

Efficient Finetuning of LLMs

Sumanth R Hegde

MS CS, UC San Diego



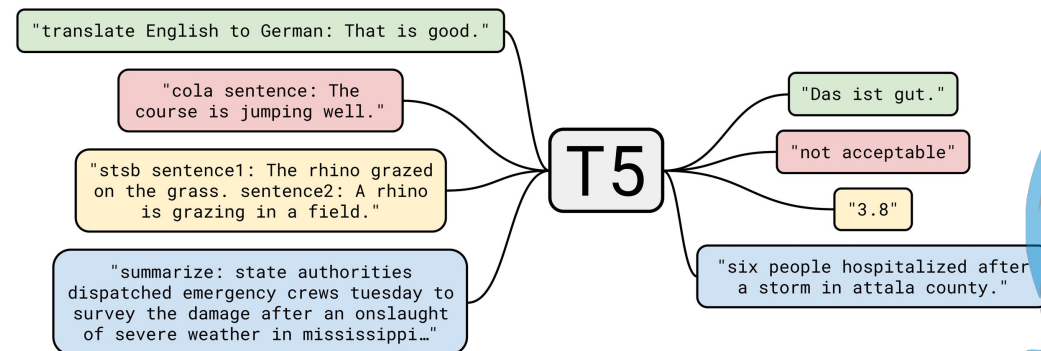
Outline

- A Primer on Large Language Models (LLMs)
- Fine-tuning LLMs
 - The What And The Why?
 - Metrics
 - Throughput/ Memory Optimizations
 - Mixed Precision
 - Parameter-Efficient Fine-tuning
 - Quantization
 - Gradient Checkpointing
 - Gradient Accumulation
- The Decision Tree



A Primer on Large Language Models

- “Large” : Billions of parameters
- “Language Model”: Predicts the probability of word(s) occurring in a sentence.
- Different variants: Causal language models (LLaMA, GPTs, etc) and Seq2Seq (T5, BART, etc)
- Underlying architecture (Usually): The Transformer (Vaswani *et al*)



T5



ABNASIA.ORG

What's so special about LLMs?

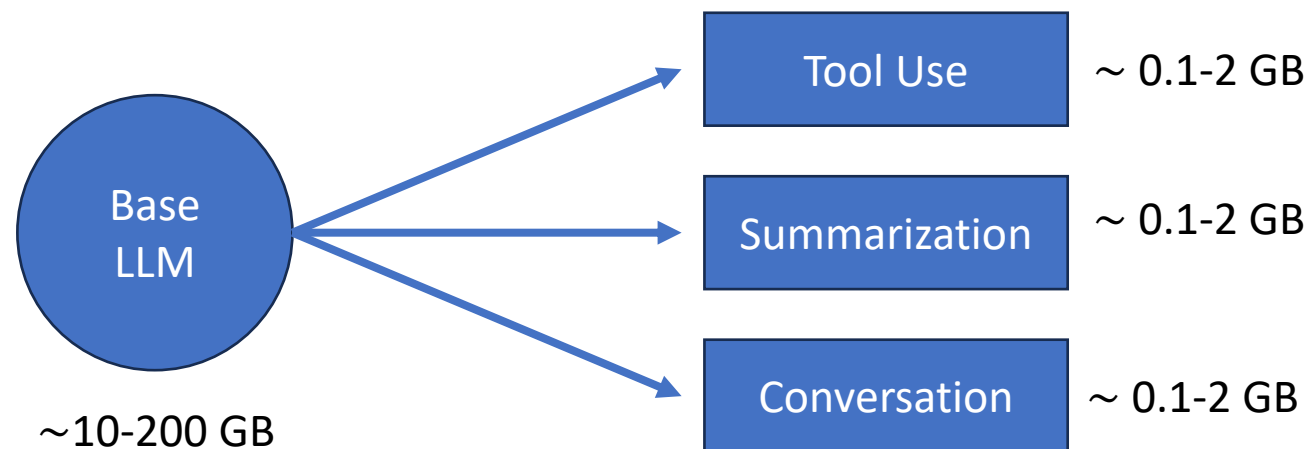
- World Knowledge: Trained on 1T+ tokens of internet-scale data
- Unsupervised Learning: Many skills learnt without explicit labels
- “Emergence”: Scaling from N to 10N params – with dataset scaling – leads to “significant” performance gains on many tasks.
- Emergence is about (1) new skills (2) new ability to *compose* different skills.¹

1: [A Theory for Emergence of Complex Skills in Language Models](#) by Prof. Sanjeev Arora



A New Era in DL Systems

- Llama 2 – 7B, 13B, 70B. GPT-3: 175B parameters
- 1T+ tokens for pre-training, 100M+ tokens for finetuning.
- 10s of GBs for inference, 100s of GBs for training even for <10B models
- *One model to do them all*: Multiple use cases, single model
 - New parameter-efficient techniques: multi-task + use-case-specific weights.



Outline

- A Primer on Large Language Models (LLMs)
- Fine-tuning LLMs
 - The What And The Why?
 - Metrics
 - Throughput/ Memory Optimizations
 - Mixed Precision
 - Parameter-Efficient Fine-tuning
 - Quantization
 - Gradient Checkpointing
 - Gradient Accumulation
- The Decision Tree



Fine-tuning LLMs

- **Pre-training:** Training an LLM from scratch (months)
- **Prompting/In-Context Learning:** Prompt an LLM with task details *in context* with a few examples. (seconds)
- **Fine-tuning:** Middle-ground, update a pre-trained model with 1000s-1M+ examples (hours/days).
 - Better use-case-specific performance than prompting.



Efficient Fine-tuning: Metrics

- Typically, we want training to:
 - Be as fast as possible (Speed)
 - Use as little as possible (Memory)
 - Make the model as performant as possible (Performance)
- “How much time will it take?” $T = T_{\text{mem}} + T_{\text{math}} + \text{Latency}$
- *GPU Utilization*: % of GPU processing power used
- *Throughput*: Rate at which we process/operate on training samples
 - Data view: Samples/ second or Tokens/second
 - Compute view: FLOPS
- *Model FLOPs Utilization (MFU)*: $\frac{\text{Observed Throughput}}{\text{Peak Achievable Throughput}}$

$\max(T_{\text{mem}}, T_{\text{math}})$ is what matters!



Efficient Fine-tuning: Optimizations

- The Memory Requirements Breakdown
- Mixed Precision
- Parameter-Efficient Fine-tuning (PEFT)
- Quantization
- Gradient Checkpointing
- Gradient Accumulation



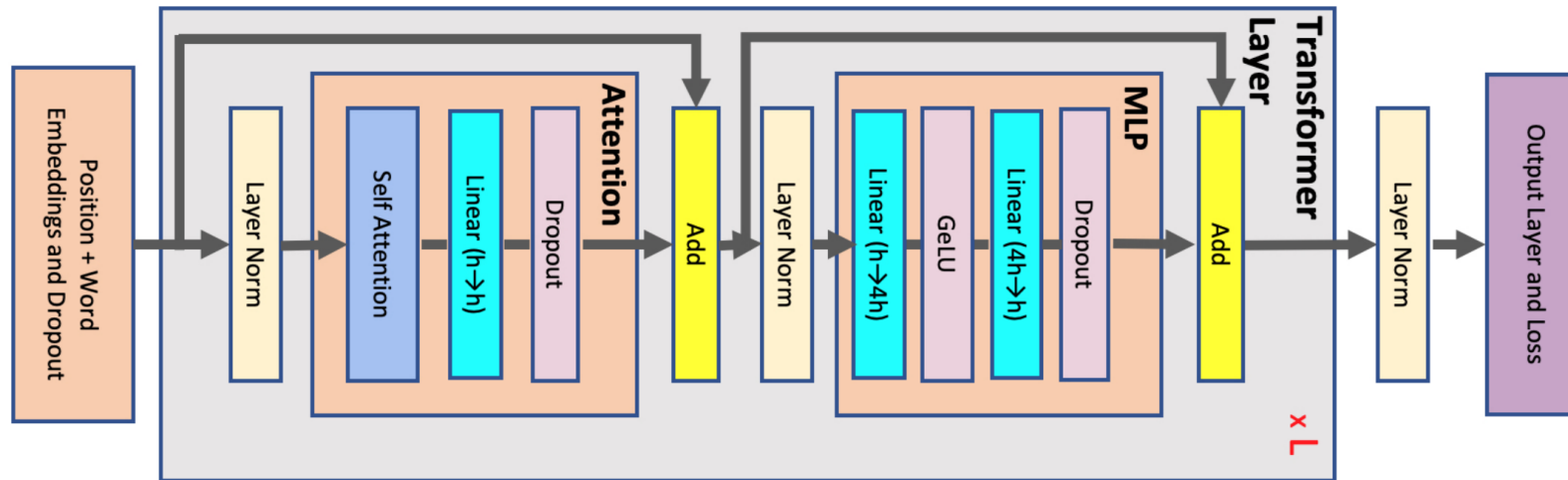
The Memory Requirements Breakdown

Total memory = Model weights + Gradients + Optimizer State + Activations

- Model weights = 4 bytes / param
- Gradients = 4 bytes / param
- Optimizer State – Depends!
- Adam/ AdamW:
 - Momentum: 4 bytes / param
 - Variance 4 bytes / param
 - Total = 8 bytes / param
- Activations?
 - Architecture and Input dependent (Not a simple “x bytes/param/token”)



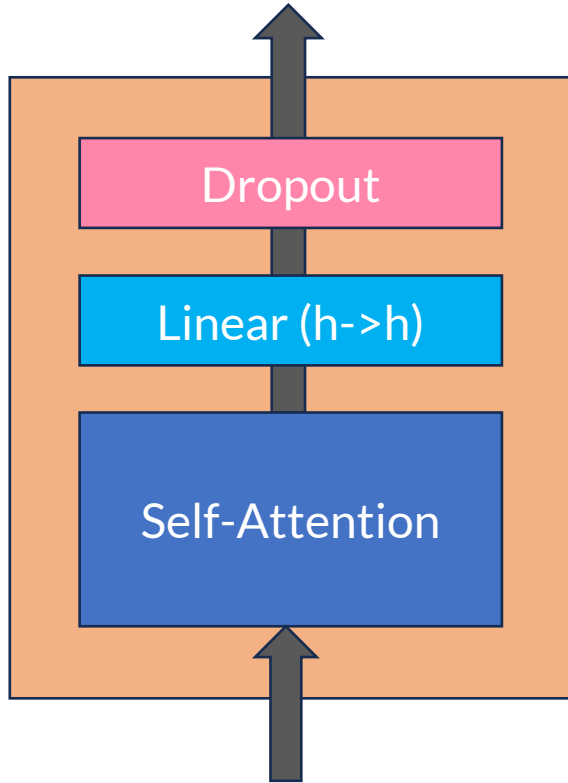
Activation memory for a Transformer



- Number of bytes per element N_b (4 – FP32, 2 – FP16/BF16)
- Sequence length s , Batch size b , Hidden dimension h
- Assume a attention heads and L transformer layers
- Input/ Output size: $N_b sbh$

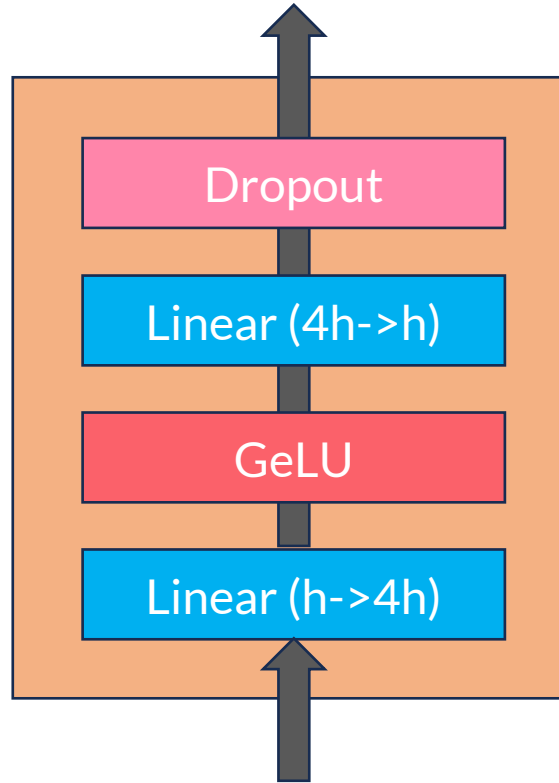


Activation memory for a Transformer



Attention block

$$(4N_b)sbh + (2N_b + 1)as^2b + N_bsbh + sbh$$



MLP block

$$N_bsbh + N_bsb(4h) + N_bsb(4h) + sbh$$

Layer Norm

Layer Norm

$$N_bsbh + N_bsbh$$



ABNASIA.ORG

Activation memory for a Transformer

- Total Activation Memory: $L((16N_b + 2)sbh + (2N_b + 1)as^2b)$
- Example for Llama-2-7B with $s=512, b=1, N_b=2$: ~5 GB
- Example for Llama-2-7B with $s=4096, b=1, N_b=2$: ~100 GB
- More: “Reducing Activation Recomputation in Large Transformer Models” by Korthikanti *et al.*

Attention block

$$(4N_b)sbh + (2N_b + 1)as^2b \\ + N_b sbh \\ + sbh$$

MLP block

$$N_b sbh \\ + N_b sb(4h) \\ + N_b sb(4h) \\ + sbh$$



ABNASIA.ORG

Efficient Fine-tuning

- Focus is on the single-GPU setting
- These are (mostly) memory optimizations
- Only two questions matter:
 - Can I train with *batchsize*=1? (0->1)
 - Can I go to *batchsize*=n? (1->n)
- The tradeoffs:
 - How much time will this take (throughput)?
 - How good will my model be (performance)?
- Keep in mind:
 - LLM finetuning can (typically) tolerate lower batch sizes (say, n=4)
 - Larger batch size != More throughput



Efficient Fine-tuning

Optimization	Memory	Throughput	Model Performance
Mixed Precision	●	↑	■
Parameter Efficient Fine tuning	↓	↑	●
Quantization	↓	↓	●
Gradient Accumulation	↓	●	■
Gradient Checkpointing	↓	↓	■

■ No Change

● It Depends



ABNASIA.ORG

Mixed Precision

- -> Weights, activations, gradients - lower precision
- -> Optimizer states - full precision (Hence, “mixed”)
- Memory:
 - Model weights and Gradients*: 2 bytes / param
 - Activations: $N_b = 2$ (~0.5x reduction)
 - Optimizer (Adam): 12 bytes / param
 - “Master copy” of weights, Momentum and Variance all need 4 bytes / param
- Throughput:
 - NVIDIA: Tensor Core – 2-4x speedup

Gradients take 4 bytes in reality! Non-activation memory has increased!

*In reality, with `torch.autocast`, assume 4 bytes

Mixed Precision: Formats

- Popular formats: FP16 and BF16 (Brain Float 16)
- FP16 has much smaller range => Affects training stability!
- FP16 needs loss scaling.
- BF16 doesn't have this issue!
- New format: FP8 (just 1 byte!)

(a) fp32: Single-precision IEEE Floating Point Format Range: $\sim 1e^{-38}$ to $\sim 3e^{38}$



(b) fp16: Half-precision IEEE Floating Point Format Range: $\sim 5.96e^{-8}$ to 65504



(c) bfloat16: Brain Floating Point Format Range: $\sim 1e^{-38}$ to $\sim 3e^{38}$

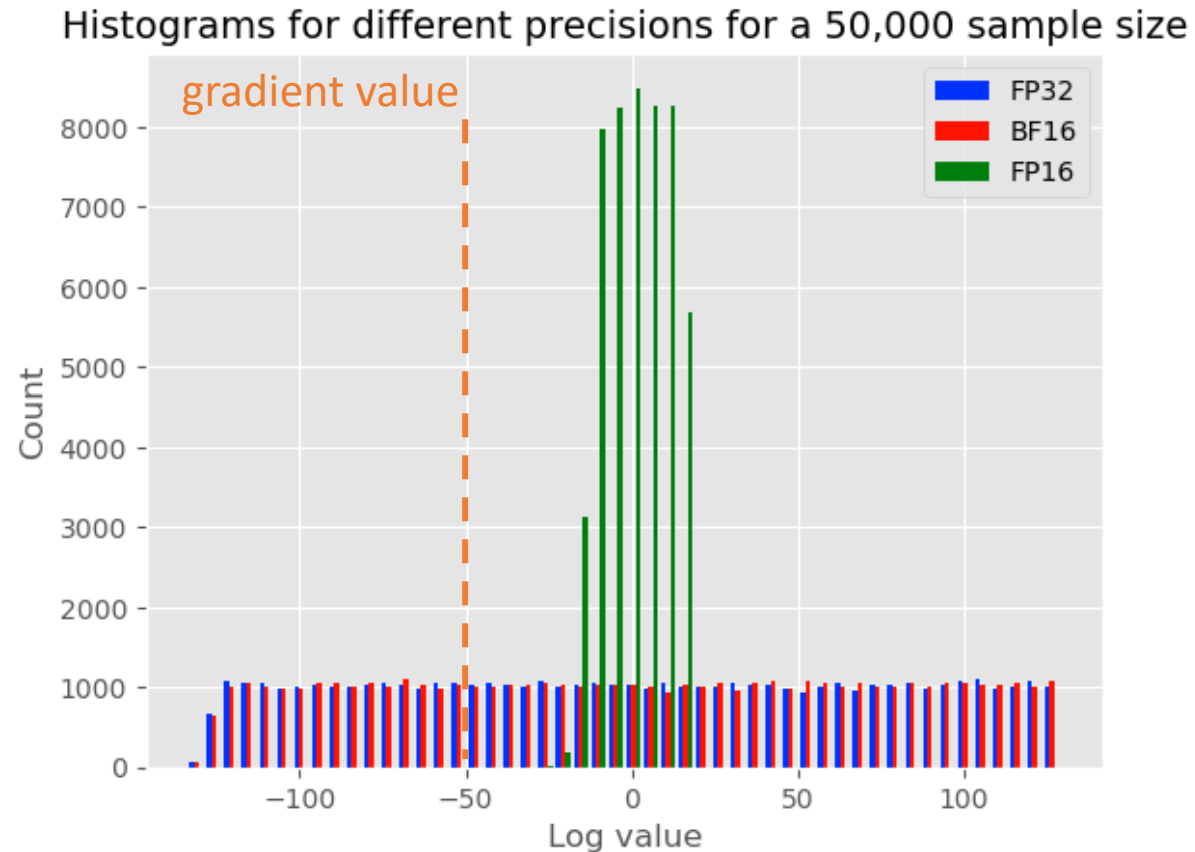


Source: [Google Cloud's BF16 guide](#)



ABNASIA.ORG

Mixed Precision: Formats



- Loss scaling is like moving FP16's fixed bucket/range across the FP32 range.
- Dynamic loss scaling implemented for you in Pytorch AMP.



Lower Precision Formats: Summary

- How should I think about FP16, BF16, FP8, etc?
- Pros:
 - Lesser memory : 4 bytes -> 2 bytes/ 1 byte
 - Lower T_{mem} and T_{math}
- Cons:
 - Accumulation is lossy
 - Direct weight updates will be noisy
 - Smaller range implies small gradients are lost

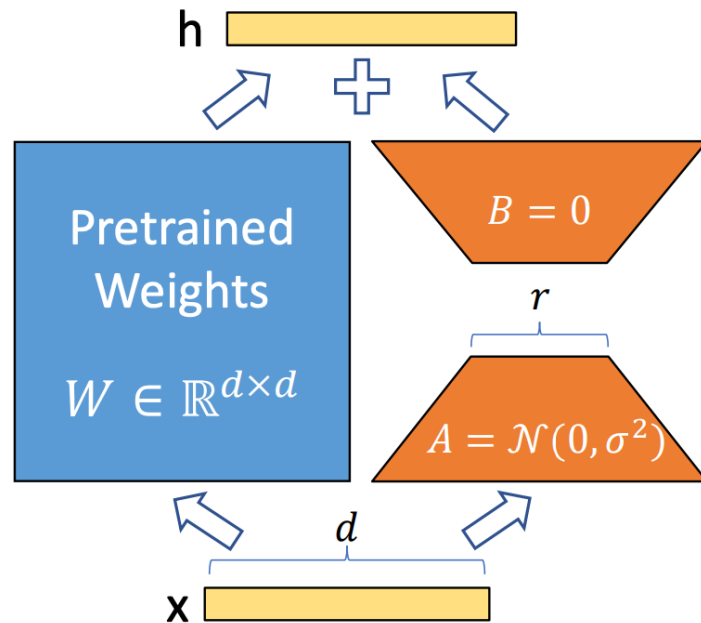


Parameter-Efficient Fine-Tuning (PEFT)

- Motivation: Do we really need to update all model weights?
- Motivation: A separate fine-tuned model for each downstream task gets expensive. (7B -> 14GB)
- Idea: Train a small number of (extra) model parameters and freeze pre-trained weights.
- Benefits: Lesser memory requirements, better throughput
- Where's the tradeoff?
 - Performance! (but not much)
 - (sometimes) Inference latency

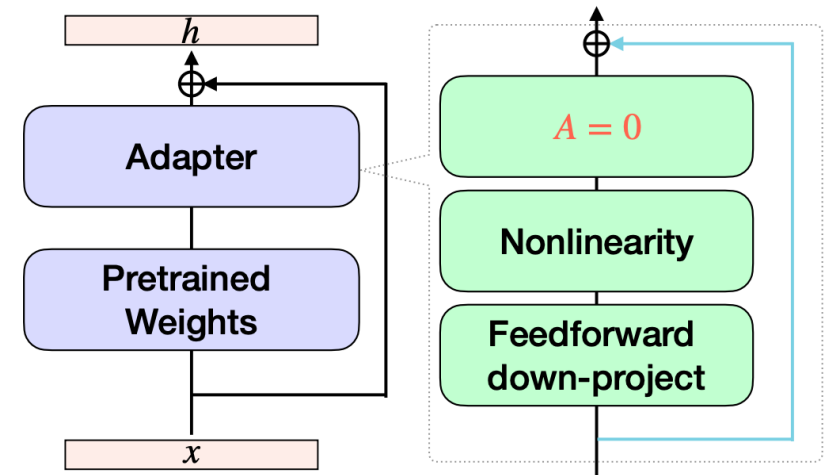


Parameter-Efficient Fine-Tuning (PEFT)

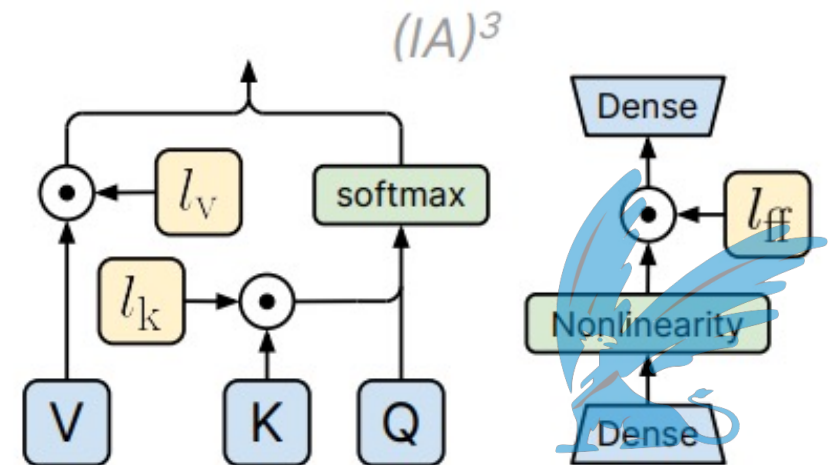


LoRA (Hu *et al*, 2021)

Adapter (Houlsby *et al*, 2019)

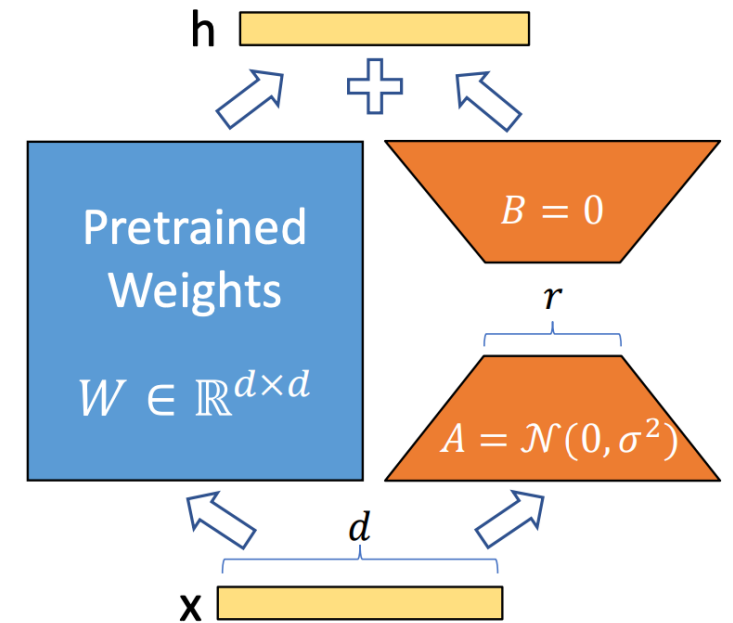


IA³ (Liu *et al*, 2022)



PEFT: Low Rank Adaptation (LoRA)

- Linear layers: $h = Wx$
- With LoRA: $h = Wx + \Delta W x = W_0x + sBA x$
- Hyperparameters:
 - rank r
 - List of layers to apply to
 - Scaling factor s
- At inference time, *merge* LoRA weights: $W = W_0 + BA$. Same inference cost!

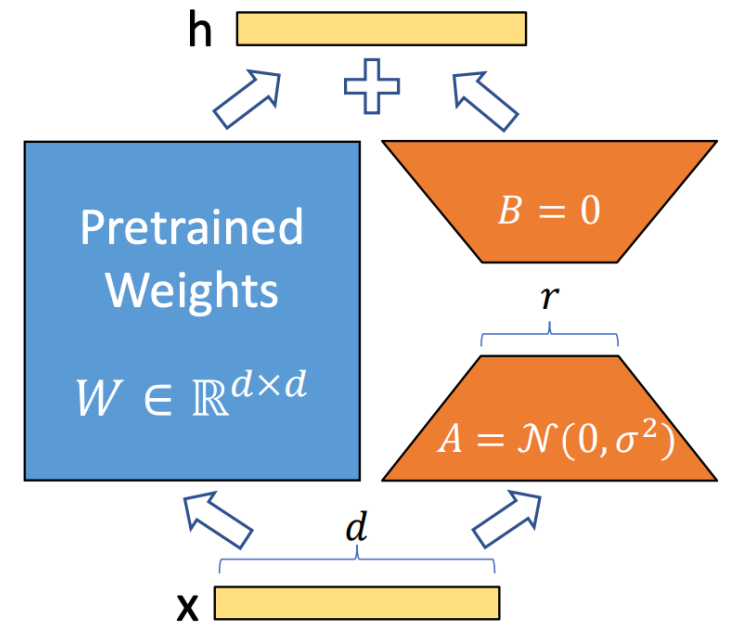


LoRA (Hu *et al*, 2021)



PEFT: LoRA

- Full-parameter fine-tuning for Llama-2-13B:
 - Weights: 2 bytes * 13B = 26GB (BF16)
 - Gradients = 52 GB
 - Optimizer State: 12 bytes * 13B = **156 GB**
 - Total = **234 GB** + Activation memory
- PEFT with LoRA (assume 0.4% more params):
 - Base weights: 2 bytes * 13B = 26GB
 - LoRA weights: 26GB * 0.4% = 0.10GB
 - Gradients: 52GB * 0.4% = **0.20 GB**
 - Optimizer State = 12 bytes/ param * 13B * 0.4% = **0.62GB**
 - Total = **26.92 GB** + Activation memory
- Checkpoints become tiny in size! (~100 MB)



LoRA (Hu *et al*, 2021)



ABNASIA.ORG

PEFT: TradeOffs

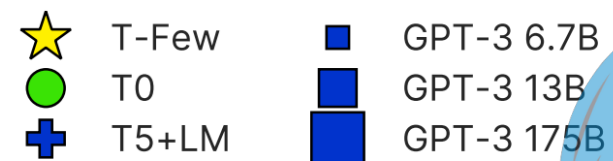
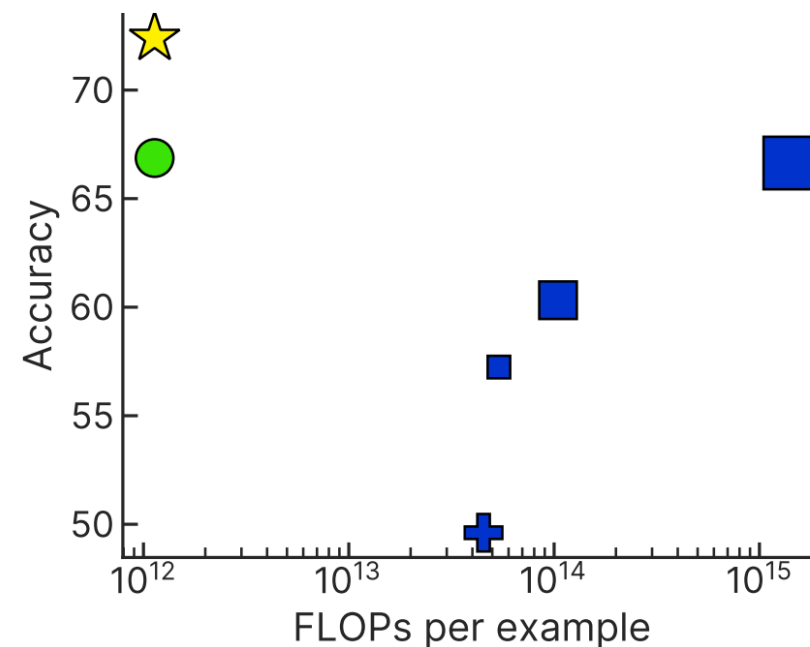
- LoRA and IA³ enable efficient training *without* increase in inference latency
- Throughput?
 - Depends! Hu *et al.* report 25% speedup for GPT-3
- Performance??
 - Sometimes worse than full-parameter at default (e.g. a hard math dataset¹)
 - On many tasks, can match full fine-tuning performance ([Dettmers et al.](#))!
 - $r=64$ for all linear layers

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



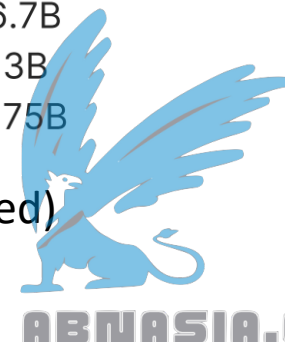
PEFT: An Ecosystem of Specialized Models

- Specialist models (vs large generalist models):
 - Usually *cheaper*
 - Can be *better* if finetuned
- PEFT methods: build a basket of finetunes on *different* tasks that *share* the same *base* model.
- Can even “route” different ex. to appropriate task at inference (like an MoE)



T-Few = T0 + IA^3 (finetuned)
T0 (zero-shot)
GPT3 – (few-shot ICL)

More: [Build an Ecosystem, Not a Monolith](#) by Prof. Colin Raffel



Quantization

- Quantization : essentially “rounding”
- But how can int8 handle the large range of fp16 ?

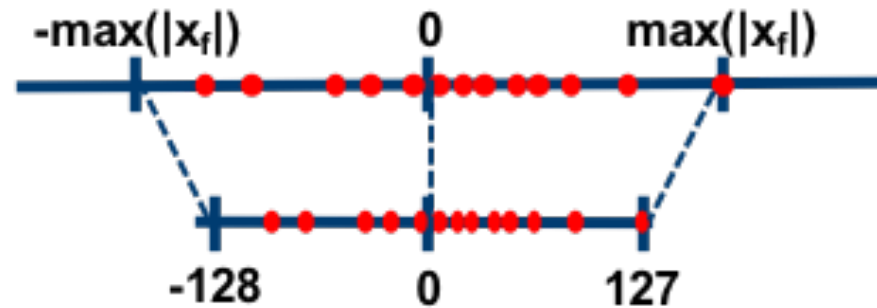
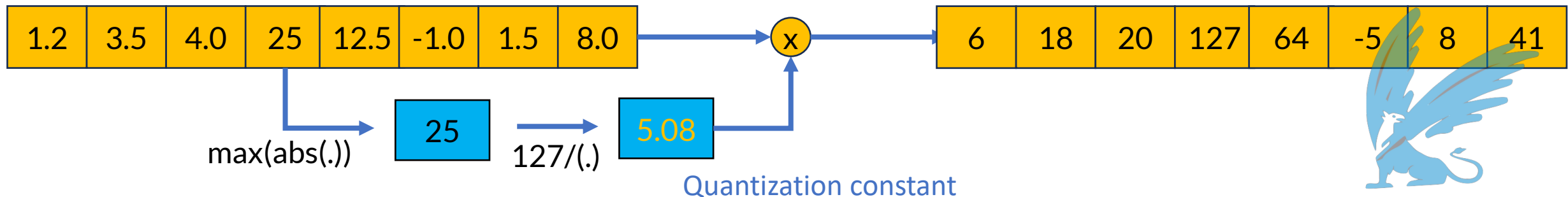
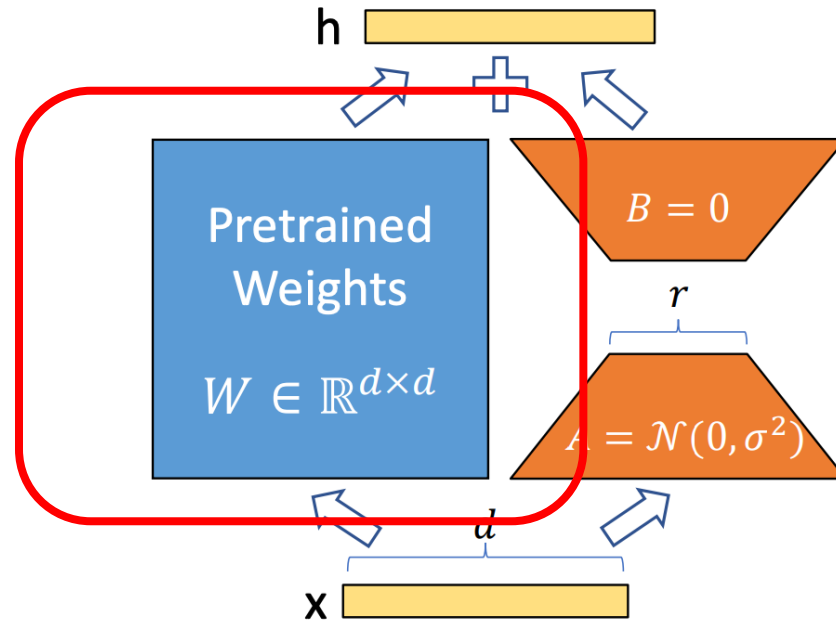


Image Source



QLoRA

Quantize base weights



During Forward Pass:
Dequantize base weights and
compute activations as usual

Simple? No!

- New 4-bit representation: NormalFloat (NF4)
- Custom 4-bit CUDA kernels for efficient matmuls
- Double Quantization: Quantize the quantization constants

QLoRA

- PEFT with LoRA (assume 0.4% more params):
 - Base weights: 2 bytes * 13B = 26GB
 - LoRA weights: 26GB * 0.4% = 0.10GB
 - Gradients: 52GB * 0.4% = 0.20 GB
 - Optimizer State = 12 bytes/ param * 13B * 0.4% = 0.62GB
 - Total = 26.92 GB + Activation memory
- QLoRA (4bit):
 - Base weights: 0.5 bytes * 13B = 6.5GB
 - LoRA weights: 26GB * 0.4% = 0.10GB
 - Gradients: 52GB * 0.4% = 0.20 GB
 - Optimizer State = 12 bytes/ param * 13B * 0.4% = 0.62GB
 - Total = 7.42 GB + Activation memory
- You can even finetune 20B models on a free Colab instance (1 T4 GPU)!



QLoRA: TradeOffs

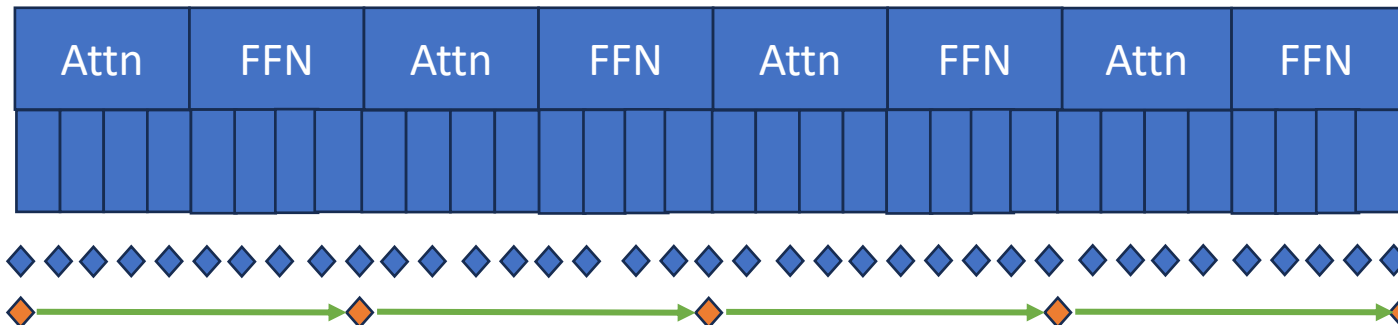
- Throughput?
 - Quantization has worse throughput. (~[30% for single-GPU](#))
 - Why? Extra dequantization step in forward pass! (T_{math})
- Performance?
 - Dettmers *et al.* show that QLoRA can match full-parameter finetuning
- QLoRA – to be used in a consumer hardware setup with 1-2 GPUs
 - Ex. 2 RTX 3090s (Total: 48GB HBM) to finetune a 33B model
- Quantization almost never needed with a multi-GPU (4+ GPUs) setting.

Optimization	Memory	Throughput	Model Performance
Quantization	↓	↓	●



Gradient/Activation Checkpointing

- Typically, all intermediate activations are saved during the forward pass
 - Needed during the backward pass.
- Llama-2-13B with $N_b=2$, $s=512$, $b=4$: 21 GB
- Activation checkpointing: instead of saving all activations, save a few!
- Reduces Activation Memory at the cost of some slowdown
 - Single-GPU: $\sim 30\%$ slowdown.
- New Memory: Depends on the strategy! For Llama-13B ex. + HF : 800 MB



Gradient Accumulation

- Goal: maximize batch size as much as possible
 - Why? Better gradient updates, Better throughput (usually...)
- Let's say you're able to train with batch size 4.
- Gradient accumulation: Do the optimization step only every few training steps
 - `gradient_accumulation_steps=4` Forward+Backward pass happen as usual
Gradients get accumulated, optimization step every 4 steps
 - Effective batch size: 16
- Tradeoffs?



Gradient Accumulation

- Slower than regular training at the same effective batch size
- Compare with training w/o accumulation.
 - Typically faster (why?)
 - Lesser all-reduce ops / communication overhead per iteration
 - Gets better with multi-GPU/ multi-node setups
- Where are the memory savings again?
 - Activations – batch size gets scaled down

Optimization	Memory	Throughput	Model Performance
Gradient Accumulation	↓	●	■



Efficient Fine-tuning: Another Look

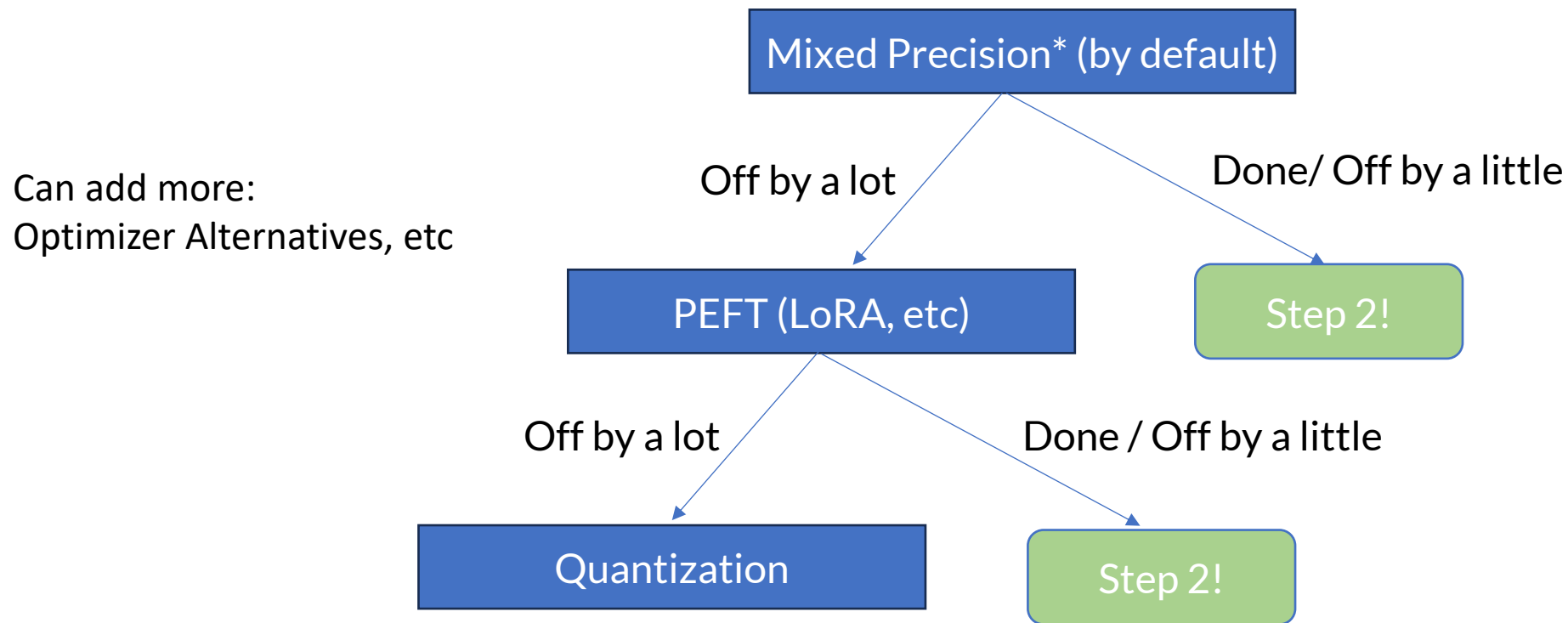
Optimization	Memory	Memory Terms affected	Throughput	Model Performance
Mixed Precision	●	All (Activation)	↑	■
PEFT	↓	Optimizer + Gradient	↑	●
Quantization	↓	Model	↓	●
Gradient Accumulation	↓	Activation	●	■
Gradient Checkpointing	↓	Activation	↓	■

■ No Change
● It Depends



Efficient Fine-tuning: The Decision Tree

Step 1: Target Weights, Optimizer and Gradients (0->1)



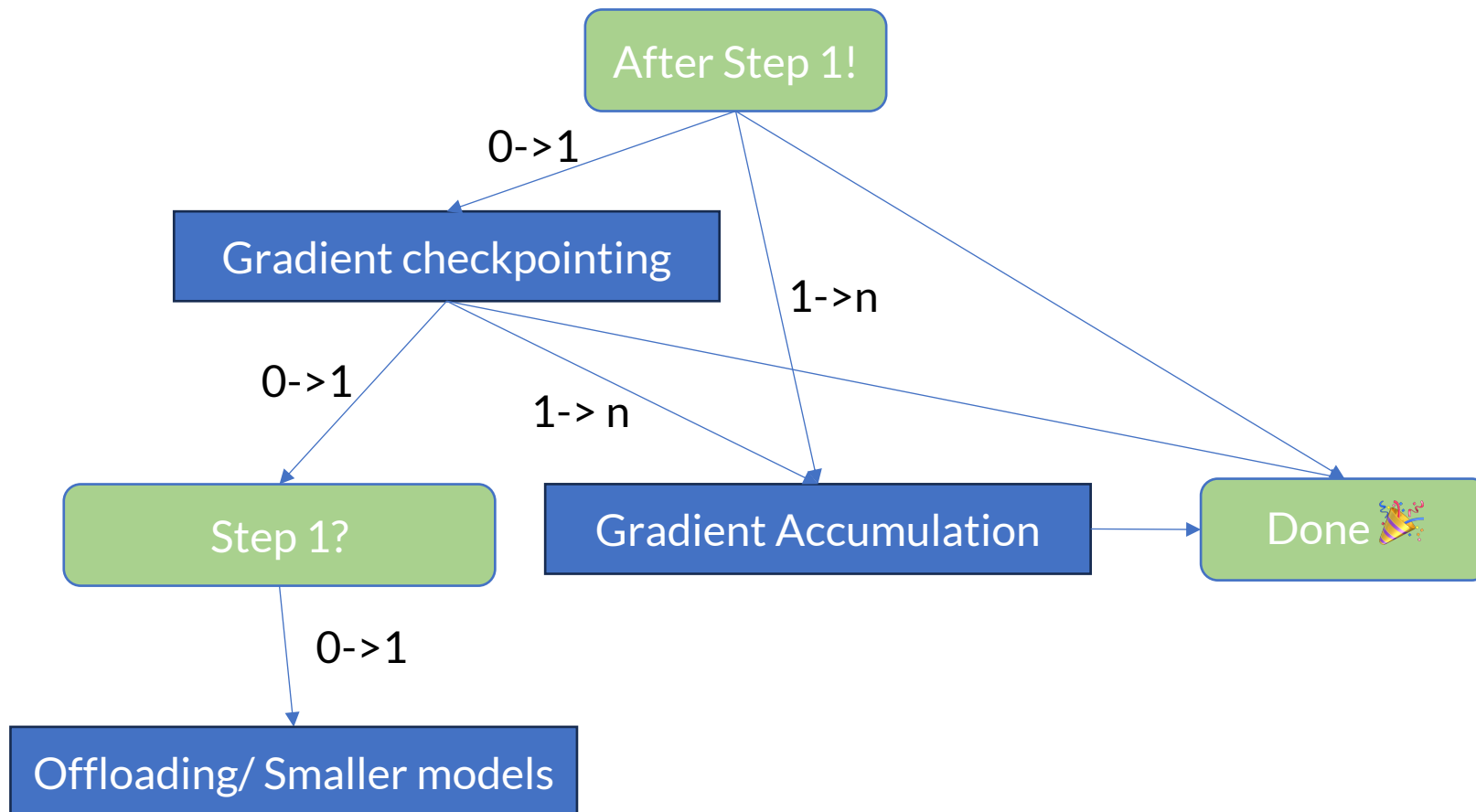
At each step, calculate: Total Memory (bs=1) - Available Memory

P.S: Flash Attention should be the default!



Efficient Fine-tuning: The Decision Tree

Step 2: Target Activations (0->1 then 1-> n)



There's More to Memory Usage...

- CUDA Kernels: Kernels loaded onto the GPU by your framework.
 - Pytorch: assume a max of [1GB](#)
- Temporary Buffers: Intermediate activations, Backward pass, etc
 - Checkpointing: All activations between two checkpoints stored at once
- Memory Fragmentation: Not enough contiguous memory
- And more.....



Takeaways

- Intuition for Efficient but Fast finetuning
 - LoRA, Quantization, etc
- Memory: Every byte per parameter matters!
- Batch size, Utilization, Throughput
- “I wanted to increase batch size, so I added quantiza...”
 - Straight to Jail! 🚩
- Lot more you can do for throughput: fused optimizers, compilation, etc
- Multi-GPU setups: Also consider communication overhead



More on LLMs

- Stas Bekman's Engineering Blog: <https://github.com/stas00/ml-engineering>
- HuggingFace's Performance Guide: <https://huggingface.co/docs/transformers/v4.20.1/en/performance>
- More from **me!**
 - Everything about Distributed Training and Finetuning: <https://sumanthrh.com/post/distributed-and-efficient-finetuning/>
 - Everything about Tokenization: <https://github.com/SumanthRH/tokenization/>



Thank you!
Questions?

