# LoRA - Low-Rank Adaptation of LLM 2021

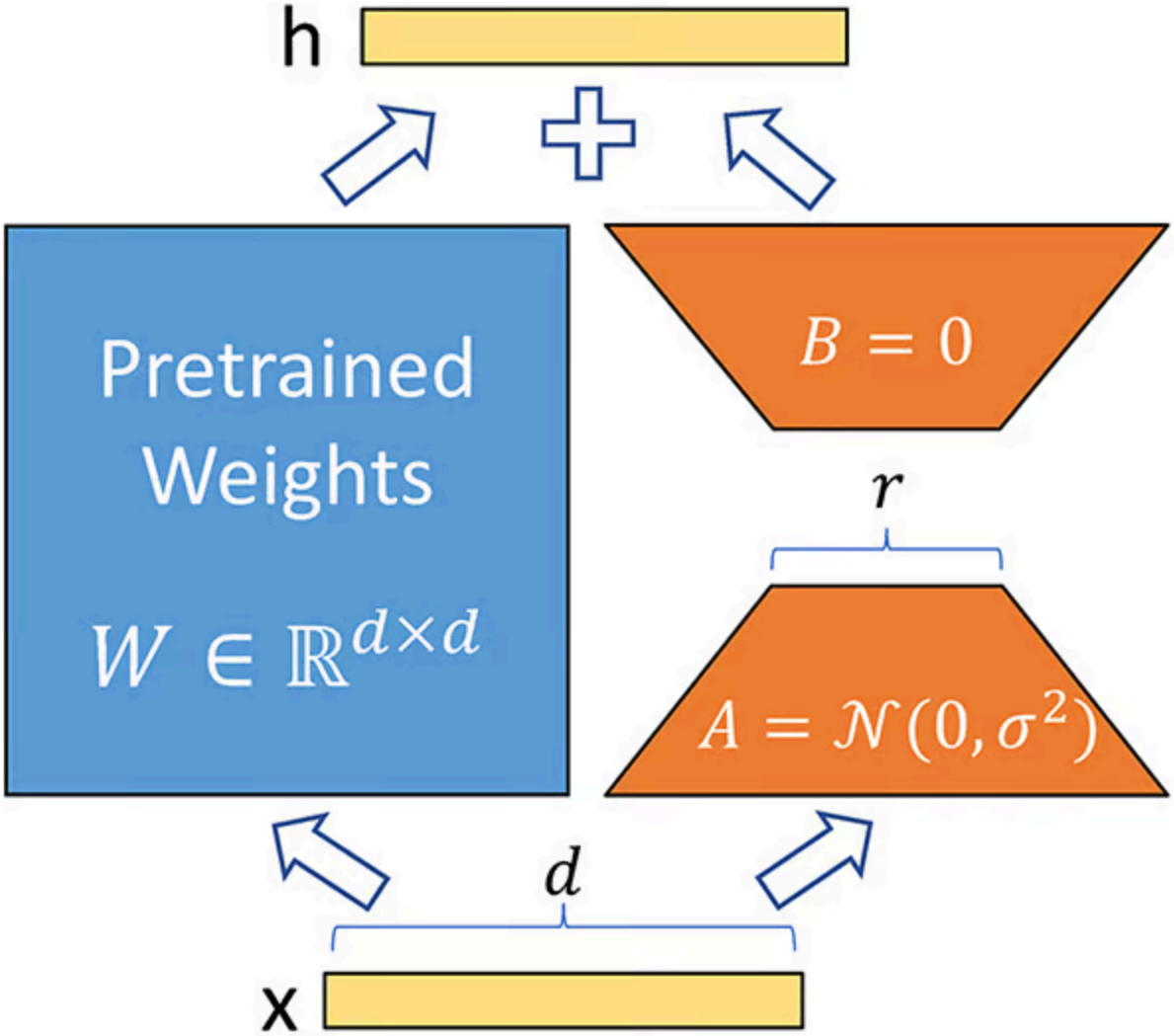
# 1. Introduction

- 🥺 문제: 기존의 대규모 모델을 **adaptation**시키기 위해 **fine-tuning**을 할 경우 기존 모델의 parameter들이 모두 다시 학습된다
- 시도: 몇몇 parameter만을 adapting 하거나 추가적인 학습 모듈을 추가하는 방법
    1. latency가 생기고 (모델 depth가 커짐) 사용 가능한 시퀀스 길이가 감소
    2. 효율성과 성능 사이에 trade-off 발생

**LoRA!** : `intrinsic rank`(**특정 작업을 처리하는데 필요한 최소한의 차원**)의 존재를 가정하고 decomposition을 이용해 최적화 진행 ➡️ 이를 통해 업데이트를 해야할 parameter의 수를 줄임

- 다양한 작업에 많고 작은 LoRA module을 적용 가능
- 효율적이고 적은 하드웨어 사용 가능 (low-rank 행렬만 업데이트)
- latency가 없음 (기존 모델 weight는 고정)

- 다양한 adaptation 방법과 같이 사용 가능




# 2. Problem Statement

언어 모델링에서 특정 작업 프롬프트가 주어졌을 때 조건부 확률을 최대화하는 것을 목표로 둔다. (maximization of conditional probability)

기존에 사전학습된 자기회귀 언어 모델을 $P_\phi(y|x)$로 둔다면 fine-tuning과 LoRA는 adaptaion에서 다음과 같은 차이를 보인다

- **Fine-Tuning의 objective:** $max_\phi \sum_{(x,y)\in Z} \sum^{|y|} \log(P_\phi(y_t|x, y_{<t}))$
  - 사전 학습된 가중치가 $\phi_0$라고 할때 $\Phi_0 + \Delta\Phi$ 로 업데이트가 됨
  - $|\Delta\Phi| = |\Phi 0|$ ➡️ 서로 다른 작업을 할 때마다, 큰 모델의 경우 비효율적
- **LoRa의 objective:**
  - $\Delta\Phi = \Delta\Phi(\Theta)$이고 $|\Theta| \ll |\Phi 0$ ➡️ 효율적인 parameter 업데이트 가능
  - GPT-3 175B 모델의 경우 0.01% 수준으로 parameter을 사용할 수 있음

# 3. Aren't Existing Solutions Good Enough?

이미 기존의 방법들로 충분하지 않을까에 대한 질문을 할 수 있다는 점에서 이 논문에서 기존 방법들의 문제점을 다음 2가지로 설명한다

1. **Inference Latency:**
   - adapter layer에 대해 추가적인 연산을 해야함
   - **adpater layer은 순차적으로 처리**(주로 FFN의 결과가 adapter layer로 입력되기 때문) ➡️ 병렬적 연산이 많은 neural network에서 latency가 생김
2. **Directly Optimizing Problem is Hard**
   - `prefix tuning`등의 방법은 최적화하기 어려움
   - 일정 부분의 시퀀스 입력을 고정적으로 차지 ➡️ downstream(다른) 작업에 사용 가능한 시퀀스 길이가 줄어듦 ➡️ 프롬프트를 바꾸더라도 성능이 낮게 나옴

# 4. Our Method

이 부분에서는 LoRA에 대한 디자인과 장점에 대해 설명을 하며 transformer에 적용하여 다른 dl 모델에 적용 가능하다는 것을 보여줌

## Low-Rank Parmeterized Update Matrics

- full rank:기존 neural network의 dense layer 가중치 행렬일 때
- **low intrinsic rank:** LoRA가 사용하는 방법으로, 더 낮은 차원으로 충분히 표현 가능하다는 전제
  - forward pass: $h = W_0x + \Delta Wx = W_0x + BAx$
  - $B \in R^{dXr}, A \in R^{rXk}, r \ll min(d, k)$ 를 통해 **저차원 행렬을 사용해 가중치 변화 가능!**

LoRA의 가중치 재구성 방법(reparmetrization)

1. 초기화: A를 랜덤 가우시안, B=0 초기화 ➡️ 초기에는 모델과 동일한 동작, 학습하면서 변화
2. scaling factor $\alpha$적용 : $\Delta W$ 너무 크면 모델 불안정 ➡️ $\frac{\alpha}{r}$ scaling 적용 ($\alpha$는 고정 상수)
   - r 값이 바뀌어도 학습 안정성 유지 가능

👍 **장점**

- **generalization:** 기존 가중치 행렬에 대해 학습 필요X ➡️ prefix 방법, full fine tuning 방법의 단점을 극복하면서 성능 유지및 상승 가능
- **no inference latency:** 매우 작은 메모리 사용으로 $BA$ 연산 가능

## Applying LoRA to Transformer

간결함과 효율적인 parameter 사용을 위해 이 논문에서는 MLP 모듈을 건드리지 않고 attention weight에 대해서만 downstream 작업에 대해 adaptation을 진행했다고 한다.

👍 **장점**

- **메모리 사용량 감소**: VRAM을 모델에 따라 2/3(transformer)에서 1/3(GPT-3 175B) 감소
  - `#VRAM` (video random access memory:gpu 연산에 최적화된 고속 메모리)
- **저장 공간 사용량 감소**: checkpoint 크기가 GPT-3 175B에서 10000배 감소
  - `#checkpoint` : 모델을 저장할 때 필요한 파일로 모델 가중치, optimizer 상태등의 정보 존재
- **적은 비용으로 다양한 작업 가능**: LoRA weight 만을 조절해 비용 낮을 추 있음 ➡️ finetuning보다 25% 속도 향상

👎 **단점**

- **다양한 작업 동시 사용**: 한번의 forward pass에서 서로 다른 작업을 처리하려면 task 마다 서로 다른 A,B를 사용해야하지만, W에 미리 합쳐서 사용하면 동시 처리하기 어려움

> ⁉️ **여러개의 작업을 한번에 처리하는 것이 어려운 이유**
>
> W+BA를 **미리 계산해서 W'W'W'로 저장**하면, 델이 기존 Transformer와 동일한 방식으로 동작해서 **추론 속도가 빠름**. 하지만 **다른 Task(작업)마다 다른 A,B를 적용해야 할 경우 문제가 생김**.
>
> - 미리 합쳐버리면 Task별로 다른 A,B를 적용 불가
> - 예제:
>   - **Task 1 (예: 감정 분석)**: W′=$W + B_1A_1W\prime = W + B1A1$
>   - **Task 2 (예: 번역)**: W′=$W + B_2A_2W\prime = W + B2A2$
> - 배치(batch) 내 샘플마다 A,B가 다르지만, 이미 WWW에 하나의 A,BA, BA,B만 합쳐졌기 때문에 **다른 Task에 대해 동시에 inference가 불가능**해짐.

# 5. Empirical Experiments

Adaptation Baseline

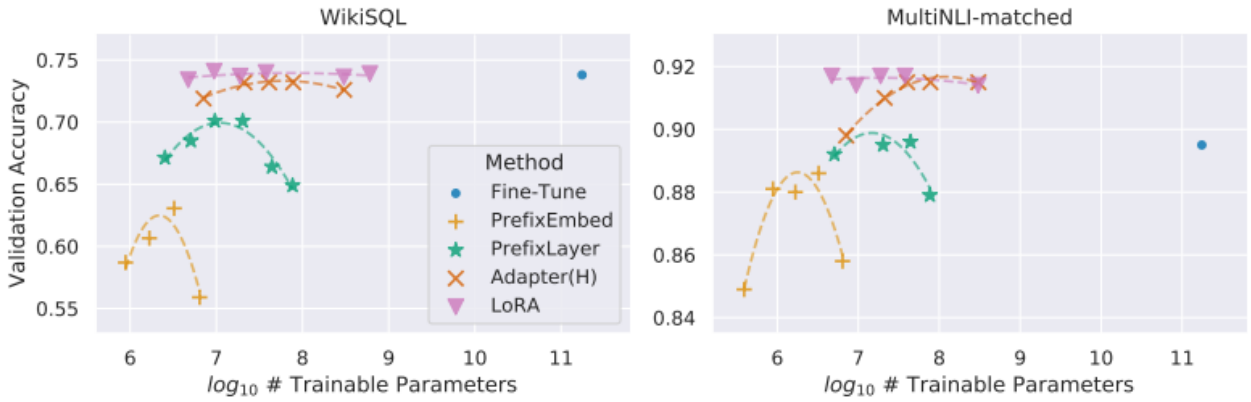
- **Fine-Tuning**: 전체 혹은 몇몇의 모델 parameter을 업데 | 트 하는 방식
- **Bias-only(BitFit)**: bias vector만 학습
- **Pre-fix-embedding tuning(PreEmbed)**: 입력 토큰에 학습 가능한 단어 임베딩인 special token을 집어넣어 특정 작업에 성능 향상을 기대
- **Prefix-layer tuning(Prelayer)**: 모든 transformer layer 이후에 학습 가능한 activation을 추가

    - trainable parameter가 성능이 올라가다가 줄어듦 ➡️ special token의 증가가 입력 분포가 기존 학습 데이터 분포로부터 멀어지게 만드는 것으로 추정
- **Adapter tuning**: 추가적인 adpater layer 사용
- ⭐ **LoRA**: 기존 가중치 행렬은 그대로 둔채 학습 가능한 rank decomposition 행렬 쌍을 추가!

**모델에 따른 결과**

- RoBERTa (NLU-understanding): 대다수의 benchmark에서 LoRA의 점수가 가장 높았으며 대체적으로 높은 점수를 유지
    - fine-tuning: parameter 400배 차이, 비슷하거나 더 높은 결과
- DeBERTa (NLU-understanding): SST-2 benchmark 제외 모든 benchmark에서 가장 좋은 성능
- GPT-2 (NLG-generation): 거의 모든 결과에서 가장 높은 성능 보여줌
    - fine-tuning: parameter 1000배 차이, 더 높은 결과
- GPT-3(175B): 3개의 데이터셋에 대해 Fine-tuning과 비슷하거나 더 나은 성능을 보임
    - fine-tuning: parameter 4000배 차이

⚠️ trainable parameter가 늘을 수록 prefix layer과 prefix embed 성능이 올라가다가 줄어드는 현상 ➡️ special token의 증가가 입력 분포가 기존 학습 데이터 분포로부터 멀어지게 만드는 것으로 추정됨




# 6. Related Works

# 7. ⭐ Understanding the Low-Rank Updates

1. Transformer의 어떤 가중치 행렬에 LoRA를 적용해야할까?

- A: **하나의 가중치 행렬에 rank를 높게 주는 것보다 여러 가중치 행렬에 rank를 작게 주는 것이 효율적** ➡️ rank가 적더라도 ΔW의 정보를 학습하기에 충분

2. 최적의 rank $r$값이 뭘까?

- A: **rank를 너무 높일 필요 없으며, r=4 정도로 충분히 좋은 성능을 보임**
  - r=1 만으로도 $W_q + W_v$ 조합에서 좋은 성능
  - rank를 크게 늘린다고 성능이 무조건 좋아지지 않음 (r=4,8일때보다 64일때 성능이 더 안좋음)

3. W와 W는 얼마나 연관이 되어 있을까

- A: **LoRA의 업데이트 행렬 ΔW는 기존 가중치 W은 중점을 두지 않는, 특정 downstream 작업에 대한 중요한 feature을 강조하여 학습**
  - ΔW는 새로운 방향 강조 ➡️ 기존 학습한 정보를 증폭하면서 특정 작업에 맞게 조정
  - rank가 너무 크면 불필요한 노이즈까지 학습

> 🔥 **rank r이 증가하면 더 많은 정보를 학습하는 것일까? (rank8 VS rank64)**
>
> 1. **SVD를 활용한 비교** #SVD
>
> - 각 rank에서 학습된 행렬의 주요 방향(principal directions) 추출 ➡️ 이를 통해 $A_{r=8}$의 상위 i개 특이 벡터가 $A_{r=64}$의 상위 j개 벡터와 얼마나 유사한지 측정
> - ⭐ $A=U\sum V^T$ ➡️ **U: 입력 데이터의 중요한 방향(principal direction)을 나타내는 벡터!** (V: 데이터 패턴 $\sum$ 특이값- 각 방향이 얼마나 중요한지)
>
> 1. **Grassmann distance 기반의 subspace similarity 측정**
>
> - Rank 8과 Rank 64의 가중치 행렬이 공유하는 부분공간의 유사도를 계산
> - 0: 완전 다른 부분 공간 1: 완전 동일한 부분공간
>
> **A**: 상위 몇 개의 특이 벡터는 Rank 8과 Rank 64에서 거의 동일한 정보를 포함 ➡️ rank를 줄여도 중요 학습 정보는 유지 & rank 너무 크면 노이즈 학습 가능성 상승

# 8. Conclusion & Future work

**결론: LoRA는 효율적인 adaptation 방법으로 inference latency와 입력 시퀀스 제한의 문제점을 해결하고 빠르게 다른 작업을 학습할 수 있으며 어떠한 nueral network에 적용 가능한 장점을 가진다**

향후 과제:

1. **다른 adaptation과 결합 가능성**: adapter, prefix-tuning, bitfit와 조합을 통해 성능 향상 가능
2. **LoRA를 통한 Fine-tuning 작동방식 이해**: fine tuning과정에서 사전학습된 feature을 어떻게 변형되는지 알 수 없는데 단순한 LoRA 방식을 통해 fine tuning을 이해할 수 있음

3. **적용 행렬**: 어떤 가중치 행렬(query,key,value)에 적용할지를 경험적 방법(heurstic)에 의존
   ➡️ 더 이론적인 방법 필요
4. **rank에 대한 연구**: ΔW뿐 아니라 W자체도 rank가 낮을 가능성이 있어 더 효율적인 adaptation 방법을 찾을 수도 있음