the model's outputs to align with the desired response characteristics or domain knowledge, providing a channel for humans to intervene with the model's behaviours.
- Instruction tuning can help LLMs rapidly adapt to a specific domain without extensive retraining or architectural changes.

Instruction Tuning

- Two methods to construct the instruction datasets:
  - Data integration from annotated natural language datasets: (instruction, output) pairs are collected from existing annotated natural language datasets by using templates to transform text-label pairs to (instruction, output) pairs.
  - Generate outputs using more advanced LLMs: employ LLMs such as GPT-3.5-Turbo or GPT4 to gather the desired outputs given the instructions instead of manually collecting the outputs.



RELAXED
SYSTEM LAB

Parameter Efficient Fine-Tuning

Parameter Efficient Fine-Tuning

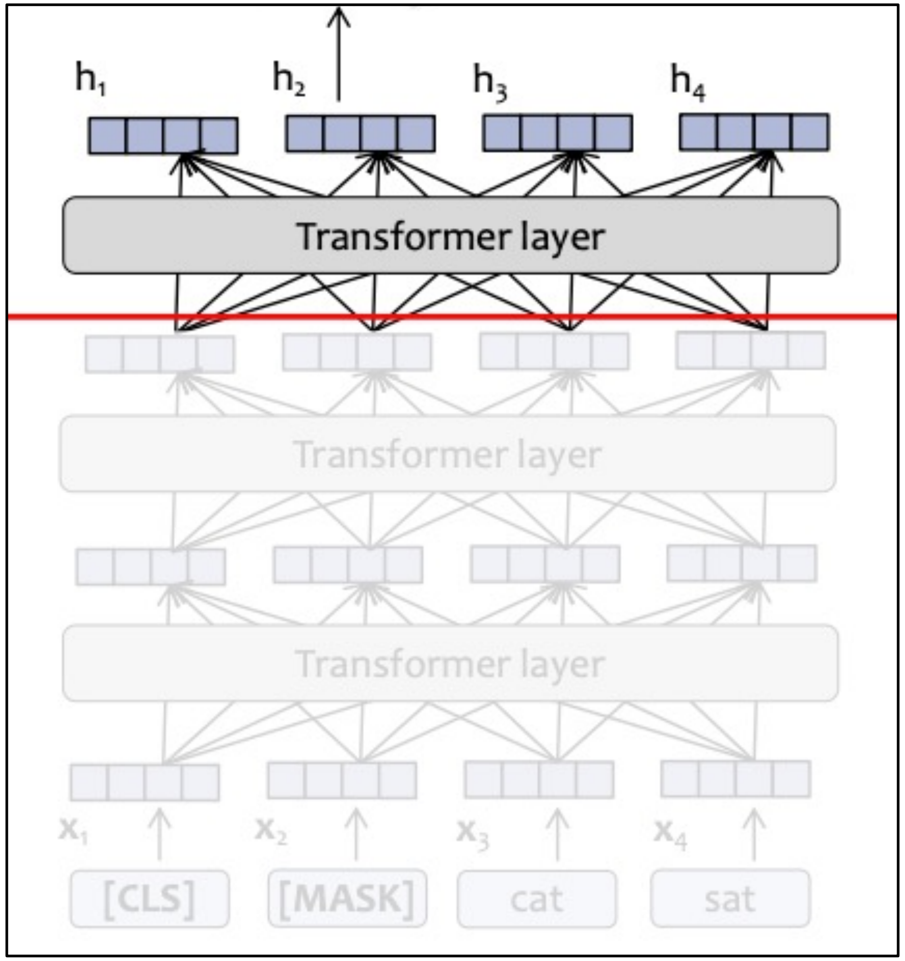
- *Parameter efficient fine-tuning (PEFT):* Rather than finetuning the entire model, we finetune only small amounts of weights.
- Goal: achieve performance on a downstream task that is comparable to fine-tuning all parameters.
- Some approaches:
  - Frozen layer/Subset fine-tuning: pick a subset of the parameters, fine-tune only those layers, and freeze the rest of the layers.
  - Adapters: add additional layers that have few parameters and tune only the parameters of those layers, keeping all others fixed.
  - Low-rank adaptation (LoRA): learn a low-rank approximation of the weight matrices.
    - VeRA: vector-based random matrix adaptation, further compress the parameter space of adaptors;
    - QLoRA: quantized memory-efficient LoRA.

Subset Fine-Tuning

- Some interpretations from NLP research:
  - Earlier layers of the transformer tend to capture linguistic phenomena and basic language understanding;
  - Later layers are where the task-specific learning happens.
- We should be able to learn new tasks by freezing the earlier layers and tuning the later ones.
- This can be a simple baseline for PEFT.

Subset Fine-Tuning

- Keep all parameters fixed except for the top K layers.
- Gradients only need to flow through top K layers instead of all of layers.
- Reduce computation load.
- Reduce memory usage.

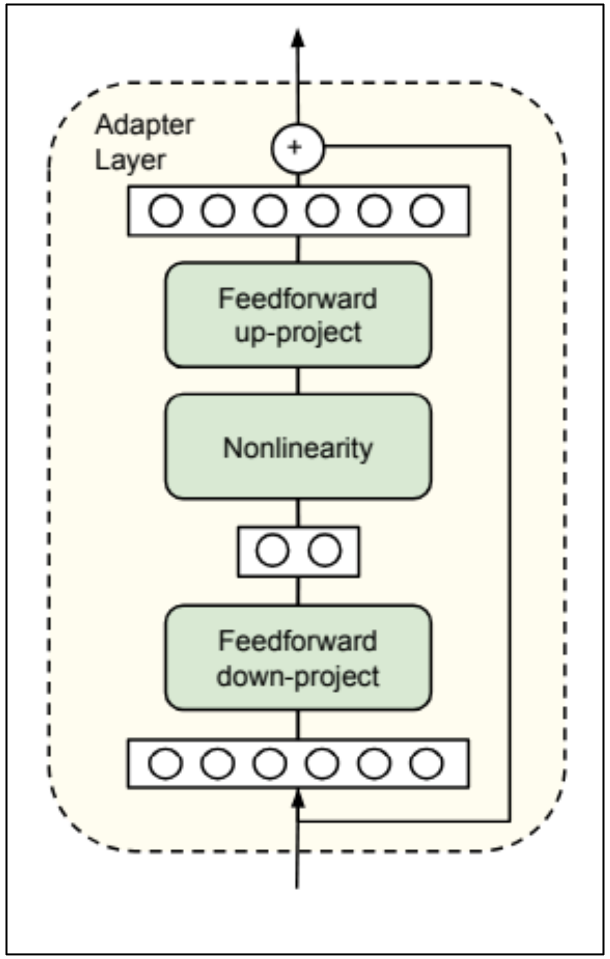


The gradient does not backpropagate to lower layers.

Adapter

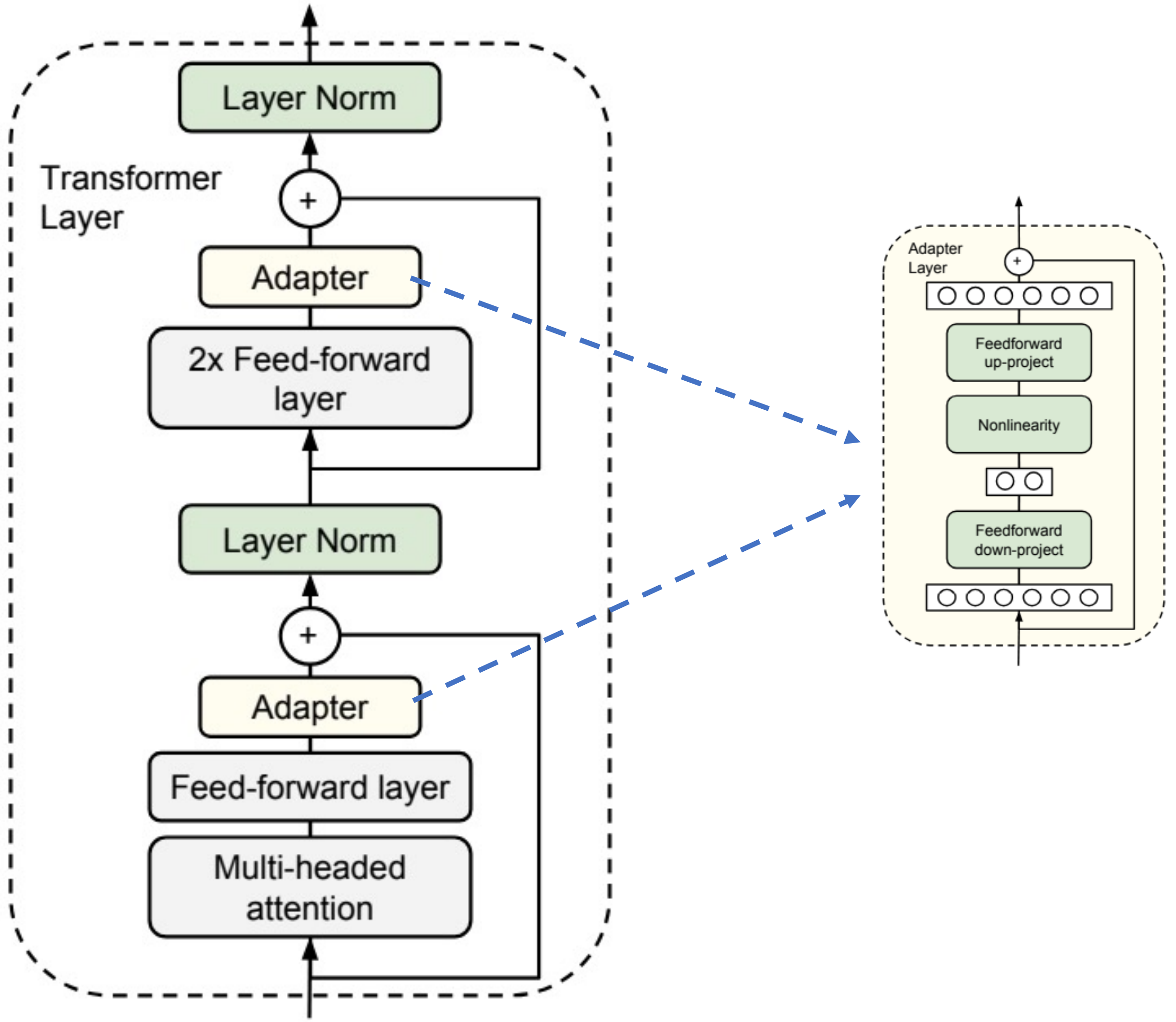
RELAXED SYSTEM LAB

- Adapters are new modules are added between layers of a pre-trained network.
  - The original model weights are fixed;
  - Just the adapter modules are tuned.
- An adapter layer is simply a feed- forward neural network with one hidden layer, and a residual connection.
- Suppose the original LLM has a model dimension of D:
  - For the down-project weight matrix: $W_A \in \mathbb{R}^{D \times R}$;
  - For the up-project weight matrix: $W_B \in \mathbb{R}^{R \times D}$;
  - We have $R \ll D$.



Adapter

- Given $R \ll D$, in practice the adapter layers contain only 0.5% – 8% of the total parameters.
- When added to a deep neural network (e.g. transformer) all the other parameters of the pretrained model are kept fixed, and only the adapter layer parameters are fine-tuned.



RELAXED
SYSTEM LAB

Low-Rank Adaptation (LoRA)

Low-Rank Adaption

- Central idea:
  - *"How can we re-parameterize the model into something more efficient to train?"*
- Finetuning has a low *intrinsic dimension*, that is, the minimum number of parameters needed to be modified to reach satisfactory performance is not very large.
- This means we can re-parameterize a subset of the original model parameters with low-dimensional proxy parameters, and just optimize the proxy.

Intrinsic Dimension

- An objective function's intrinsic dimension measures the minimum number of parameters needed to reach a satisfactory solution to the objective.
- Can also be thought of as the lowest dimensional subspace in which one can optimize the original objective function to within a certain level of approximation error.
- Details in this paper: https://arxiv.org/abs/2012.13255
  - Suppose we have model parameters $\theta^{(D)} \in \mathbb{R}^D$, D is the number of parameters;
  - Instead of optimizing $\theta^{(D)}$, we could optimize a smaller set of parameters $\theta^{(d)} \in \mathbb{R}^d$, where $d \ll D$.
  - This done through clever factorization:
    - $\theta^{(D)} = \theta_0^{(D)} + P\left(\theta^{(d)}\right)$; where $P\colon\ \mathbb{R}^d \to \mathbb{R}^D$
    - P is typically a linear projection: $\theta^{(D)} = \theta_0^{(D)} + \theta^{(d)}M$.
  - Intuitively, we do an arbitrary random projection onto a much smaller space; usually, a linear projection, we then solve the optimization problem in that smaller subspace. If we reach a satisfactory solution, we say the dimensionality of that subspace is the intrinsic dimension.

Low-Rank Adaption

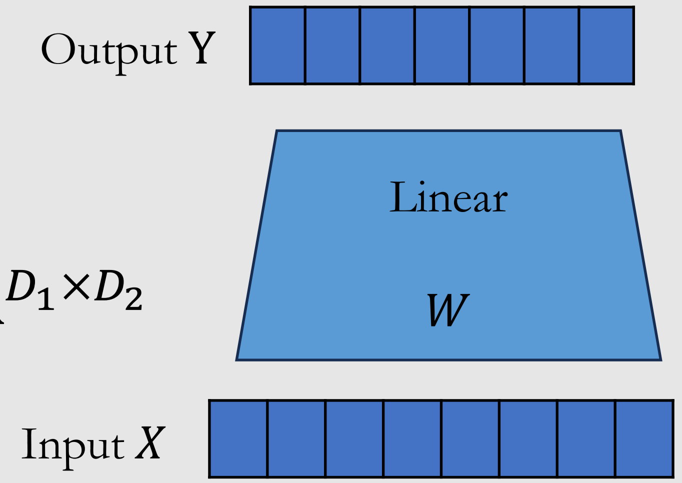
- Full paper: https://arxiv.org/pdf/2106.09685.
- Intuition: It's not just the model weights that are low rank, updates to the model weights are also low-rank.
- LoRA freezes the pre-trained model weights and injects trainable rank decomposition matrices into some or all layers.

LoRA Key Idea

- Keep the original pre-trained parameters W fixed during fine-tuning;
- Learn an additive modification to those parameters ΔW;
- Define ΔW as a low-rank decomposition: $\Delta W = AB$.

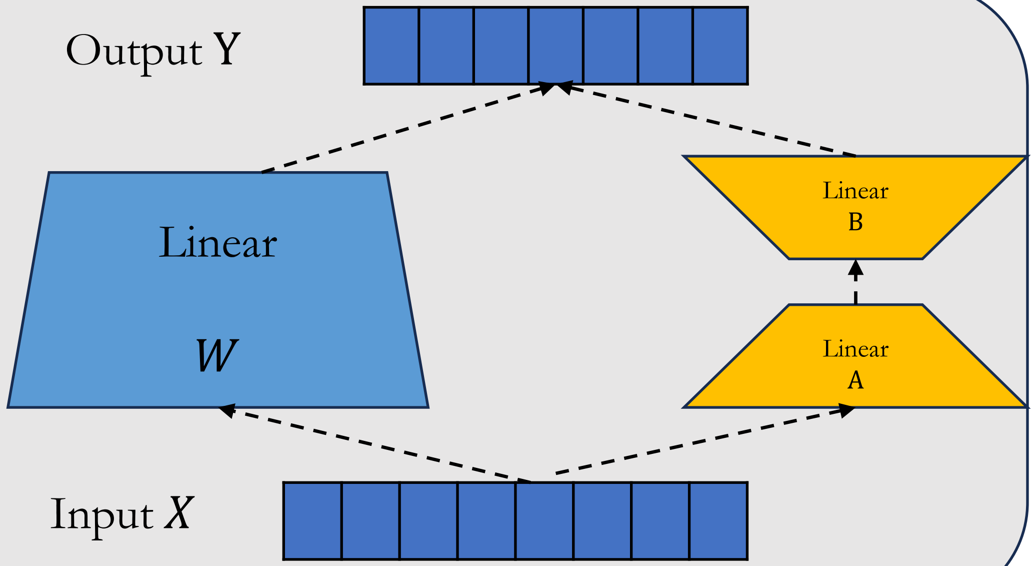
Standard Linear Layer

- $Y = XW$
- $X \in \mathbb{R}^{D_1}, Y \in \mathbb{R}^{D_2}, W \in \mathbb{R}^{D_1 \times D_2}$



LoRA Linear Layer

- $Y = XW + XAB = X(W + AB)$
- $X \in \mathbb{R}^{D_1}, Y \in \mathbb{R}^{D_2}, W \in \mathbb{R}^{D_1 \times D_2}$
- $A \in \mathbb{R}^{D_1 \times R}, B \in \mathbb{R}^{R \times D_2}$

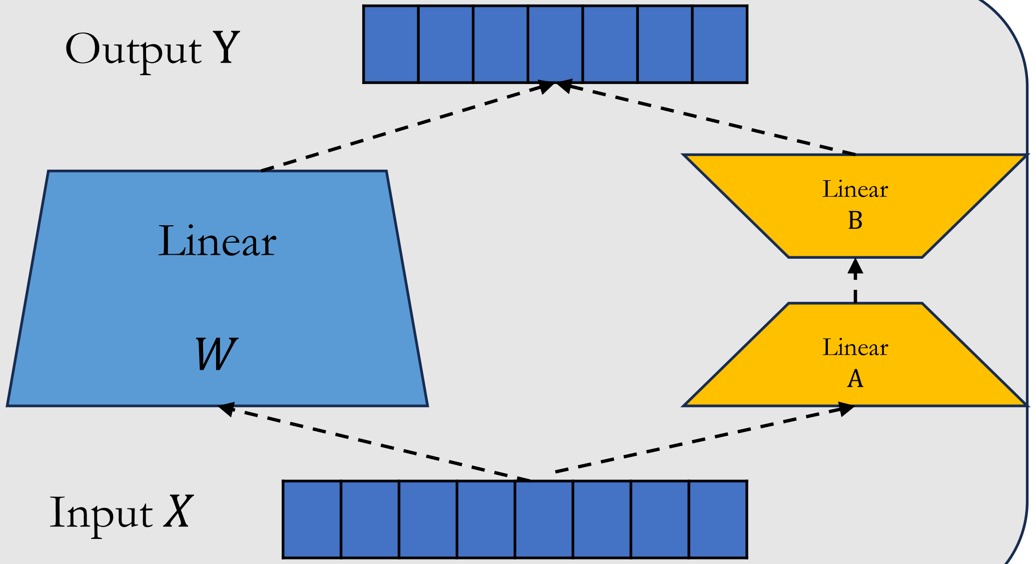


LoRA Initialization

- We initialize the trainable parameters:
  - $A_{ij} \sim \mathcal{N}(0, \sigma^2) \quad B_{ij} = 0$
- This ensures that, at the start of fine-tuning, the parameters have their pre-trained values:
  - $\Delta W = AB = 0$

LoRA Linear Layer

- $Y = XW + XAB = X(W + AB)$
- $X \in \mathbb{R}^{D_1}, Y \in \mathbb{R}^{D_2}, \mathrm{W} \in \mathbb{R}^{D_1 \times D_2}$
- $A \in \mathbb{R}^{D_1 \times R}, B \in \mathbb{R}^{R \times D_2}$



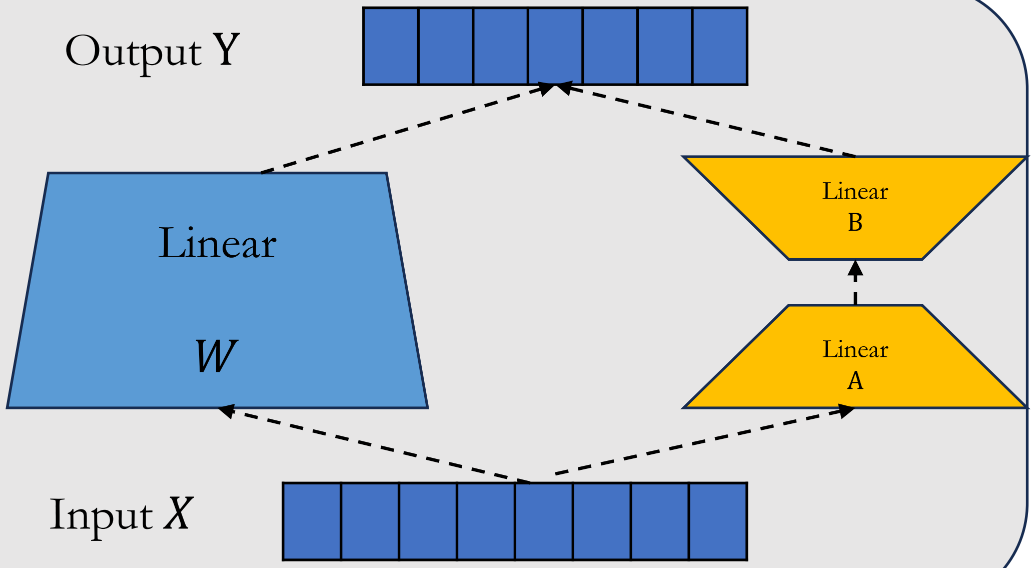
LoRA Hot Swapping Parameters

- W and AB the same dimension, so we can swap the LoRA parameters in and out of a Standard Linear Layer.

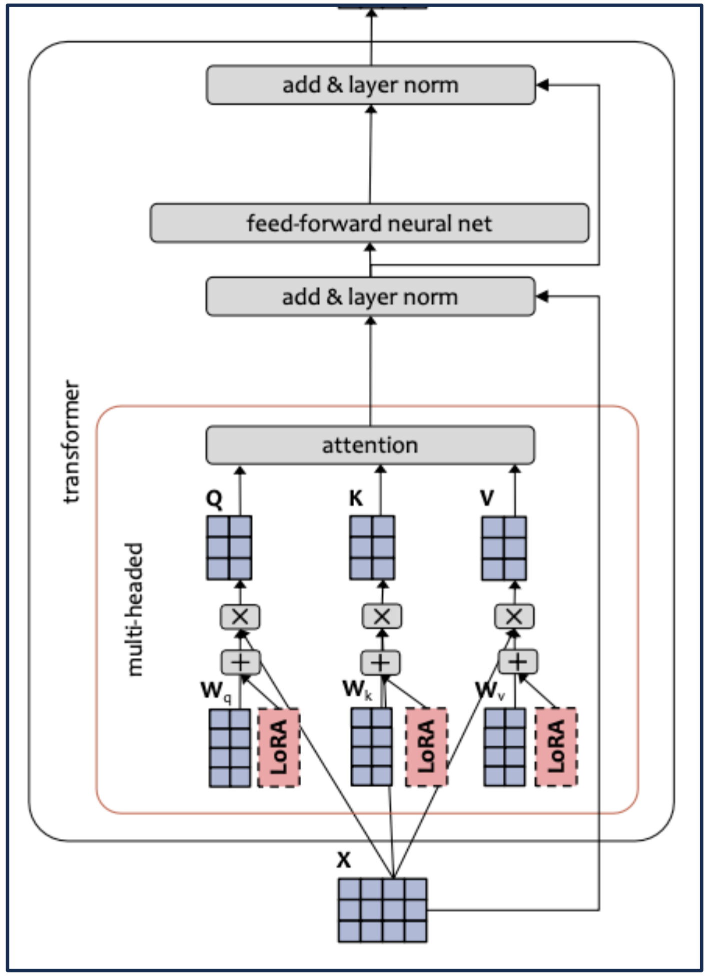
- To include LoRA:
  - $W' \leftarrow W + AB$
- To remove LoRA:
  - $W \leftarrow W' - AB$

LoRA Linear Layer

- $Y = XW + XAB = X(W + AB)$
- $X \in \mathbb{R}^{D_1}, Y \in \mathbb{R}^{D_2}, \mathrm{W} \in \mathbb{R}^{D_1 \times D_2}$
- $A \in \mathbb{R}^{D_1 \times R}, B \in \mathbb{R}^{R \times D_2}$



LoRA for Transformers



- LoRA linear layers could replace every linear layer in the Transformer layer;
- But the original paper only applies LoRA to the attention weights:
  - $Q = \text{LoRALinear}(X, W_q, A_q, B_q)$
  - $K = \text{LoRALinear}(X, W_k, A_k, B_k)$
  - $V = \text{LoRALinear}(X, W_v, A_v, B_v)$
  - Some further research found that most efficient to include LoRA only on the query and key linear layers.

LoRA Results

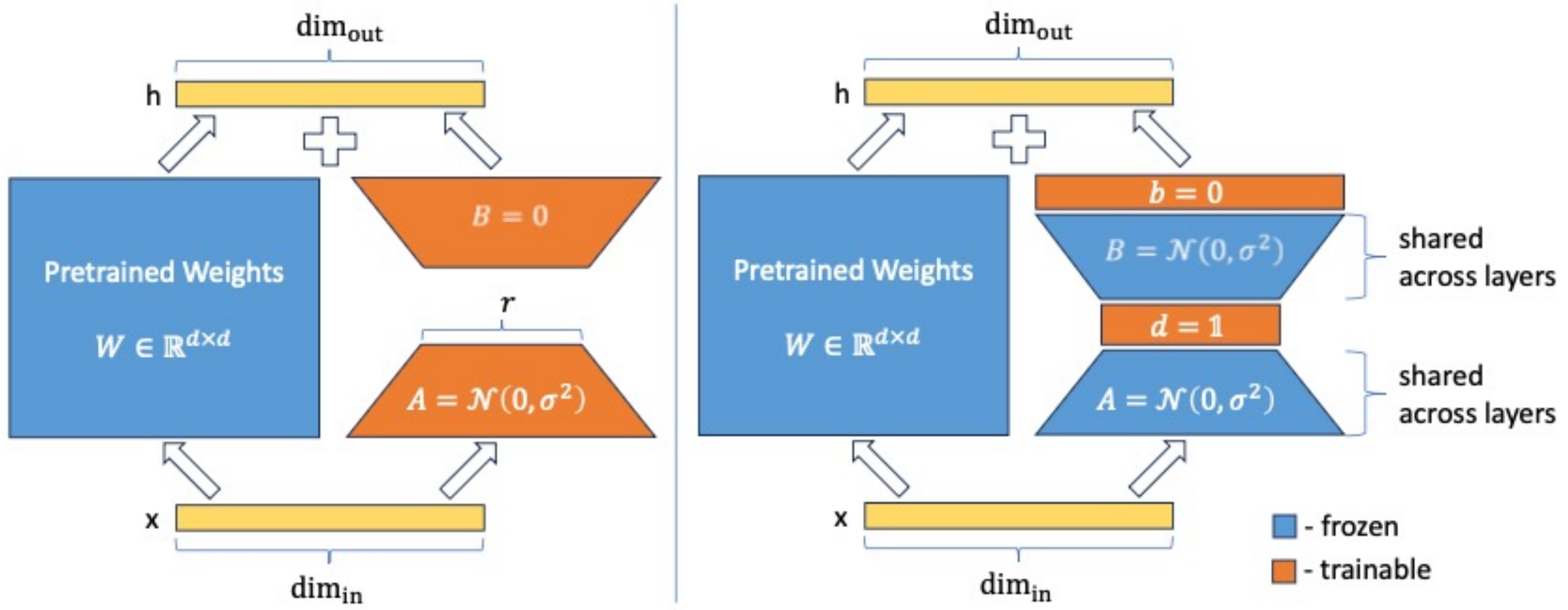
- Some emprical takeaways:
  - Applied to GPT-3, LoRA achieves performance almost as good as full parameter fine-tuning, but with far fewer parameters.
  - On some tasks, it even outperforms full fine-tuning.
  - For some datasets, a rank of $R = 1$ is sufficient.
  - LoRA performs well when the dataset is large or small.

VeRA: Vector-based Random Matrix Adaptation

- VeRA is a parameter-efficient fine-tuning technique that is similar to LoRA but requires even fewer extra parameters while promising similar or even better performance.
- Employ "scaling vectors" to adapt a pair of frozen random matrices shared between layers:
  - The reduction of the count of trainable parameters is achieved by sharing the same low-rank matrices across all layers, and only training two additional vectors per layer.
- LoRA for each adaptor:
  - $Y = XW + XAB,$
  - Tunable parameters: $A \in \mathbb{R}^{\mathrm{dim}_{in} \times r}, B \in \mathbb{R}^{\mathrm{r} \times \mathrm{dim}_{out}}$
- VeRA:
  - $Y = XW + XA\Lambda_d B\Lambda_b,$
  - $A \in \mathbb{R}^{\mathrm{dim}_{in} \times r}, B \in \mathbb{R}^{\mathrm{r} \times \mathrm{dim}_{out}}$ are frozen, random, and shared across layers;
  - Tunable parameters: $d \in \mathbb{R}^{r}, \Lambda_d \in \mathbb{R}^{r \times r}, b \in \mathbb{R}^{\mathrm{dim}_{out}}, \Lambda_b \in \mathbb{R}^{\mathrm{dim}_{out} \times \mathrm{dim}_{out}}$

VeRA: Vector-based Random Matrix Adaptation

RELAXED SYSTEM LAB



- LoRA for each adaptor:
  - $Y = XW + XAB,$
  - Tunable parameters: $A \in \mathbb{R}^{\dim_{in}\times r}, B \in \mathbb{R}^{r\times \dim_{out}}$
- VeRA:
  - $Y = XW + XA\Lambda_d B\Lambda_b,$
  - $A \in \mathbb{R}^{\dim_{in}\times r}, B \in \mathbb{R}^{r\times \dim_{out}}$ are frozen, random, and shared across layers;
  - Tunable parameters: $d \in \mathbb{R}^r, \Lambda_d \in \mathbb{R}^{r\times r}, b \in \mathbb{R}^{\dim_{out}}, \Lambda_b \in \mathbb{R}^{\dim_{out}\times \dim_{out}}$

VeRA: More Parameter-Efficient

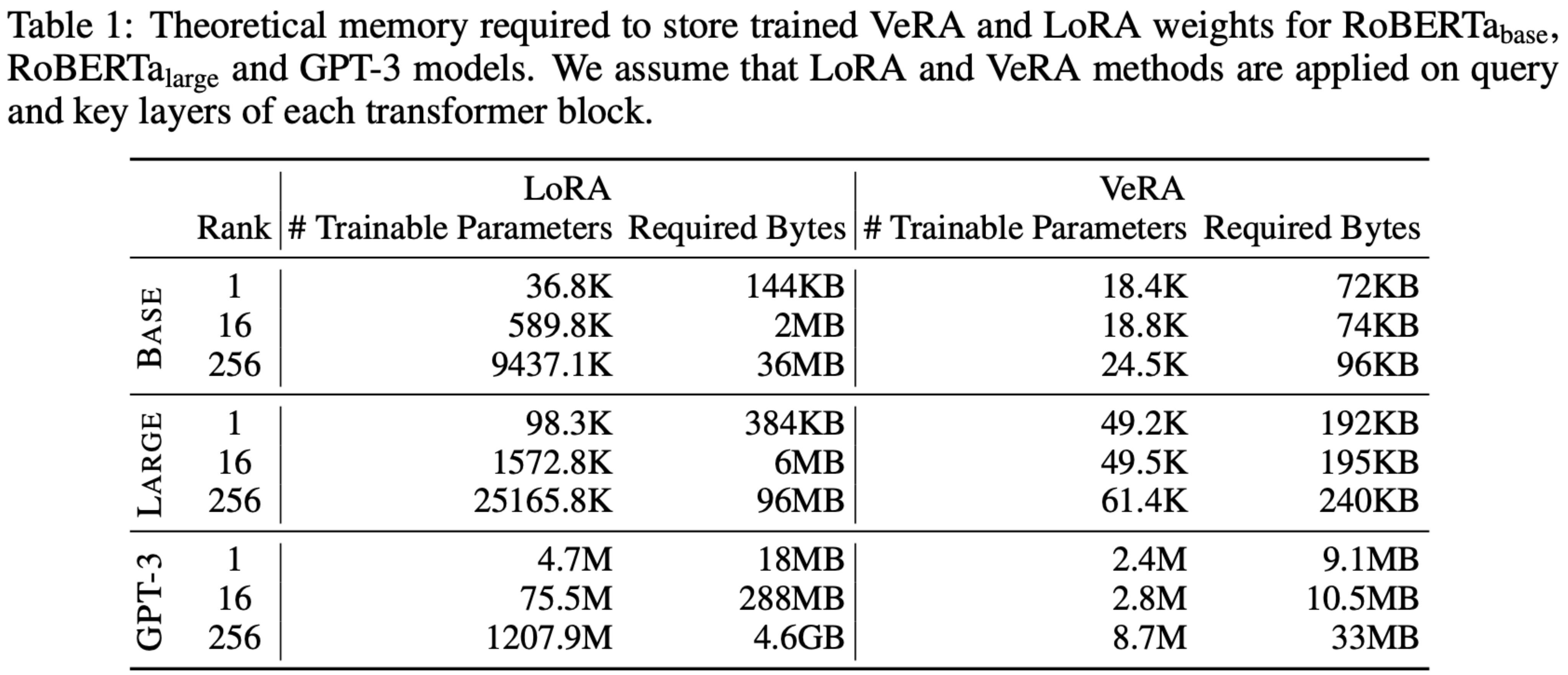
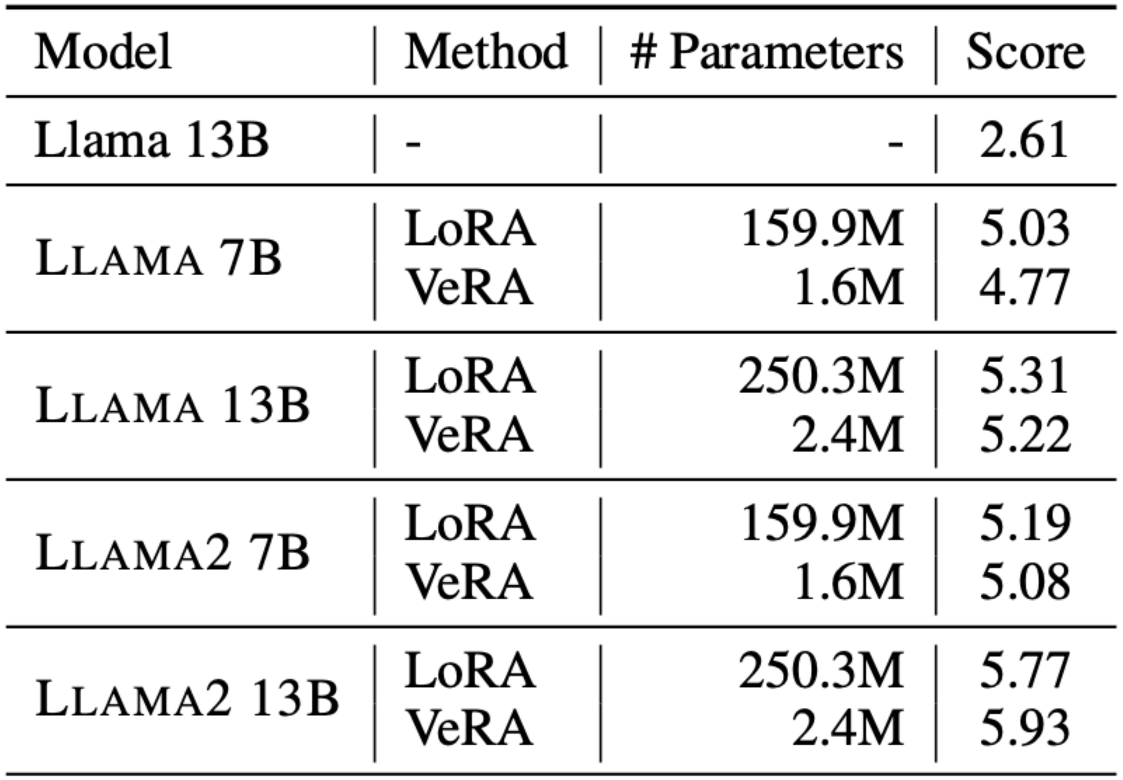


Table 1: Theoretical memory required to store trained VeRA and LoRA weights for RoBERTa$_{\text{base}}$, RoBERTa$_{\text{large}}$ and GPT-3 models. We assume that LoRA and VeRA methods are applied on query and key layers of each transformer block.

		LoRA		VeRA	
	Rank	# Trainable Parameters	Required Bytes	# Trainable Parameters	Required Bytes
BASE	1	36.8K	144KB	18.4K	72KB
	16	589.8K	2MB	18.8K	74KB
	256	9437.1K	36MB	24.5K	96KB
LARGE	1	98.3K	384KB	49.2K	192KB
	16	1572.8K	6MB	49.5K	195KB
	256	25165.8K	96MB	61.4K	240KB
GPT-3	1	4.7M	18MB	2.4M	9.1MB
	16	75.5M	288MB	2.8M	10.5MB
	256	1207.9M	4.6GB	8.7M	33MB

VeRA: Similar Performance with LoRA



Model	Method	# Parameters	Score
Llama 13B	-	-	2.61
LLAMA 7B	LoRA	159.9M	5.03
	VeRA	1.6M	4.77
LLAMA 13B	LoRA	250.3M	5.31
	VeRA	2.4M	5.22
LLAMA2 7B	LoRA	159.9M	5.19
	VeRA	1.6M	5.08
LLAMA2 13B	LoRA	250.3M	5.77
	VeRA	2.4M	5.93

- Average scores on MT-Bench generated by models fine-tuned with VeRA and LoRA methods, and the base Llama 13B model. VeRA closely matches the performance of LoRA on the instruction-following task, with a 100× reduction in trainable parameters.

QLoRA: Efficient Finetuning of Quantized LLMs

- QLoRA reduces the memory usage of LLM finetuning without performance tradeoffs compared to standard fp16 model finetuning.
- QLoRA uses 4-bit quantization to compress a pretrained language model:
  - The LM parameters are frozen and stored as 4-bit W^{NF4};
  - A relatively small number of trainable parameters stored as bf16 (A^{BF16}, B^{BF16}) are added to the model in the form of Low-Rank Adapters.
- During finetuning, QLoRA dequantizes (i.e., doubleDequantize) the frozen 4-bit quantized pretrained language model parameters for the Low-Rank Adapters computation:
  - $Y^{BF16} = X^{BF16}\text{doubleDequantize}(W^{NF4}) + X^{BF16}A^{BF16}B^{BF16}$

QLoRA: Efficient Finetuning of Quantized LLMs

Tim Dettmers* Artidoro Pagnoni* Ari Holtzman

Luke Zettlemoyer

University of Washington
{dettmers,artidoro,ahai,lsz}@cs.washington.edu

Abstract

We present QLoRA, an efficient finetuning approach that reduces memory usage enough to finetune a 65B parameter model on a single 48GB GPU while preserving full 16-bit finetuning task performance. QLoRA backpropagates gradients through a frozen, 4-bit quantized pretrained language model into Low Rank Adapters (LoRA). Our best model family, which we name **Guanaco**, outperforms all previous openly released models on the Vicuna benchmark, reaching 99.3% of the performance level of ChatGPT while only requiring 24 hours of finetuning on a single GPU. QLoRA introduces a number of innovations to save memory without sacrificing performance: (a) 4-bit NormalFloat (NF4), a new data type that is information theoretically optimal for normally distributed weights (b) Double Quantization to reduce the average memory footprint by quantizing the quantization constants, and (c) Paged Optimizers to manage memory spikes. We use QLoRA to finetune more than 1,000 models, providing a detailed analysis of instruction following and chatbot performance across 8 instruction datasets, multiple model types (LLaMA, T5), and model scales that would be infeasible to run with regular finetuning (e.g. 33B and 65B parameter models). Our results show that QLoRA finetuning on a small high-quality dataset leads to state-of-the-art results, even when using smaller models than the previous SoTA. We provide a detailed analysis of chatbot performance based on both human and GPT-4 evaluations showing that GPT-4 evaluations are a cheap and reasonable alternative to human evaluation. Furthermore, we find that current chatbot benchmarks are not trustworthy to accurately evaluate the performance levels of chatbots. A lemon-picked analysis demonstrates where **Guanaco** fails compared to ChatGPT. We release all of our models and code, including CUDA kernels for 4-bit training.[2]

1 Introduction

Finetuning large language models (LLMs) is a highly effective way to improve their performance, [40, 62, 43, 61, 59, 37] and to add desirable or remove undesirable behaviors [43, 2, 4]. However, finetuning very large models is prohibitively expensive; regular 16-bit finetuning of a LLaMA 65B parameter model [57] requires more than 780 GB of GPU memory. While recent quantization methods can reduce the memory footprint of LLMs [14, 13, 18, 66], such techniques only work for inference and break down during training [65].

We demonstrate for the first time that it is possible to finetune a quantized 4-bit model without any performance degradation. Our method, QLoRA, uses a novel high-precision technique to quantize a pretrained model to 4-bit, then adds a small set of learnable Low-rank Adapter weights [28]

*Equal contribution.
[2] https://github.com/artidoro/qlora and https://github.com/TimDettmers/bitsandbytes

QLoRA: Benchmark Results

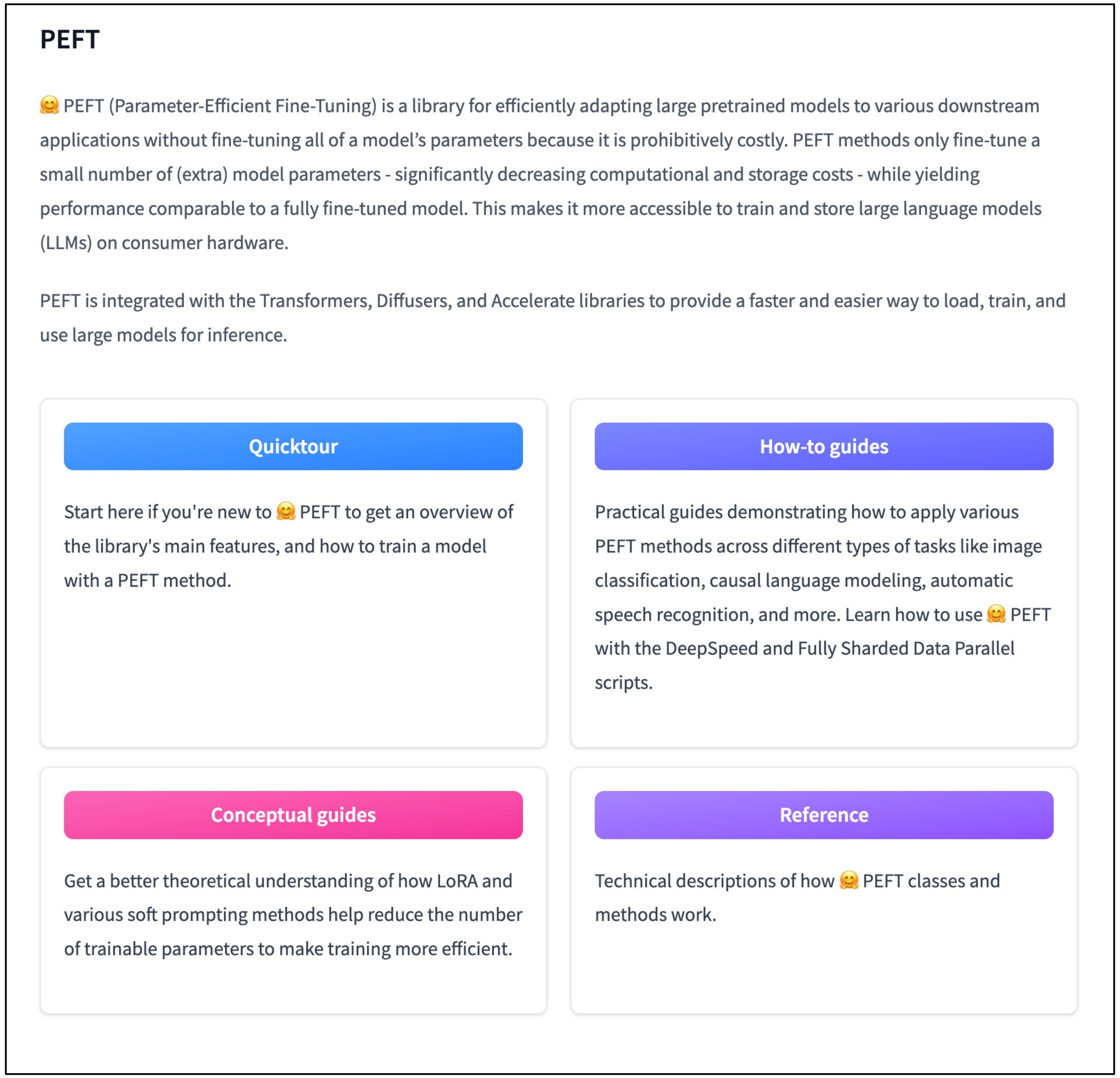
- QLoRA does have a small impact on the model performance compared to regular LoRA.
- https://lightning.ai/pages/community/lora-insights/

	TruthfulQA MC1	TruthfulQA MC2	Arithmetic 2ds	Arithmetic 4ds	BLiMP Causative	MMLU Global Facts
Llama 2 7B base	0.2534	0.3967	0.508	0.637	0.787	0.32
LoRA default 1	0.2876	0.4211	0.3555	0.0035	0.75	0.27
LoRA default 2	0.284	0.4217	0.369	0.004	0.747	0.26
LoRA default 3	0.2815	0.4206	0.372	0.004	0.747	0.27
QLoRA (nf4)	0.2803	0.4139	0.4225	0.006	0.783	0.23
QLoRA (fp4)	0.2742	0.4047	0.5295	0.016	0.744	0.23

RELAXED
SYSTEM LAB

PEFT Practical Examples

PEFT



PEFT

🤗 PEFT (Parameter-Efficient Fine-Tuning) is a library for efficiently adapting large pretrained models to various downstream applications without fine-tuning all of a model's parameters because it is prohibitively costly. PEFT methods only fine-tune a small number of (extra) model parameters - significantly decreasing computational and storage costs - while yielding performance comparable to a fully fine-tuned model. This makes it more accessible to train and store large language models (LLMs) on consumer hardware.

PEFT is integrated with the Transformers, Diffusers, and Accelerate libraries to provide a faster and easier way to load, train, and use large models for inference.

Quicktour

Start here if you're new to 🤗 PEFT to get an overview of the library's main features, and how to train a model with a PEFT method.

How-to guides

Practical guides demonstrating how to apply various PEFT methods across different types of tasks like image classification, causal language modeling, automatic speech recognition, and more. Learn how to use 🤗 PEFT with the DeepSpeed and Fully Sharded Data Parallel scripts.

Conceptual guides

Get a better theoretical understanding of how LoRA and various soft prompting methods help reduce the number of trainable parameters to make training more efficient.

Reference

Technical descriptions of how 🤗 PEFT classes and methods work.

https://huggingface.co/docs/peft/en/index

LoRA Example: Common Prerequisites

RELAXED SYSTEM LAB

Common Prerequisites

```
import torch
from torch.utils.data import DataLoader
from transformers import (
    AutoTokenizer,
    AutoModelForSequenceClassification,
    DataCollatorWithPadding,
)
from datasets import load_dataset
```

LoRA Example: Load & Tokenize

RELAXED SYSTEM LAB

Load & Tokenize

```
# 1. Load SST-2 (a binary sentiment task)
raw_ds = load_dataset("glue", "sst2")

# 2. Tokenizer
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
def preprocess(batch):
    return tokenizer(batch["sentence"], truncation=True)

tokenized_ds = raw_ds.map(preprocess, batched=True)
tokenized_ds.set_format(type="torch",
columns=["input_ids","attention_mask","label"])
```

LoRA Example: PEFT & Model Setup

RELAXED SYSTEM LAB

PEFT & Model Setup

```
from peft import LoraConfig, get_peft_model

# Wrap a pretrained BERT for classification
model = AutoModelForSequenceClassification.from_pretrained("bert-base-
uncased")

# LoRA config: low-rank adapters on Q/K/V projections
lora_cfg = LoraConfig(
    r=8,
    lora_alpha=32,
    target_modules=["query","value"],
    lora_dropout=0.1,
    bias="none",
    task_type="SEQ_CLS"
)
model = get_peft_model(model, lora_cfg)
```

LoRA Example: DataLoader & Training Loop

RELAXED SYSTEM LAB

DataLoader & Training Loop

```
# DataLoader
train_dl = DataLoader(
    tokenized_ds["train"],
    batch_size=128,
    shuffle=True,
    collate_fn=DataCollatorWithPadding(tokenizer),
    num_workers=4
)
# Optimizer & device
device = "cuda" if torch.cuda.is_available() else "mps" if torch.backends.mps.is_available() else "cpu"
model.to(device)
optim = torch.optim.AdamW(model.parameters(), lr=1e-6)
# Training loop
model.train()
for epoch in range(3):
    for batch in train_dl:
        batch = {k:v.to(device) for k,v in batch.items()}
        outputs = model(**batch)
        loss = outputs.loss
        optim.zero_grad()
        loss.backward()
        optim.step()
        print(f"Epoch {epoch+1} loss: {loss.item():.4f}")
```

References

- https://arxiv.org/abs/2308.10792
- https://www.andrew.cmu.edu/course/11-667/lectures/W4L2_PETM.pptx.pdf
- https://www.cs.cmu.edu/~mgormley/courses/10423//slides/lecture11-peft-ink.pdf
- https://arxiv.org/abs/2012.13255
- https://arxiv.org/pdf/2106.09685
- https://arxiv.org/abs/2310.11454
- https://arxiv.org/abs/2305.14314
- https://huggingface.co/blog/4bit-transformers-bitsandbytes
- https://huggingface.co/docs/peft/en/index