research paper explained

Here are the sections we are going to see about

Abstract

Introduction

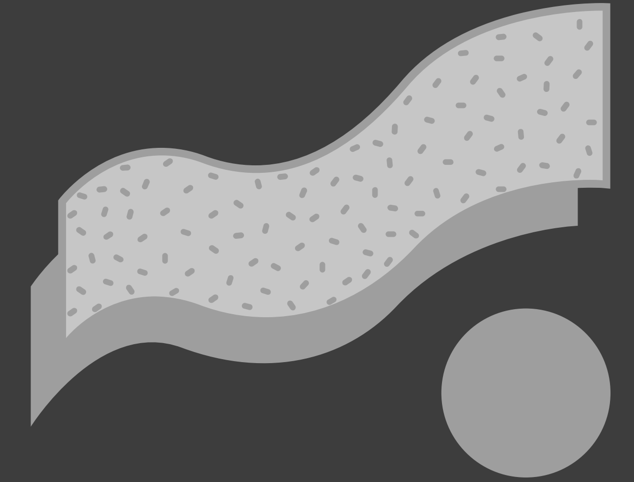
Problem Statement

Problems in Existing Solution

LoRA Method

Understanding the Low Rank Updates

ABSTRACT



With increased larger pretraining of models full finetuning requires all params to be tuned as well which becomes less feasible. For eg: GPT3(175 B)

LoRA technique proposed in this paper freezes pre-trained weights and injects a trainable rank decomposition matrix to train in reduced params for downstream tasks

No additional inference latency unlike conventional adapters, higher training throughput, and performance on par or better than a fully finetuned model are some of the key features of LoRA

INTRODUCTION

Challenges Existing: Full finetuning tunes all the parameters of a large pre-trained model making it inconvenient to train such pre-trained models

Existing Solutions - Adapters: With only a few parameters needed for new tasks it is only needed to store and load a small number of task-specific parameters for each task in addition to the pre-trained model

Problems in Existing Solution: Adapters failed to match the performance of a fully finetuned model

INTRODUCTION - KEY INSPIRATION

"Over-parameterized models, in fact, reside in a low intrinsic dimension" is a key inspiration for this paper which means that large models can learn with low dimensional inputs

INTRODUCTION - HYPOTHESIS

"Change in weights during model adaptation also has a low intrinsic rank" which states that model weight can adapt with very less linearly independent vectors

Note:
dimension - all rows
rank - linearly independent rows

So for eg: 1000 dimension can be expressed in terms of 10 linearly independent rows(rank)

INTRODUCTION

“

LoRA allows training some dense layers indirectly by optimizing rank decomposition matrices of dense layer

INTRODUCTION

Advantages of LoRA

01 Efficiently train multiple tasks by switching decomposition matrices

02 Lowers the hardware barrier by 3 times since gradient is calculated only for injected matrices

03 Simple linear design allows merging trainable weights with frozen weights which results in no additional inference latency

04 LoRA can be combined with some other previous adapter methods like previous tuning

PROBLEM STATEMENT

Given a pre-trained language model, **$P_{\Phi}(y|x)$** parameterized by **Φ** while finetuning for different tasks in context target pairs **$Z = \{(x_i, y_i)\}_{i=1,..,N}$** where x_i and y_i are a sequence of tokens based on the use case.

During full finetuning pre-trained weights are updated as follows:

$$\Phi_0 + \Delta\Phi$$

Here the main drawback is that dimension of finetuning is the same as pretraining

$$|\Delta\Phi| = |\Phi_0|$$

PROBLEM STATEMENT - SOLN.

Considering a parameter-efficient approach where

$$\Delta\Phi = \Delta\Phi(\Theta)$$

In the above equation, Θ represents the smaller set of params with,

$$|\Theta| \ll |\Phi_0|$$

With this smaller set of params, optimization is easy since the trainable params($|\Theta|$) can be as small as **0.01 % of $|\Phi_0|$**

PROBLEMS IN EXISTING SOLN

1. **Adapter Layers Inference Latency:**

Though there are different types of adapters like two or one per block experimented it wasn't possible to bypass the extra compute. Large NN expects parallelism but adapters expect sequential. In a sequential computing scenario in GPT, the latency was been able to notice

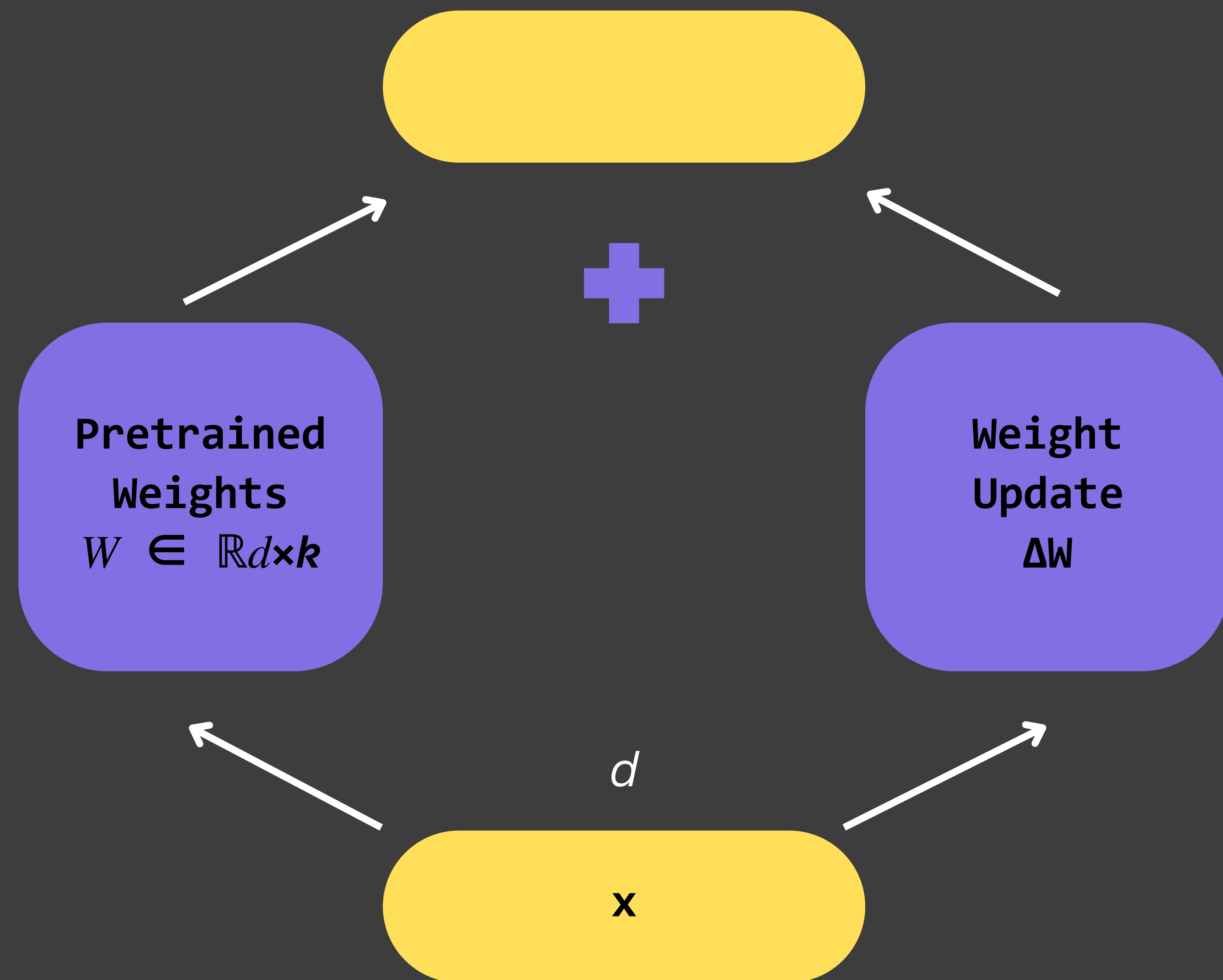
2. Directly Optimizing the Prompt is Hard:

Some of the previous adapter techniques focusing on compute aspects such as prefix tuning where it reserves part of sequence length for adaptation to reduce compute by seq length for downstream tasks. But here the performance wasn't as expected

LoRA Method

COMPARISON FULL VS LORA

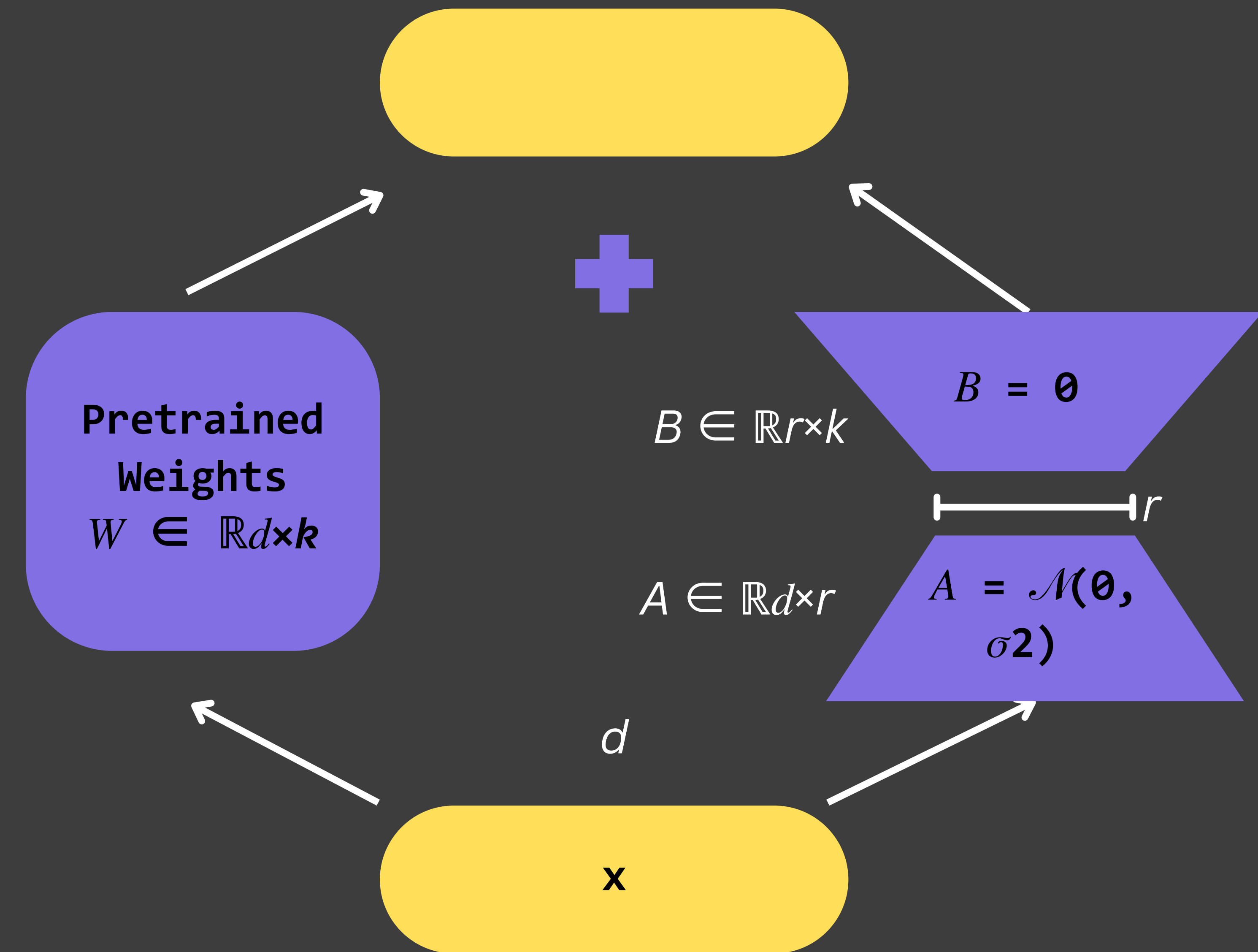
Full Fine-tuning



Wnew = Wold + ΔW

If ΔW = d x k is 100 x 500 then params in total = 100 x 500 = 50000

LoRA Fine-tuning



Wnew = Wold + WA.WB

Here with r = 4, ΔW in params in total = 100 x 4 + 4 x 500 = 2400

LoRA METHOD - UPDATION METRICS

As per the hypothesis, we can represent the weight matrix as follows

$$W_0 \in \mathbb{R}^{d \times k}$$

The weight updation can be done as follows,

$$W_{new} = W_{old} + \Delta W$$

Here from the architecture diagram of LoRA, we can see that **$\Delta W = W_A . W_B$** where,

$$A \in \mathbb{R}^{d \times r}, B \in \mathbb{R}^{r \times k}, r \text{ -> intrinsic rank}$$

During finetuning W0 is frozen so no gradient updation on the params of the pre-trained model. Only the decomposition matrices A and B contain trainable parameters that update.

LoRA METHOD - UPDATION METRICS

A forward pass for **h = W0x** can be written as follows

h0 = W0x + BAx

Initially, **ΔW = 0**, where there is weight initialization for A and B as follows

A -> Gaussian Initialization(A = $\mathcal{N}$(0, σ 2)), B -> Zero(B = 0)

Then we scale the weight updation **ΔWx = BAx** by

α/r, where **α** -> lora alpha is a constant in r

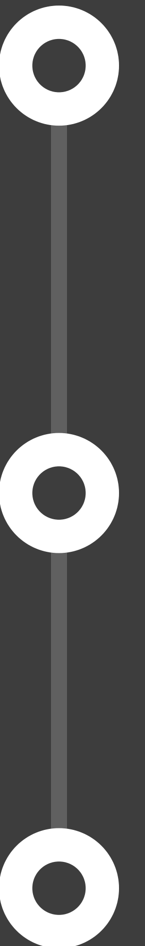
While optimizing with Adam it is almost similar to the learning rate(LR). Unlike previous adapter models which converge to an MLP and can't take long sequences here, LoRA converges to the original training model.

LoRA METHOD ON TRANSFORMER

Initially, the study is limited to adapting only attention weights for the downstream tasks freezing MLP modules for parameter efficiency and simplicity

Wq(Wk, Wv) is treated as a single matrix

Usually Wq and Wv are considered as target modules for adaptation

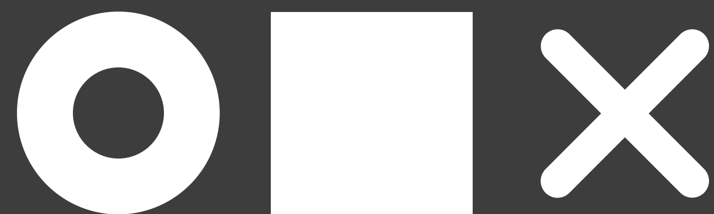


PRACTICAL BENEFITS OF LORA

Memory Reduction and Storage Usage: This is the most significant usage of LoRA where it reduced the memory consumption drastically. For eg: With r=4, target_modules = ["query_value"] the checkpoint of GPT3 was 35 MB which is 10000x lesser than the original one. The conversion came in the following way:

- 1.2TB -> 350GB since we don't store the optimizer state for frozen params
- With the above specified hyperparams, the model was reduced from 350GB -> 35 MB. Hence the adapter module size was only 35 MB

Task Switching: Another key benefit in practice of LoRA is that it is possible to swap the lora modules for specific downstream task



UNDERSTANDING THE LOW-RANK UPDATES

(i) Which matrices should we apply to?

Referring the table it is very clear that we can achieve very competitive performances by just focusing on Wq, Wv
=> Matrices to be applied are Wq, Wv

	# of Trainable Parameters = 18M						
Weight Type Rank r	W_q 8	W_k 8	W_v 8	W_o 8	W_q, W_k 4	W_q, W_v 4	W_q, W_k, W_v, W_o 2
WikiSQL ($\pm 0.5\%$)	70.4	70.0	73.0	73.2	71.4	**73.7**	**73.7**
MultiNLI ($\pm 0.1\%$)	91.0	90.8	91.0	91.3	91.3	91.3	**91.7**

UNDERSTANDING THE LOW-RANK UPDATES

(ii) What is the optimal rank for lora?

From the table of comparison, we can conclude that we can use 4 or 8.
=> r = 4 or 8 based in params budget

	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

UNDERSTANDING THE LOW-RANK UPDATES

(iii) How much does W and ΔW correlate?

	$r = 4$			$r = 64$		
	ΔW_q	W_q	Random	ΔW_q	W_q	Random
$\|\|U^\top W_q V^\top\|\|_F =$	0.32	21.67	0.02	1.90	37.71	0.33
$\|\|W_q\|\|_F = 61.95$	$\|\|\Delta W_q\|\|_F = 6.91$			$\|\|\Delta W_q\|\|_F = 3.57$		

From the table, we can infer the following,

(a) **Random << ΔWq** by correlation which shows that the ΔWq amplifies some features in W

(b) **Amplification factor: 21.5 ≈ 6.91/0.32** r = 4 has a huge amplification factor with r = 64 smaller

Thus we can conclude that W and ΔW correlate and amplifies the important features for the downstream tasks

CONCLUSIONS

LoRA is an memory and computation efficient strategy which achieves comparable or better performance than a fully finetuned model

Swappable modules, No additional inference latencies are some key features of LoRA