Introduction

The goal of Transfer Learning is to adapt a Pre-Trained model for a Source task (the "base" model) to solve a new Target task.

Adapting a base model is typically performed by Fine-Tuning

- allowing the weights of the base model (and any additional "head") layers to adapt
- by training with a relatively small number of examples from the Target task.

Although Fine-Tuning is effective, there is a problem, especially with LLM base models

- LLM models can have a very large number N of parameters
- They are increasingly deep: number of stacked Transformer blocks n_{layers} is growing
    - latency in training

Even training on a small number of Target task examples is expensive in time and memory.

The question we address in this module

- Can we adapt a base model *without* modifying *all* of the parameters of the base model ?

We will refer to this problem as *Parameter Efficient Transfer Learning*

- or *Parameter Efficient Fine-Tuning* when Fine-Tuning is used as the method for adaptation

We want the number of *adapted* parameters to be small relative to the total number of base model parameters.

We will use this fraction as a metric in comparing adaptation methods.

We note that the number of parameters in a Transformer is $N = \mathcal{O}\left(n_{\text{layers}} * d^2\right)$

- where d is the internal dimension of the Transformer
- calculations may be found in [our notebook (Transformer.ipynb#Number-of-parameters)](Transformer.ipynb#Number-of-parameters) and [here (https://arxiv.org/pdf/2001.08361.pdf#page=6)](https://arxiv.org/pdf/2001.08361.pdf#page=6)

Motivation for Parameter Efficient Transfer Learning

A base model may have a large number of parameters (e.g., an LLM)

- Adapting *all* the parameters may require large quantities of time and space
- Reducing the number of adapted parameters may have efficiency advantages

Beyond the obvious efficiency advantage

- there is a space advantage
- the specialization of the Base Model to a Target Task can be represented by the small number of adapted parameters

This means that the parameters of the same base model can be *shared*

- across models for different Target tasks
- with one set of separate (but small) adapted parameters for each Target

This is also potentially a way to enable per-user instances of a Target task

- with user-specific training examples kept private to each user's instance

Adapters

References

- Parameter Efficient Transfer Learning for NLP (https://arxiv.org/pdf/1902.00751.pdf)
- LLM Adapters (https://arxiv.org/pdf/2304.01933.pdf)

Adapters are modules (implemented as Neural Networks)

- that are inserted into the existing modules (layers) of the base model.

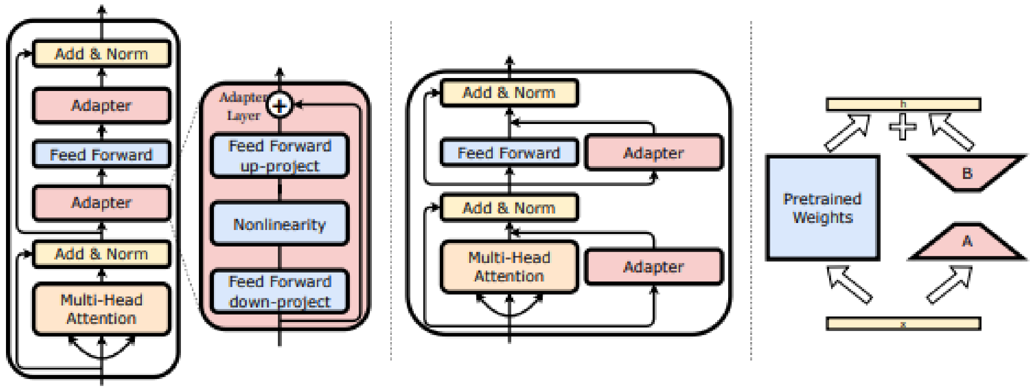
In the general case:

- we can insert one or more adapters *anywhere* within the NN comprising the base model.

Within a *single* Transformer block, typical arrangements are

- Series
  - Adapter inserted between modules
- Parallel
  - Adapter inserted parallel to a module
    - provided an alternate path *by-passing* the module

Various Adapter designs

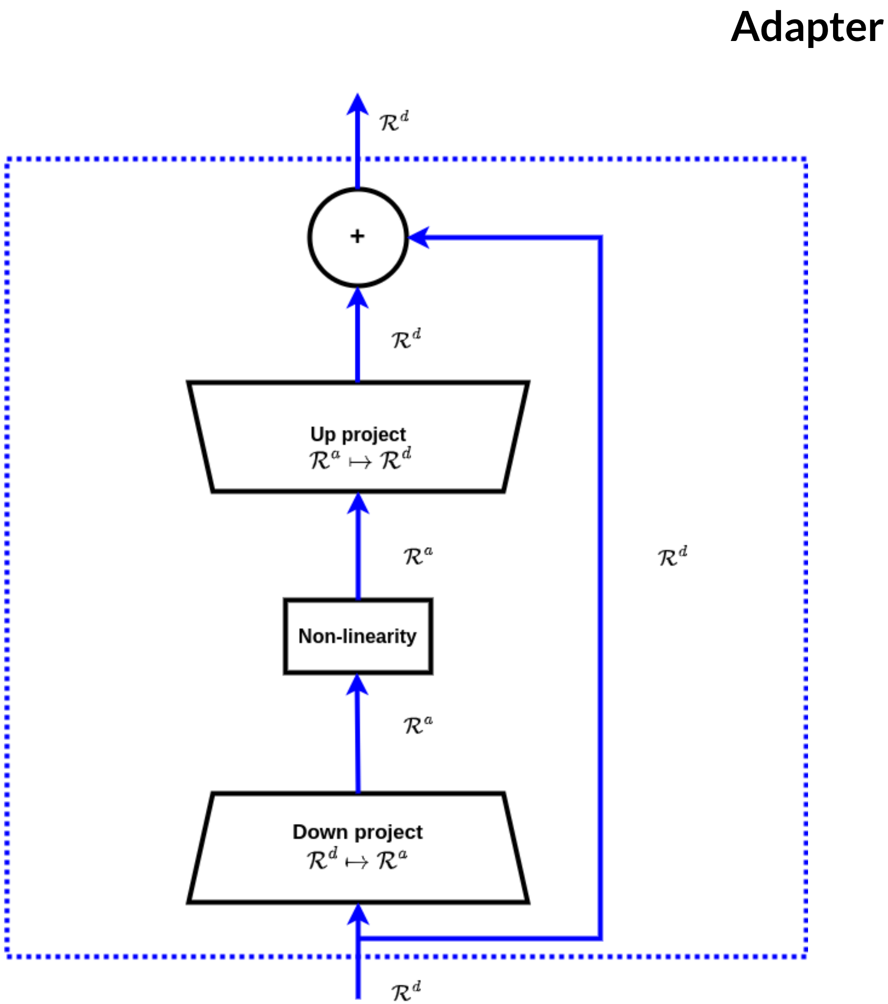


(a) Series Adapter (b) Parallel Adapter (c) LoRA

Figure 1: A detailed illustration of the model architectures of three different adapters: (a) Series Adapter (Houlsby et al., 2019), (b) Parallel Adapter (He et al., 2021), and (c) LoRA (Hu et al., 2021).

Attribution: https://arxiv.org/pdf/2304.01933.pdf#page=2

Here is a diagram of a common adapter



The dimensions of the input and output of the adapter

- are the same d (common vector dimension) used for all layers in a Transformer
- facilitates inserting adapters anywhere in the Transformer

The usual architecture

- usually two modules, with a bottleneck of dimension $a < d$
    - Project down to reduced dimension; Project up to original dimension
- skip connection around the two projection modules

We are already familiar with adaptation via Adapter-like modules

- adding a new "head" layer to a head-less base model
    - often a Classifier to adapt the base model to the particular Target classes
- [Feature based transfer learning (NLP_Language_Models.ipynb#Other-uses-of-a-Language-Model:-Feature-based-Transfer-Learning)](NLP_Language_Models.ipynb#Other-uses-of-a-Language-Model:-Feature-based-Transfer-Learning)
    - feeding the representation created by the base model to another module.
- these are not technically adapters
    - input and output dimensions don't match
    - architecture may differ

Regardless of where Adapters are placed

- they derive a new function g from the function f computed by the base model

Formally:

- f_Θ denotes the function computed by the base model which is parameterized by Θ
- $g_{\Theta,\Phi}(\mathbf{x})$ denotes the function computed by the adapted model
    - Φ are the Adapter parameters
    - Θ are the base model parameters

Adapter Tuning occurs when we train only the parameters Φ of the Adapter modules

- on a small number of examples from the Target task
- freezing the parameters of the base model

During epoch t of Adapter Tuning, we learn $\Phi_{(t)}$

- initialing $\Phi_{(0)}$ such that
$$g_{\Theta, \Phi_{(0)}}(\mathbf{x}) \approx f_{\Theta}(\mathbf{x})$$
- can be achieved by setting $\Phi = 0$
    - because of the skip connection, the adapter output becomes $f_{\Theta}(\mathbf{x})$

Bottleneck size

Since Adapter Tuning does not change base model parameters Θ,

- the space used depends on the size of Φ
- this is the key to adapting the base model using a small number of parameters

The number of parameters of the projection components of the Adapter are $\mathcal{O}\left(d * a\right)$, multiplied by the number k of Adapters.

Recall that a number of parameters in a Transformer are $\mathcal{O}\left(n_{\text{layers}} * d^2\right)$.

Expressing the size of Φ as a fraction of the size of Θ:

$$
\begin{aligned}
r &= \frac{|\Phi|}{|\Theta|} \\
&\approx \frac{d*a*n_{\text{layers}}}{n_{\text{layers}}*d^2} & \text{since} \\
& \quad |\Phi| = \mathcal{O}\left(d * a * n_{\text{layers}}\right) \text{ assuming } k = n_{\text{layers}} \\
& \quad |\Theta| = \mathcal{O}\left(n_{\text{layers}} * d^2\right) \text{ for a Transformer} \\
&\approx \frac{a}{d}
\end{aligned}
$$

For reference, $d = 12,288$ for GPT-3; a is chosen to satisfy a target for r

- e.g., $r = 0.1\%$, results in bottleneck size $a = 12$

In [experiments (https://arxiv.org/pdf/1902.00751.pdf#page=4)](https://arxiv.org/pdf/1902.00751.pdf#page=4), the the botttleck was varied
$$a \in \{2, 4, 8, 16, 32, 64\}$$
so typical a is a fraction of 1%.

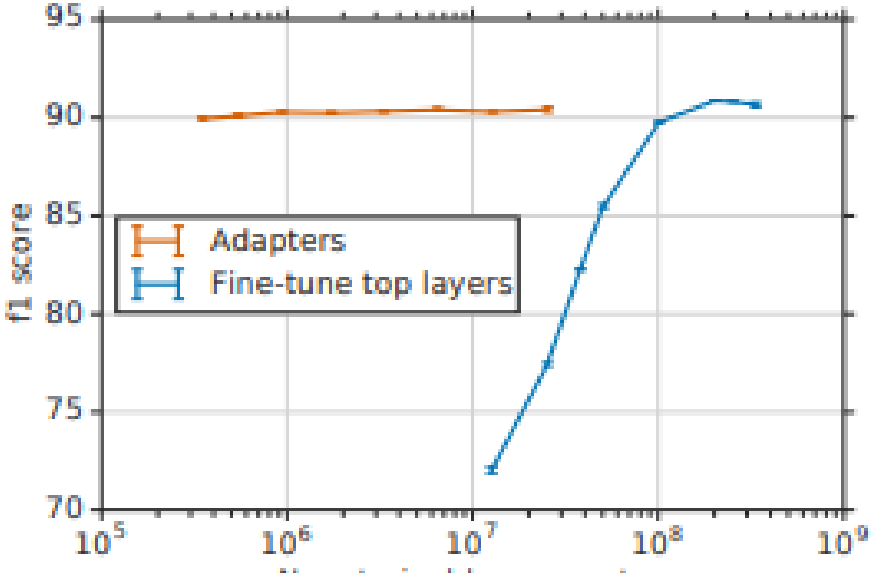
[The effect of varying a (https://arxiv.org/pdf/1902.00751.pdf#page=7)](https://arxiv.org/pdf/1902.00751.pdf#page=7) are shown in the orange line in the diagram below

- the horizontal axis is the total number of trainable parameters, which is linear in a
- it seems to show that increasing the size of the bottleneck does not impact performance greatly

The table also compares adaptation via Adapters to adaptation by Fine-Tuning only the top layers of the base model

- the total number of trainable parameters increases with the number of top layers fine-tuned
- the results show that adaptation via Adapters is better than Fine Tuning top layers
    - *unless* we Fine-Tune *many* top layers

Adapter vs Fine Tuning



95
90
85
80
75
70
f1 score
Adapters
Fine-tune top layers
10^5
10^6
10^7
10^8
10^9

Adapter placement

Recall that Transformer blocks are usually stacked into n_{layers} in a Transformer for an LLM.

Initially, Adapters were placed at *each* level of the stack.

However, [experiments (https://arxiv.org/pdf/1902.00751.pdf#page=8)](https://arxiv.org/pdf/1902.00751.pdf#page=8) show that the most impactful adapters are located at the *top* of the stack.

In the study, adapters are *removed* within a span of levels of the stacked blocks.

- the models are **not re-trained** after removing the adapters

The horizontal/vertical axes indexes the *end/start* of the span.

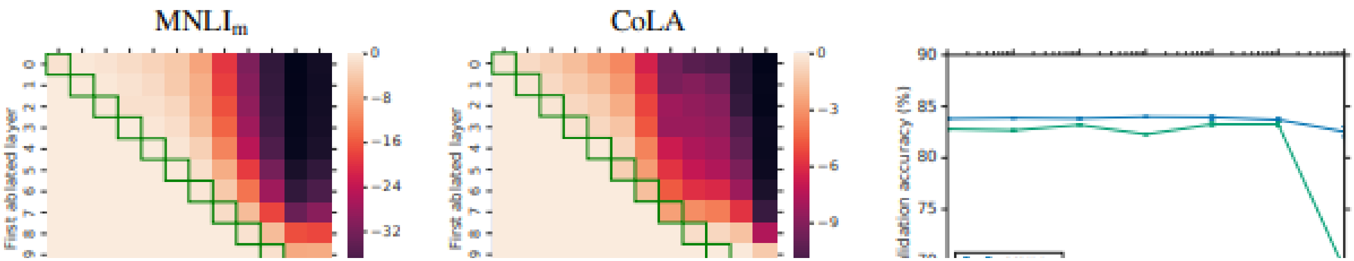
Columns 7 and beyond indicates the removing adapters does not decrease performance

- until the adapter at level 7 is removed

The last column indicates that the largest performance decrease occurs

- when removing the single adapater at the top level

Adapter placement



MNLI_m
CoLA
First ablated layer
Validation accuracy (%)

This is interesting

- Recall, our hypothesis of Deep learning is that increasing levels of abstraction of the inputs are created as layers become deeper
- The early layers create representations that transfer across most tasks
- The deepest layer representations are most task-specific

The decrease in performance corresponding to deeper layers

- may indicate that the Target task specific adaptation
- occurs in the region which we associate most with the Source task

LoRA

References

- [LoRA:Low Rank Adaptation of Large Language Models (https://arxiv.org/pdf/2106.09685.pdf)](https://arxiv.org/pdf/2106.09685.pdf)

Additional reading

- [Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning (https://arxiv.org/abs/2012.13255)](https://arxiv.org/abs/2012.13255)
- [LoRA Learns Less and Forgets Less (https://arxiv.org/pdf/2405.09673)](https://arxiv.org/pdf/2405.09673)

The Adapter method of Fine Tuning uses a module involving

- Down projecting to a lower dimension
- Up projecting back to the original dimension
- with an intervening non-linearity
- where the projections are achieved via Dense layers

We now show the Low Rank Adaptation (LoRA) method that is similar

- Down and Up Projections
- without an intervening non-linearity
- where the projections are achieved via matrix multiplication

Let $\mathbf{W}$ denote the parameters of the Pre-Trained Model.

Fine-Tuning updates the parameters to

$$\mathbf{W}' = \mathbf{W} + \Delta\mathbf{W}$$

The usual method is to use Gradient Descent to create a sequence of parameter updates

- one per mini-batch
- equal to negative one times the learning-rate scaled gradient of the Loss

$$\begin{aligned}
\mathbf{W}_{(0)} &= \mathbf{W} \\
\text{update}_t &= -\alpha_t * \frac{\partial \mathcal{L}_{\mathbf{W}_{(t-1)}}}{\partial \mathbf{W}_{(t-1)}} \\
\mathbf{W}_{(t)} &= \mathbf{W}_{(t-1)} + \text{update}_t
\end{aligned}$$

$$\Delta\mathbf{W} = \sum_t \text{update}_t$$

LoRA uses a different method

- using Gradient Descent to approximate the *cumulative* change $\mathbf{\Delta W}$.

Computing $\Delta \mathbf{W}$

LoRA does not **learn** $\Delta \mathbf{W}$ directly.

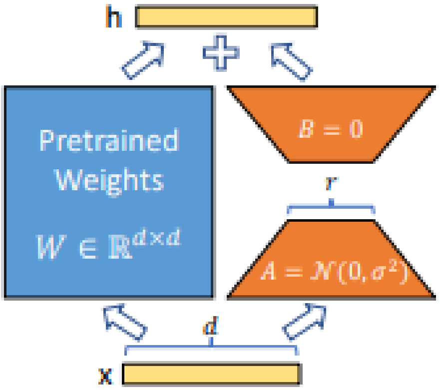
- It factors $\Delta \mathbf{W}$ as the product of two *smaller* lower rank matrices A, B:
$$\Delta \mathbf{W} = A * B$$

out		down project		up project
$\Delta \mathbf{W}$	=	A	*	B
$(d \times d)$		$(d \times r)$		$(r \times d)$

where $r \leq \operatorname{rank}(\Delta \mathbf{W})$

Here is the architecture

LoRA adapting Pre-Trained matrix W



Attribution: https://arxiv.org/pdf/2106.09685.pdf#page=1

Given input $\mathbf{x}$, this arrangement results in output h

$$
\begin{aligned}
h &= \mathbf{W}_0 * \mathbf{x} && \text{the left branch} \\
&\quad + \mathbf{x} * A * B && \text{the sum operator on top} \\
&= \mathbf{W}_0 * \mathbf{x} + \Delta\mathbf{W} * \mathbf{x} && \Delta\mathbf{W} = A * B \\
&= (\mathbf{W}_0 + \Delta\mathbf{W}) * \mathbf{x} && \text{distributive property} \\
&= \mathbf{W}' * \mathbf{x} && \mathbf{W}' = \mathbf{W}_0 + \Delta\mathbf{W}
\end{aligned}
$$

Thus, the output is $\mathbf{W}' * \mathbf{x}$, satisfying the goal of adapting $\mathbf{W}$ to $\mathbf{W}'$.

Note the computational advantage of computing

$$(\mathbf{x} * A) * B$$

over

$$\mathbf{x} * (A * B)$$

- We avoid constructing the $(d \times d)$ matrix $(A * B)$
- in favor of constructing *short* vectors

down project		**up project**	
$(\mathbf{x} * A)$	*	B	
$d * (d \times r)$	*	$(r \times d)$	dimensions
r	*	$(r \times d)$	left product: dimensions r
d			final dimensions: d

The two dimensions of A and B are d and r.

Thus, the resulting number of parameters

- is $2 * d * r$ parameters
- rather than d^2

So, not only is the representation of $\mathbf{\Delta W}$ smaller, there are fewer parameters to Fine-Tune.

Matrix B is initialized to 0 so that

- when Fine-Tuning begins

$$
\begin{aligned}
\mathbf{W}' &= \mathbf{W}_0 &+& (A * B) \\
&= \mathbf{W}_) &+& (A * 0) \\
&= \mathbf{W}_0
\end{aligned}
$$

- the initial output is the *same* as the unmodified weights

A, B get updated during Fine-Tuning

- by gradient descent on the elements of the matrices

The original weights $\mathbf{W}_0$ are **frozen** and not updated by Gradient Descent.

Note the similarity to the Adapter used in a Parallel arrangement.

The advantage of the Parallel arrangement compared to a Series arrangement

- the Series introduces an added layer
- each time it appears
- which slows *inference*

The Parallel arrangement used in LoRA does not introduce latency at inference time.

How big does r have to be ?

Not much ! Values of $r \leq 2$ seem to do very well in an experiment

The accuracy reported when $r = 2$ is almost the same as when $r = 64$

LoRA: accuracy versus rank r

7.2 WHAT IS THE OPTIMAL RANK r FOR LORA?

We turn our attention to the effect of rank r on model performance. We adapt $\{W_q, W_v\}$, $\{W_q, W_k, W_v, W_c\}$, and just W_q for a comparison.

	Weight Type	$r=1$	$r=2$	$r=4$	$r=8$	$r=64$
WikiSQL($\pm 0.5\%$)	W_q	68.8	69.6	70.5	70.4	70.0
	W_q, W_v	73.4	73.3	73.7	73.8	73.5
	W_q, W_k, W_v, W_o	74.1	73.7	74.0	74.0	73.9
MultiNLI ($\pm 0.1\%$)	W_q	90.7	90.9	91.1	90.7	90.7
	W_q, W_v	91.3	91.4	91.3	91.6	91.4
	W_q, W_k, W_v, W_o	91.2	91.7	91.7	91.5	91.4

Table 6: Validation accuracy on WikiSQL and MultiNLI with different rank r. To our surprise, a rank as small as one suffices for adapting both W_q and W_v on these datasets while training W_q alone needs a larger r. We conduct a similar experiment on GPT-2 in Section H.2.

Attribution: https://arxiv.org/pdf/2106.09685.pdf#page=10

Results

How do the various adaptation methods compare according to the authors ?

LoRa with 37.7MM parameters ($.02\%$ of GPT-3) *outperforms* full Fine-Tuning.

LoRA: Performance, by method of adaptation

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

Table 4: Performance of different adaptation methods on GPT-3 175B. We report the logical form validation accuracy on WikiSQL, validation accuracy on MultiNLI-matched, and Rouge-1/2/L on SAMSum. LoRA performs better than prior approaches, including full fine-tuning. The results on WikiSQL have a fluctuation around $\pm 0.5\%$, MNLI-m around $\pm 0.1\%$, and SAMSum around $\pm 0.2/\pm 0.2/\pm 0.1$ for the three metrics.

Attribution: https://arxiv.org/pdf/2106.09685.pdf#page=8

is LoRA as good as full Fine-Tuning ?

Compare the $\Delta \mathbf{W}$ of LoRA to that of full Fine-Tuning

- $\Delta \mathbf{W}_\text{LoRA}$ is low rank
- $\Delta \mathbf{W}_\text{Fine tuning}$ is of unconstrained rank

It has been [shown (https://arxiv.org/pdf/2405.09673)](https://arxiv.org/pdf/2405.09673) that for some Target tasks

- $\Delta \mathbf{W}$ is of *high* rank
- so LoRA will under-perform Fine Tuning for these Target tasks

On the other hand:

- $\mathbf{W}'_{\text{LoRA}} = \mathbf{W}_0 + \Delta\mathbf{W}_{\text{LoRA}}$
- is more similar to $\mathbf{W}_0$
- than $\mathbf{W}'_{\text{Fine tuning}}$
$$= \mathbf{W}_0 + \Delta\mathbf{W}_{\text{Fine tuning}}$$

so it has been found that LoRA is *less likely forget* the Source Task than full Fine-Tuning.

Summary

LoRA

- learns less (Target task)
- forgets less (Source task)

Technical aside: what does ”similar” mean above ?

Note

"Similar" is used in a very loose manner above.

The relation between modified $\mathbf{W}'$, base $\mathbf{W}_0$, and "perturubation" $\Delta\mathbf{W}$ was evaluated

- Using SVD (recall: used in PCA)
- To determine the number of singular vectors required to capture 90% of the variance of each

For Fine-tuning

- The number of singular vectors required
- was similar for $\mathbf{W}'_{\text{Fine tuning}}$, $\mathbf{W}_0$ and $\Delta\mathbf{W}_{\text{Fine tuning}}$

For LoRA

- the number of singular vectors for $\Delta \mathbf{W}_{\text{LoRA}}$ is much smaller, for low rank r
- the rank r required
    - in order for the number of singular vectors of $\Delta \mathbf{W}_{\text{LoRA}}$
    - to approach the number of singular vectors of $\mathbf{W}_0$
    - is 10 to 100 times greater than the typical (small) r used in LoRA

So the low rank r typically used for $\Delta \mathbf{W}_{\text{LoRA}}$

- is unable to capture the true (i.e., Fine-tuned) rank of $\mathbf{W}'_{\text{Fine tuning}}$

This is dependent on the Source Task (i.e., $\mathbf{W}_0$) and the Target Task (i.e., $\mathbf{W}'$)

BitFit

References

- BitFit: Simple Parameter-efficient Fine-tuning for Transformer-based Masked Language-models (https://arxiv.org/pdf/2106.10199.pdf)

Our goal remains

- to fine-tune a base model
- without having to adapt many parameters

LoRA achieves this goal

- by leaving base model parameters unchanged
- adding Adapters
    - training only Adapter weights

This paper takes a different approach

- adapt a *small number* of base model parameters

Surprisingly: just fine-tuning the *bias* terms ("intercept") works pretty well !

To be specific: the bias parameters of Attention lookup layers are modified.

Recall 1

From the [Attention Lookup module (Attention_Lookup.ipynb#Projecting-queries,-keys-and-values)](Attention_Lookup.ipynb#Projecting-queries,-keys-and-values)

- Attention creates queries, keys, and values
    - based on the sequences (states) produced by earlier layers of the Transformer
- Rather than using the raw states of the Transformer as queries (resp., keys/values)
- we can map them through projection/embedding *matrices* $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V$
    - each mapping matrix shape is $(d \times d)$
    - thus, the mapping preserves the shapes of Q, K, V

- Mapping through these matrices:

out		left		right
Q	=	Q	*	$\mathbf{W}_Q$
$(T \times d)$		$(T \times d)$		$(d \times d)$

K | = | K| | $\boldsymbol{W}_K$ | V | = | V| |$\mathbf{W}_V$ | $(\bar{T} \times d)$ | | $(\bar{T} \times d)$ | | $(d \times d)$

Recall 2

Our notational practice in dealing with the "bias" term

- when computing a dot product $\mathbf{w} \cdot \mathbf{x}$ we add
    - a constant "1" as first element of $\mathbf{x}$ (let's call the augmented vector $\mathbf{x}'$)
    - the bias parameter b as the first element of $\mathbf{w}$ (let's call this $\mathbf{w}'$)

So

$$\mathbf{w} \cdot \mathbf{x} + b = \mathbf{w}' \cdot \mathbf{x}'$$

This paper

- keeps $\mathbf{w}$ frozen
- modifies b

where these terms are parts of $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V$.

On small to medium fine-tuning datasets

- performance comparable to fine-tuning *all* parameters

on large fine-tuning datasets

- performance comparable to other sparse methods

Conclusion: Fine-Tuning is easy for everyone !

Fine-Tuning a huge model like GPT-3 seemed out of the realm of possibility for individuals or small organizations.

- huge memory requirements
- time intensive
    - even with the *much smaller* number of examples in the Fine-Tuning dataset compared to the Pre-Training datasets

Parameter Efficient Transfer learning shows

- Fine-Tuning is now accessible on consumer grade hardware
- Without negligible loss of performance (maybe even better) than full Fine-Tuning

Our module on Transformer Scaling (Transformers_Scaling.ipynb)

- highlighted a trend
- to *smaller* Large Language Models
- with performance matching very large models (like GPT-3).

Combined with Parameter Efficient Fine-Tuning

- it is now possible to Fine-Tune a model (LLaMA 7B) (https://arxiv.org/pdf/2303.16199.pdf)
- with performance equivalent to GPT-3 (175B parameters)
- using 8 A100 GPU's
- in one hour !

In [2]:

```
print("Done")
```

Done