PEFT / Quantization

Lecture 10

EN.705.743: ChatGPT from Scratch

Lecture Outline

- Fine-Tuning is Expensive
- Survey of Parameter-Efficient Fine-Tuning (PEFT)
  - Common Idea: Add a few parameters
  - Learning Prompts
  - Adapters
  - LoRA
- Mixed-Precision
  - Data Types Review
  - Training with Mixed Precision
- Quantization
  - Post-Training Quantization
  - QLoRA
  - Extreme Quantization

Fine-Tuning ≈ Training

Fine-Tuning Compute

Although fine-tuning has much smaller data requirements than pre-training, we are still updating the entire model.

This is very memory intensive, and even “small” models may be prohibitive.

Consider a 7B parameter model (a small state-of-the-art model like Mistral or LLaMA):

- 7B x 4 bytes per parameter = 28 GB to store model
- + 1 gradient value per parameter (+28GB)
- + 2 rolling values in the Adam optimizer (+56GB)
- + the memory needed for forward/backward pass (this can be significant unless you use a tiny batch size)

So we need a system with >100 GB of VRAM ($10k with consumer hardware (slow), or $50k+ for professional-grade).

Fine-Tuning Compute

This level of investment is typically out of reach for small labs, academic labs, etc. Completely out of reach for a hobbyist looking to have a professional-level setup.

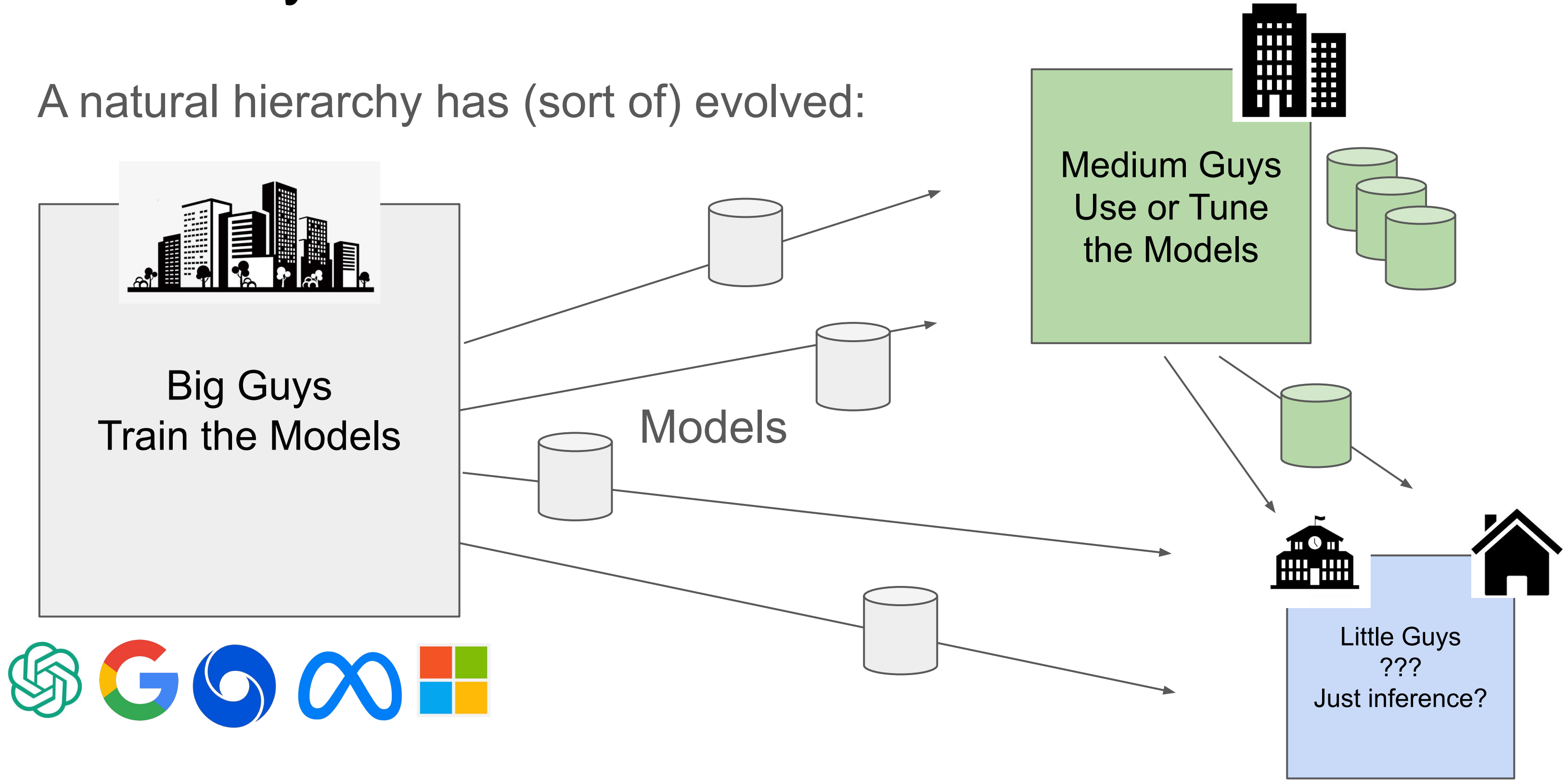
(You can also do this on AWS but if you are doing this several times over this adds up too).

This is a problem! If no one can fine-tune the models:

- Research progress slows down
- LLMs become less desirable as a “product”
- Open-source community kind of can’t exist (other than inference)

Hierarchy

A natural hierarchy has (sort of) evolved:



Solutions

Probably out of necessity, several solutions have arisen for this problem.

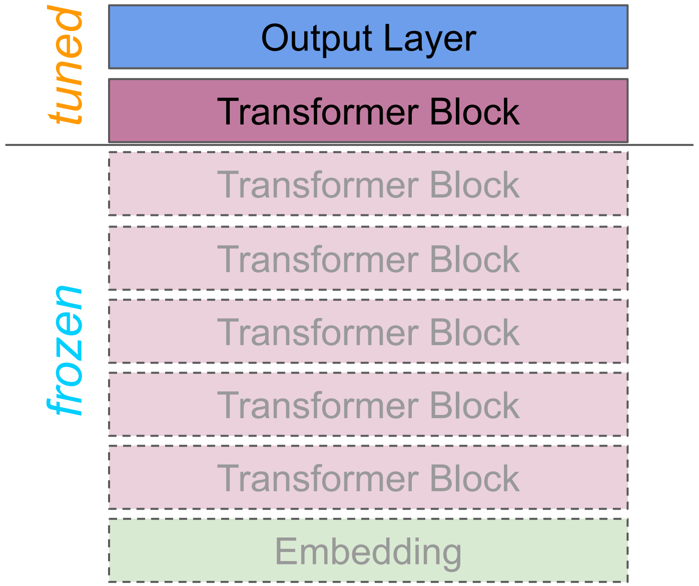
This is the subject of today’s lecture. How to tune a model without $$$.

Quick Note

One (bad) solution is to freeze the LLM except for a few layers at the end. This is a fine-tuning technique that has been around for a while.

In this view, the bulk of the model is treated as a “feature extractor”, and we learn a small model that operates on these features.

This is not done much with LLMs, because it doesn’t combine well with how transformers build up an understanding layer by layer.

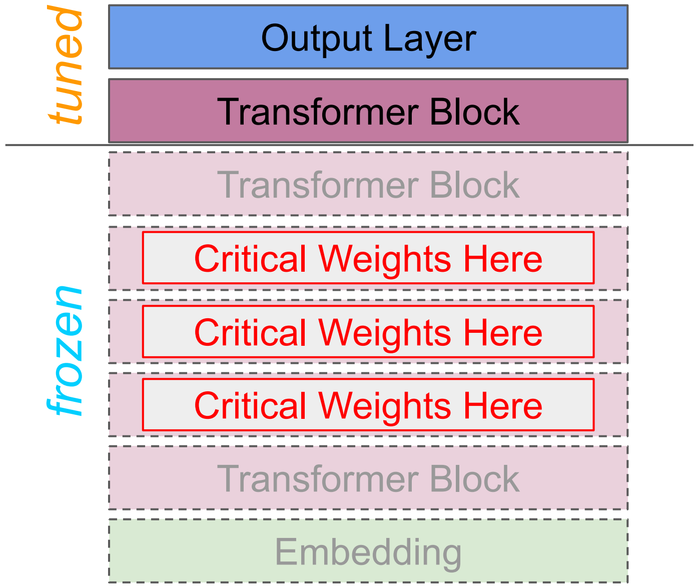


Quick Note

Why doesn’t this work for LLMs?

Transformers build up knowledge throughout the network, with low layers learning word-level meanings and higher layers broad concepts.

Unless we have a task that is nearly identical in domain (general language) and task (self-supervised text continuation), it is best if we edit the model across all layers.



Parameter Efficient Fine-Tuning

Adding Non-Frozen Parameters

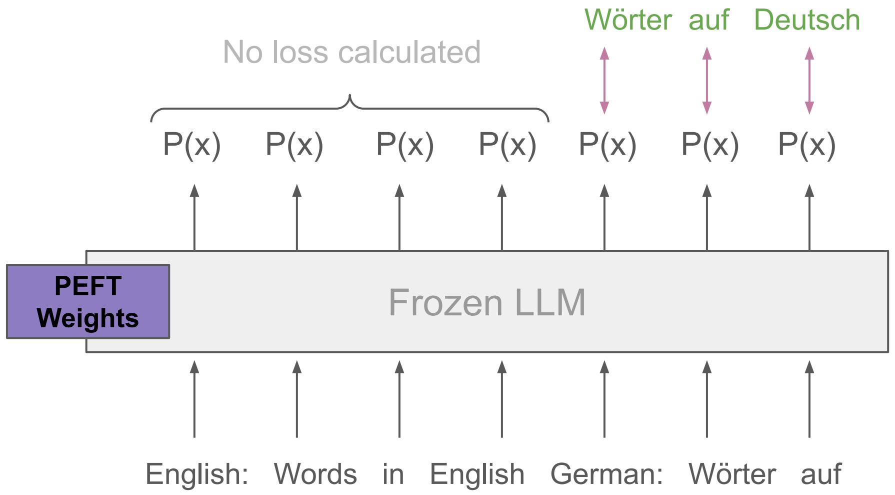
A group of techniques have arisen which (perhaps paradoxically) add parameters to the LLM to make fine-tuning more efficient. We then tune only these parameters and keep the original weights frozen.

These techniques are called PEFT, **P**arameter-**E**fficient **F**ine-**T**uning.

The number of added parameters is often very small, less than 1% of the original model size.

Training Diagram

PEFT is used with SFT. We add a small number of weights to our model and keep the original weights frozen. Only the new weights are updated.



Why do we do this?

Calculating and storing the gradient is really expensive (~3x the model storage size). If we are only updating a few weights, this becomes negligible.

We still need to pass through the entire model, but only retain optimization information about a small fraction of it.

Where do the new parameters go?

There are many options here! Most PEFT methods (there are many of them) are defined by where they place the new parameters. There are three key possibilities:

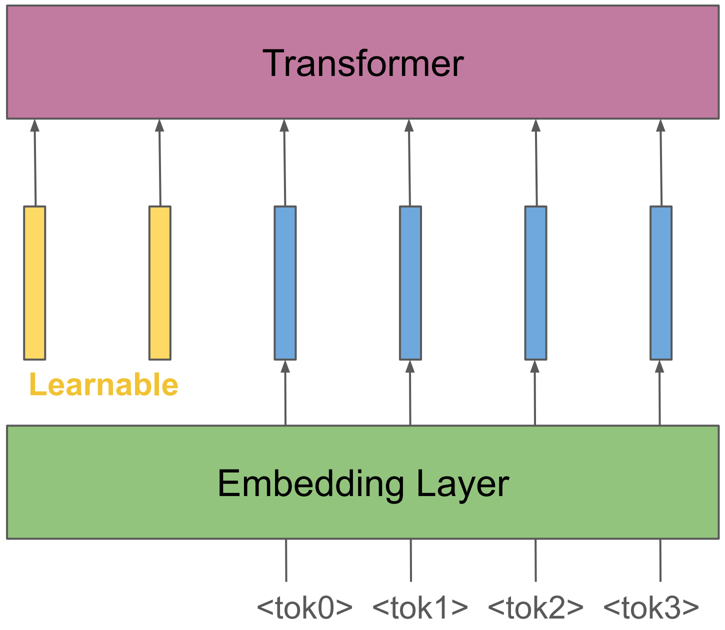
1) In the sequence itself, as learnable tokens
2) Augmenting the existing weight matrices
3) As entirely new weight matrices or layers

(1) Adding Parameters in the Sequence

This family of PEFT methods all use a common idea- extra vectors are added to the beginning of a sequence (after embedding), and we can backpropagate into these.

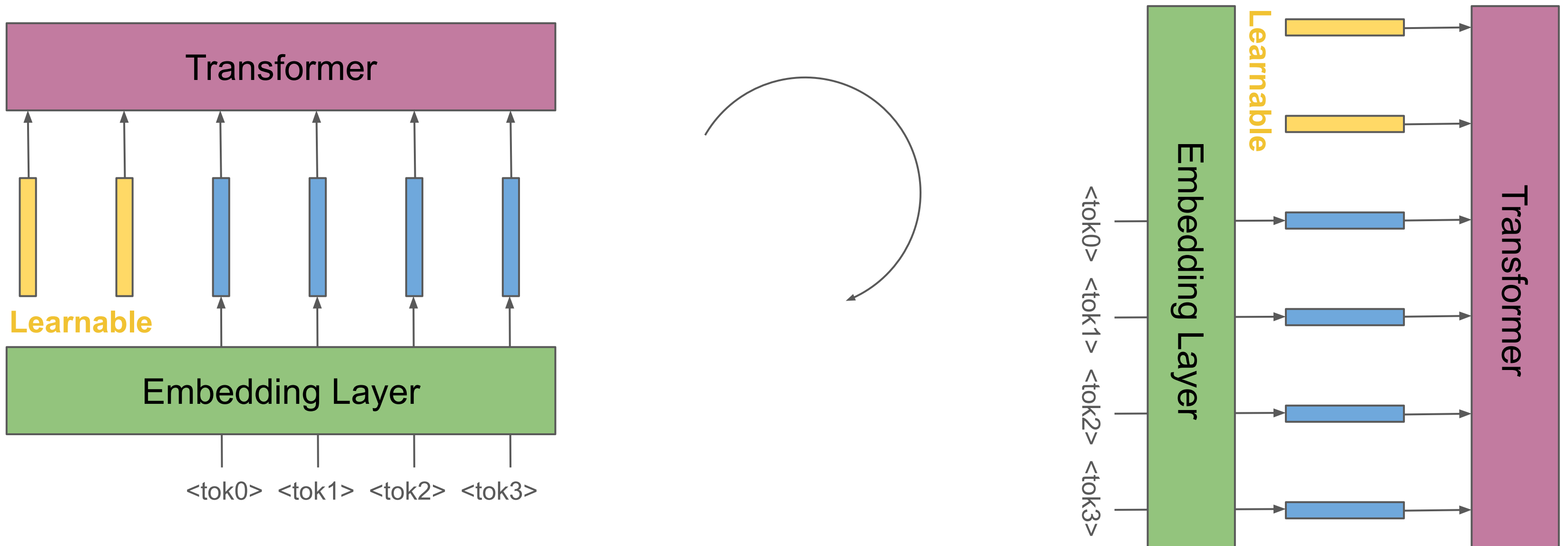
Each of these represents a tunable point in the embedding space. Since these are at the beginning, all “real” tokens can attend to them.

Intuitively, you could think of these as a “soft prompt” that is learned, and describes to the model how to behave.



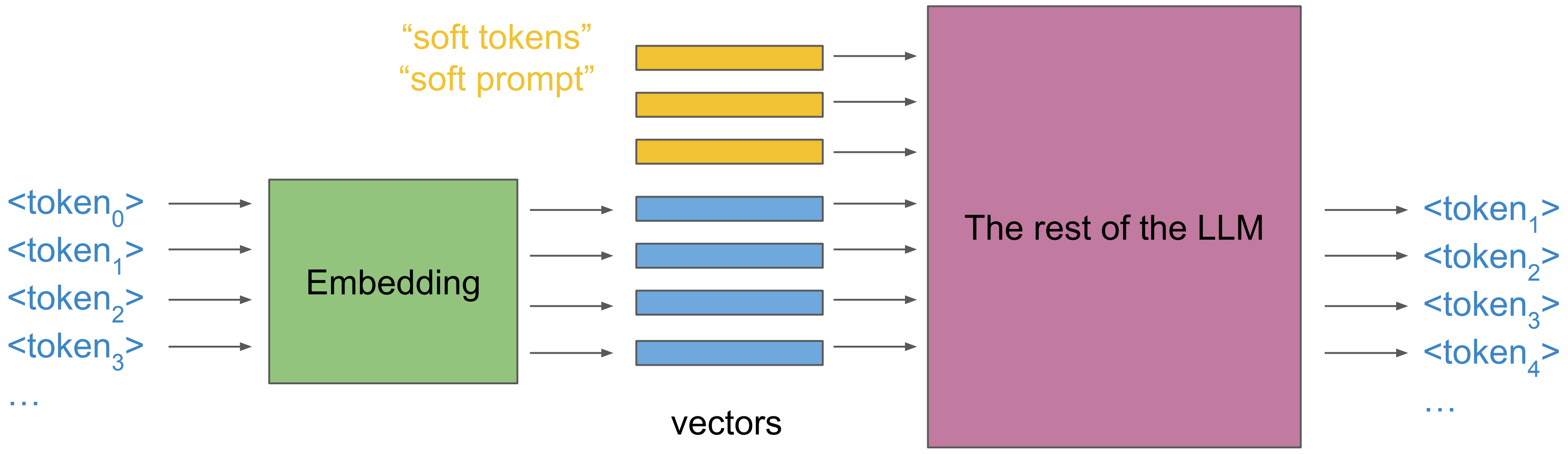
Rotating the diagrams.

For this section I am turning the diagrams so there is more room:



Prompt Tuning (Lester et al, Sep 2021)

Prompt tuning adds a tunable “prompt” (prepended inputs), but they only exist in embedding space. Everything is frozen except these vectors.



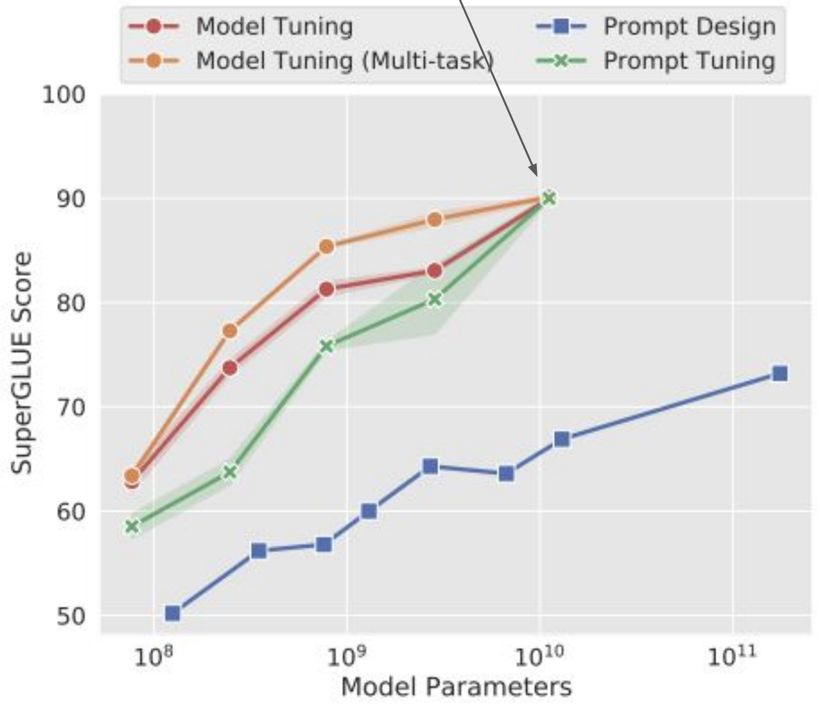
Prompt Tuning Results

They specifically test prompt tuning on T5 models on individual SuperGLUE tasks (SuperGLUE is a collection of challenging NLP tasks).

The key results are:

- **Prompt Tuning is competitive to SFT for large modes (10B and up)**
- Crazy small parameter usage (can be 0.01% of original model size)
- Initialize the soft prompt with embeddings of real tokens, not just random.

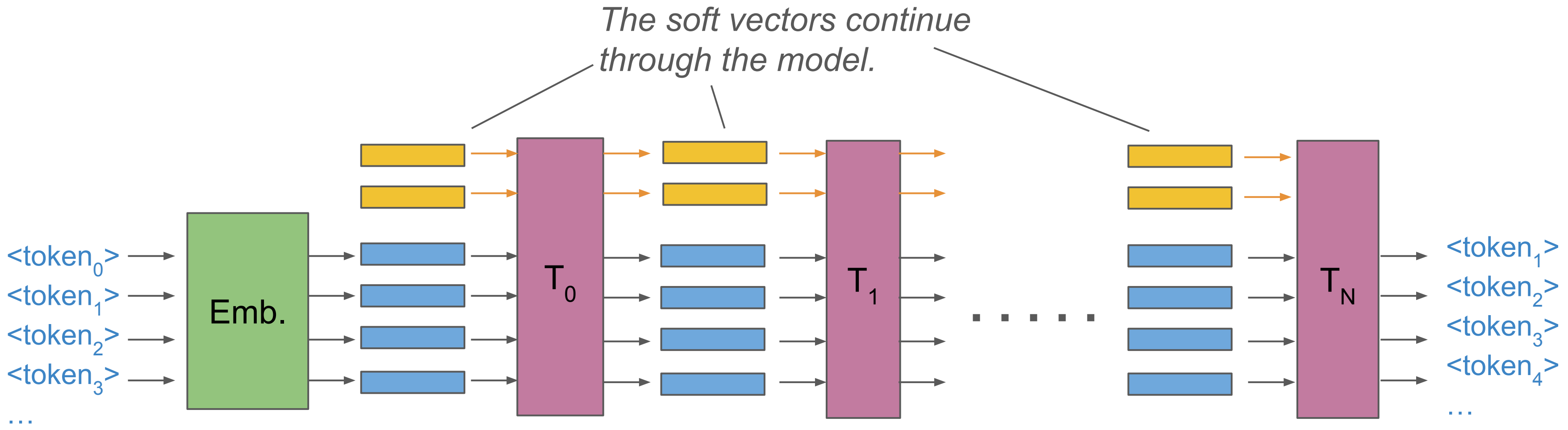
All their experiments converge here.



Prefix Tuning (Li & Liang, Jan 2021)

This is more complex than prompt tuning (and actually came first). We can start with prompt tuning:

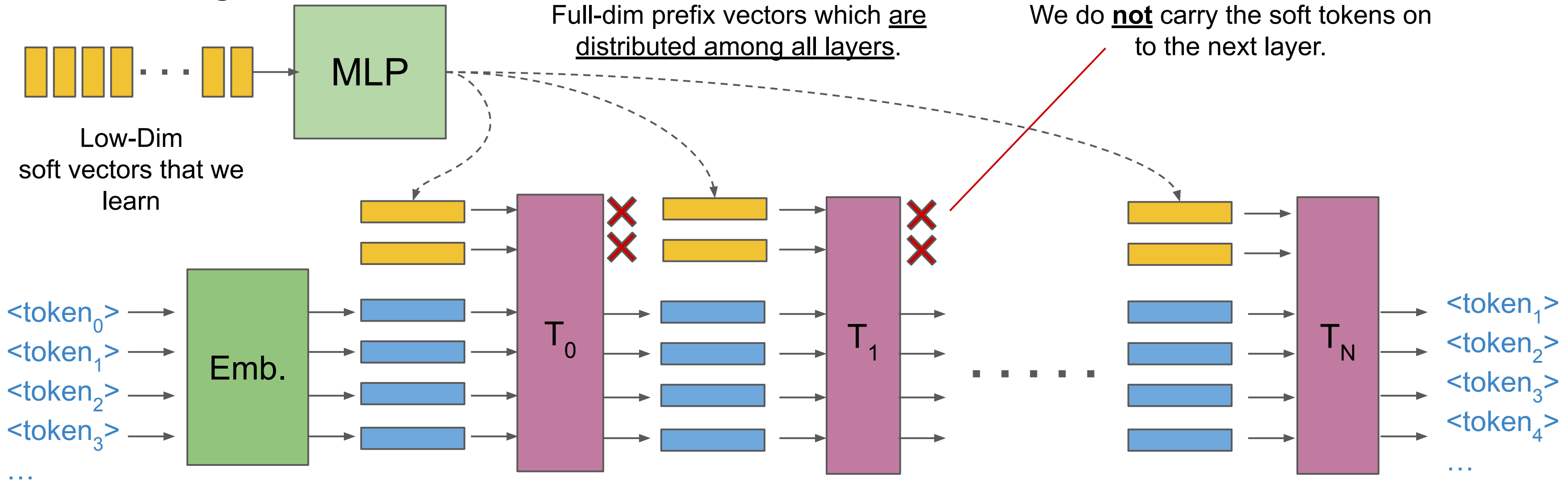
Prompt Tuning, Expanded Network View:



Prefix Tuning

This is the same as prompt tuning except (a) soft tokens are tuned for every layer, and (b) an MLP helps stabilize learning the soft tokens:

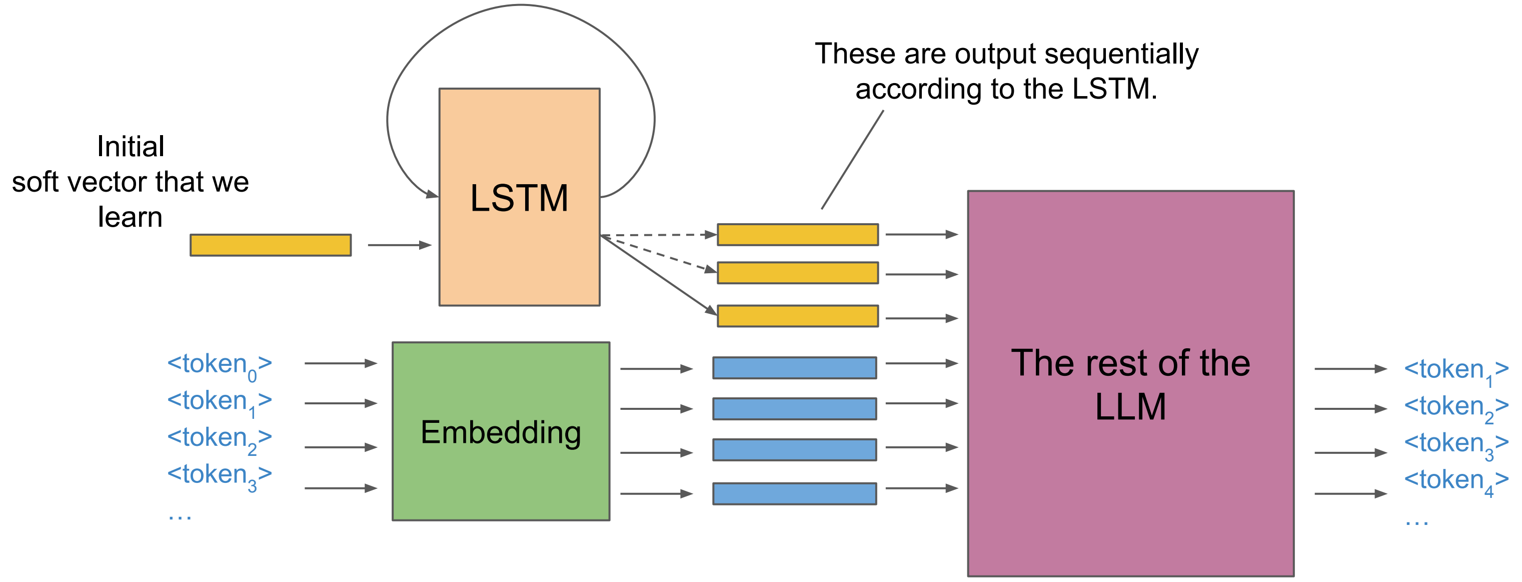
Prefix Tuning:



MLP
Low-Dim soft vectors that we learn
Full-dim prefix vectors which are distributed among all layers.
We do **not** carry the soft tokens on to the next layer.
Emb.
T_0
T_1
T_N
<token$_0$>
<token$_1$>
<token$_2$>
<token$_3$>
…
<token$_1$>
<token$_2$>
<token$_3$>
<token$_4$>
…

P-Tuning (Liu et al, Mar 2021)

Another option ("P-Tuning") outputs the soft vectors sequentially, since they are technically in sequence after all. We don't necessarily need to use the LSTM at test-time: it just enforces a sequential dependence on our soft vectors.



Summary

Works like *Prompt Tuning* add soft vectors that we can tune as prefixes to our inputs (a “soft prompt”).

We can make this more complex if we want to: Works like *Prefix-Tuning* and *P-tuning* try ideas like adding a separate network to generate the soft vectors. Prefix-Tuning also experiments with injecting soft vectors at each layer.

These other methods seem to also work, but they are much more complex than Prompt-Tuning. P-Tuning only studied small models. Prefix tuning only studied a table processing problem.

(2) Augment the weights that are already there

By far the most popular method here is LoRA.

LoRA

LoRA is kind of the odd one out here- it is a general technique for fine-tuning that can be applied to any deep neural net.

What does updating a weight matrix look like?

$$W' = W + \nabla W$$

LoRA

LoRA is kind of the odd one out here- it is a general technique for fine-tuning that can be applied to any deep neural net.

What does updating a weight matrix look like?

$$W' = W + \nabla W$$
$$W'' = W' + \nabla W'$$

LoRA

LoRA is kind of the odd one out here- it is a general technique for fine-tuning that can be applied to any deep neural net.

What does updating a weight matrix look like?

$$W' = W + \nabla W$$

$$W'' = W' + \nabla W'$$

$$= W + \nabla W + \nabla W' \ldots$$

LoRA

LoRA is kind of the odd one out here- it is a general technique for fine-tuning that can be applied to any deep neural net.

What does updating a weight matrix look like?

$$W' = W + \nabla W$$

$$W'' = W' + \nabla W'$$

$$= W + \nabla W + \nabla W' \ldots$$

$$= W_{frozen} + [\text{overall}$$

LoRA

You can represent updating a model by learning a new weight matrix that will be added to the frozen model:

$$W_{\text{tuned}} = W_{\text{frozen}} + \boxed{W_{\text{delta}}}$$

Only update this.

LoRA

Problem: W is huge (could be 10s of millions of parameters).

$$W_{tuned}^{MxN} = W_{frozen}^{MxN} + \boxed{W_{delta}^{MxN}}$$

Only update this.

If we stopped here, we would just have doubled our parameter count :(

LoRA

Problem: W is huge (could be 10s of millions of parameters).

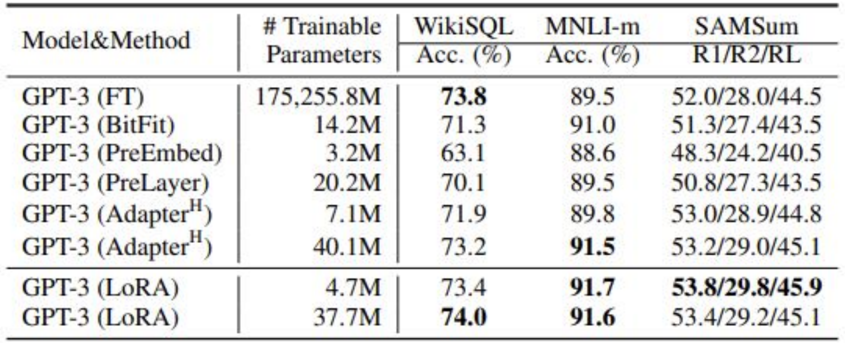
Solution: Factorize W_{delta} into two smaller matrices.

$$W^{MxN}_{tuned} = W^{MxN}_{frozen} + \boxed{A^{Mxr} B^{rxN}}$$

Here r << M or N. A typical value of r would be 8 or 16.

LoRA Paper (Hu et al, Oct 2021)

LoRA paper shows that LoRA is competitive with full fine-tuning:



Model&Method	# Trainable Parameters	WikiSQL Acc. (%)	MNLI-m Acc. (%)	SAMSum R1/R2/RL
GPT-3 (FT)	175,255.8M	**73.8**	89.5	52.0/28.0/44.5
GPT-3 (BitFit)	14.2M	71.3	91.0	51.3/27.4/43.5
GPT-3 (PreEmbed)	3.2M	63.1	88.6	48.3/24.2/40.5
GPT-3 (PreLayer)	20.2M	70.1	89.5	50.8/27.3/43.5
GPT-3 (Adapter^H)	7.1M	71.9	89.8	53.0/28.9/44.8
GPT-3 (Adapter^H)	40.1M	73.2	**91.5**	53.2/29.0/45.1
GPT-3 (LoRA)	4.7M	73.4	**91.7**	**53.8/29.8/45.9**
GPT-3 (LoRA)	37.7M	**74.0**	**91.6**	53.4/29.2/45.1

r = 8

Why? Their explanation is that for a specific downstream task, the set of features needed is much smaller than the set of features the model computes. So the low-rank W_{delta} basically just amplifies the features that are needed for the specific downstream task, and it can do this with low rank.

(3) Add new layers

The last type of alteration we can make is to insert new layers into the model. Since we do not want to just tune the “top” of the model, we add these throughout.

These are called “Adapters”.

Adapters

There is a nice diagram in a review by Hu et al:

https://arxiv.org/pdf/2304.01933

We already talked about these.

Adds small bottlenecked MLPs inside the transformer block (inline).

Adds small bottlenecked MLPs inside the transformer block (parallel).

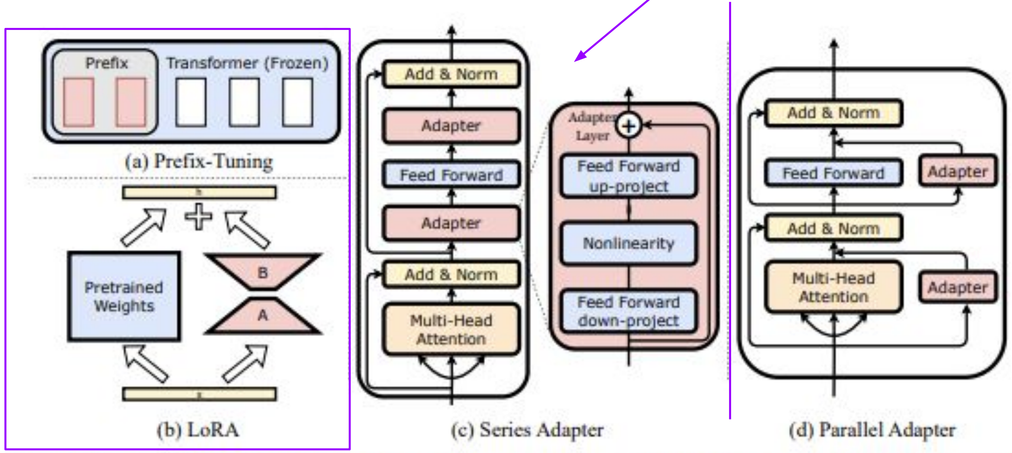


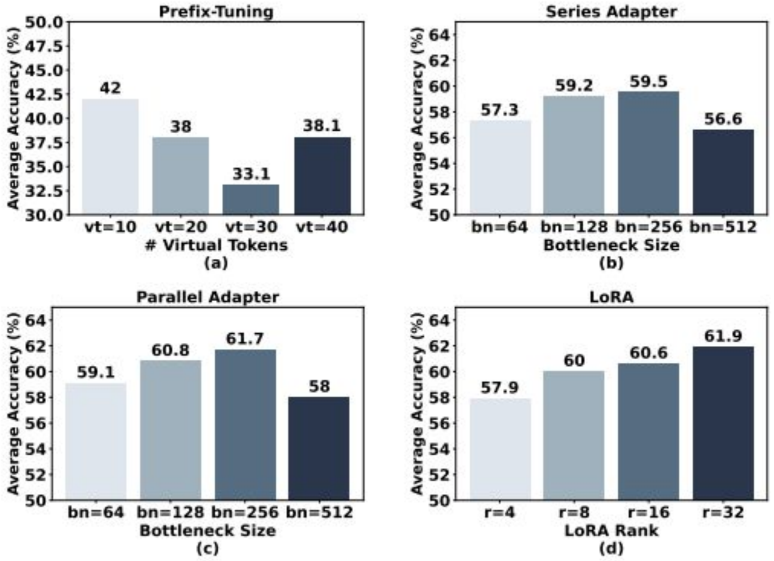
Figure 1: A detailed illustration of the model architectures of three different adapters: (a) Prefix-Tuning, (b) LoRA, (c) Series Adapter, and (d) Parallel Adapter.

Hu et al results / Summary

In their experiments, Hu et al found that series and parallel adapters work about as well as LoRA. They find that prefix tuning does not work well, although I do not believe they compared to prompt tuning.

Generally these papers have a good deal of disagreement about which method is the best.

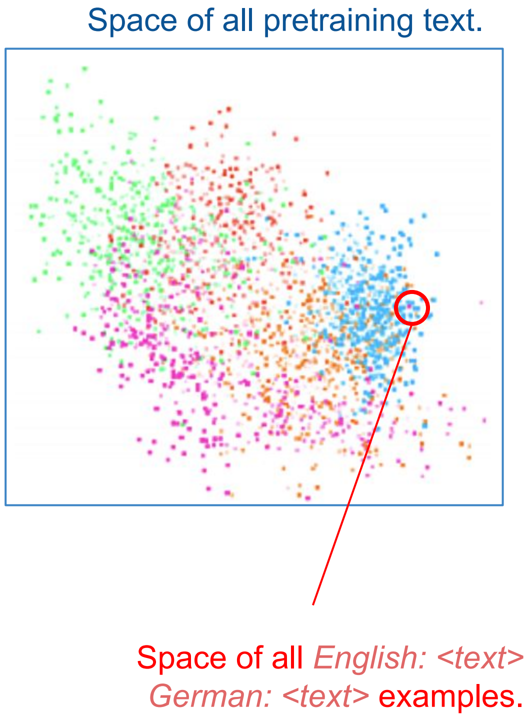
I think all of them are options for a given problem. They all offer similar benefits: Extremely low tunable parameter counts, very good performance.



Why does PEFT work?

It seems that PEFT works because large LLMs are a great starting point, and with a few small changes in critical spots we can change them.

Additionally, the fine-tuning tasks are typically much smaller is scope than the original LLM training scheme. There is an idea that PEFT methods simply **“bring relevant knowledge to the surface”**, meaning they reinforce existing knowledge or behavior that is already present in the LLM.


Space of all pretraining text.
Space of all *English: <text> German: <text>* examples.

Mixed-Precision Training

Mixed Precision Training

Instead (or in addition to) PEFT methods, a very approachable technique is to change the data type of the model.

Generally, this is only applied to the forward pass.

The update step is usually done in full precision (float32) because it requires high precision:

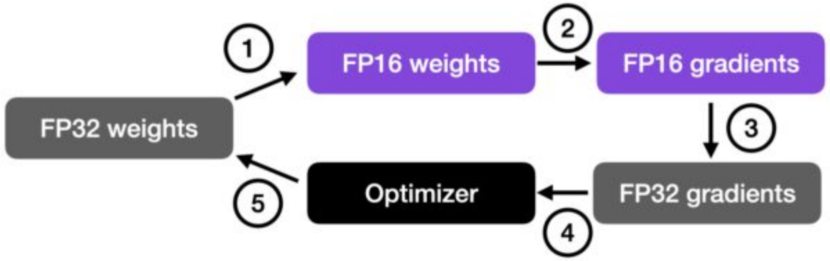
- Want to capture small changes to weights
- Weights may be small numbers to begin with ($-1 < x < 1$)
- Learning rate can be very small (1e-4, 1e-5, etc), so we are multiplying our updates by a tiny number
- Methods like Adam will also track rolling statistics of the gradient, and we want to be able to capture changes with enough precision that they matter

Mixed Precision Training

We flip back and forth between data types as needed:

- Forward pass: low precision
- Backward pass: low precision
- Convert gradient to high precision (float 32)
- Apply gradient in high precision
- Convert back to low precision

This is called “mixed-precision” training and can be very effective at reducing memory requirements.

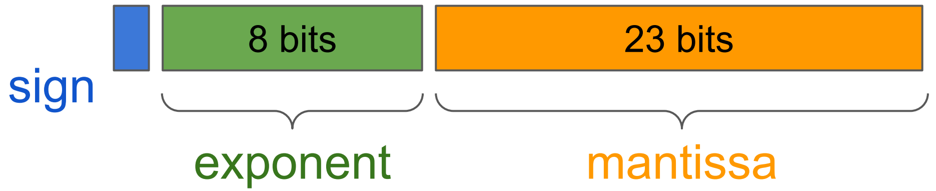


https://lightning.ai/pages/community/tutorial/accelerating-large-language-models-with-mixed-precision-techniques/

Data Types

If we are using something other than float32, what are we using?

The Float32 data type uses 32 bits to store a float:

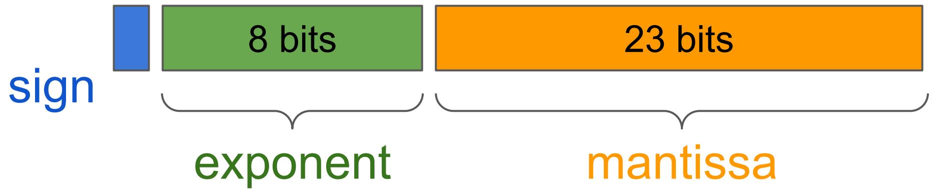


Number = [sign] * 2^[exponent] * [1+mantissa]

Data Types

If we are using something other than float32, what are we using?

The Float32 data type uses 32 bits to store a float:



Controls the magnitude of the number (base 2)

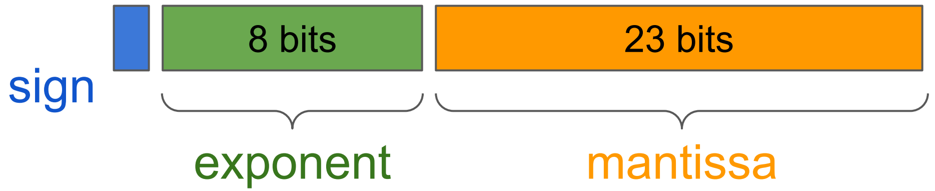
We can represent exponents of **-127 to 127**

Controls the digits of the number

Data Types

If we are using something other than float32, what are we using?

The Float32 data type uses 32 bits to store a float:

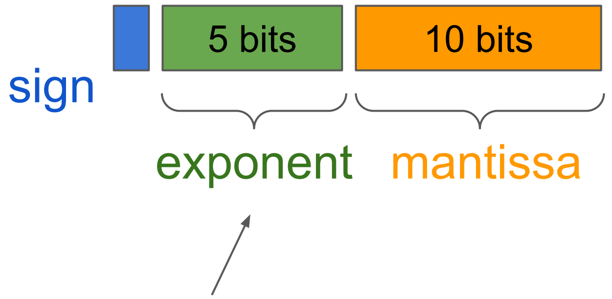


However, we get reduced precision towards the “fringes”.

If the exponent is bumping us much further than digits we can represent in the mantissa, we incur error.

Data Types: float16

One alternative is float16 (“half precision”). This works the same way except we use less bits for each component:



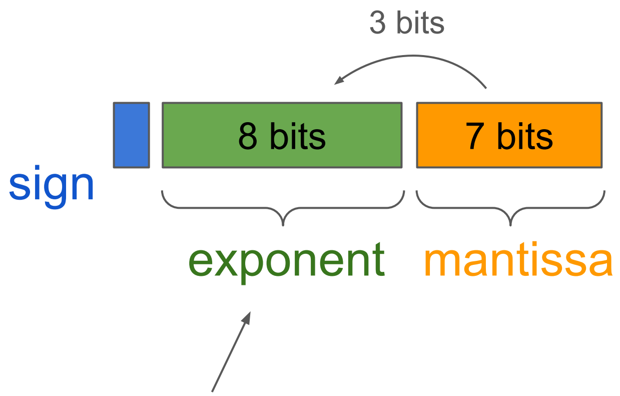
Number = [sign] * $2^{[\text{exponent}]}$ * [1+mantissa]

We can represent exponents of **-32 to 32.**

This may cause problems with very high or **very low numbers.**

Data Types: bfloat16

To aid in representing very low numbers, Google Brain invented “bfloat16”, where the b stands for “Brain”. They **move 3 bits from the mantissa to the exponent.**



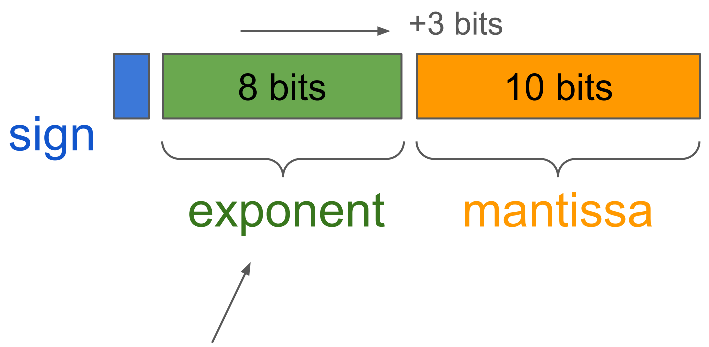
$$\text{Number} = [\text{sign}] * 2^{[\text{exponent}]} * [1+\text{mantissa}]$$

We now have the same range of float32, although we have less precision.

bfloat16 is extremely common for mixed-precision training.

Data Types: tf32

NVIDIA wanted the best of both worlds, so instead of moving three bits they **added three bits**, creating the “TensorFloat-32” data type (which is very misleading as it is actually 19 bits).



Number = [sign] * $2^{\text{[exponent]}}$ * [1+mantissa]

We now have the same range of float32, **and the same precision as float16.**

tf32 is not widely supported, mainly on recent high-end GPUs from NVIDIA (Starting with the A100 line).

Comments

What you use depends on your hardware and DNN library. Pytorch has a very utilities that handle most of the data typing for you.

If you have support for tf32, use that. Otherwise try bfloat16.

You can also try using lower precision for everything (not mixed), and see if your model still converges.

Quantization

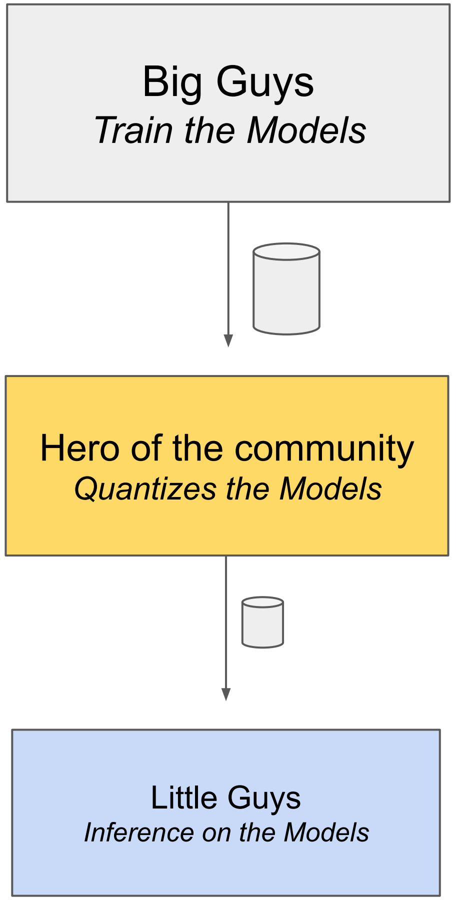
LLM Pipeline

We still largely have an environment in which organizations with large compute train models, and less-resourced organizations try to tune them or use them for inference.

Another application of data types is to ***quantize*** a model: convert it to lower-precision so you can run inference on less hardware.

This is very different from mixed-precision training. When training, we need high precision to perform updates.

If we are running inference, we may not need this anymore!



Quantization

Generally models are quantized into data types that are much more extreme than float32 or float16.

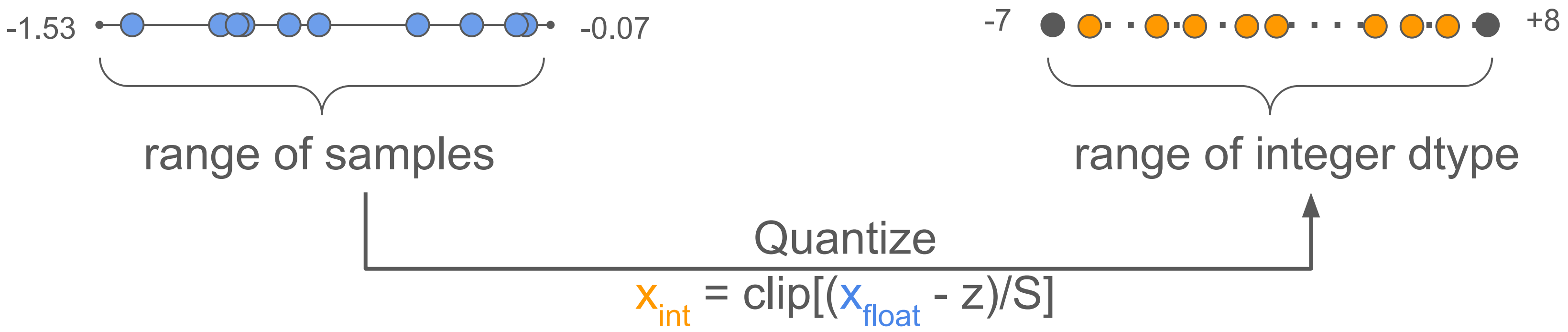
Often int8 or int4 are used. int8 can represent 256 values (8 bits) and int4 can represent 16 (4 bits). In addition to lowering memory, we can now use integer routines for some operations (much faster).

NOTE: Sometimes “int4” means an integer stored with 4 bytes (32 bits, the typical storage for an integer). This is NOT what we mean. Here, “int4” means **4 bits.**

How to Quantize

To quantize our weights or activations, we first sample some of them and determine the range of values we are dealing with.

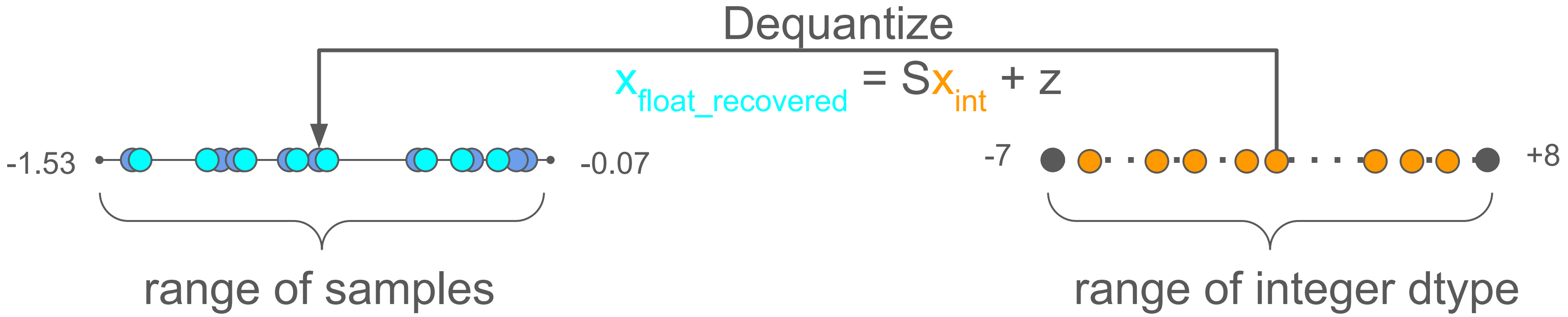
We then map this range of values onto our range of representable integers. The easiest method is just to find a scale factor **S** and offset **z:**



$$x_{int} = \text{clip}[(x_{float} - z)/S]$$

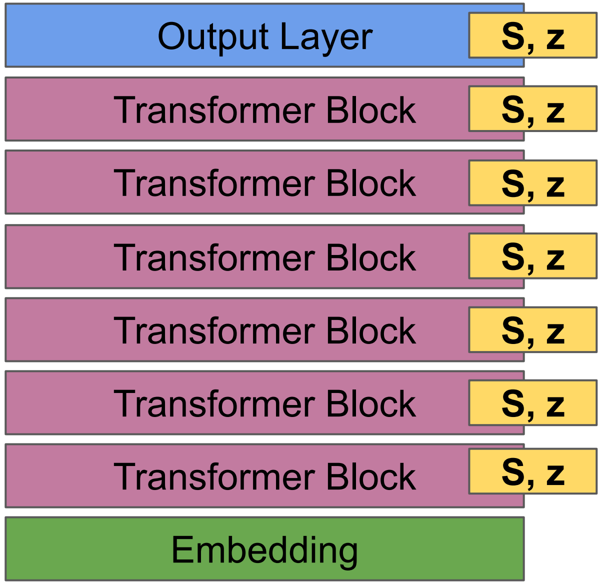
Quantize and Dequantize

We can return to floating point by reversing the mapping. However, this is of course a lossy operation- we can only reconstruct at most 16 distinct values (or 256 for int8):



Quantizing Chunks

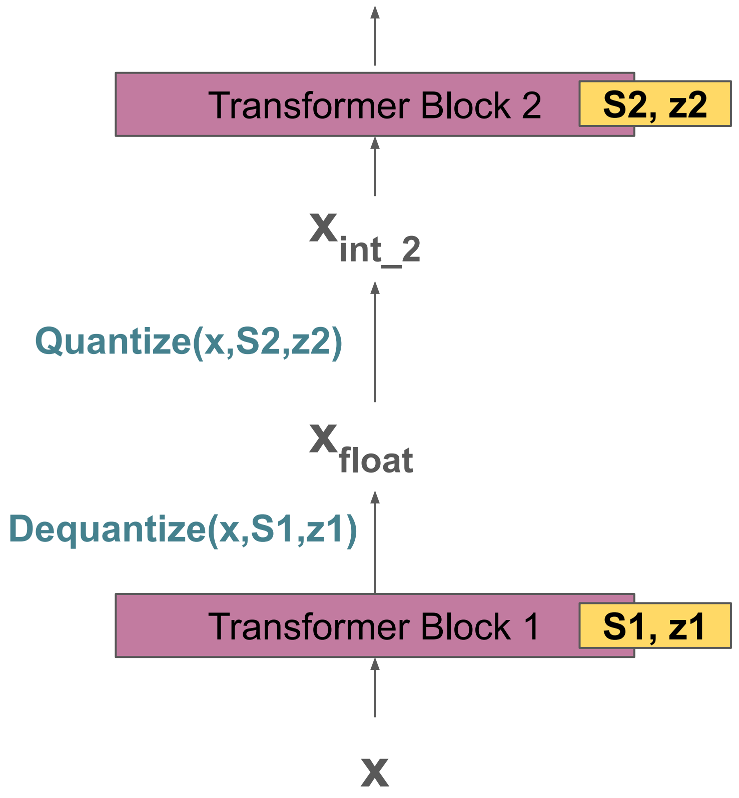
Quantizing an entire model is not practical, since the range of values will be quite large and the loss of information will be too destructive. Instead, we can learn a separate (S,z) for each piece of the model (we can decide how granular these pieces are). A common option is to learn (S,z) for each layer or each tensor.



Quantize and Dequantize

Since each chunk will have a different range of floats that it is meant to represent, we need to dequantize and re-quantize between chunks.

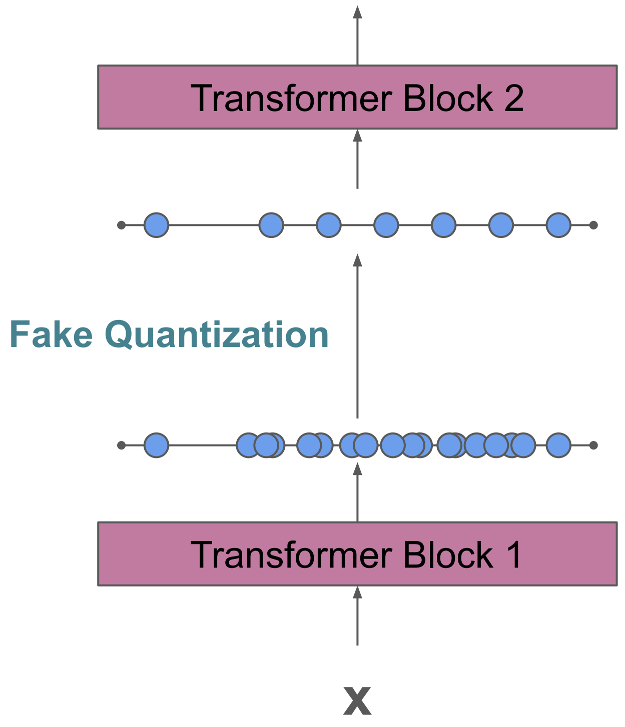
Our data is first mapped back to floats, and then mapped to integers that the next layer expects.



Quantization-Aware Training

Quantizing post-training provides memory and speed improvements, but we are degrading the model (snapping all values to one of 16 or 256 points).

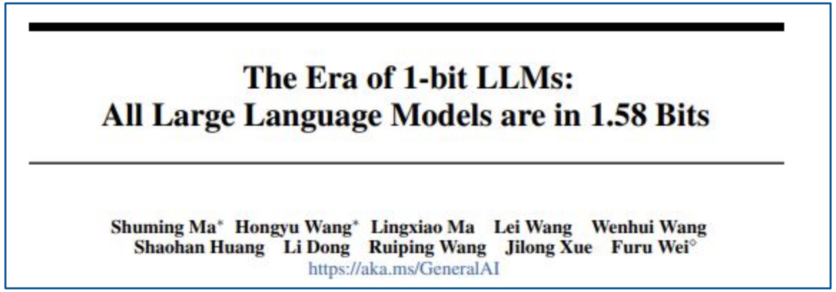
To improve the process, we can train a model such that it is prepared to be quantized. The idea is simple- we keep our model in float32, but throughout the model we snap our value to a discrete set of points. This simulates the effects of data loss due to quantization, and our model can learn how to behave under these conditions.



BitNet b1.58

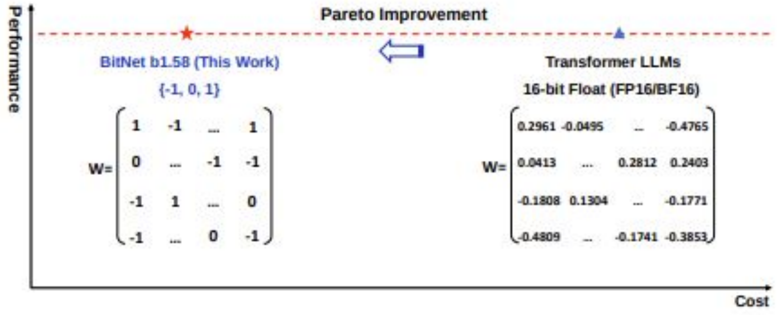
An extreme example of quantization is a paper from February 2024 called BitNet b1.58, which quantized values into 3 bins represented as -1, 0, or 1. During inference they get roughly 10x improvement (either a batch 10x as large or 10x more batches per second).

They find that BitNet b1.58 with 3B parameters performs similarly to LLaMA 3B.



The Era of 1-bit LLMs:
All Large Language Models are in 1.58 Bits

Shuming Ma* Hongyu Wang* Lingxiao Ma Lei Wang Wenhui Wang
Shaohan Huang Li Dong Ruiping Wang Jilong Xue Furu Wei◇
https://aka.ms/GeneralAI



QLoRA

A very popular PEFT+Quantization method is QLoRA, which applies LoRA to a 4-bit quantized model that is frozen (instead of the the original float32 model). They introduce a 4-bit floating point type called NF4, and switch between this (quantized for storage) and bfloat16 (dequantized for calculation) as needed.

The QLoRA paper introduces a new LLM named “Guanaco”. **They fine-tune LLaMA on one GPU in 24 hours.**

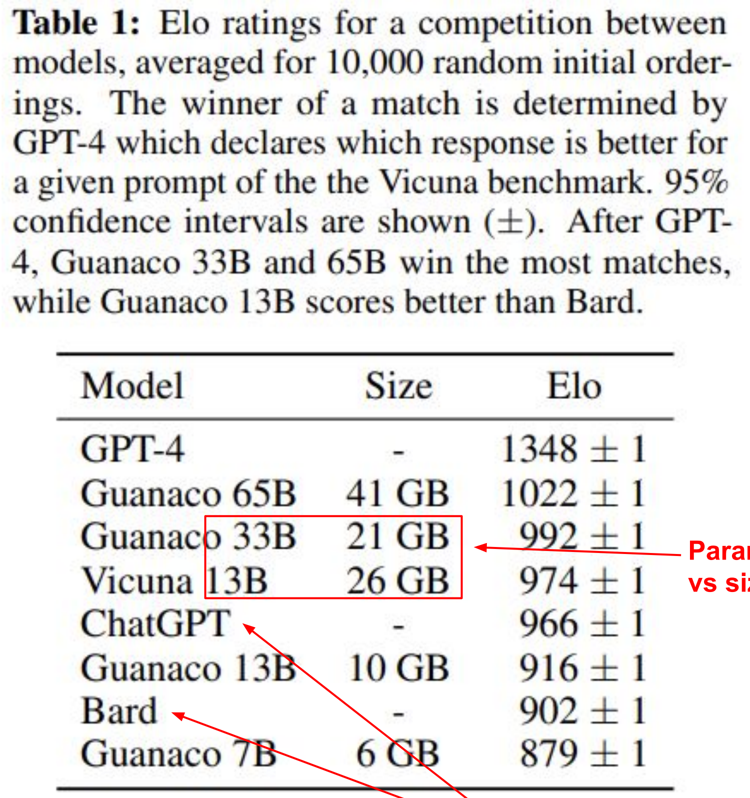


Table 1: Elo ratings for a competition between models, averaged for 10,000 random initial orderings. The winner of a match is determined by GPT-4 which declares which response is better for a given prompt of the the Vicuna benchmark. 95% confidence intervals are shown (±). After GPT-4, Guanaco 33B and 65B win the most matches, while Guanaco 13B scores better than Bard.

Model	Size	Elo
GPT-4	-	1348 ± 1
Guanaco 65B	41 GB	1022 ± 1
Guanaco 33B	21 GB	992 ± 1
Vicuna 13B	26 GB	974 ± 1
ChatGPT	-	966 ± 1
Guanaco 13B	10 GB	916 ± 1
Bard	-	902 ± 1
Guanaco 7B	6 GB	879 ± 1

Parameters vs size!

Stiff competition.

End of Lecture Summary

PEFT is a family of techniques to fine-tune LLMs which limited resources. PEFT methods strategically add parameters and tune them, keeping the rest of the model frozen.

You can also make a model smaller (after training) with quantization- converting to a more efficient data type.

Combinations of these techniques (QLoRA) have yielded very powerful but also very compact models.