LoRA
Low-Rank Adaptation of Large Language Models

강동규

DeepSync, South Korea

Problem State

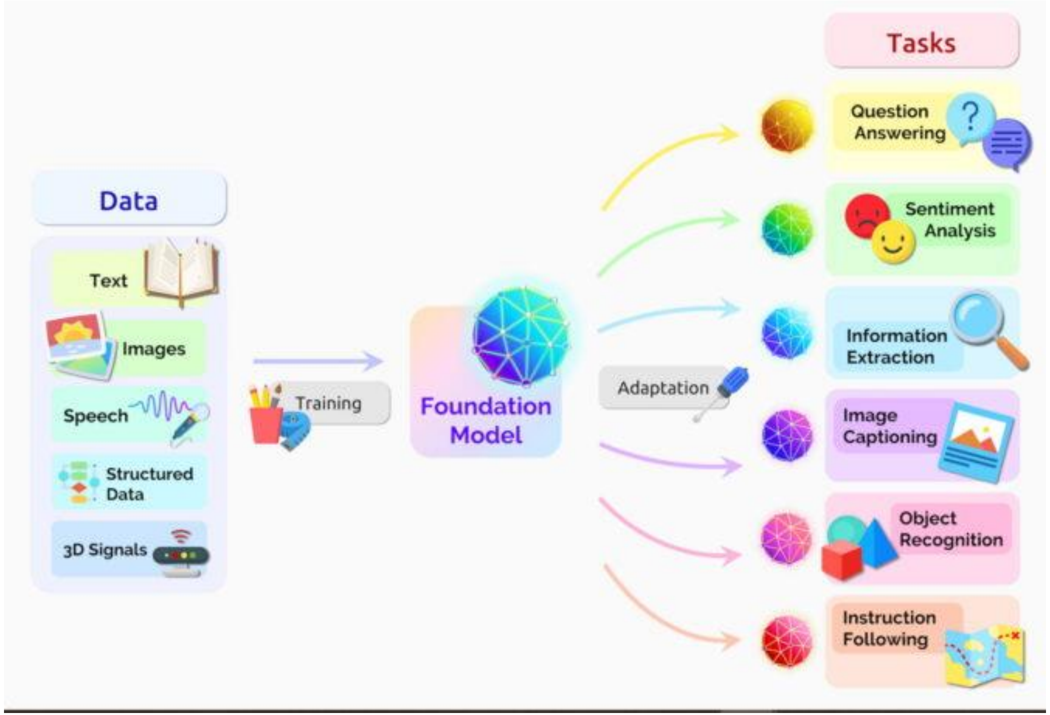

Fig1

- Downstream task에 대해 적용하는 방식
- Transfer Learning, Fine Tuning 방식 활용

Problem State

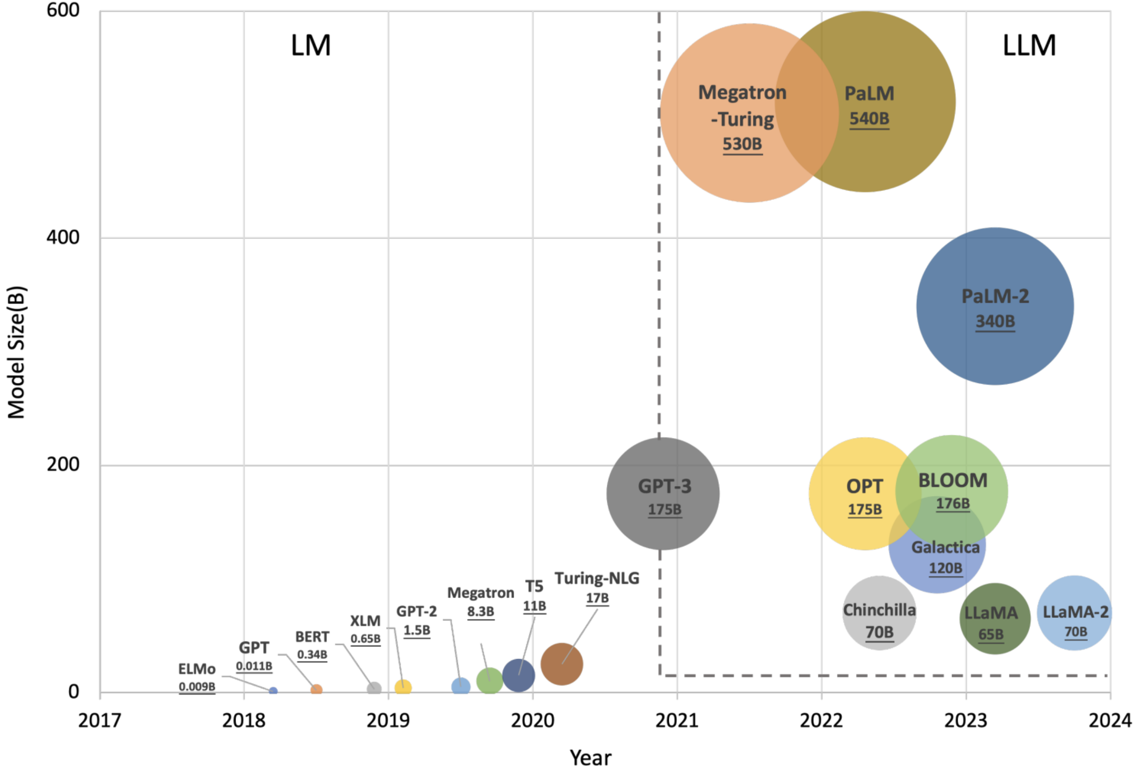

Fig2

Background

- (x, y) : 훈련 데이터. (입력, 출력)
- Φ : 전체 파라미터
- Fully Fine-Tuning : x가 주어졌을 때 y를 출력하는 모든 파라미터를 조정한다.
- Problem → 너무나 많은 시간, 비용 발생 (GPT-3의 경우 $|\Delta\Phi| \approx$ 175 Billion)

- Parameter-Efficient Approach : 전체보다 훨씬 작은 특정 파라미터 Θ를 업데이트 시킨다.
- LoRA의 핵심 → $\Delta\Phi(\Theta)$를 Pretrained 파라미터에 더해준다. ($|\Theta| \ll |\Phi_0|$, 0.01% 정도로 매우 작다)

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

Fully Fine Tuning

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

Parameter-efficient approach

Background

- Fine Tuning : Pre-Trained, Foundation 모델을 다양한 downstream task(요약, Seq2SQL 등)에 adaptation하는 방식
- 해당 논문 → Fine Tuning : 모든 파라미터를 업데이트 시키는 것
- Adaptor Tuning : 일부를 추가해 adaptor만 학습시키는 방식
- Prompting(in-context learning) : 자연어로 된 instruction과 예시를 input에 덧붙이는 방식
- Prefix Tuning : Prefix라 불리는 vector 를 input에 덧붙여 prefix만을 학습하는 방식

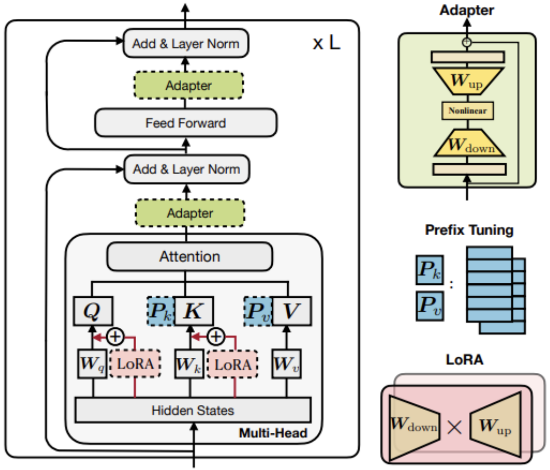

Figure 1: Illustration of the transformer architecture and several state-of-the-art parameter-efficient tuning methods. We use blocks with dashed borderlines to represent the added modules by those methods.

Fig3

Background

- Adaptor Tuning → Layer를 추가하는 방식 → Inference Latency 발생
- Large neural network → 병렬 처리를 통해 latency를 낮추지만, adapter는 순차적으로 연결되어 있어 작은 병목 현상이더라도 latency를 증가시킨다.

+ 병렬 연산 시 모델의 분할로 인해 중복된 파라미터 저장, 동기화로 인한 latency가 발생한다

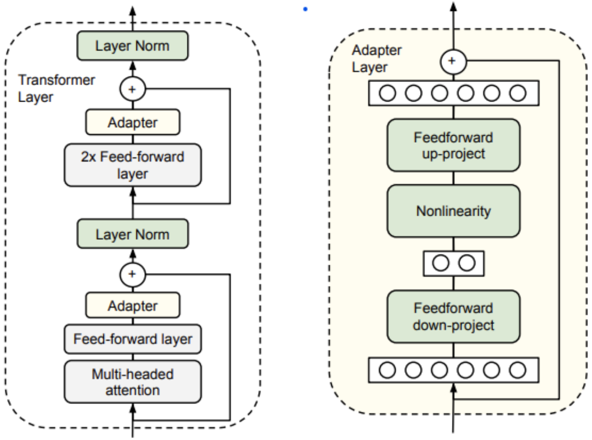

Fig4-Adaptor

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

Fig5-GPT-2 Model

Background

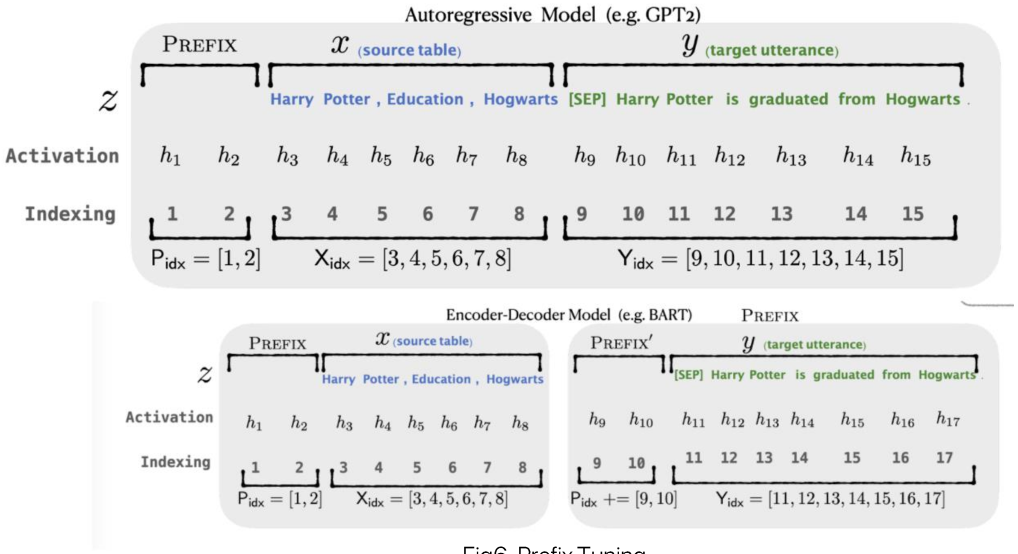

Fig6-Prefix Tuning

- Prefix Tuning
- Prefix를 input에 붙여 사용하기 때문에 sequence length에 제한이 더 발생한다.
- 최적화하기 어렵고 성능이 non-monotonically하게 파라미터가 변화한다.

Terminology

Deep Sync

- d_{model} : input/output dimension size
- W_q, W_k, W_v, W_o: query/key/value/output projection matrices in self-attention module
- W or W_0: pre-trained weight matrix
- ΔW : accumulated gradient update during adaptation
- r : rank of a LoRA module

Idea

- LoRA : Low-Rank Adaptation
- 기존 연구 → 학습된 over-parameterized model이 실제로는 낮은 low intrinsic dimension에 있다는 점에서 착안하여 low-rank matrices를 통해 adaptation을 충분히 할 수 있다!
- Measuring the Intrinsic Dimension of Objective Landscapes Li et al., 2018a
- Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning Aghajanyan et al., 2020

- LoRA 방식

1. Pre-trained weight를 고정된 상태로 유지
2. Adaptation 중 dense layer의 변화에 대한 rank decomposition matrices를 최적화

Low-Rank Parameterized Update Matrices

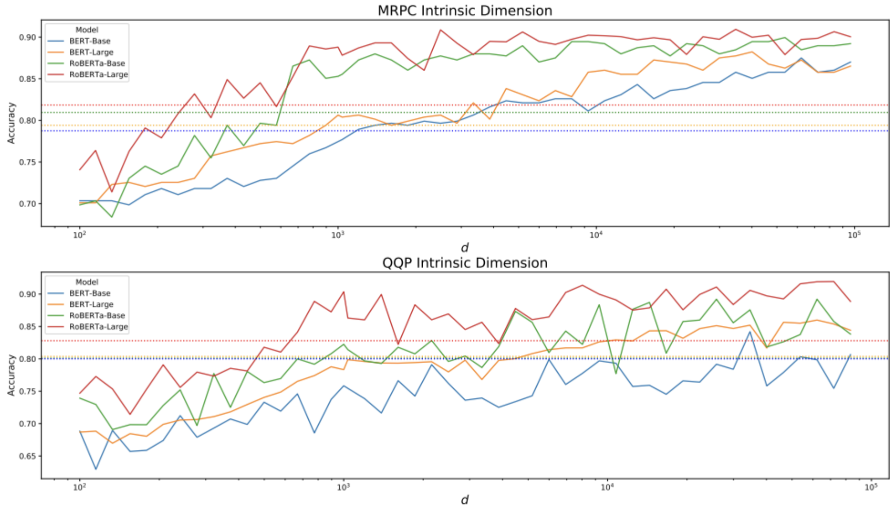

Figure 1: The following figures show the evaluation accuracy on two datasets and four models across a range of dimensions d for the DID method. The horizontal lines in each figure represent the 90% solution of the respective full model.

Fig7

- LoRA → 모든 dense layer에 적용 가능
- Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning Aghajanyan et al., 2020에서는 RoBERTa의 경우 오직 200 trainable parameter만을 가지고도 본래의 90%의 성능을 달성했다고 주장한다.

Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning

Low-Rank Parameterized Update Matrices

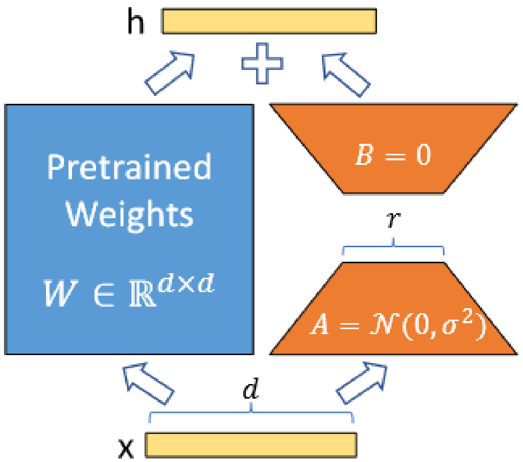

Fig8

- Adaptor와 비슷한 형태
- low rank, r로 down projection, 본래 차원으로 up projection을 한다.

$$B \in \mathbb{R}^{d \times r}, A \in \mathbb{R}^{r \times k} \quad r \ll \min(d, k)$$

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

Low-Rank Parameterized Update Matrices

Deep Sync

```python
class Linear(nn.Linear, LoRALayer):
    # LoRA implemented in a dense layer
    def __init__(
        self,
        in_features: int,
        out_features: int,
        r: int = 0,
        lora_alpha: int = 1,
        lora_dropout: float = 0.,
        fan_in_fan_out: bool = False, # Set this to True if the layer to replace stores weight like (fan_in, fan_out)
        merge_weights: bool = True,
        **kwargs
    ):
        nn.Linear.__init__(self, in_features, out_features, **kwargs)
        LoRALayer.__init__(self, r=r, lora_alpha=lora_alpha, lora_dropout=lora_dropout,
                           merge_weights=merge_weights)

        self.fan_in_fan_out = fan_in_fan_out
        # Actual trainable parameters
        if r > 0:
            self.lora_A = nn.Parameter(self.weight.new_zeros((r, in_features)))
            self.lora_B = nn.Parameter(self.weight.new_zeros((out_features, r)))
            self.scaling = self.lora_alpha / self.r
            # Freezing the pre-trained weight matrix
            self.weight.requires_grad = False
        self.reset_parameters()
        if fan_in_fan_out:
            self.weight.data = self.weight.data.transpose(0, 1)
```

```python
def reset_parameters(self):
    nn.Linear.reset_parameters(self)
    if hasattr(self, 'lora_A'):
        # initialize B the same way as the default for nn.Linear and A to zero
        # this is different than what is described in the paper but should not affect performance
        nn.init.kaiming_uniform_(self.lora_A, a=math.sqrt(5))
        nn.init.zeros_(self.lora_B)
```

- A → Random Gaussian initialization
- B → Zero initialization

```python
if r > 0:
    self.lora_A = nn.Parameter(self.weight.new_zeros((r, in_features)))
    self.lora_B = nn.Parameter(self.weight.new_zeros((out_features, r)))
    self.scaling = self.lora_alpha / self.r
    # Freezing the pre-trained weight matrix
    self.weight.requires_grad = False
```

https://github.com/microsoft/LoRA/blob/main/loralib/layers.py

Low-Rank Parameterized Update Matrices

Deep Sync

```
def forward(self, x: torch.Tensor):
    def T(w):
        return w.transpose(0, 1) if self.fan_in_fan_out else w
    if self.r > 0 and not self.merged:
        result = F.linear(x, T(self.weight), bias=self.bias)
        result += (self.lora_dropout(x) @ self.lora_A.transpose(0, 1) @ self.lora_B.transpose(0, 1)) * self.scaling
        return result
    else:
        return F.linear(x, T(self.weight), bias=self.bias)
```

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

https://github.com/microsoft/LoRA/blob/main/loralib/layers.py

Contribution

- A Generalization of Full Fine-tuning
- No Additional Inference Latency
- 명시적으로 W를 계산, 저장하고 추론을 할 수 있다.
- Downstream task의 파라미터를 언제든 더하고 뺄 수 있다. (활용성이 높다.)

```python
def train(self, mode: bool = True):
    def T(w):
        return w.transpose(0, 1) if self.fan_in_fan_out else w
    nn.Linear.train(self, mode)
    if mode:
        if self.merge_weights and self.merged:
            # Make sure that the weights are not merged
            if self.r > 0:
                self.weight.data -= T(self.lora_B @ self.lora_A) * self.scaling
            self.merged = False
    else:
        if self.merge_weights and not self.merged:
            # Merge the weights and mark it
            if self.r > 0:
                self.weight.data += T(self.lora_B @ self.lora_A) * self.scaling
            self.merged = True
```

https://github.com/microsoft/LoRA/blob/main/loralib/layers.py

Result

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

Fig9

Result

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 (Adapter^{L})*	0.37M	66.3	8.41	45.0	69.8	2.40
GPT-2 M (Adapter^{L})*	11.09M	68.9	8.71	46.1	71.3	2.47
GPT-2 M (Adapter^{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 (FT^{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	**70.4**$_{\pm.1}$	**8.85**$_{\pm.02}$	**46.8**$_{\pm.2}$	**71.8**$_{\pm.1}$	**2.53**$_{\pm.02}$
GPT-2 L (FT)*	774.03M	68.5	8.78	46.0	69.9	2.45
GPT-2 L (Adapter^{L})	0.88M	$69.1_{\pm.1}$	$8.68_{\pm.03}$	$46.3_{\pm.0}$	$71.4_{\pm.2}$	**2.49**$_{\pm.0}$
GPT-2 L (Adapter^{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	**70.4**$_{\pm.1}$	**8.89**$_{\pm.02}$	**46.8**$_{\pm.2}$	**72.0**$_{\pm.2}$	$2.47_{\pm.02}$

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

Fig10

Result

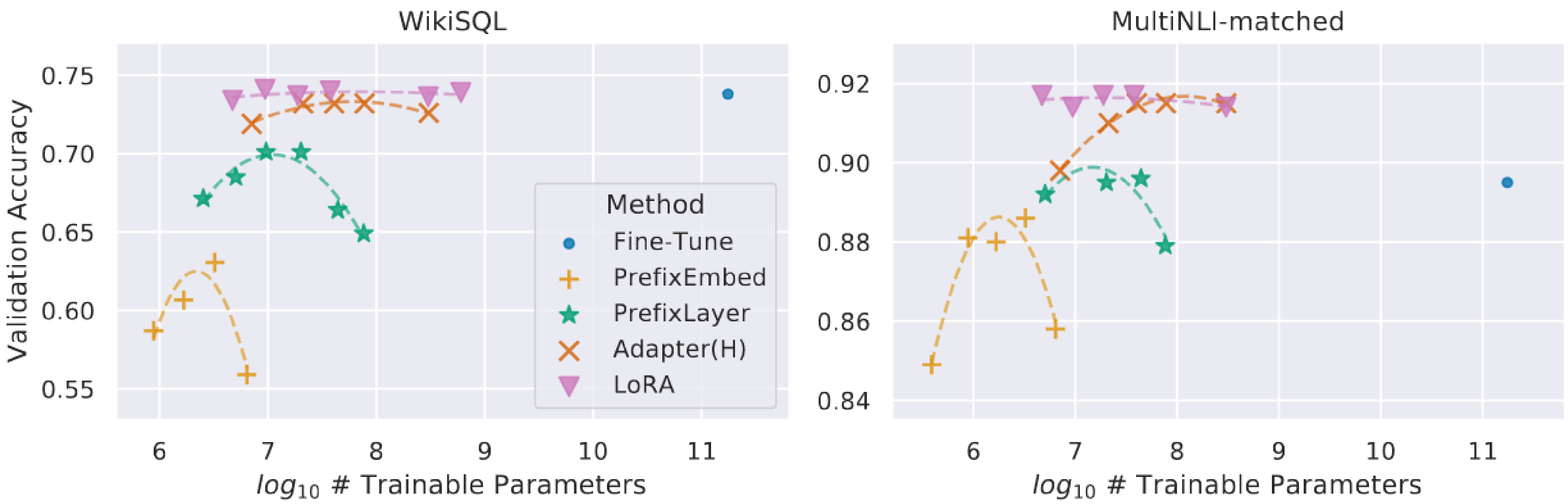

Fig11

Reference

- Fig1 - https://blogs.nvidia.co.kr/2023/04/04/what-are-foundation-models/
- Fig2 - This Artificial Intelligence Survey Research Provides A Comprehensive Overview Of Large Language Models Applied To The Healthcare Domain
- Fig3 - Towards A Unified View of Parameter-Efficient Transfer Learning
- Fig4 - Parameter-Efficient Transfer Learning for NLP
- Fig5, 8, 9, 10, 11 – LoRA: Low-Rank Adaptation of Large Language Models
- Fig6 - Prefix-Tuning: Optimizing Continuous Prompts for Generation
- Fig7 - Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning

Thank you for your time