Large Language Models

Efficient fine-tuning and inference

Flavio Giobergia



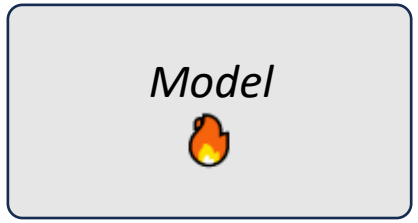
The need for fine-tuning

- Language models are few-shot, or even zero-shot learners
  - GPT-3 (*"Language Models are Few-shot Learners"*)
  - FLAN (*"Finetuned Language Models Are Zero-Shot Learners"*)

- However, the upper bound in performance is generally set by fine-tuned models
  - Even much smaller models, when fine-tuned on a task, can outperform bigger models used with In-Context Learning

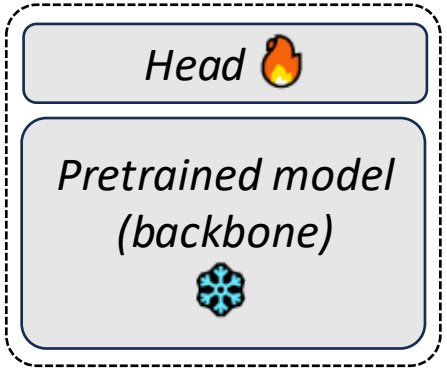
- So, we may still need to fine-tune models!

Fine-tuning



- Fine-tuning is a simple (and intuitive) approach to improve a model's performance on new tasks
- In fine-tuning, we *continue the training* of a pretrained model on the new (fine-tuning) data
  - All of the original model's weights can change (🔥)
- Pros
  - Allows achieving performance comparable to training from scratch
  - Smaller datasets can be used to change the behavior of pretrained models
    - (See instruction tuning, model alignment)
- Cons
  - Resource intensive for large models

Feature-based transfer



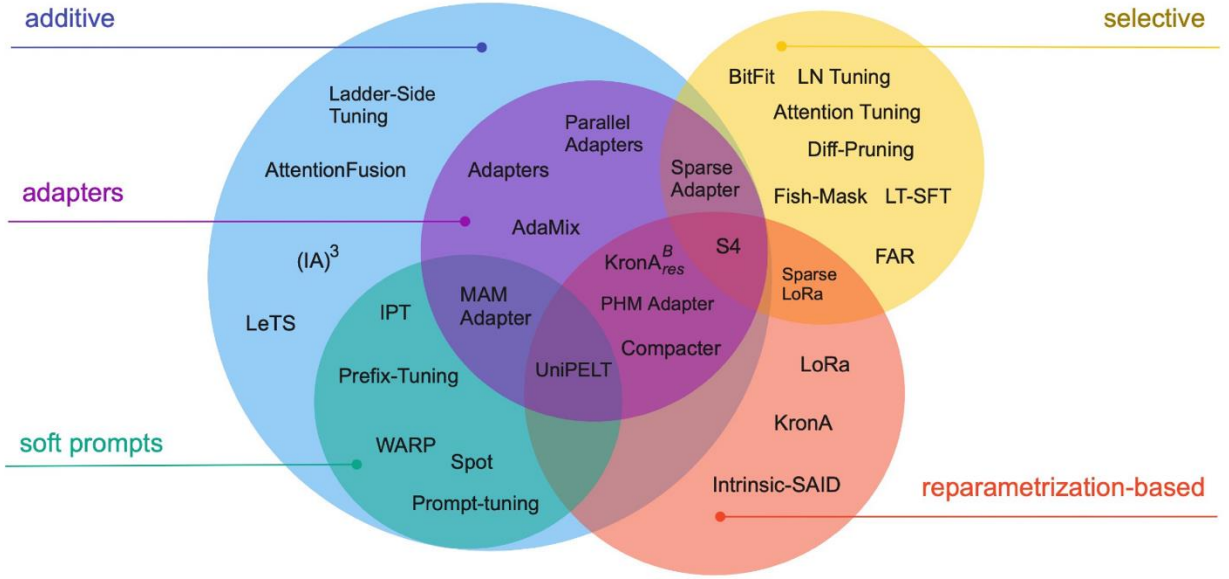
- We *freeze* (❄) the *backbone model*
  - i.e., no gradient updates will be computed for those weights
- We add a *trainable head*
  - Only the weights of the head are updated
  - Optionally, we can "unfreeze" the last few layers of the backbone
- The original model is used as a *feature extractor*
- The head uses the extracted features
- Pros
  - Only changes a small fraction of all possible weights (less resource intensive)
  - Works well when the original and the new task(s) are similar
- Cons
  - Complex (or more diverse) tasks require deeper changes
  - The performance on these tasks will be sub-optimal

Problems with "classic" approaches

- Fine-tuning and feature-based transfer introduce opposite benefits/problems
- As *models grow larger* and *tasks diversify*, we typically struggle to use one or the other approach
- Other families of approaches emerged to overcome the limitations of the abovementioned techniques
- *Parameter-efficient Fine-Tuning* (PEFT)

Parameter-efficient Fine-Tuning

- Techniques used to reduce the computational cost required for the fine-tuning of models
- PEFT techniques reduce the number of parameters to update
- Among the others, there are:
  - BitFit
  - Adapter layers
  - LoRA (Low Rank Adaptation)
  - Prompt tuning



Scaling Down to Scale Up: A Guide to Parameter-Efficient Fine-Tuning (https://arxiv.org/pdf/2303.15647)

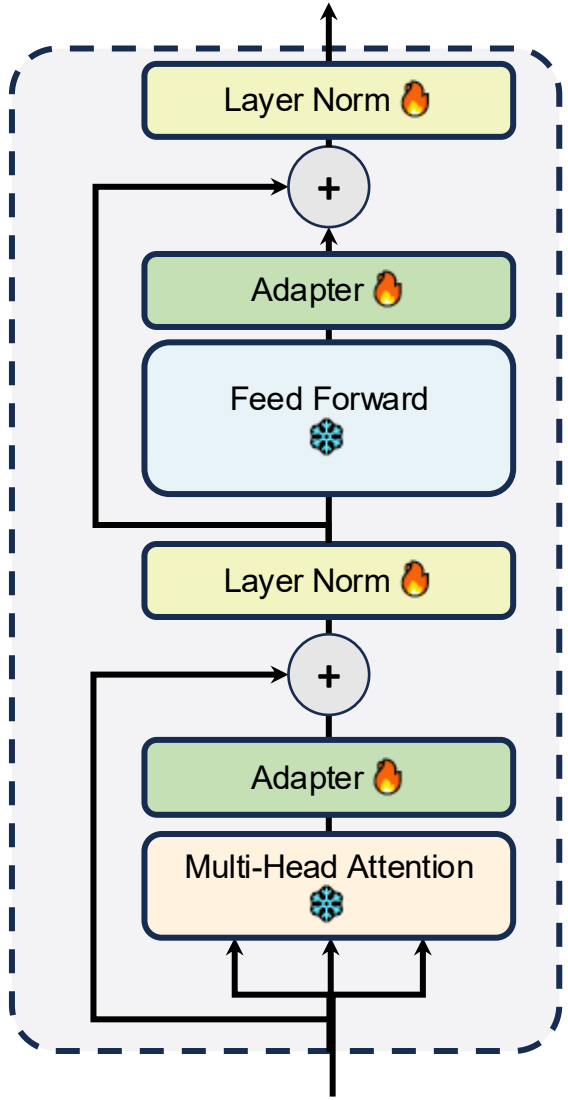
Bias-terms Fine-tuning (BitFit)

- BitFit is a very simple sparse fine-tuning technique, where only the bias terms of the model (or a subset) are modified
- Bias terms characterize a small fraction of model weights
  - E.g., 0.1% for BERT
- Zaken et al., 2021 show that tuning the bias terms is sufficient to get performance comparable to fine-tuning
- *"Bias terms and their importance are rarely discussed in the literature. Indeed, the equations in the paper introducing the Transformer model (Vaswani et al., 2017) do not include bias terms at all, and their existence in the BERT models might as well be a fortunate mistake"*

Zaken, Elad Ben, Shauli Ravfogel, and Yoav Goldberg. "Bitfit: Simple parameter-efficient fine-tuning for transformer-based masked language-models." *arXiv preprint arXiv:2106.10199* (2021). https://arxiv.org/pdf/2106.10199

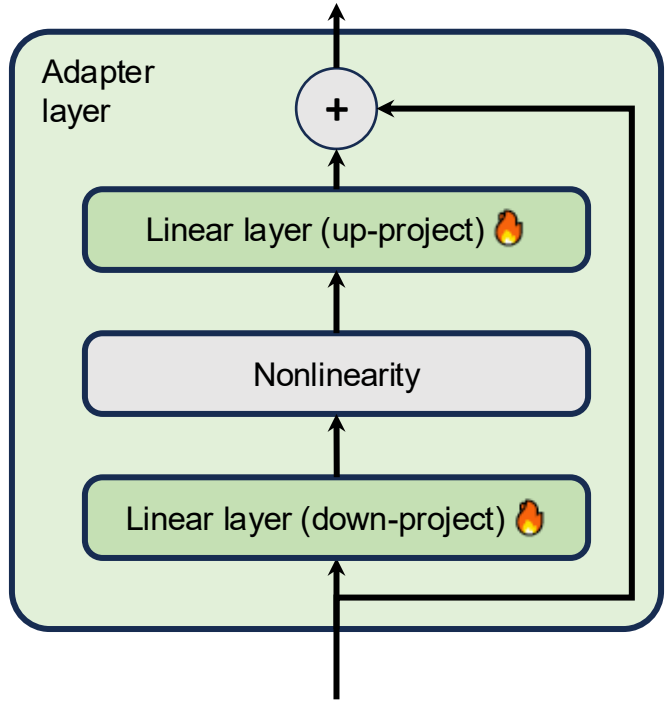
Adapters

- Introduces additional layers inbetween existing (attention or fully-connected) layers
- Only the newly introduced layers (adapters) are trained
- The rest of the model is frozen
  - Except for Layer Norms (small, and introduce some more flexibility)
- This reduces the overall number of trained parameters
- But allows changing the model at various depths (unlike feature-based transfer!)



Houlsby, Neil, et al. "Parameter-efficient transfer learning for NLP." International conference on machine learning. PMLR, 2019. http://proceedings.mlr.press/v97/houlsby19a/houlsby19a.pdf

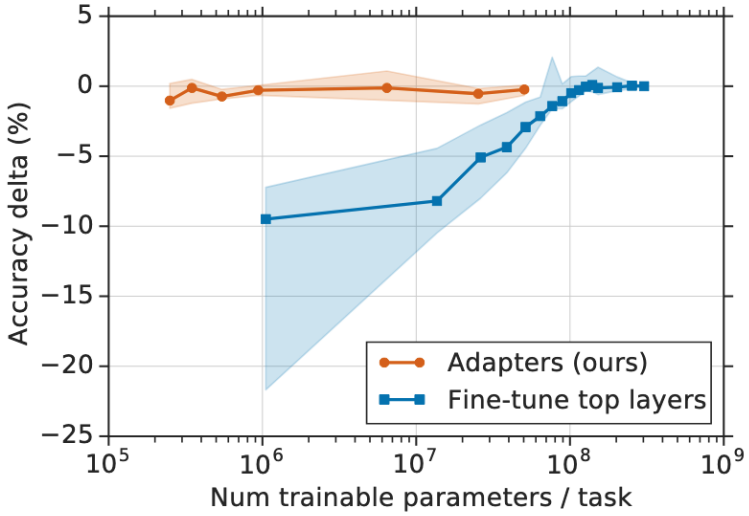
Adapter layer



- The adapter layer is characterized by a simple fully-connected model
  - Down-projecting layer (e.g., 768 → 32)
  - Non-linearity (e.g., ReLU)
  - Up-projecting layer (e.g., 32 → 768)
- More complex architectures have been studied in the original paper, but with no added benefits
- The adapters are "injected" in a pretrained (already working) model
  - If initialized randomly, it could "ruin" the intermediate results
  - Initially, the adapter should behave as an identity function (adapter(x) ≈ x)
  - Then, during fine-tuning, the layer adapts the intermediate results
  - To get an identity-like behavior, a residual connection is used
    - And the up-projecting layer is initialized with values close to 0

Results of adapters

- In terms of *reduction of number of trainable parameters*
  - A single transformer layer in BERT has ~7M parameters
  - Adapters (32 dimensions) have:
    - 768*32 + 32 + 32*768+768 → ~50K parameters
    - Two adapters for each layer → 100K parameters/layer
    - 70x fewer parameters to tune!
- In terms of *performance*
  - Authors show that, empirically, they achieve similar performance to using classic fine-tuning
  - (But at a fraction of the cost!)

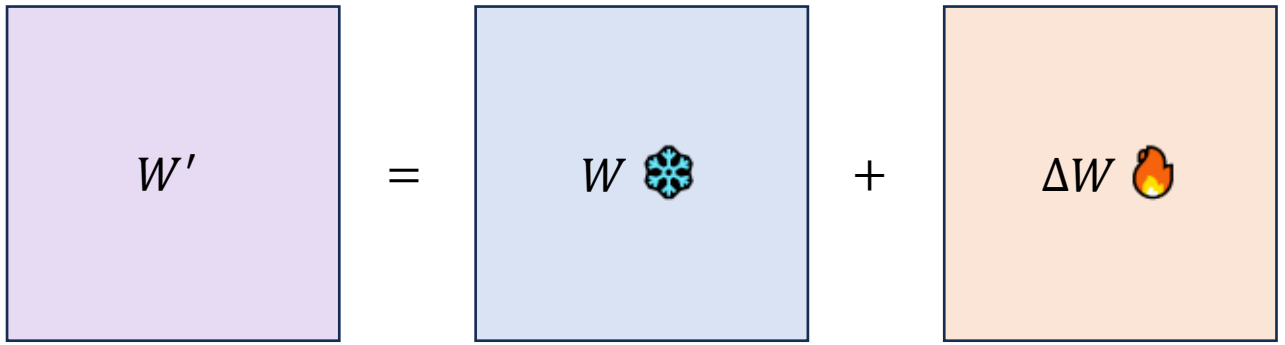


(Source: original paper)

Low Ranking Adaptation (LoRA)

- Fine-tuning of a model (e.g., a single layer W) produces an incremental change ΔW applied to the weights
- After fine-tuning, we have new weights W' such that:

$$W' = W + \Delta W$$



- We can fix (freeze) W and only learn the incremental change ΔW during fine-tuning

Hu, Edward J., et al. "Lora: Low-rank adaptation of large language models." *arXiv preprint arXiv:2106.09685* (2021). https://arxiv.org/pdf/2106.09685

Rank of matrices (digression)

- The rank of a matrix is the number of linearly independent rows (columns)

- An $n \times n$ matrix F is full rank if all rows (columns) are linearly independent
  - It can be factorized as $F = AB^T$, both A and B are $n \times n$

- An $n \times n$ matrix L is low rank if some rows (columns) are linear combinations of the other
  - $L = AB^T$, A and B are $n \times r$ (r being the rank of L)

Low-rank factorization (digression)

- This 3x3 matrix is low rank
  - row 2 = row 1 * 2
  - row 3 = row 1 * -1
  - rank = 1

- It can be factorized into two 3x1 matrices
  - A = [1 2 -1], B = [1 2 1]

	1	2	1
1	1	2	1
2	2	4	2
-1	-1	-2	-1

Near low-rank (digression)

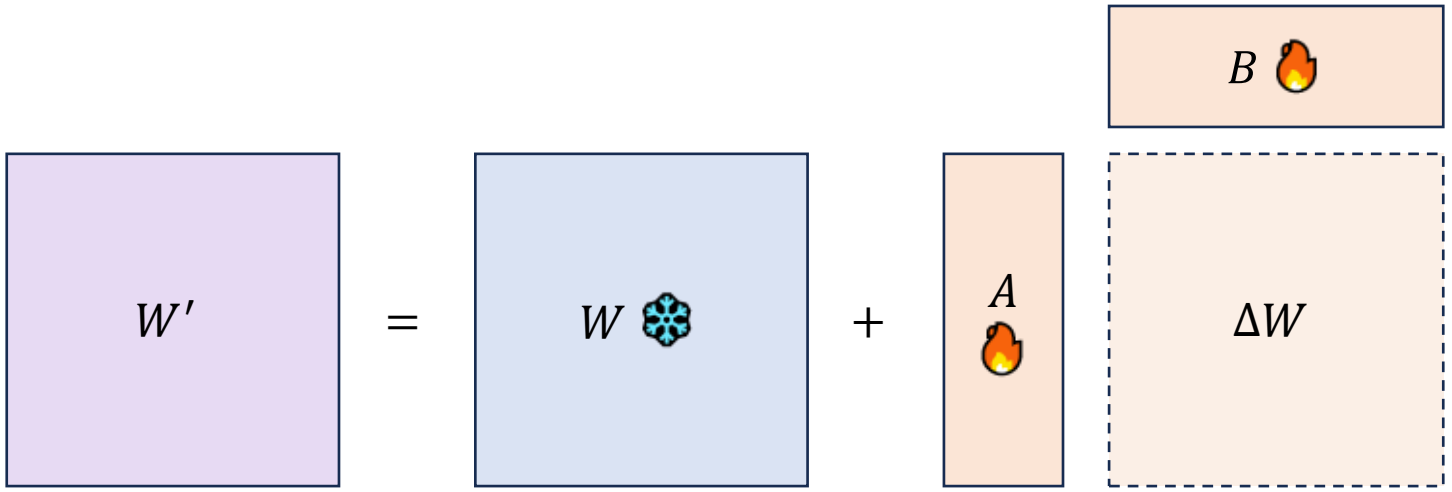
- What about this matrix?

1	2	1
2.01	4	2
-1	-2	-1.01

- It is *technically* full rank
- However, it's just the previous matrix, with very small changes!
- We call these matrices, *near low-rank*
- Even if we cannot write it as AB^T, we can still *approximate* it
- The closer the matrix is to actually being low rank, the better the approximation will be

LoRA (low rank assumption of ΔW)

- Learning ΔW implies having to fine-tune the same number of parameters, so by itself is not particularly useful
- But, if we assume (verify empirically) that ΔW is generally near low-rank, we can approximately factorize it into smaller matrices A, B



Low-Rank Adaptation

$$W' = W + AB^T$$

- Since we approximate ΔW , we can compute the gradients for (and optimize) A, B directly
- We *learn* A, B such that AB^T will be *approximately* the desired ΔW
- If ΔW is *near low-rank*, we can produce a good approximation with A, B
- So we can choose the rank to use (r) in the shapes of A, B
- If $r < \frac{n}{2}$, then A, B collectively have fewer parameters than ΔW
- Before fine-tuning, we want $\Delta W = 0$ (no update w.r.t. the pretrained model)
  - This is achieved by sampling A from $\mathcal{N}(0, \sigma^2)$ and $B = 0$ (so, $AB^T = 0$)

Results of LoRA

(Source: original paper)

Model & Method	# Trainable Parameters	E2E NLG Challenge BLEU	NIST	MET	ROUGE-L	CIDEr
GPT-2 M (FT)*	354.92M	68.2	8.62	46.2	71.0	2.47
GPT-2 M ($\text{Adapter}^{\text{L}}$)*	0.37M	66.3	8.41	45.0	69.8	2.40
GPT-2 M ($\text{Adapter}^{\text{L}}$)*	11.09M	68.9	8.71	46.1	71.3	2.47
GPT-2 M ($\text{Adapter}^{\text{H}}$)	11.09M	$67.3_{\pm.6}$	$8.50_{\pm.07}$	$46.0_{\pm.2}$	$70.7_{\pm.2}$	$2.44_{\pm.01}$
GPT-2 M (FT^{Top2})*	25.19M	68.1	8.59	46.0	70.8	2.41
GPT-2 M (PreLayer)*	0.35M	69.7	8.81	46.1	71.4	2.49
GPT-2 M (LoRA)	0.35M	$\mathbf{70.4}_{\pm.1}$	$\mathbf{8.85}_{\pm.02}$	$\mathbf{46.8}_{\pm.2}$	$\mathbf{71.8}_{\pm.1}$	$\mathbf{2.53}_{\pm.02}$
GPT-2 L (FT)*	774.03M	68.5	8.78	46.0	69.9	2.45
GPT-2 L ($\text{Adapter}^{\text{L}}$)	0.88M	$69.1_{\pm.1}$	$8.68_{\pm.03}$	$46.3_{\pm.0}$	$71.4_{\pm.2}$	$\mathbf{2.49}_{\pm.0}$
GPT-2 L ($\text{Adapter}^{\text{L}}$)	23.00M	$68.9_{\pm.3}$	$8.70_{\pm.04}$	$46.1_{\pm.1}$	$71.3_{\pm.2}$	$2.45_{\pm.02}$
GPT-2 L (PreLayer)*	0.77M	70.3	8.85	46.2	71.7	2.47
GPT-2 L (LoRA)	0.77M	$\mathbf{70.4}_{\pm.1}$	$\mathbf{8.89}_{\pm.02}$	$\mathbf{46.8}_{\pm.2}$	$\mathbf{72.0}_{\pm.2}$	$2.47_{\pm.02}$

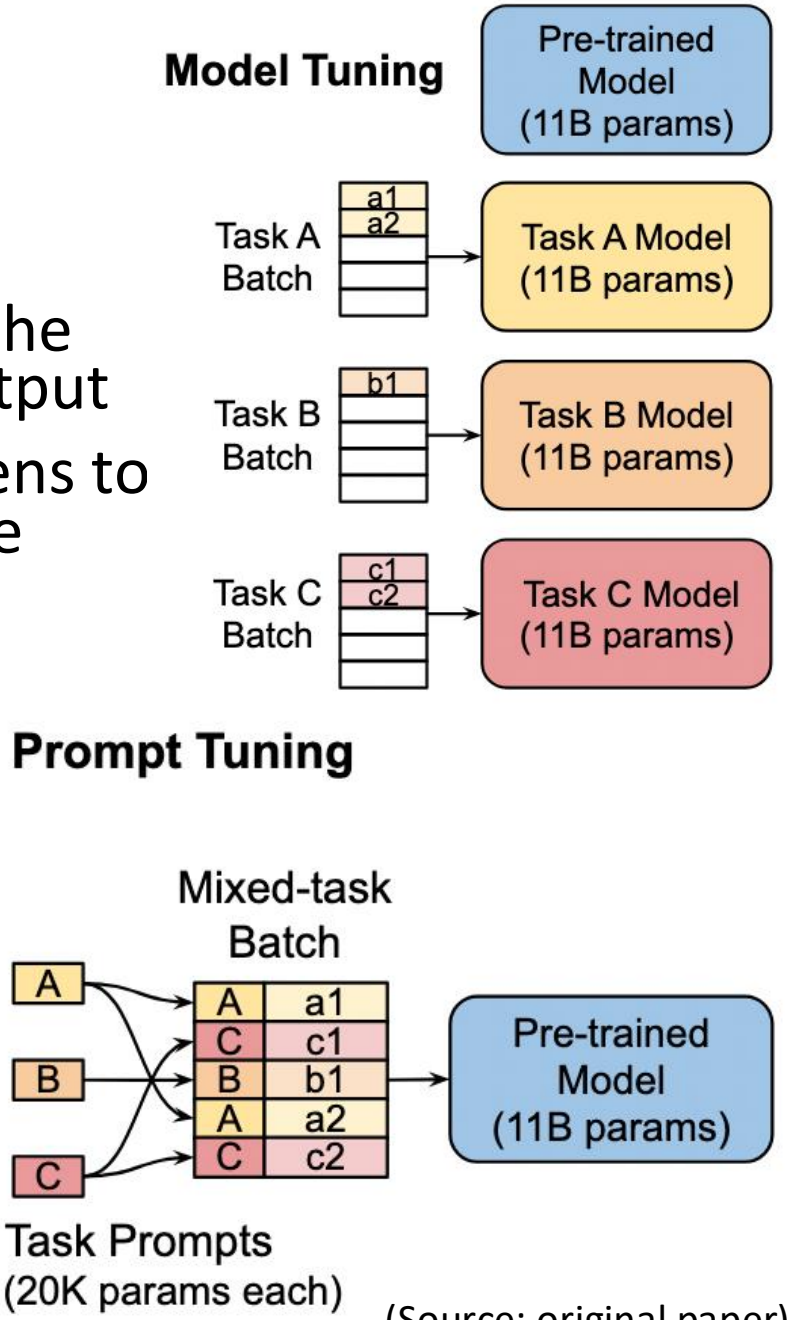
- Authors show, empirically, that the updates are indeed near low-rank
- By using values as low as r = 4, we get good approximations of ΔW
- For BERT, 1 transformer layer = 7M parameters
  - If we apply LoRA on W_q, W_k, W_v, W_o (768x768) and the FF layers (768 → 3072 → 768), with r = 4, we get:
    - 768*4*2*4 + (768*4+3072*4)*2 = 50k parameters (~140x fewer parameters)
- We can choose to fine-tune only parts of the transformer, e.g. only queries and values, producing fewers parameters!

Considerations on LoRA

- The benefits increase as we increase the model size
  - with $d_{model} = 12{,}288$ (GPT-3), we get up to 10,000x fewer parameters!
  - Authors claim the hardware requirements go down by a factor of 3
    - (forward pass still needed "in full")
- We can fine-tune the same model on *multiple tasks*
  - Each task t will have its A_t, B_t adapters
- We can store a *single instance* of the pretrained model, and add the appropriate A_t, B_t for each task needed (space efficient!)
- At inference time, we can *precompute* W' from W, A, B
  - In this way, there's no overhead introduced by the additional multiplications/additions required to compute $W'x = Wx + AB^T x$

Prompt tuning

- *Prompting*: approach of adding *extra information* for the model *to condition* on, during its generation of the output
- Normally, prompting is done by prepending some tokens to the input, attempting to maximize the likelihood of the correct output
- In GPT-3, for instance, these prompt tokens are part of the model's vocabulary
  - In *prompt design*, the "right" words exist, we just need to choose them
- In *prompt tuning*, instead, we add a fixed set of special tokens to the prompt
  - Then, we *allow fine-tuning only those special tokens*
  - (Instead of choosing the right "words", we create them)



Lester, Brian, Rami Al-Rfou, and Noah Constant. "The power of scale for parameter-efficient prompt tuning." arXiv preprint arXiv:2104.08691 (2021). https://arxiv.org/pdf/2104.08691

(Source: original paper)

Other optimization techniques

- The previous techniques fall under the umbrella of PEFT
  - Focus on fine-tuning
- Other techniques are more generally focused on reducing the footprint of the model
  - In terms of memory used, and/or computational cost
  - Either at training, or inference time
- This is done by either making the *same model smaller*
  - (Quantization, reduction in floating point precision)
- Or, by building *smaller versions of models*
  - (Distillation)

Quantization

- *Quantization* is the process of mapping continuous (floating-point) values to discrete ones
- In other words, reduce the precision of the model's weights and/or activations by limiting the number of representable values
- Pros
  - Reduces storage/memory requirements for models
  - Improves computational efficiency (operations on fewer bits, possibly integer)
- Cons
  - Loss in precision typically has some effect on the model's performance

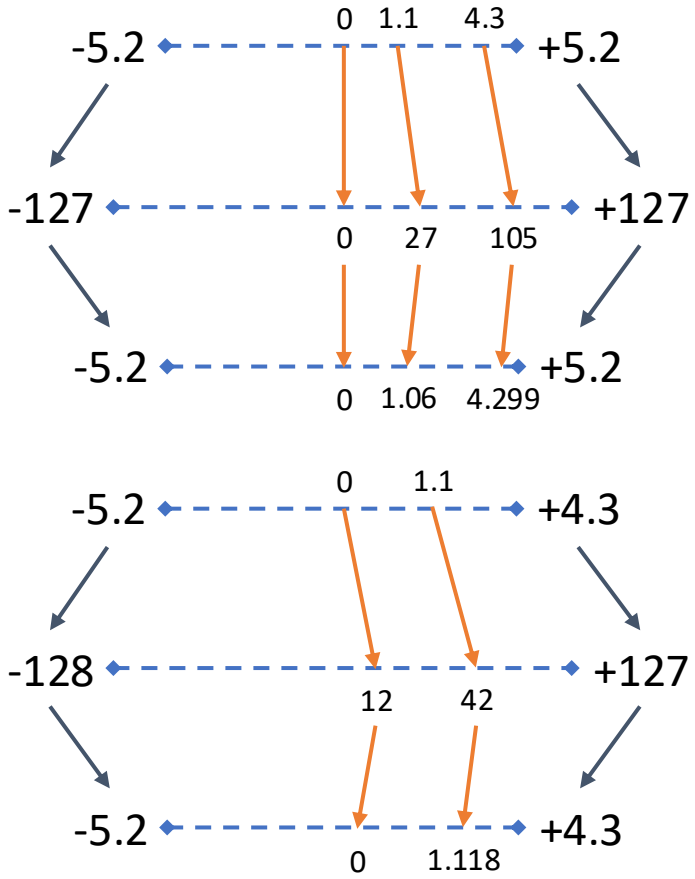
Numerical representations

- *float32*
  - 1 bit for sign, 8 bits for exponent, 23 bits for fraction
  - Range (positive): ~ $1.18 \cdot 10^{-38}$ (normalized) to $3.4 \cdot 10^{38}$
- *float16*
  - 1 bit for sign, 5 bits for exponent, 10 bits for fraction
  - Range (positive): ~ $6.1 \cdot 10^{-5}$ (normalized) to $6.5 \cdot 10^{4}$
- *bfloat16*
  - 1 bit for sign, 8 bits for exponent, 7 bits for fraction
  - (same range of values as float32, with lower precision)
- *int8*
  - -128 to 127

Quantization process

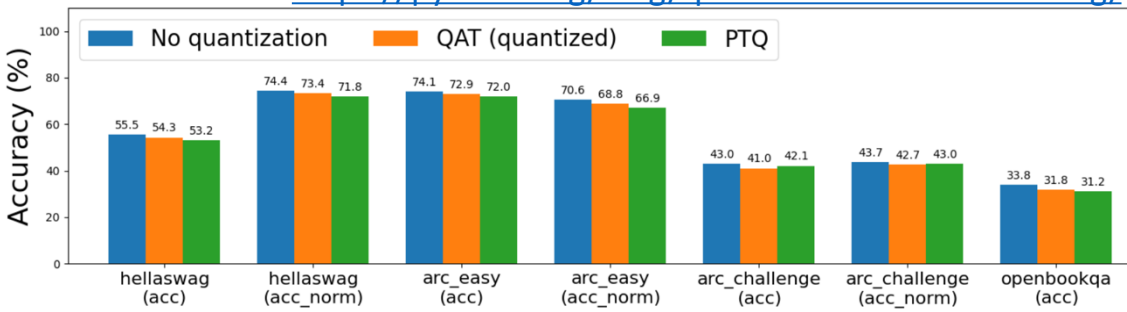
-5.2	4.3	1.1	0

- Mapping scheme
  - Specifies how values from one domain (e.g. values of a tensor, in float32) should map to the target domain (e.g. int8)
  - Various approaches (e.g., *absmax*, *zero-point*)
  - All based on *scale* (size of step) and *zero-point* (z, what is "0"?)
  - $q_{\min} = -128, q_{max} = 127,\ b = \max_i x_i, a = \min_i x_i$
  - $q = clip(round(x/s) + z, q_{\min}, q_{\max})$
  - $\hat{x} = (q - z)s$
- *Absmax quantization*
  - Scale values symmetrically
  - If a is the abs-largest value in the tensor, values are mapped so as -a = -127, +a = +127
  - The original value "0" is preserved to "0" in the quantized version
  - $scale = \max_i |x_i| / 127,\ z = 0$
- *Zero-point quantization*
  - Scale values asymmetrically
  - If min = a, max = b, values are mapped so that a = -128, b = +127
  - More efficient use of the range of possible values for asymmetric distributions
- $scale = \frac{b-a}{q_{max} - q_{min}}, z = clip(round\left(q_{min} - \frac{a}{scale}\right), q_{min}, q_{max})$



https://pytorch.org/blog/quantization-aware-training/

Types of quantization



- *Post-Training Quantization* (*PTQ*)
  - A model is trained "normally", its weights and/or activations are quantized afterwards
  - Easy to apply to a model without retraining, but can produce sub-optimal results

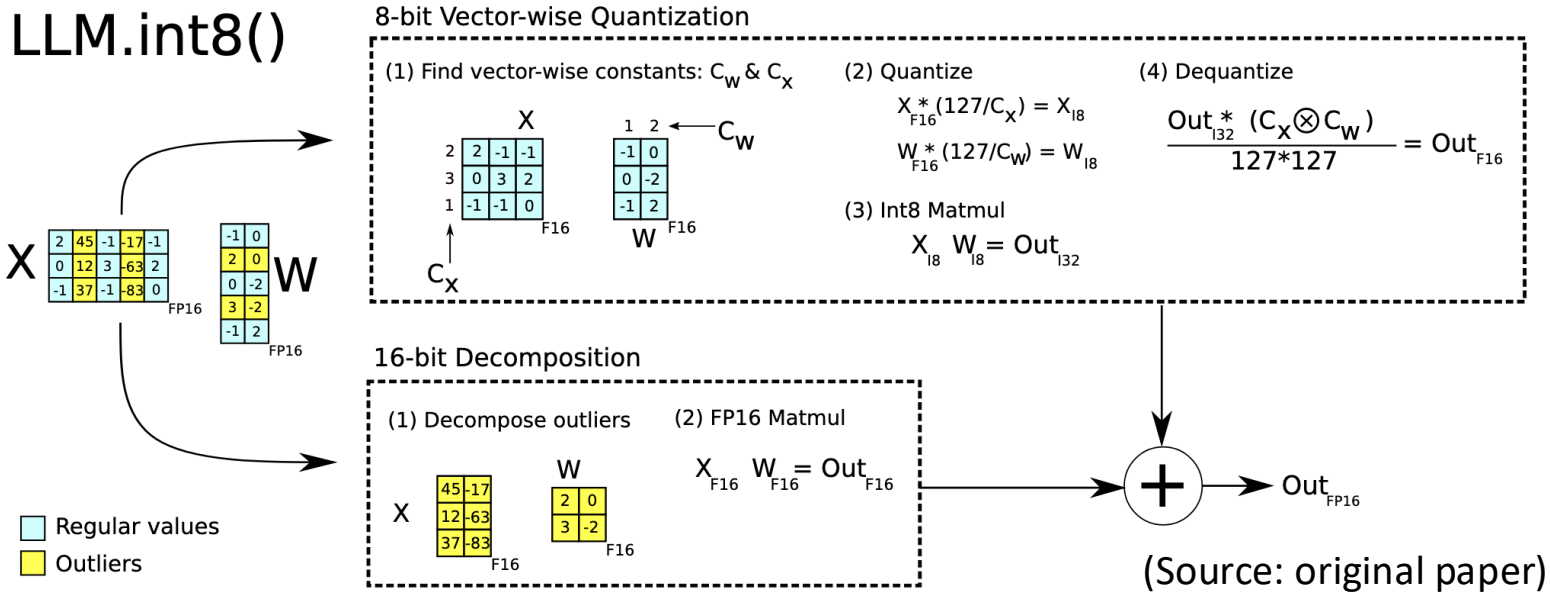
- *Quantization-Aware Training* (*QAT*)
  - Incorporates quantization during training
  - Forward pass
    - Using (fake) quantized version of weights/activations
    - All values are kept to full precision (but rounded)
  - Backward pass
    - Gradients computed in full precision
    - Quantization parameters (e.g., scale, zero-point) are learned
    - Some tricks to bypass non-differentiable functions (e.g., rounding)
  - Better performance, but requires intervening on the training process

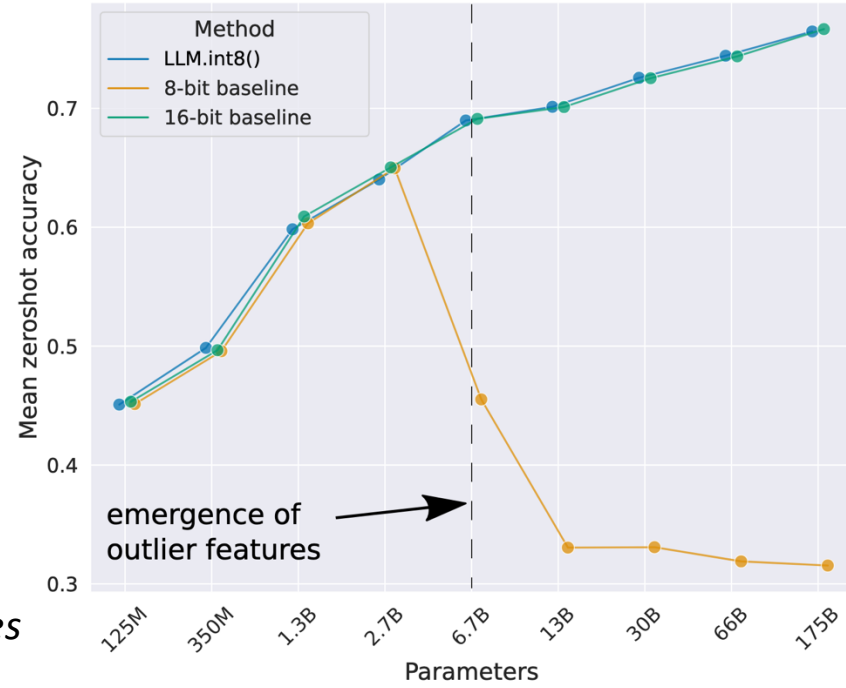
Static vs dynamic quantization

- To quantize weights/activations, we need to know *scale* and *zero-point*
- For *weights*, we can compute the information beforehand
- For *activations*, there are two approaches: static & dynamic quantization
- *Static quantization*
  - Pre-compute scale, zero-point, then used them in a fixed way
  - A validation set is used to measure distributions (calibration phase)
  - Faster, more consistent
- *Dynamic quantization*
  - Compute scale and zero-point for each activation separately
  - Makes better use of the range of possible values
  - No calibration step required
  - More computationally expensive (at inference, we compute values on the activations)

LLM.int8()

- *Vector-wise quantization*
  - Compute scaling constants for each row/vector of matrices
  - Instead of 1 per matrix
  - This produces better-quantized dot products
- *Mixed-precision decomposition*
  - Authors observe that there are some outlier weights/activations (features) in larger models, with large magnitudes
    - Approximately 0.1% of all features
    - These outliers are useful for LLMs
  - To better handle these outlier features, we can decompose each matrix into non-outliers (8-bit precision) and outliers (16-bit precision)





Dettmers, Tim, et al. "Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale." *Advances in Neural Information Processing Systems* 35 (2022): 30318-30332.
https://arxiv.org/pdf/2208.07339

Reduced floating point precision

- A less drastic measure, which we can easily adopt, is converting a model to *half precision* (float16)
- We typically use float32 (single precision) models
- However, we don't always need that much precision
- By simply converting a model to half precision, we can save half the space of the model!
- We preserve values in a continuous range of values, so this is not considered quantization
- Rather easy in PyTorch!
  - `model = model.half()`

Model distillation

- A smaller model can be *distilled* from a larger model, so as to obtain behaviors similar to those of the original model

- Distillation is generally done with a teacher/student paradigm:
  - *Teacher*: the original (larger) model, we want to reduce in size
  - *Student*: a smaller model we want to use to mimic the teacher

- The student is trained to *predict* the teacher's *probability distribution* (across all words) instead of the correct word

- By learning from the teacher, the student receives information unavailable in the ground truth
  - E.g., the correct word could be *cat*
    - (*cat*: 1, *everything else*: 0)
  - But the teacher may provide a more semantically rich target
    - (*cat*: 0.6, *kitten*: 0.2, *kitty*: 0.1, *cats*: 0.1)

Hinton, Geoffrey. "Distilling the Knowledge in a Neural Network." *arXiv preprint arXiv:1503.02531* (2015). https://arxiv.org/pdf/1503.02531

Model distillation in LLMs

- It's been shown that distilled models can achieve comparable performance to the original ones, despite smaller architectures
- Various LMs have distilled versions:
  - *DistilBERT* (from BERT)
    - https://arxiv.org/pdf/1910.01108
    - 40% smaller, retains 97% of performance, 60% faster
  - *TinyBERT* (from BERT)
    - https://arxiv.org/pdf/1909.10351
    - 7.5x smaller, retains 97% of performance, 9.4x faster inference
  - *MiniLM*
    - https://arxiv.org/pdf/2002.10957
    - 50% smaller, retains 99% of performance
  - *DistilGPT-2*