# LoRA Review

Paper: Low-Rank Adaptation of Large Language Models

Sangho Kim

# Contents

# Introduction

- 최근 GPT와 같은 LLM 또는 Foundation model이 등장함에 따라 task-specific model을 위한 fine-tuning 방법이 소개됨
- 이때 full fine-tuning을 수행하게 되면 무수히 많은 GPU가 필요
- 한정된 하드웨어로 fine-tuning을 원활히 수행하고 충분히 좋은 성능의 모델을 뽑아내기 위해 LoRA를 소개
- 간단히 말해, pre-trained weight는 freezing하고 trainable rank decomposition matrices를 network 중간에 삽입
- 이후 기존 모델에 비해 훨씬 더 적은 parameter를 학습
- 논문에서는 GPT-3와 같은 original model에 비해 trainable parameter가 약 10,000배 줄어든다고 소개
- 또한 추가적인 inference latency가 없다는 장점도 존재
- 다양한 LLM model에 LoRA를 적용함에도 불구하고 비슷하거나 오히려 더 좋은 성능을 달성하는 실험 결과도 존재

# Introduction

- LoRA의 구조를 간단하게 그리면 오른쪽 그림과 같음
- 즉, SVD의 개념처럼 Decomposition Low Rank Matrix를 삽입
- 이때 B는 0으로 initialization
- A는 $N(0, \sigma^2)$를 따르는 random variable로 initialization

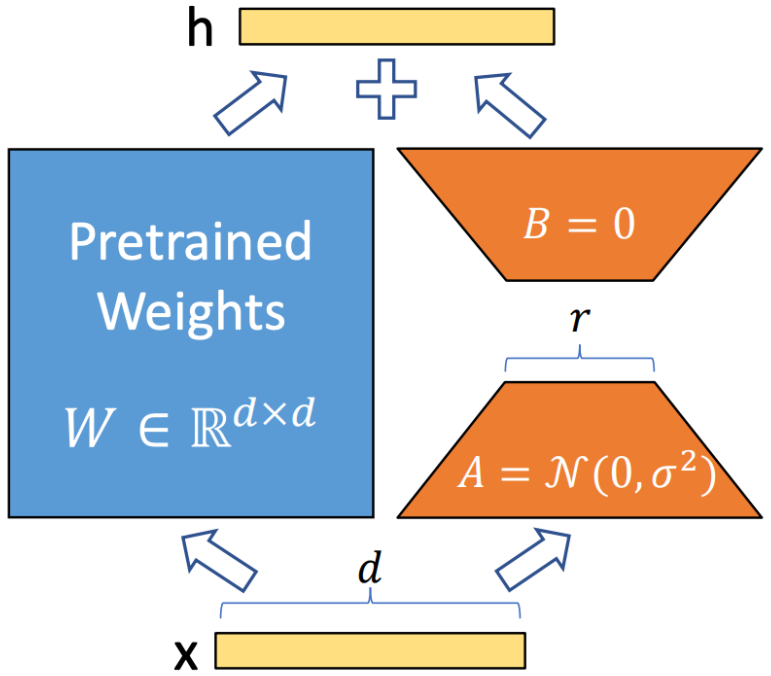



Figure 1: Our reparametrization. We only train $A$ and $B$.

# Contents

# Problem Statement

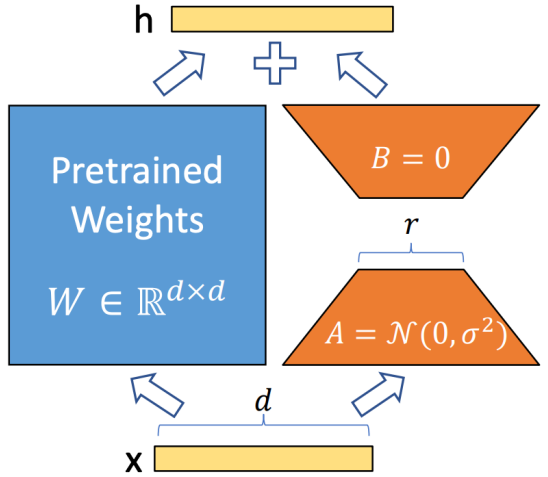
- 먼저, pre-trained weight인 $\Phi$가 존재하는 model $P_\Phi(y|x)$가 주어지며 이는 GPT, LLaMA 등과 같은 LLM으로 간주
- 이후, downstream task에 맞춰 tuning하기 위한 context-target pairs가 필요
- 이를 $Z = \{(x_i, y_i)\}_{i=1,\cdots,N}$이라 가정, 이때 $x_i, y_i$는 sequences of tokens
- 만약 full fine-tuning을 수행하게 된다면 pre-trained weight $\Phi_0$와 추가적인 weight $\Delta\Phi$를 가지고 initialization
- 즉, 식으로 표현하면 다음과 같음

$$\max_{\Phi} \sum_{(x,y)\in\mathcal{Z}} \sum_{t=1}^{|y|} \log\left(P_\Phi(y_t|x, y_{<t})\right)$$

- 그러나 LoRA는 다음의 objective를 설정 가능

$$\max_{\Theta} \sum_{(x,y)\in\mathcal{Z}} \sum_{t=1}^{|y|} \log\left(p_{\Phi_0+\Delta\Phi(\Theta)}(y_t|x, y_{<t})\right)$$

- 즉, pre-trained weight인 $\Phi_0$는 freezing, $\Delta\Phi$만 학습 (오른쪽 그림에서 주황색 부분)



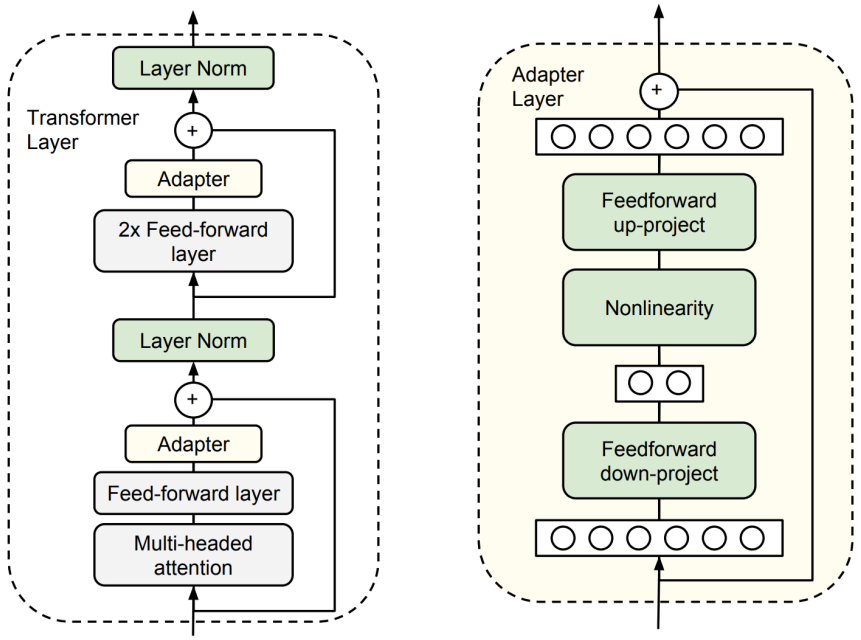
# Contents

# Existing Solutions

- 본 연구에서 해결하고자 하는 문제에 대한 기존 방법론들은 크게 다음 2가지로 나뉘어짐
  - 1. adding adapter layer
  - 2. optimizing some forms of the input layer activation

# Existing Solutions

- 이 중 1번 방법론은 다음 그림과 같음

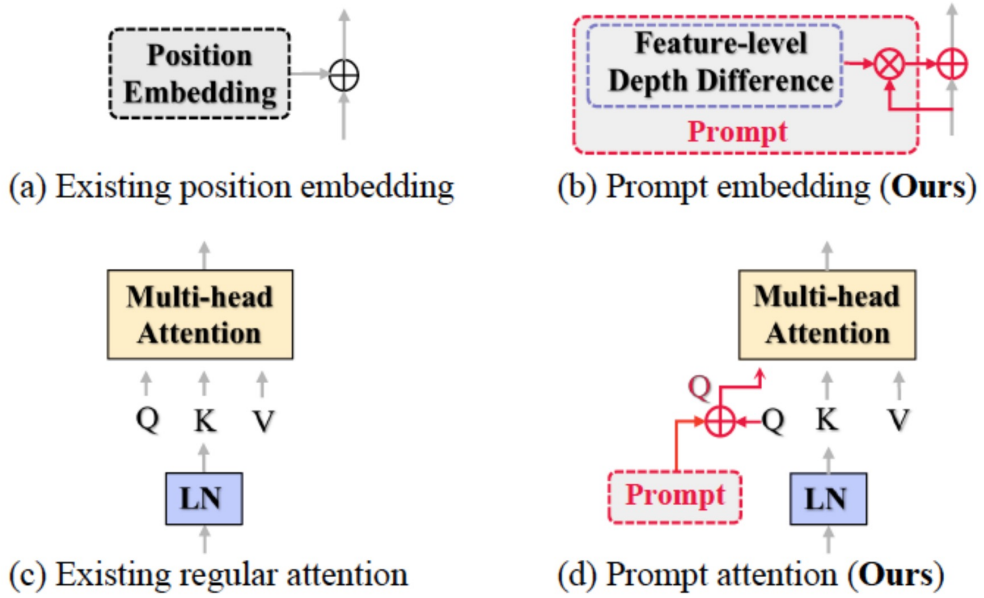



- 즉, Transformer Block 안에 작은 사이즈의 trainable module을 추가해 fine-tuning하는 방법

Source: Parameter-Efficient Transfer Learning for NLP

# Existing Solutions

- 2번 방법론은 다음 그림과 같음




- 즉, input embedding에 prompt embedding을 추가하고 이를 다양한 방법으로 학습해 fine-tuning 대체

Source: SelfPromer: Self-Prompt Dehazing Transformers with Depth-Consistency

# Existing Solutions

- 그러나 1번 방법론은 다음의 문제점을 가지고 있음
- Transformer는 Parallelism에 의존하는데, 1번 스타일의 Adapter를 적용하면 additional computing + bottleneck 발생
- 이에 따른 실험은 다음 table에서 볼 수 있음

| Batch Size<br>Sequence Length<br>$\|\Theta\|$ | 32<br>512<br>0.5M | 16<br>256<br>11M | 1<br>128<br>11M |
|---|---|---|---|
| Fine-Tune/LoRA | 1449.4±0.8 | 338.0±0.6 | 19.8±2.7 |
| $\text{Adapter}^{\text{L}}$ | 1482.0±1.0 (+2.2%) | 354.8±0.5 (+5.0%) | 23.9±2.1 (+20.7%) |
| $\text{Adapter}^{\text{H}}$ | 1492.2±1.0 (+3.0%) | 366.3±0.5 (+8.4%) | 25.8±2.2 (+30.3%) |

Table 1: Infernece latency of a single forward pass in GPT-2 medium measured in milliseconds, averaged over 100 trials. We use an NVIDIA Quadro RTX8000. "$|\Theta|$" denotes the number of trainable parameters in adapter layers. $\text{Adapter}^{\text{L}}$ and $\text{Adapter}^{\text{H}}$ are two variants of adapter tuning, which we describe in Section 5.1. The inference latency introduced by adapter layers can be significant in an online, short-sequence-length scenario. See the full study in Appendix B.

# Existing Solutions

- 2번 방법론도 문제점을 가지고 있음
- 이 방법론의 대표적인 예로 prefix tuning이 존재하는데 이는 optimize하기 어려움
- 성능 또한 trainable parameter에 따라 단조적으로 증가하지 않는다는 문제점도 존재

# Contents

# Method

- 이제 본격적으로 LoRA에 대해 자세히 알아보자
- $W_0 \in R^{d \times k}$는 neural network의 pre-trained weight (freezing parameters)
- LoRA에 해당하는 adapter는 $\Delta W = BA$이며 이때 $B \in R^{d \times r}, A \in R^{r \times k}$ (trainable parameters)
- 여기서 $r \ll \min(d, k)$은 hyperparameter로 LoRA의 rank를 결정
- $x$를 input, h를 LoRA가 탑재된 network module의 output이라고 하면 다음과 같이 쓸 수 있음

$$h = W_0 x + \Delta W x = W_0 x + BAx$$

- 여기서 $\Delta W x$는 $\frac{\alpha}{r}$ 에 의해 scaling

# Method

- Generalization of Full Fine-tuning
  - LoRA를 적용해도 full fine-tuning을 할 때만큼의 expressiveness를 가질 수 있음

- 앞에서 소개했던 기존 2가지 방법론은 문제를 야기
  - 1. Adapter-based method:
    - 추가적인 MLP layer 학습 필요
  - 2. Prefix-based method:
    - input에 prompt embedding을 추가해야 하기 때문에 input sequence를 추가하는 데 한계가 존재

- No Additional Inference Latency
  - 특정 task에 적용하기 위해 $BA$ matrix를 fine-tuning한 후 다른 task에 적용하고 싶다면 새로운 $BA$ matrix를 fine-tuning을 수행하면 된다는 장점
  - 이 때문에 no additional inference latency라고 표현한 듯 하다

# Method

- 이 논문에서는 attention module에만 LoRA를 적용
- LoRA의 가장 큰 장점은 적은 수의 trainable parameter + efficient memory storage usage
- VRAM을 최대 2/3까지 줄일 수 있으며 $r \ll d_{model}$이어야 함
- GPT-3를 예로 들자면
  - 원래 GPT-3 (175B)를 학습하는 데 필요한 VRAM은 1.2TB
  - Attention Module에 LoRA를 적용하면 350GB까지 감소 가능
  - 또한 $r = 4$로 설정하고 query, value projection matrices에만 적용한다면 35MB까지 감소
  - 이를 통해 25%의 speedup을 유도

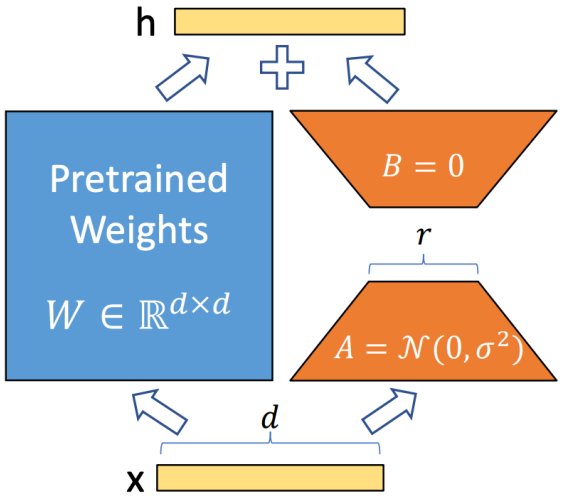
- LoRA의 한계점
  - 다른 downstream task에 적용할 때마다 $BA$를 바꿔줘야 한다는 것

# Method

- LoRA가 왜 엄청나게 효율적인 모델인지 살펴보자
- 먼저 LoRA를 식으로 표현하면 다음과 같다

$$h = W_0x + \Delta Wx = W_0x + BAx$$

- 이때 $BA$ 부분만 학습을 한 후, $(W_0+BA)x = W_{new}x$로 나타낼 수 있음
- 즉, 새로 tuning한 weight는 $W_{new}$로 표현할 수 있고 이때 모델의 parameter 수는 변함이 없음
- 또한 다양한 task에 대해 fine-tuning하여 얻은 여러 $BA$들을 가지고 적재적소에 활용 가능



# Contents

# Experimental Results

- 실험 내용들을 보기 전에 몇 가지 notation을 정리하고 가자
  - $FT$ : Fine-Tuning
  - $FT^{Top2}$ : 마지막 2개의 layer만 Tuning
  - BitFit (Bias-only) : bias vector만 training
  - Prefix-embedding tuning : input token 사이에 special token을 삽입
  - Prefix-layer tuning : 논문에서는 we learn the activations after every Transformer Layer라고 표현을 했지만 나름 내가 이해하기로는 "activation을 취한 이후의 output을 scaling한다는 것 아닐까? 추측한다"
  - Adapter tuning
    - $Adapter^{H}$: Self-attention, MLP, Resudial connection 이후에 adapter layer 탑재
    - $Adapter^{P}$: MLP, LayerNorm 이후에만 적용
    - $Adapter^{D}$: Dropout concept을 활용한 Adapter Layer

# Experimental Results

- 다양한 adaptation method를 GLUE benchmark에 실험한 결과

| Model & Method | # Trainable Parameters | MNLI | SST-2 | MRPC | CoLA | QNLI | QQP | RTE | STS-B | Avg. |
|---|---|---|---|---|---|---|---|---|---|---|
| $\text{RoB}_{\text{base}}$ (FT)* | 125.0M | **87.6** | 94.8 | 90.2 | **63.6** | 92.8 | **91.9** | 78.7 | 91.2 | 86.4 |
| $\text{RoB}_{\text{base}}$ (BitFit)* | 0.1M | 84.7 | 93.7 | **92.7** | 62.0 | 91.8 | 84.0 | 81.5 | 90.8 | 85.2 |
| $\text{RoB}_{\text{base}}$ ($\text{Adpt}^{\text{D}}$)* | 0.3M | $87.1_{\pm.0}$ | $94.2_{\pm.1}$ | $88.5_{\pm1.1}$ | $60.8_{\pm.4}$ | $93.1_{\pm.1}$ | $90.2_{\pm.0}$ | $71.5_{\pm2.7}$ | $89.7_{\pm.3}$ | 84.4 |
| $\text{RoB}_{\text{base}}$ ($\text{Adpt}^{\text{D}}$)* | 0.9M | $87.3_{\pm.1}$ | $94.7_{\pm.3}$ | $88.4_{\pm.1}$ | $62.6_{\pm.9}$ | $93.0_{\pm.2}$ | $90.6_{\pm.0}$ | $75.9_{\pm2.2}$ | $90.3_{\pm.1}$ | 85.4 |
| $\text{RoB}_{\text{base}}$ (LoRA) | 0.3M | $87.5_{\pm.3}$ | $\mathbf{95.1}_{\pm.2}$ | $89.7_{\pm.7}$ | $63.4_{\pm1.2}$ | $\mathbf{93.3}_{\pm.3}$ | $90.8_{\pm.1}$ | $\mathbf{86.6}_{\pm.7}$ | $\mathbf{91.5}_{\pm.2}$ | **87.2** |
| $\text{RoB}_{\text{large}}$ (FT)* | 355.0M | 90.2 | **96.4** | **90.9** | 68.0 | 94.7 | **92.2** | 86.6 | 92.4 | 88.9 |
| $\text{RoB}_{\text{large}}$ (LoRA) | 0.8M | $\mathbf{90.6}_{\pm.2}$ | $96.2_{\pm.5}$ | $\mathbf{90.9}_{\pm1.2}$ | $\mathbf{68.2}_{\pm1.9}$ | $\mathbf{94.9}_{\pm.3}$ | $91.6_{\pm.1}$ | $\mathbf{87.4}_{\pm2.5}$ | $\mathbf{92.6}_{\pm.2}$ | **89.0** |
| $\text{RoB}_{\text{large}}$ ($\text{Adpt}^{\text{P}}$)† | 3.0M | $90.2_{\pm.3}$ | $96.1_{\pm.3}$ | $90.2_{\pm.7}$ | $\mathbf{68.3}_{\pm1.0}$ | $\mathbf{94.8}_{\pm.2}$ | $\mathbf{91.9}_{\pm.1}$ | $83.8_{\pm2.9}$ | $92.1_{\pm.7}$ | 88.4 |
| $\text{RoB}_{\text{large}}$ ($\text{Adpt}^{\text{P}}$)† | 0.8M | $\mathbf{90.5}_{\pm.3}$ | $\mathbf{96.6}_{\pm.2}$ | $89.7_{\pm1.2}$ | $67.8_{\pm2.5}$ | $\mathbf{94.8}_{\pm.3}$ | $91.7_{\pm.2}$ | $80.1_{\pm2.9}$ | $91.9_{\pm.4}$ | 87.9 |
| $\text{RoB}_{\text{large}}$ ($\text{Adpt}^{\text{H}}$)† | 6.0M | $89.9_{\pm.5}$ | $96.2_{\pm.3}$ | $88.7_{\pm2.9}$ | $66.5_{\pm4.4}$ | $94.7_{\pm.2}$ | $92.1_{\pm.1}$ | $83.4_{\pm1.1}$ | $91.0_{\pm1.7}$ | 87.8 |
| $\text{RoB}_{\text{large}}$ ($\text{Adpt}^{\text{H}}$)† | 0.8M | $90.3_{\pm.3}$ | $96.3_{\pm.5}$ | $87.7_{\pm1.7}$ | $66.3_{\pm2.0}$ | $94.7_{\pm.2}$ | $91.5_{\pm.1}$ | $72.9_{\pm2.9}$ | $91.5_{\pm.5}$ | 86.4 |
| $\text{RoB}_{\text{large}}$ (LoRA)† | 0.8M | $\mathbf{90.6}_{\pm.2}$ | $96.2_{\pm.5}$ | $\mathbf{90.2}_{\pm1.0}$ | $68.2_{\pm1.9}$ | $\mathbf{94.8}_{\pm.3}$ | $91.6_{\pm.2}$ | $\mathbf{85.2}_{\pm1.1}$ | $\mathbf{92.3}_{\pm.5}$ | **88.6** |
| $\text{DeB}_{\text{XXL}}$ (FT)* | 1500.0M | 91.8 | **97.2** | 92.0 | 72.0 | **96.0** | 92.7 | 93.9 | 92.9 | 91.1 |
| $\text{DeB}_{\text{XXL}}$ (LoRA) | 4.7M | $\mathbf{91.9}_{\pm.2}$ | $96.9_{\pm.2}$ | $\mathbf{92.6}_{\pm.6}$ | $\mathbf{72.4}_{\pm1.1}$ | $\mathbf{96.0}_{\pm.1}$ | $\mathbf{92.9}_{\pm.1}$ | $\mathbf{94.9}_{\pm.4}$ | $\mathbf{93.0}_{\pm.2}$ | **91.3** |

- LoRA는 다른 method들과 비슷하거나 오히려 더 성능이 좋은 경우가 많음
- 여기서 주목할 만한 점은 trainable parameter의 갯수이다

# Experimental Results

- E2E NLG Challenge에서 GPT-2를 가지고 다양한 adapter를 적용한 실험

| 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 ($\text{FT}^{\text{Top2}}$)* | 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 | $\mathbf{70.4}_{\pm.1}$ | $\mathbf{8.89}_{\pm.02}$ | $\mathbf{46.8}_{\pm.2}$ | $\mathbf{72.0}_{\pm.2}$ | $2.47_{\pm.02}$ |

- 이 실험에서는 LoRA가 대부분의 실험에서 근소하게 가장 좋은 성능을 보여주고 있음
- 여기서도 역시 trainable parameter 수에 비해 좋은 성능을 뽑아내는 엄청난 효율성을 보여줌

# Experimental Results

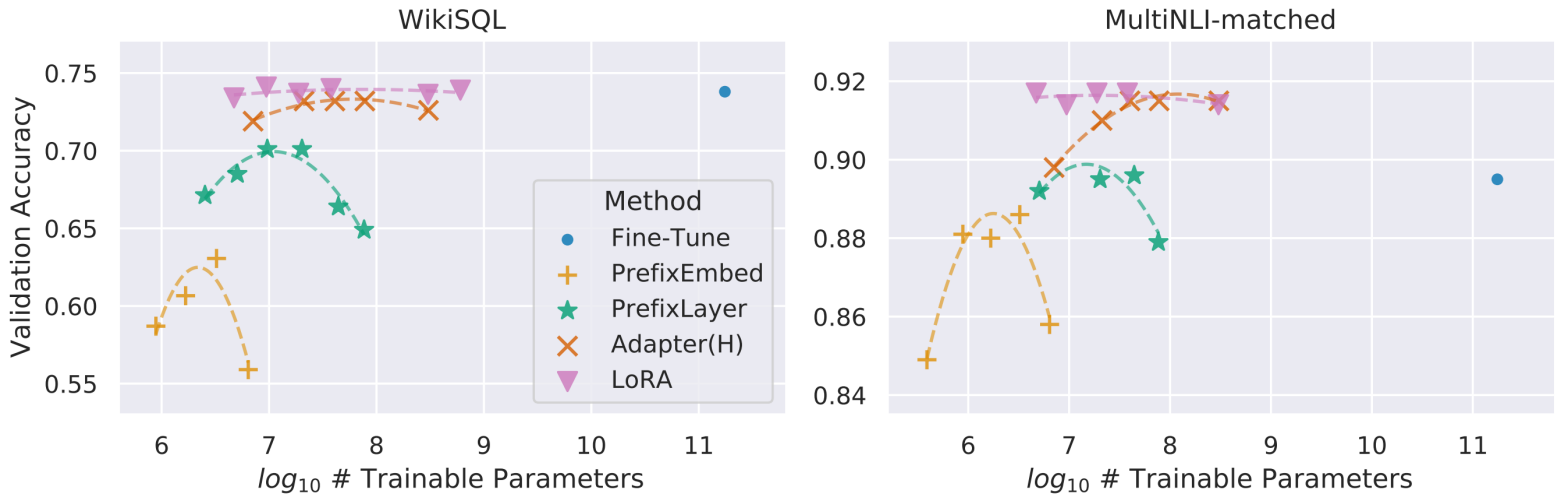
- WkikSQL에서 GPT-3를 가지고 다양한 adapter를 적용한 실험

| 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 ($\text{Adapter}^{\text{H}}$) | 7.1M | 71.9 | 89.8 | 53.0/28.9/44.8 |
| GPT-3 ($\text{Adapter}^{\text{H}}$) | 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 |

- 이 실험에서도 마찬가지로 LoRA는 다른 method와 비교해도 뒤지지 않는 성능 보유

# Experimental Results

- GPT-3에서 각 benchmark를 가지고 trainable parameter 수에 따른 성능 비교 실험




- 다른 method들은 trainable parameter의 수가 특정 임계값을 넘으면 성능이 떨어진다
- 반면 LoRA는 robust한 성능을 유지

# Contents

# Conclusion

- 넉넉하지 않은 GPU를 갖고 LLM에서 downstream task에 적용하기 위한 method인 LoRA를 소개
- 다양한 실험을 통해 LoRA의 엄청난 효율성과 성능을 입증
- 특히, GPT-3의 경우, trainable parameter의 수를 10,000배까지 감소

- 개인적 의견
  - Low Rank Decomposition이라는 간단한 아이디어로 누구나 LLM을 Fine-tuning할 수 있게 만들었단 것이 대단하다
  - 그리고 SVD라는 개념이 LLM fine-tuning method에 활용될 줄 몰랐다
  - 선형대수학을 공부한 사람이라면 누구나 아는 SVD를 이런 분야에 쓰다니 저자들의 아이디어가 정말 대단하다
  - 아마 내가 읽은 딥러닝 논문 중 다섯 손가락 안에 들 정도로 굉장한 논문이지 않을까 생각한다
  - 오랜만에 논문다운 논문을 읽은 것 같다