Fine-tuning with QLoRA

AI in News: Applying Generative AI in Production with Confidence

Bilyana Taneva-Popova

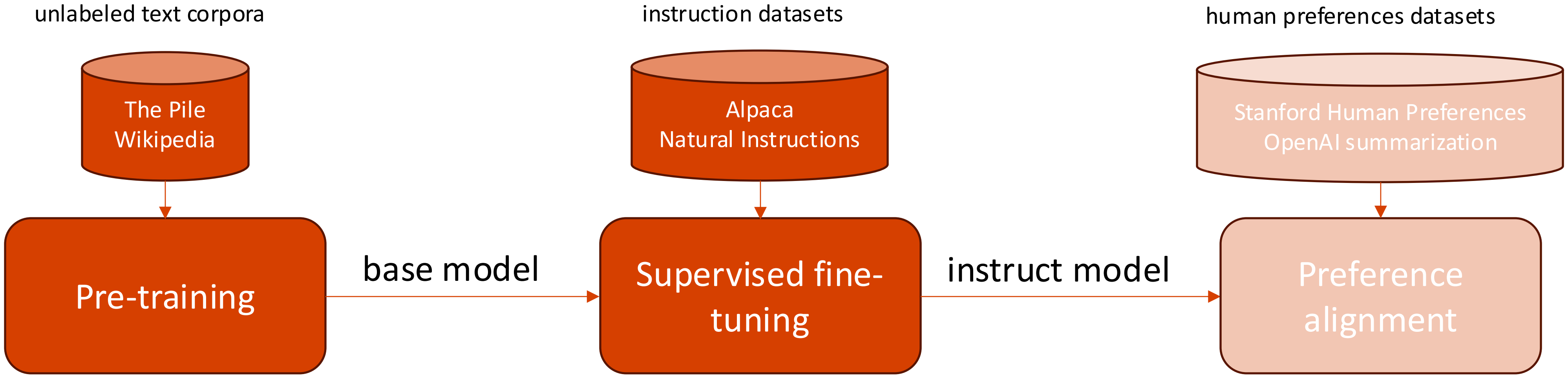
Thomson Reuters™

Content

- Introduction
- Fine-tuning with LoRA
- Fine-tuning with QLoRA
- Practical exercise

Introduction

- Supervised fine-tuning takes a base model (trained to predict the next token) and turns it into a model that generates more useful completions
- Still trained on next token prediction, but now on more targeted instruction datasets



Full Fine-tuning Feasibility

- Full fine-tuning common prior to the rise of LLMs
  - Small scale models (100M-300M params) typically fine-tuned for domain applications

- For larger models (>1B params) full fine-tuning is typically infeasible
- To load 7B model in full precision -> 7B * 4bytes = 28 GB RAM
- To fine-tune 7B model in half-precision and mixed-precision mode
  - 2 bytes for the weight + 2 bytes for the gradient + 12 bytes for the Adam optimizer state = 16 bytes per trainable param -> 7B * 16 bytes = 112 GB RAM

- Parameter-efficient fine-tuning methods (LoRA and QLoRA)
  - Aim at drastically reducing the number of trainable parameters of a model while keeping the same performance as full fine-tuning

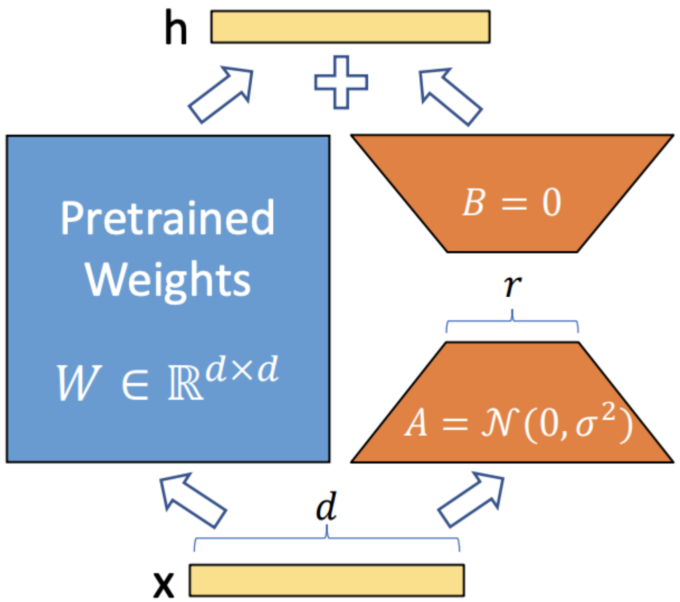
When to Fine-tune

- Specific domains (e.g., legal, medical, finance)
- Cost reduction
- Tone, style, formatting
- New and specific tasks
- Prompt engineering not sufficient

Parameter Efficient Fine-tuning (PEFT)

- Low-rank adaptation (LoRA)
- Drastically reduces the number of trainable parameters while maintaining performance
- Freeze the base model and add only a few trainable parameters (called adapters)

- LoRA learns a weight matrix W' = W + AB
- W are the frozen weights of the base model.
  They don't receive any further updates
- A and B are trainable low-rank matrices that
  adapt to the new data. Their product has the same shape as W



Thomson Reuters

Reduction of Trainable Parameters

- Suppose we have embedding vectors of 1000 dimensions
- This results in K, Q and V matrices of 1000 x 1000 = 10^6 trainable parameters for full fine-tuning
- If we choose r = 8, then we learn matrices A (1000 x 8) and B (8 x 1000)
- A and B have only 16000 parameters -> 1% of the initial 10^6 parameters

Advantages of LoRA

- The base model can be shared and used to build many small LoRA modules for different tasks
- LoRA makes training much more efficient and lowers the hardware requirements up 3 times in practice
- The performance of models fine-tuned with LoRA is comparable to the performance of fully fine-tuned models
- LoRA does not add any inference latency when adapter weights are merged with the base model

QLoRA

- Reduces greatly memory requirements during training
    - Allows fine-tuning of SOTA models on consumer hardware

- Keeps weights of base model quantized, dequantizes on demand for forward/backward pass
    - Uses 4-bit NormalFloat (NF4), an information theoretically optimal quantization data type for normally distributed data
    - Double Quantization, a method that quantizes the quantization constants
    - Reduces memory requirements by 90%
    - Possible because base model is frozen, thus quantization is pre-computed, weight gradients are computed only for the LoRA parameters

- Uses 16-bit BrainFloat for computations
- Uses paged optimizers, preventing memory spikes during gradient checkpointing from causing out-of-memory errors

4-bit NormalFloat Quantization

- Array of FP32 numbers ($f_1 \dots f_n$), want to store them in Int4 ($i_1 \dots i_n$)
- Linear quantization:
  - Divide by the global maximum to arrive at [-1…1] range: $q_k = f_k / c$, where $c = \max_j(\text{abs}(f_j))$
  - Convert to 4-bit integer with range [-7…7]: $i_k = \text{round}(q_k * 7)$
  - Quantization bins are equally distributed
- Pretrained neural network weights usually have zero-centered normal distribution ➔ quantization bins are not utilized well with linear quantization
- QLoRA splits [-1…1] in an information-theoretically optimal way, using quantiles for bin boundaries. This results in bins with normally distributed lengths, which better matches the distribution of pretrained neural network weights

Double Quantization

- Array of FP32 numbers($f_1 \ldots f_n$), want to store them in Int4 ($i_1 \ldots i_n$)
    - Quantize using global maximum, $c = \max_j(\mathrm{abs}(f_j))$
- Single constant will likely not work well if n is large ➔ divide into blocks
    - Use c_1 for $f_1 \ldots f_{64}$, c_2 for $f_{65} \ldots f_{128}$, etc.
    - Adds overhead
- QLoRA quantizes $c_1 \ldots c_{\{n/64\}}$, using 8-bit precision
    - Again, in blocks

Paged optimizers

- Gradient checkpointing creates memory spikes, when re-computing activations
  - Can lead to OOM errors
- To prevent this, QLoRA keeps optimizer states in unified pageable memory
  - GPU can move optimizer states to CPU memory if a spike leads to an OOM condition
  - Optimizer states only needed at the very end (after the forward and backward passes), thus efficient