PEFT

Lecture 11

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

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)

Solutions

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

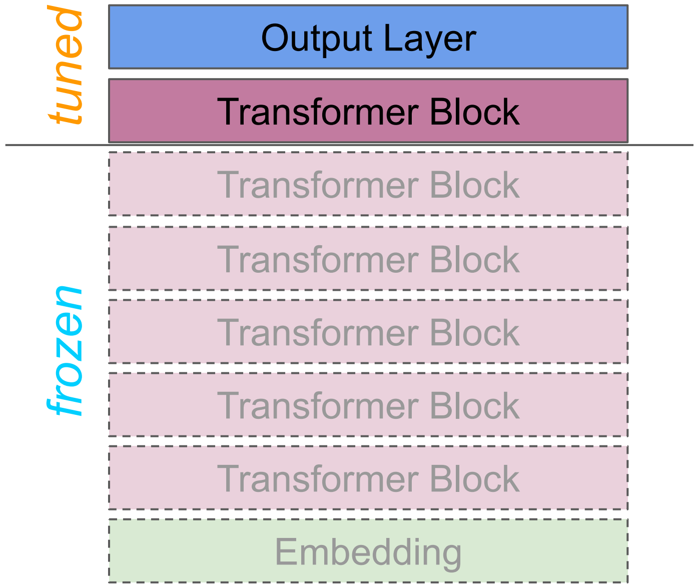
This is the subject of today’s lecture. How to tune a model without $$$.

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.

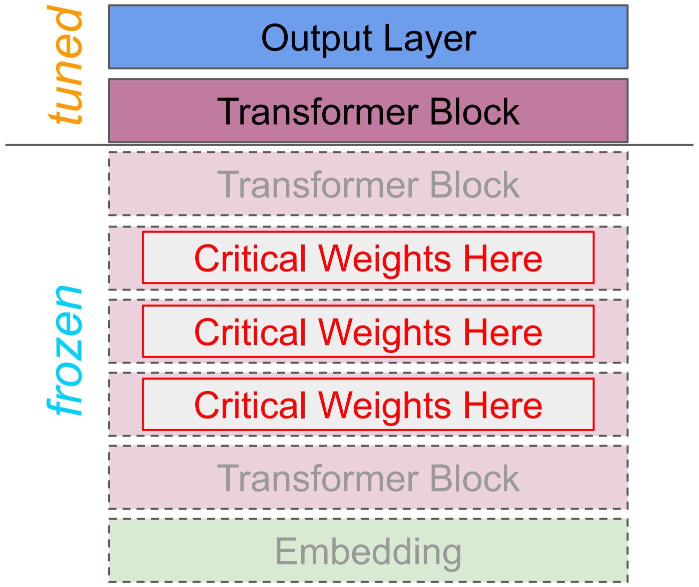


Quick Note

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

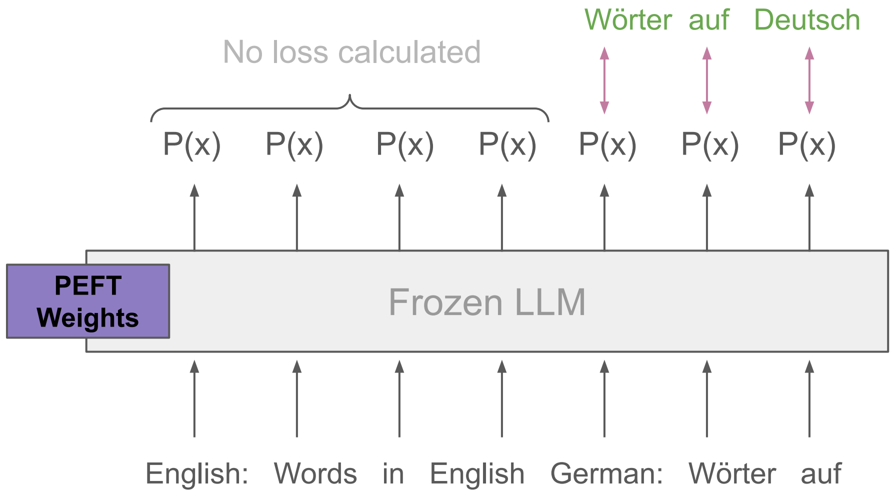
A group of techniques have arisen which (perhaps paradoxically) add parameters to the LLM to make fine-tuning more efficient. We then tune only these parameters and keep the original weights frozen.

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.



Wörter auf Deutsch
No loss calculated
P(x) P(x) P(x) P(x) P(x) P(x) P(x)
PEFT Weights
Frozen LLM
English: Words in English German: Wörter auf

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:

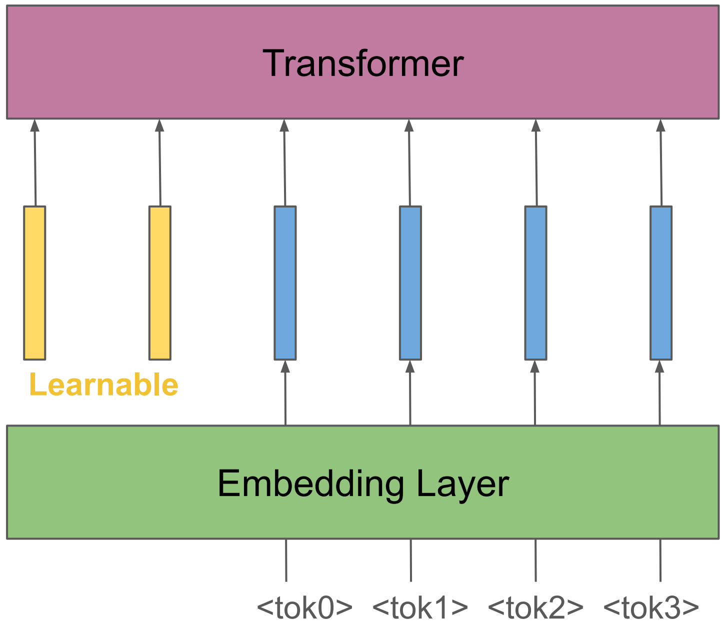
1) In the sequence itself, as learnable tokens
2) Augmenting the existing weight matrices
3) As entirely new weight matrices or layers

(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.

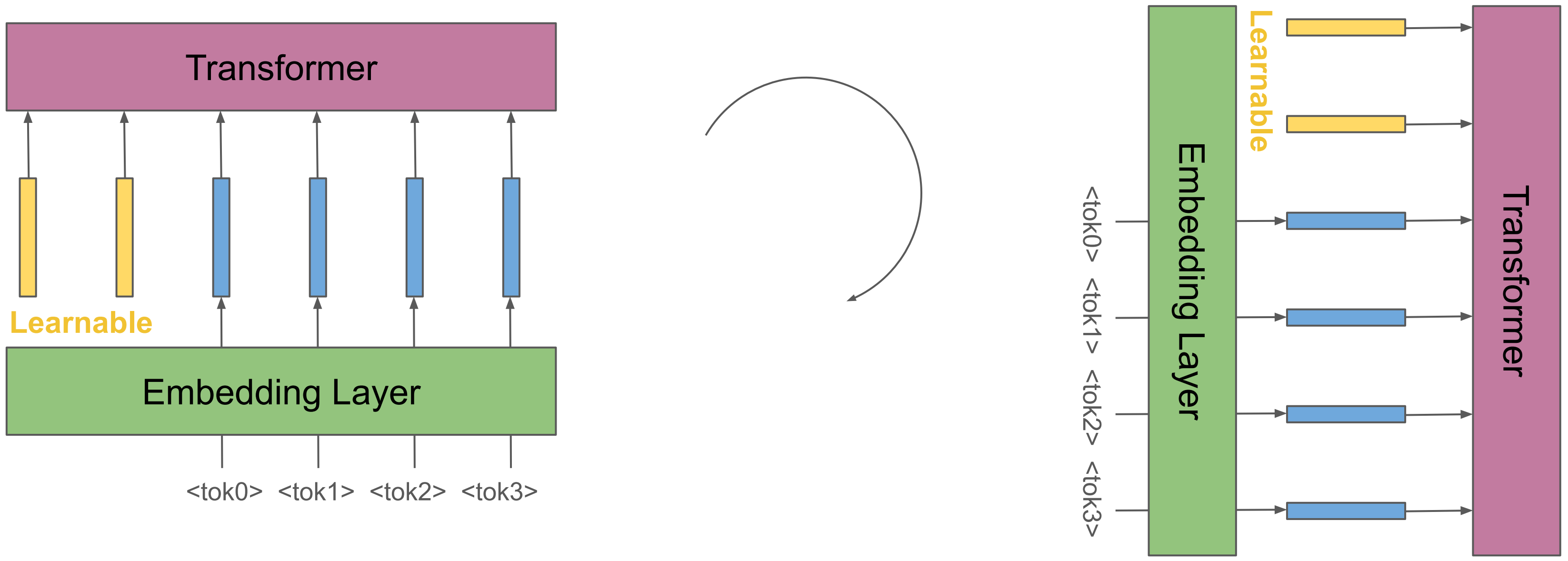
Each of these represents a tunable point in the embedding space. Since these are at the beginning, all “real” tokens can attend to them.

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



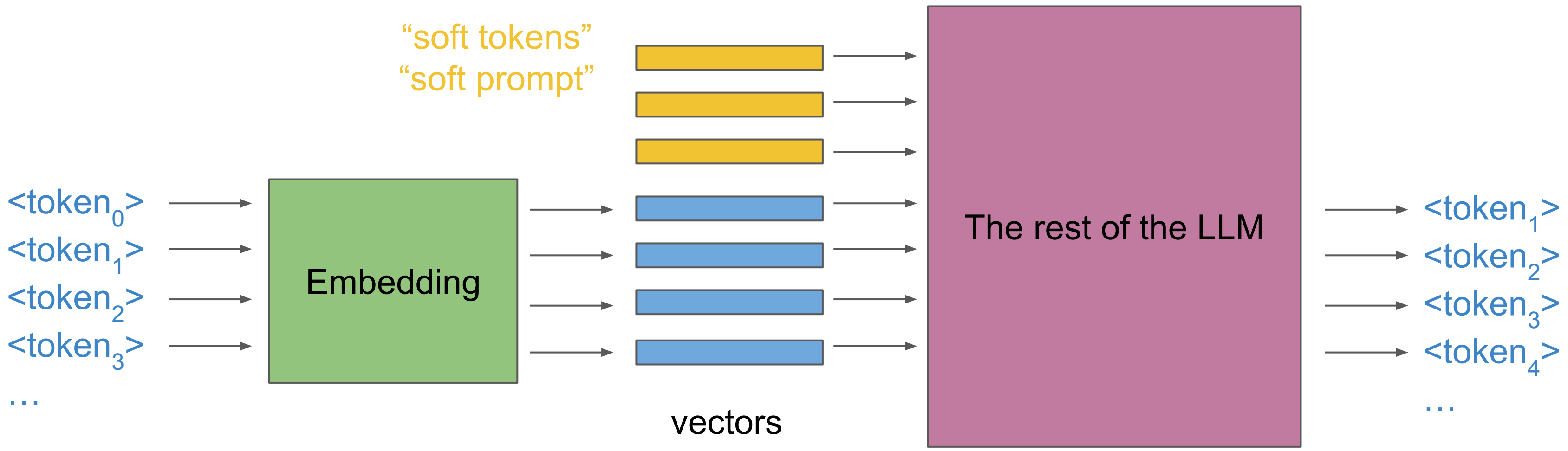
Rotating the diagrams.

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.



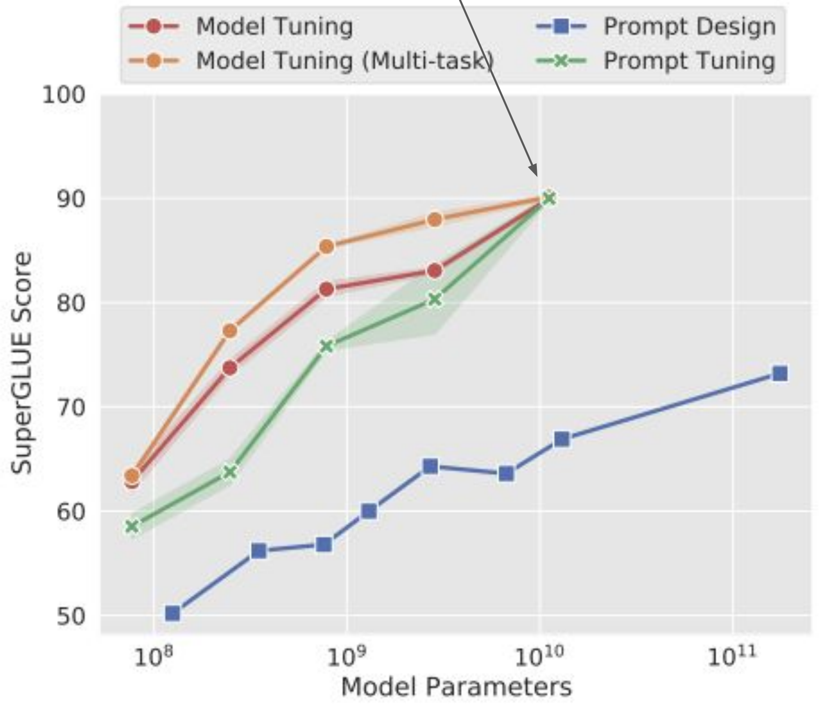
Prompt Tuning Results

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.

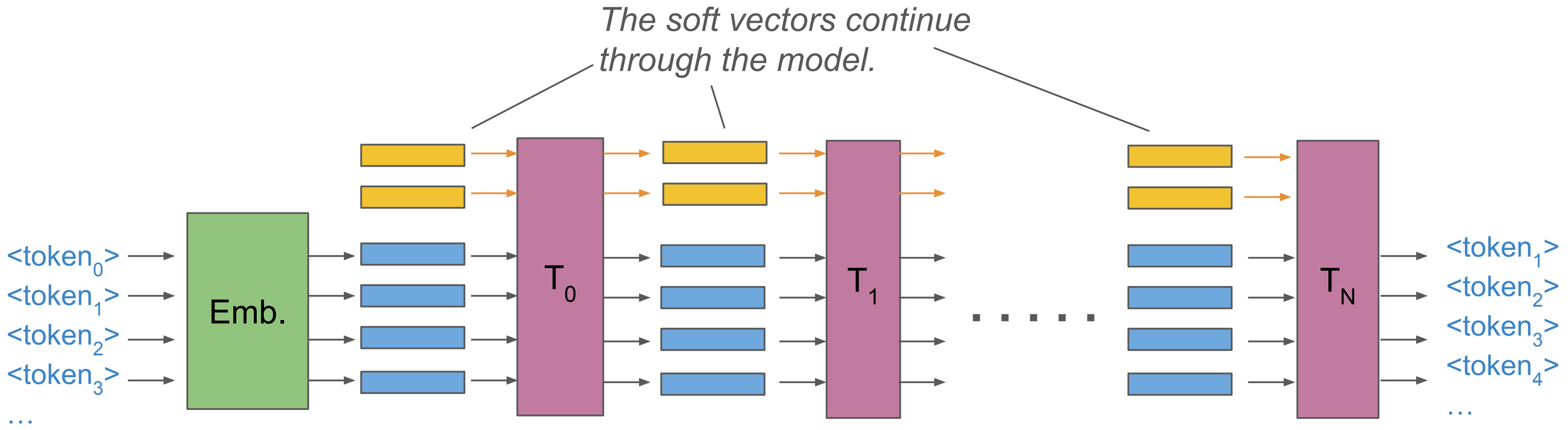
All their experiments converge here.



Prefix Tuning (Li & Liang, Jan 2021)

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

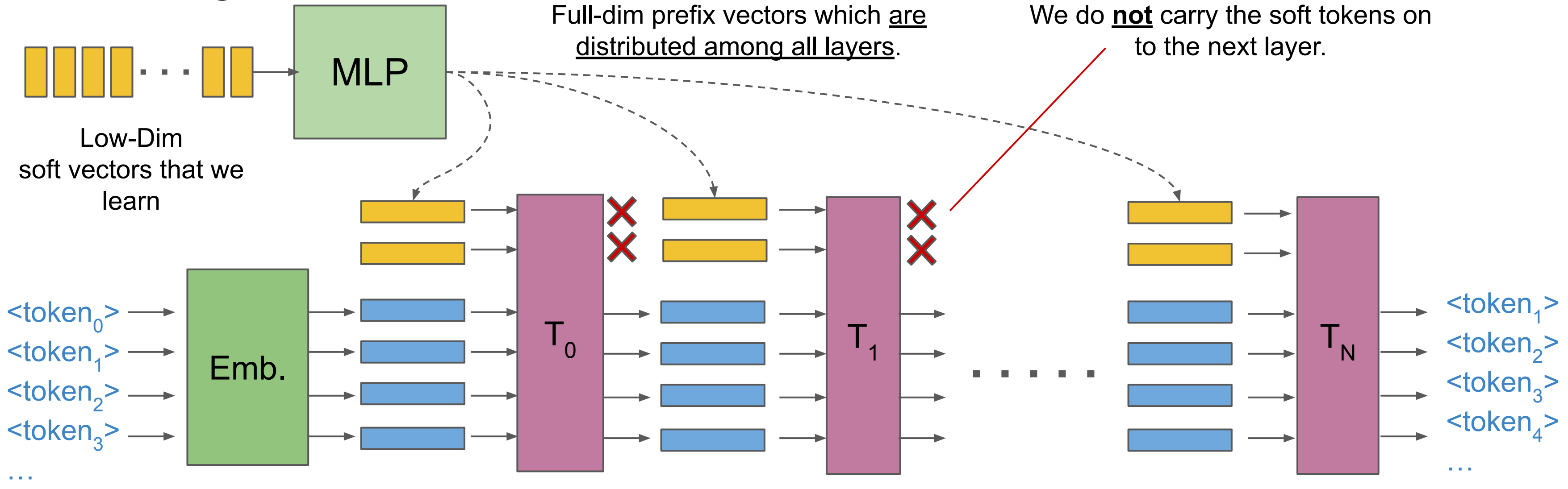
Prompt Tuning, Expanded Network View:


The soft vectors continue through the model.
$\langle token_0 \rangle$
$\langle token_1 \rangle$
$\langle token_2 \rangle$
$\langle token_3 \rangle$
…
Emb.
T_0
T_1
T_N
$\langle token_1 \rangle$
$\langle token_2 \rangle$
$\langle token_3 \rangle$
$\langle token_4 \rangle$
…

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:

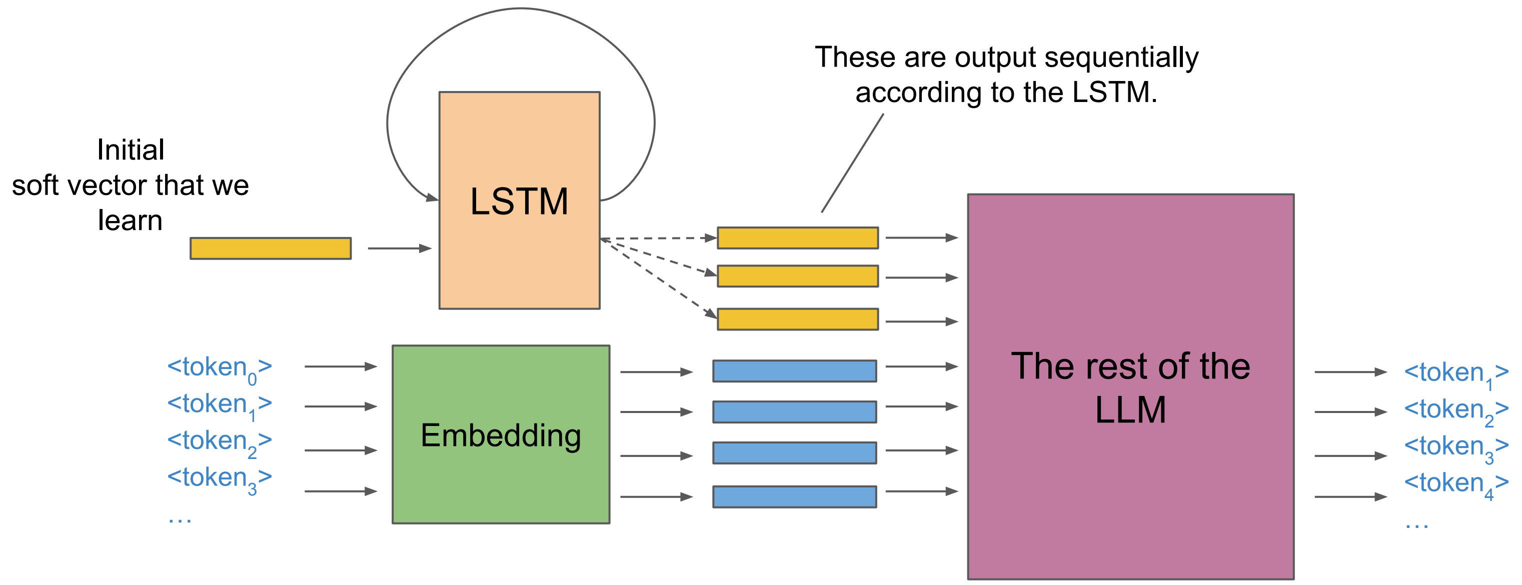
Prefix Tuning:



MLP
Full-dim prefix vectors which are distributed among all layers.
We do not carry the soft tokens on to the next layer.
Low-Dim soft vectors that we learn
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.

(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?

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

Here I just mean “the updated weights”, not a derivative or gradient.

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?

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

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

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

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

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).

$$W_{\text{tuned}}^{M\text{x}N} = W_{\text{frozen}}^{M\text{x}N} + \boxed{W_{\text{delta}}^{M\text{x}N}}$$

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).

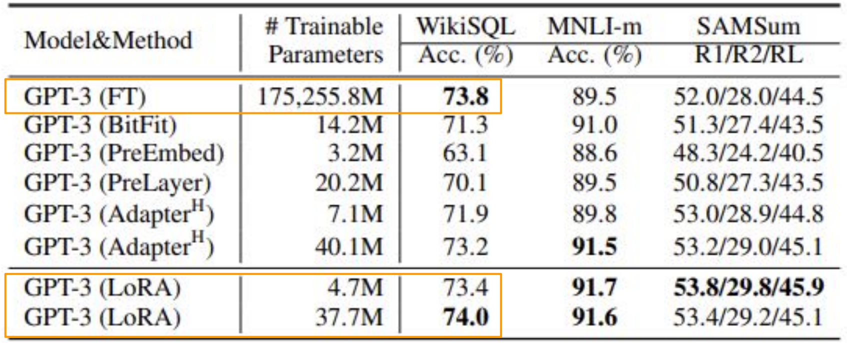
Solution: Factorize W_{delta} into two smaller matrices.

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

Here $r \ll M$ 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).

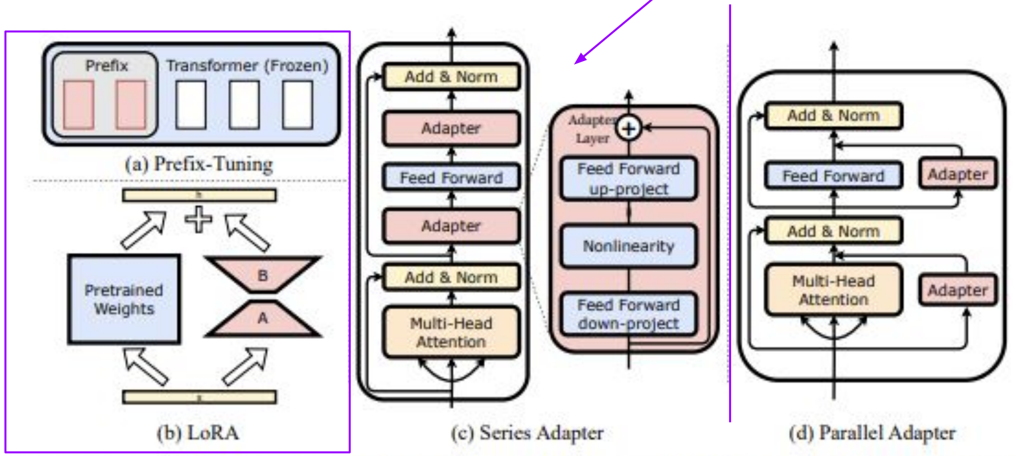
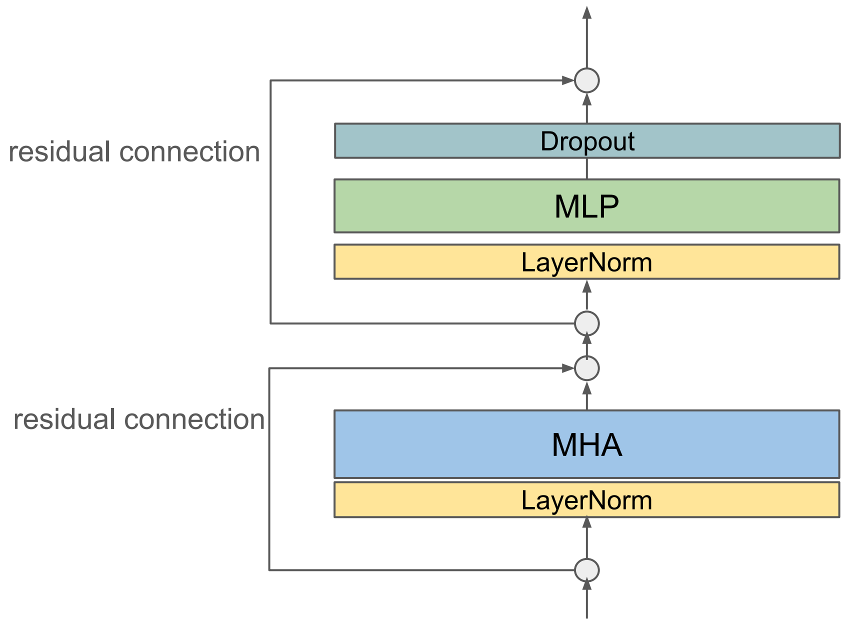
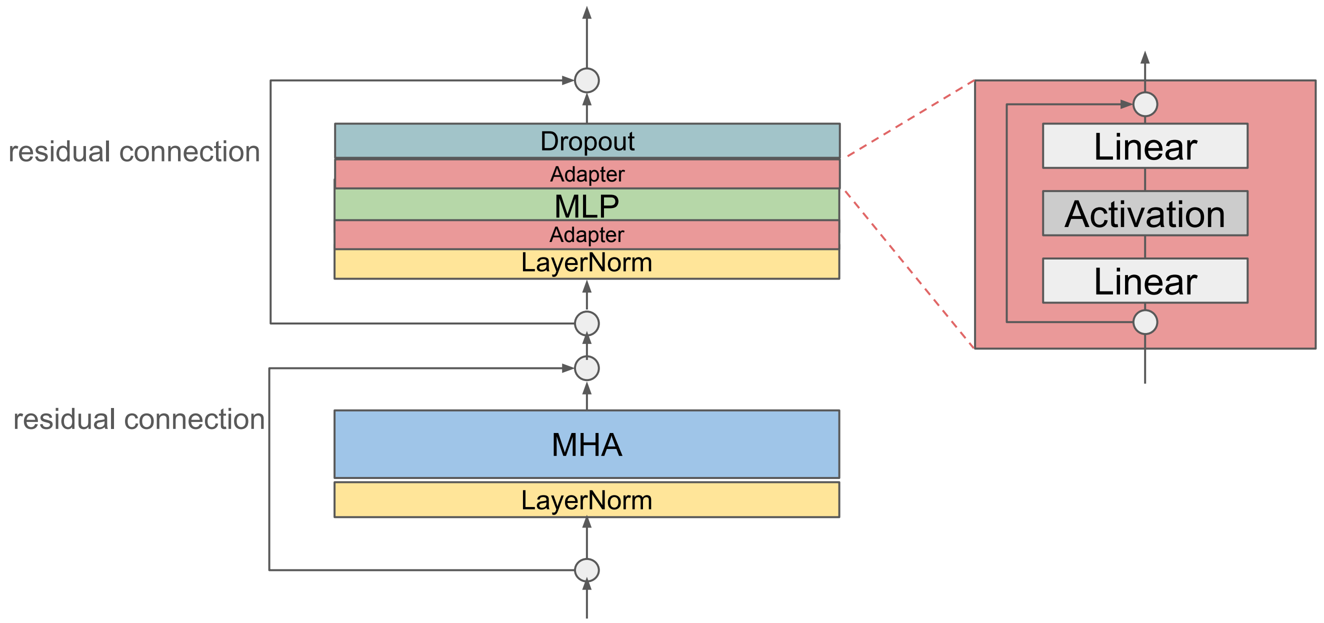


Figure 1: A detailed illustration of the model architectures of three different adapters: (a) Prefix-Tuning, (b) LoRA, (c) Series Adapter, and (d) Parallel Adapter.

Adapters (Our Diagrams)



Adapters (Our Diagrams)

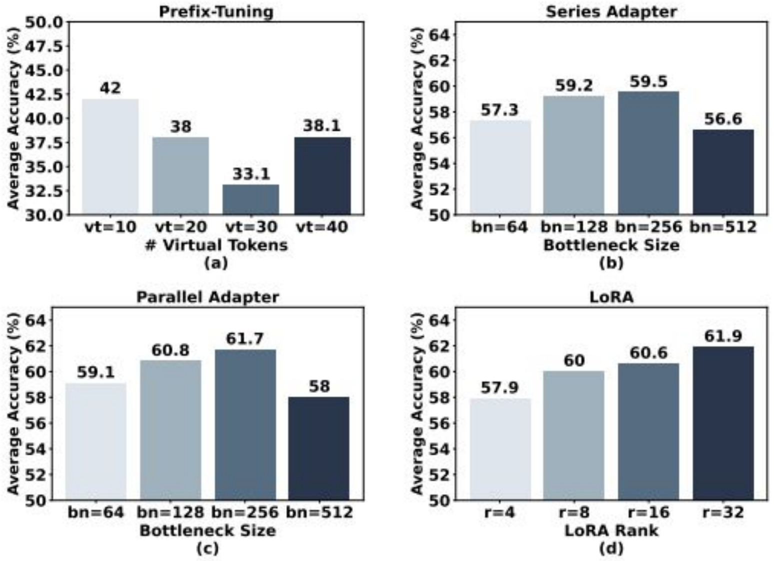


Hu et al results / Summary

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.

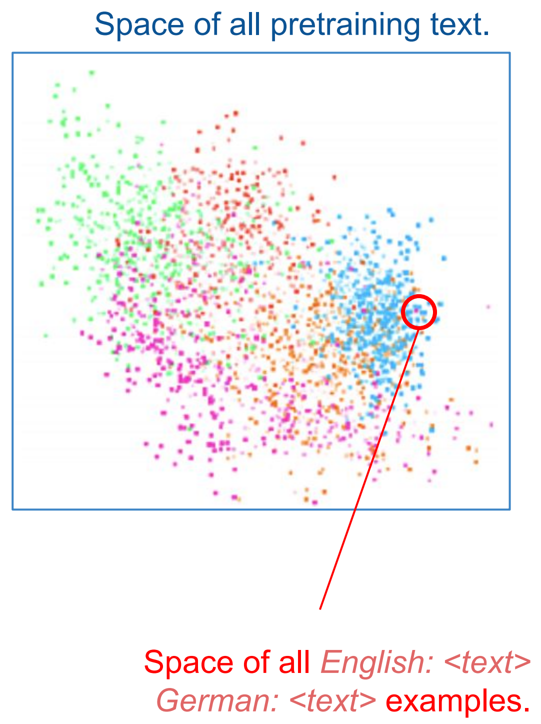
I think all of them are options for a given problem. They all offer similar benefits: Extremely low tunable parameter counts, very good performance.



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.

Additional Resources

PEFT Methods:

Prompt Tuning: https://arxiv.org/abs/2104.08691

Prefix Tuning: https://arxiv.org/abs/2101.00190

P-Tuning: https://arxiv.org/abs/2103.10385

LoRA: https://arxiv.org/abs/2106.09685

Adapters: https://arxiv.org/pdf/2304.01933