LoRA: Low-Rank Adaptation of Large Language Models

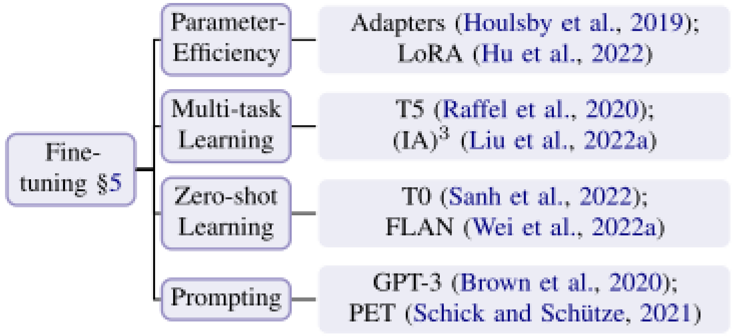
Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen
Microsoft Corporation

발제자: 윤예준

https://arxiv.org/abs/2106.09685

00. PEFT

- Parameter-Efficiency Fine-tuning
- pre-　trained된 Model을 downstream task에 적용할 때, 가중치의 일부만 업데이트하는 파인 튜닝 방법

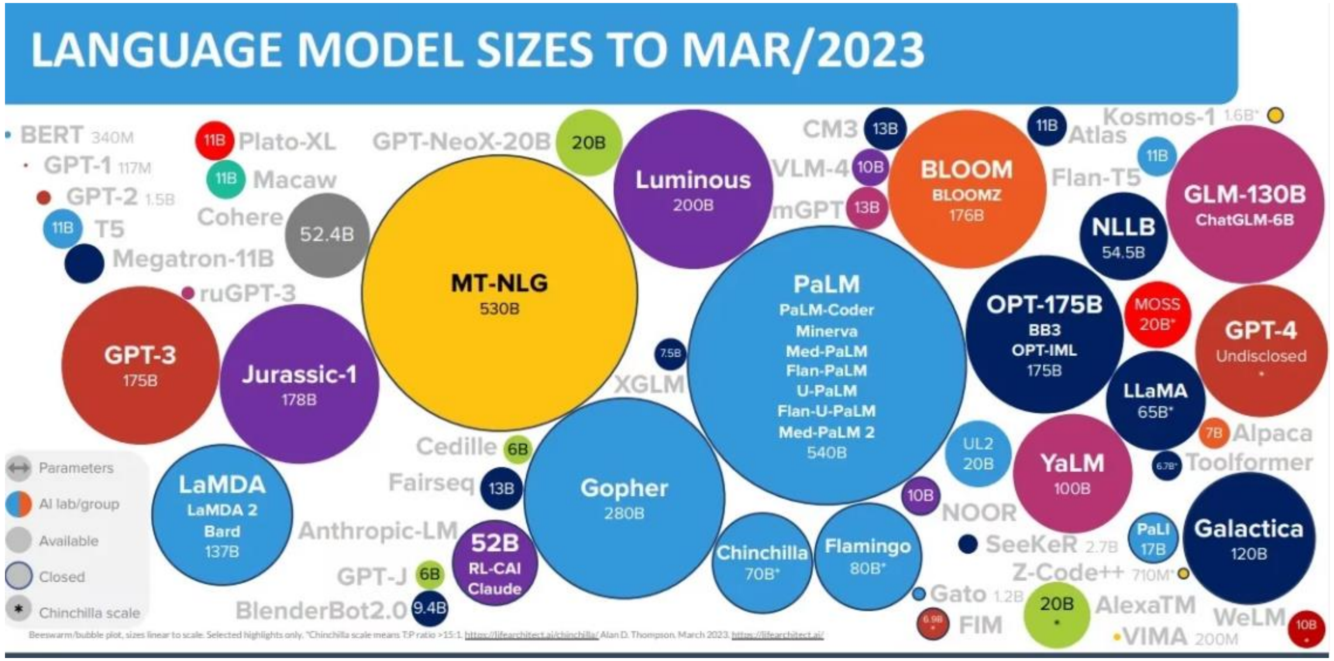


- 이전 방법
  - BitFit: bias vector만 학습
  - Adapter: 학습 가능한 adapter 추가

…

01. Introduction

- LLM의 모든 파라미터를 fine-tuning하는 것은 실현 가능성이 낮음
- 이를 해결하고자 LoRA를 제안
    - greatly reducing the number of trainable parameters for downstream tasks
    - fewer trainable parameters, a higher training throughput, and, unlike adapter, no additional inference latency에도 불구하고 파인튜닝 보다 비슷하거나 더 좋음



Inside language models (from GPT-4 to PaLM) – Dr Alan D. Thompson – Life Architect

01.Introduction

- "Measuring the Intrinsic Dimension of Objective Landscapes", "Intrinsic Dimensionality Explanins the Effectiveness of Language Model Fine-Tuning"
  ⇒ over-parameterized model은 low intrinsic dimension에 존재한다 사실에 기반

- Major downside of fine-tuning is that the new model contains as many parameters.
- Existing study limitation
  - adapting only some parameters or learning external modules for new tasks
    ⇒ introduce inference latency by extending model depth or reduce the models usable sequence length

Batch Size Sequence Length $\lvert\Theta\rvert$	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%)

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. "$\lvert\Theta\rvert$" denotes the number of trainable parameters in adapter layers. Adapter^L and Adapter^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.

01. Introduction

This work

- Propose Low-Rank Adaptation (LoRA) approach
  - inspiration from Li et al. (2018a); Aghajanyan et al. (2020)

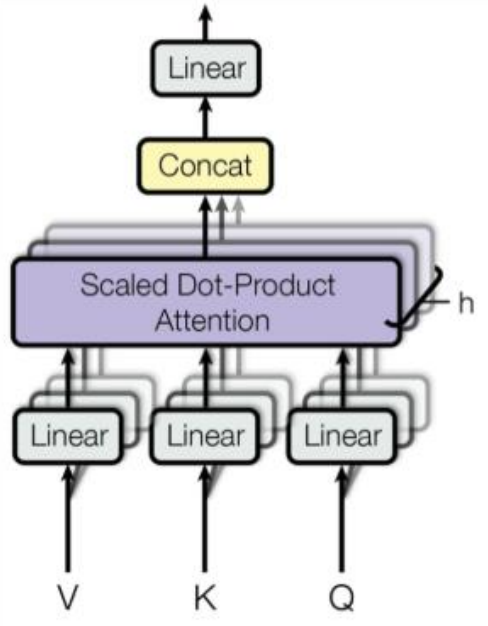
  => hypothesize that the change in weights during model adaptation also has a low "intrinsic rank"

  - 사전학습된 모델의 weight는 freeze하고 trainable rank decomposition matrices를 inject하는 것.

02. 제안 방법

단순화를 위해 self-attention에만 적용

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



02. 제안 방법

LoRA

- $h = W_0 x + \Delta W = W_0 x + BAx$
  - A: random Gaussian initialization
  - B: zero initialization
  - α is roughly the same as tuning the learning rate if we scale the initialization appropriately
  - $\Delta W = BA \left(\Delta W x \ scaling \ by \frac{\alpha}{r} \right)$ is zero at the beginning of training

- merge하면 No Additional Inference Latency
- 하지만 A, B를 merge하면 다른 여러 작업에 대한 입력을 일괄 처리하는 것은 어려움

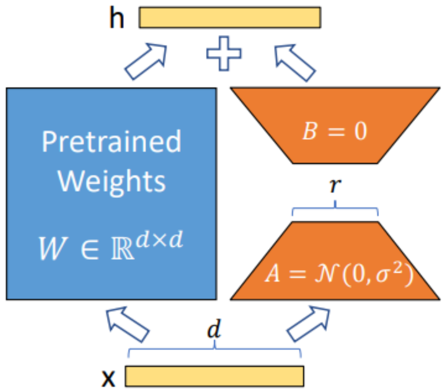


Figure 1: Our reparametrization. We only train A and B.

04. 실험 결과

Model & Method	# Trainable Parameters	MNLI	SST-2	MRPC	CoLA	QNLI	QQP	RTE	STS-B	Avg.
RoB_{base} (FT)*	125.0M	**87.6**	94.8	90.2	**63.6**	92.8	**91.9**	78.7	91.2	86.4
RoB_{base} (BitFit)*	0.1M	84.7	93.7	**92.7**	62.0	91.8	84.0	81.5	90.8	85.2
RoB_{base} (Adpt^{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_{base} (Adpt^{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_{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**
$\text{RoB}_{\text{large}}$ (Adpt^{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
$\text{RoB}_{\text{large}}$ (Adpt^{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
$\text{RoB}_{\text{large}}$ (Adpt^{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
$\text{RoB}_{\text{large}}$ (Adpt^{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**
DeB_{XXL} (FT)*	1500.0M	91.8	**97.2**	92.0	72.0	**96.0**	92.7	93.9	92.9	91.1
DeB_{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**

Table 2: $\text{RoBERTa}_{\text{base}}$, $\text{RoBERTa}_{\text{large}}$, and $\text{DeBERTa}_{\text{XXL}}$ with different adaptation methods on the GLUE benchmark. We report the overall (matched and mismatched) accuracy for MNLI, Matthew's correlation for CoLA, Pearson correlation for STS-B, and accuracy for other tasks. Higher is better for all metrics. * indicates numbers published in prior works. † indicates runs configured in a setup similar to Houlsby et al. (2019) for a fair comparison.

04. 실험 결과

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 $\pm0.5\%$, MNLI-m around $\pm0.1\%$, and SAMSum around $\pm0.2/\pm0.2/\pm0.1$ for the three metrics.

- BitFit: only train the bias vectors
- Prefix-embedding tuning(PreEmbed): inserts special tokens among the input tokens and just learning the word embeddings (or equivalently, the activations after the embedding layer)
- Prefix-layer tuning(Prelayer)
    - extension to prefix-embedding tuning
    - learn the activations after every Transformer layer
- Adapter
  inserts adapter layers between the self-attention module (and the MLP module) and the subsequent residual connection.
- LoRA
  only apply LoRA to W_q, W_v in most experiments for simplicity

04. 실험 결과

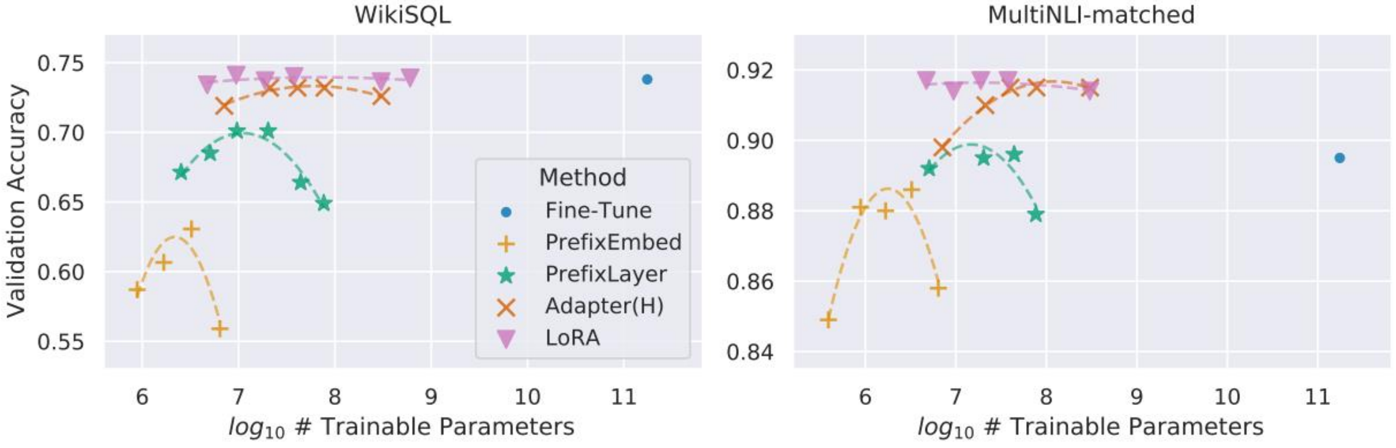


Figure 2: GPT-3 175B validation accuracy vs. number of trainable parameters of several adaptation methods on WikiSQL and MNLI-matched. LoRA exhibits better scalability and task performance. See Section F.2 for more details on the plotted data points.

04. 실험 결과

Which Weight Matrices in Transformer should we apply LoRA to?

	# 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**

Table 5: Validation accuracy on WikiSQL and MultiNLI after applying LoRA to different types of attention weights in GPT-3, given the same number of trainable parameters. Adapting both W_q and W_v gives the best performance overall. We find the standard deviation across random seeds to be consistent for a given dataset, which we report in the first column.

What is The Optimal Rank r For LoRA?

	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.

05. 코드 분석

```python
class LoRALayer():
    def __init__(
        self,
        r: int,
        lora_alpha: int,
        lora_dropout: float,
        merge_weights: bool,
    ):
        self.r = r
        self.lora_alpha = lora_alpha
        # Optional dropout
        if lora_dropout > 0.:
            self.lora_dropout = nn.Dropout(p=lora_dropout)
        else:
            self.lora_dropout = lambda x: x
        # Mark the weight as unmerged
        self.merged = False
        self.merge_weights = merge_weights
```

```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)

    def reset_parameters(self):
        nn.Linear.reset_parameters(self)
        if hasattr(self, 'lora_A'):
            # initialize A the same way as the default for nn.Linear and B to zero
            nn.init.kaiming_uniform_(self.lora_A, a=math.sqrt(5))
            nn.init.zeros_(self.lora_B)

    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

    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)
```

06. Conclusion

- Propose LoRA
  - an efficient adaptation strategy that neither introduces inference latency nor reduces input sequence length while retaining high model quality
- It allows for quick task-switching when deployed as a service by sharing the vast majority of the model parameters.
- Generally applicable to any neural networks with dense layers

Few-Shot Parameter-Efficient Fine-Tuning is Better and Cheaper than In-Context Learning

NeurIPS 2022

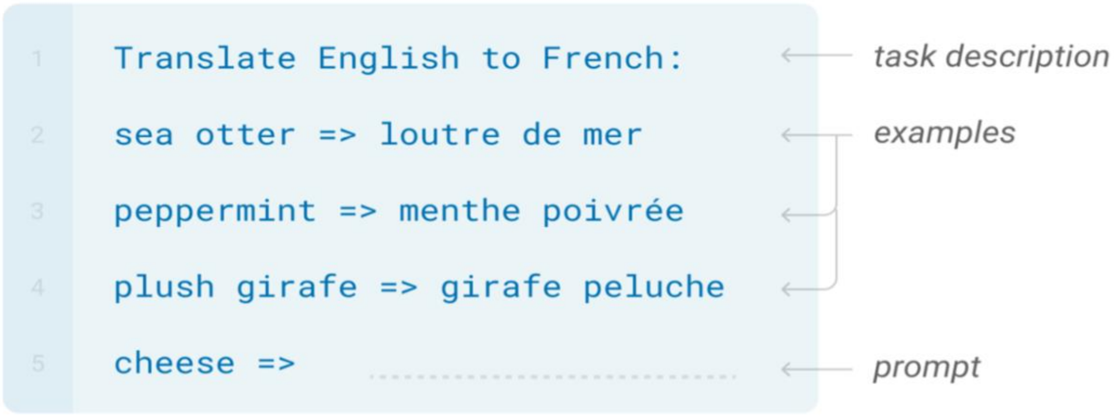
Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, Colin Raffel
Department of Computer Science University of North Carolina at Chapel Hill

발제자: 윤예준

https://arxiv.org/abs/2205.05638

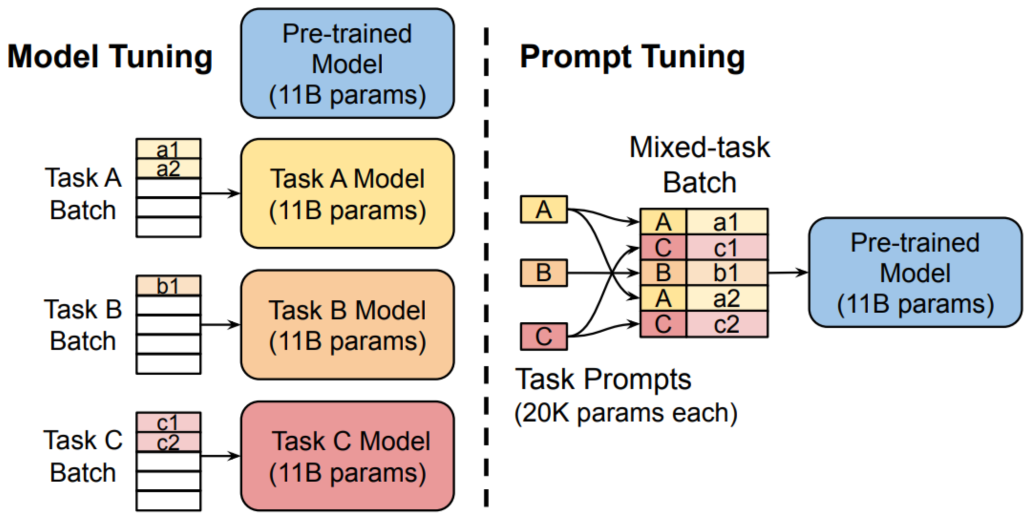
01. Introduction

- In-context Learning(ICL)의 등장
  - input-target pair에 대한 computational cost 높음
  - fine-tuning에 비해 성능이 떨어짐
  - prompt template에 따라 결과가 크게 달라짐



01. Introduction

- Parameter-Efficient Fine-tuning(PEFT)의 등장
  - 작은 수의 파라미터만을 학습하고도 fine-tuning과 비슷한 성능 => 적은 training cost
  - Prompt Tuning: mixed-task batches 가능
  - <u>하지만, very little labeled data is available 상황에서의 연구가 부족</u>



01. Introduction

Few-shot ICL에 경쟁력 있는 PEFT가 가져야할 조건

In-Context Learning
- 학습 없음
- New task 수행 가능
- Mixed-task batches 가능

PEFT
- 학습 시 적은 파라미터 사용
- New task에 대해 높은 성능
- Mixed-task batches 가능

01. Introduction

- Our primary goal in this paper is to close this gap by proposing a recipe – i.e., a model, a PEFT method, and a fixed set of hyperparameters – that attains strong performance on novel, unseen tasks while only updating a tiny fraction of the model's parameters.

- T-few recipe
  - Base Model(T0) + IA3 + Loss Function

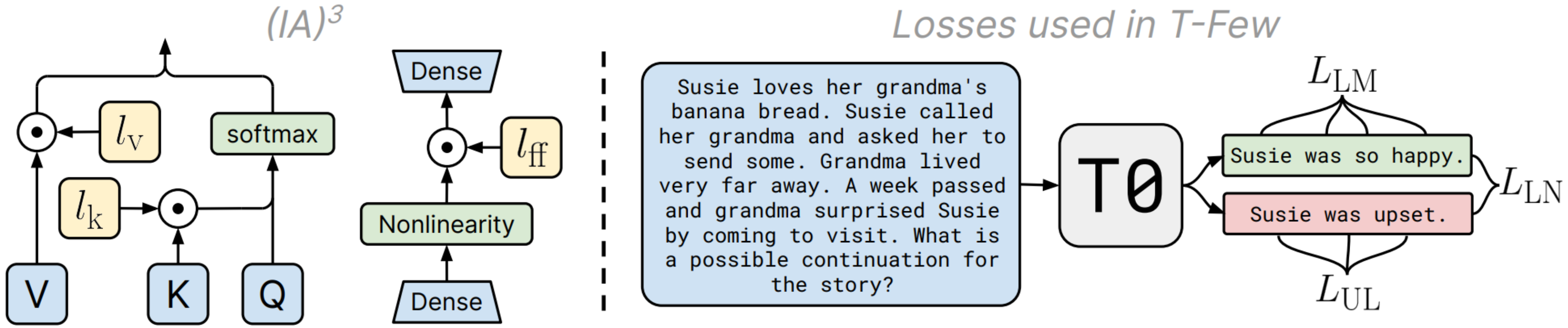
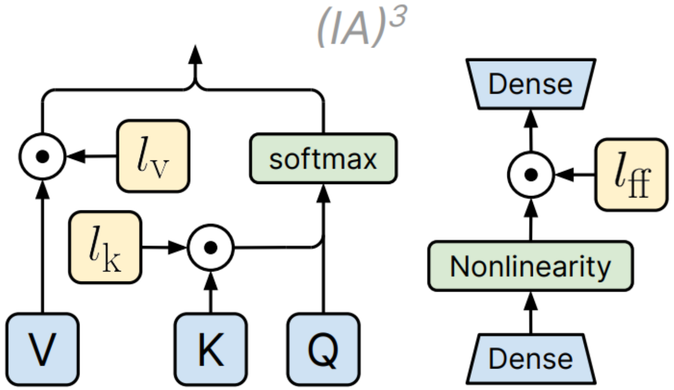


Figure 1: Diagram of (IA)^3 and the loss terms used in the T-Few recipe. *Left:* (IA)^3 introduces the learned vectors l_k, l_v, and l_{ff} which respectively rescale (via element-wise multiplication, visualized as $\odot$) the keys and values in attention mechanisms and the inner activations in position-wise feed-forward networks. *Right:* In addition to a standard cross-entropy loss L_{LM}, we introduce an unlikelihood loss L_{UL} that lowers the probability of incorrect outputs and a length-normalized loss L_{LN} that applies a standard softmax cross-entropy loss to length-normalized log-probabilities of all output choices.

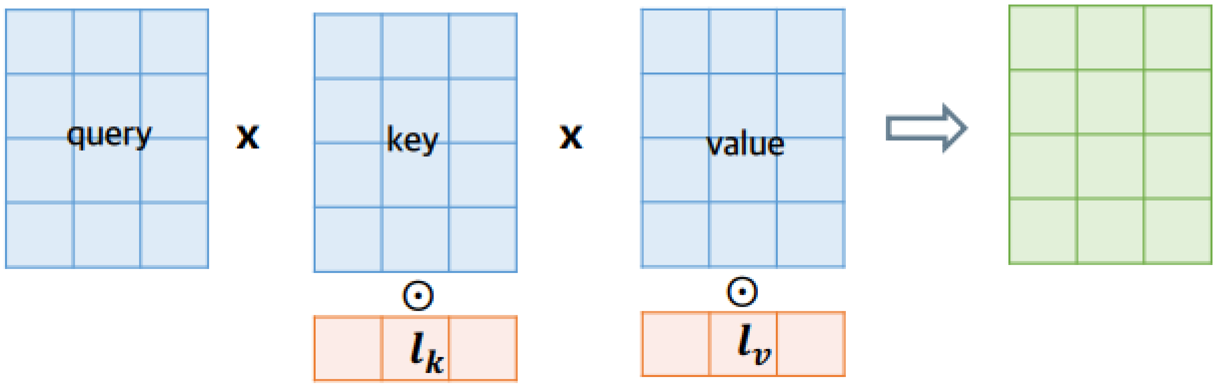
02. 제안 방법

IA^3: Infused Adapter by Inhibiting and Amplifying Inner Activations

1) <u>rescaling vectors on the keys and values in attention mechanism</u>
2) intermediate activation of the position-wise feed-forward networks

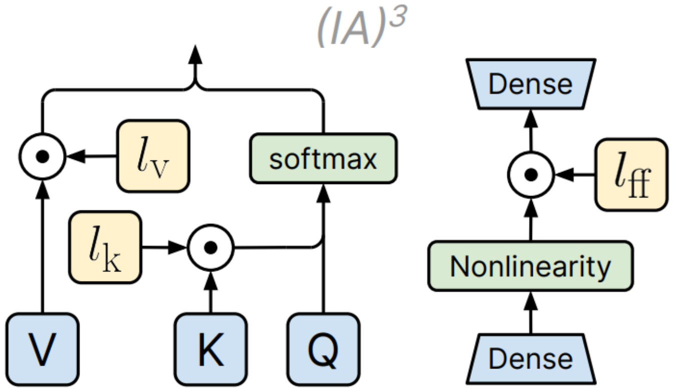


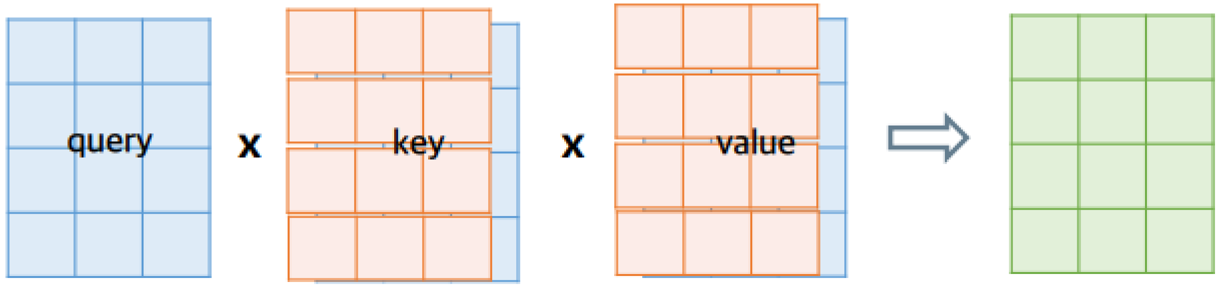
$$\mathrm{softmax}\left(\frac{Q(l_k \odot K^T)}{\sqrt{d_k}}\right)(l_v \odot V)$$



02. 제안 방법

IA^3: Infused Adapter by Inhibiting and Amplifying Inner Activations
1) rescaling vectors on the keys and values in attention mechanism
2) intermediate activation of the position-wise feed-forward networks



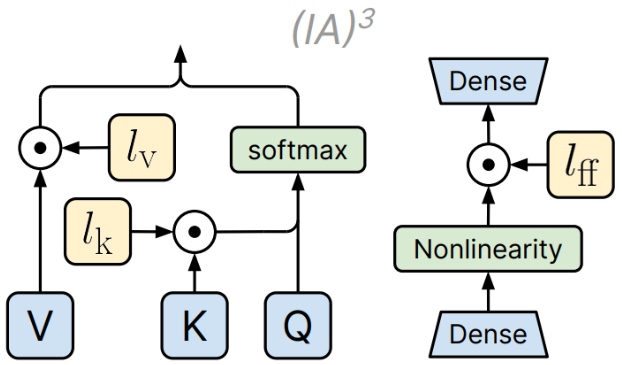


$$\mathrm{softmax}\left(\frac{Q(l_k \odot K^T)}{\sqrt{d_k}}\right)(l_v \odot V)$$

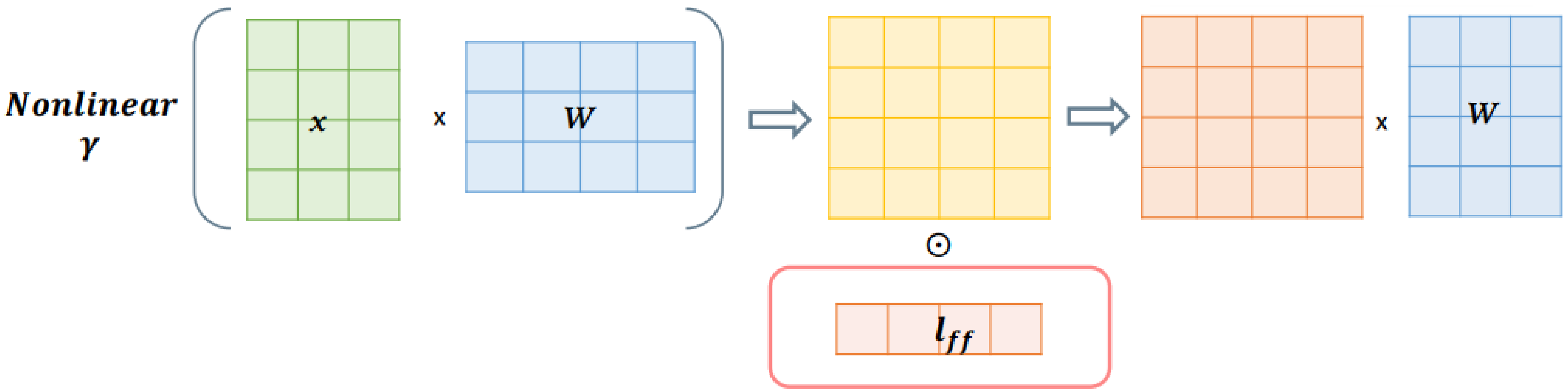
02. 제안 방법

IA^3: Infused Adapter by Inhibiting and Amplifying Inner Activations

1) rescaling vectors on the keys and values in attention mechanism
2) intermediate activation of the position-wise feed-forward networks



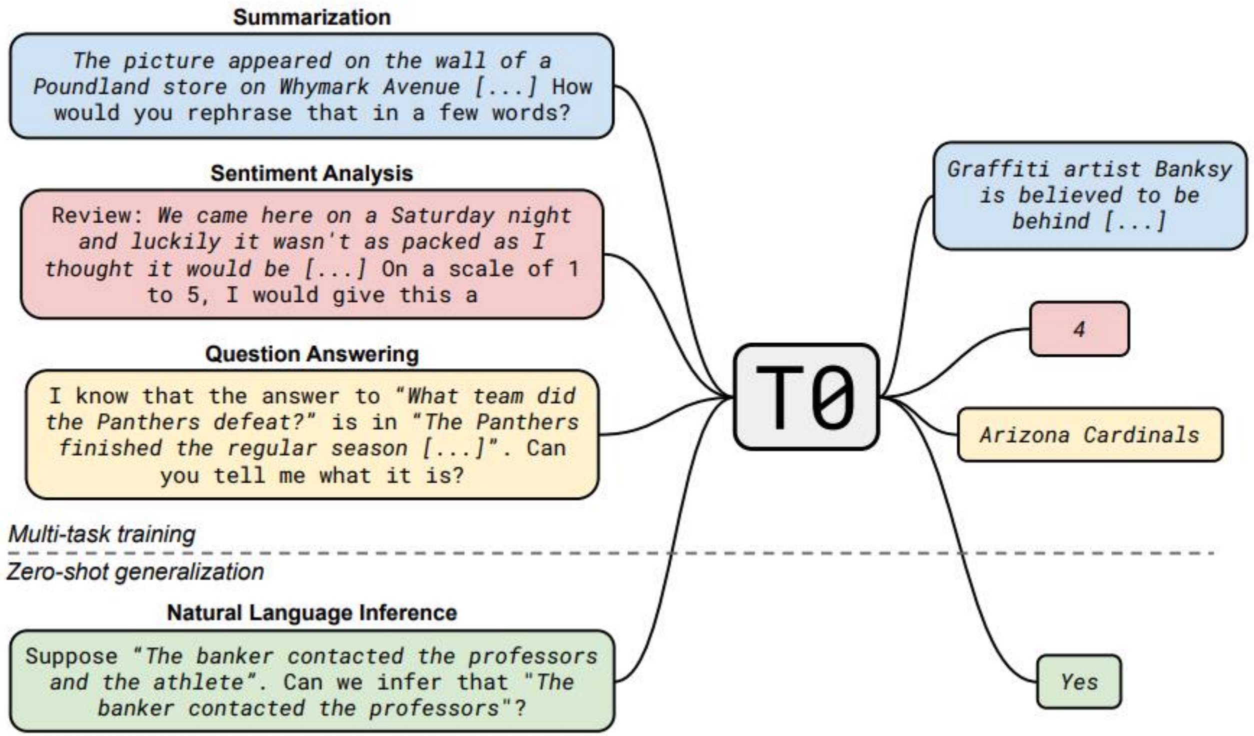
$$\mathrm{softmax}\left(\frac{Q(l_k \odot K^T)}{\sqrt{d_k}}\right)(l_v \odot V)$$



02. 제안 방법

Baseline Model: T0

pre-trained T5 + fine-tuning on multitask mixture of dataset for zero-shot generalization



02. 제안 방법

Loss Function

- Unlikelihood Training
- T0 evaluation: rank classification
  - model의 log-likelihood를 이용하여 가장 높은 확률을 가진 것이 정답일 경우 “prediction is correct”라고 함
  - 문제점: Multi choice Question Answering task 에서,
    correct choice와 in-correct choice를 모두 고려한 model update가 이루어짐
  - 효과: 원치 않은 prediction을 할 확률을 줄임. => Incorrect choice sequence에 낮은 확률을 주어, rank classification 향상

$\hat{y}^{(n)} = \left(\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_{T^{(n)}}\right)$ is the n-th of N incorrect target sequences
T^(n): 각 choice를 구성하는 토큰 수

$$L_{\mathrm{UL}} = -\frac{\sum_{n=1}^{N} \sum_{t=1}^{T^{(n)}} \log\left(1 - p\left(\hat{y}_i^{(n)} \middle| \mathbf{x}, \hat{y}_{<t}^{(n)}\right)\right)}{\sum_{n=1}^{N} T^{(n)}}$$

Neural text generation with unlikelihood training에서 처음 제안 됨

Loss Function

- Unlikelihood Training
- T0 evaluation: rank classification
  - 문제점: 짧은 길이의 answer에 더 많은 probability가 주어지는 경향이 있기 때문에, length normalization이 필요함.
- T0 논문에서,
  - "For simplicity, we do not apply length normalization to the log-likelihoods of the target options."

Q: 이번 주 일정이 어떻게 되나요?

- A1: 바빠요.(0.7)
- A2: 매일(0.2) * 오전에(0.5) * 일이(0.6) * 있어서(0.75) * 바빠요. (0.8) = 0.036
- A3: 매일(0.2) * 오전에(0.5) * 기업(0.4) * 프로젝트(0.5) * 미팅이(0.6) * 있어서(0.75) * 바빠요. (0.8) = 0.0072

$\beta(x,y) = \frac{1}{T}\sum_{t=1}^{T} \log p\,(y_t|x, y < t)$: 문장 길이로 정규화한 log-probability

$$L_{\mathrm{LN}} = -\log \frac{\exp(\beta(\mathbf{x},\mathbf{y}))}{\exp(\beta(\mathbf{x},\mathbf{y})) + \sum_{n=1}^{N} \exp(\beta(\mathbf{x},\hat{\mathbf{y}}^{(n)}))}$$

T-few recipe

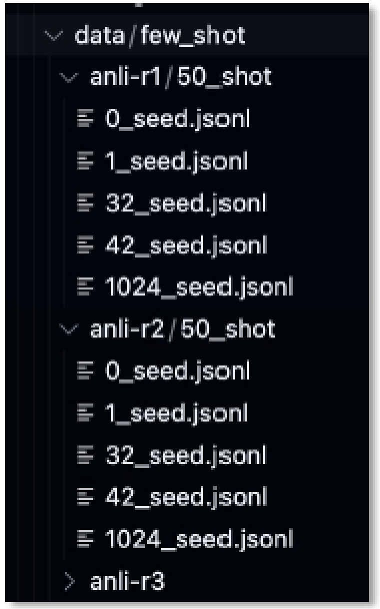
- Base: T0 + pre-training (IA)^3
- loss = standard LM loss + unlikelihood loss + length-normalized loss
- Training
  - 1000 steps, a batch size of 8
  - Adafactor optimizer
  - 3e^-3 learning rate
  - linear decay schedule with a 60-step warmup
  - Task마다 하이퍼 파라미터 튜닝 x

03. 실험 결과

held-out T0 datasets

- T0 학습 시 사용한 데이터셋
- 구성: held-out dataset
- 20~70개 examples

Task	Dataset
sentence completion	COPA, H-SWAG, Story Cloze
natural language inference	ANLI, CB, RTE
coreference resolution	WSC, Winograde
word sense disambiguation	WiC



03. 실험 결과

RAFT: Real-world Annotated Few-shot Tasks

- long input text, a public training set with 50 examples and a larger unlabeled test set
- 정답 공개 X, huggingface leaderboard를 통해 평가 가능

Dataset Name	*Long inputs*	*Domain expertise*	*Detailed instructions*	*Number of classes*	*Test set size*
ADE Corpus V2 (*ADE*)	–	✓	–	2	5,000
Banking77 (*B77*)	–	–	–	77	5,000
NeurIPS impact statement risks (*NIS*)	✓	–	–	2	150
OneStopEnglish (*OSE*)	✓	–	–	3	516
Overruling (*Over*)	–	✓	–	2	2,350
Semiconductor org types (*SOT*)	–	–	–	3	449
Systematic review inclusion (*SRI*)	✓	–	✓	2	2,243
TAI safety research (*TAI*)	✓	✓	✓	2	1,639
Terms of Service (*ToS*)	–	✓	✓	2	5,000
TweetEval Hate (*TEH*)	–	–	✓	2	2,966
Twitter complaints (*TC*)	–	–	–	2	3,399

Table 1: Overview of the tasks in RAFT. *Long inputs*, *Domain expertise*, and *Detailed instructions* are some of the real-world challenges posed by RAFT.

Rank	Submitter	Submission Name	Submission Date	Overall
1	Raldir	AuT-Few (H)	Nov 19, 2022	0.773
2	AaronLi	yiwise	Jul 08, 2022	0.768
3	jtmohta	T-Few	May 06, 2022	0.758
4	Raldir	AuT-Few	Dec 17, 2022	0.747

Method	Acc.
T-Few	75.8%
Human baseline [2]	73.5%
PET [50]	69.6%
SetFit [51]	66.9%
GPT-3 [4]	62.7%

Table 2: Top-5 best methods on RAFT as of writing. T-Few is the first method to outperform the human baseline and achieves over 6% higher accuracy than the next-best method.

04. 실험 결과

Results: held-out T0 datasets

Method	Inference FLOPs	Training FLOPs	Disk space	Acc.
T-Few	1.1e12	2.7e16	4.2 MB	72.4%
T0 [1]	1.1e12	0	0 B	66.9%
T5+LM [14]	4.5e13	0	16 kB	49.6%
GPT-3 6.7B [4]	5.4e13	0	16 kB	57.2%
GPT-3 13B [4]	1.0e14	0	16 kB	60.3%
GPT-3 175B [4]	1.4e15	0	16 kB	66.6%

Table 1: Accuracy on held-out T0 tasks and computational costs for different few-shot learning methods and models. T-Few attains the highest accuracy with 1,000× lower computational cost than ICL with GPT-3 175B. Fine-tuning with T-Few costs about as much as ICL on 20 examples with GPT-3 175B.

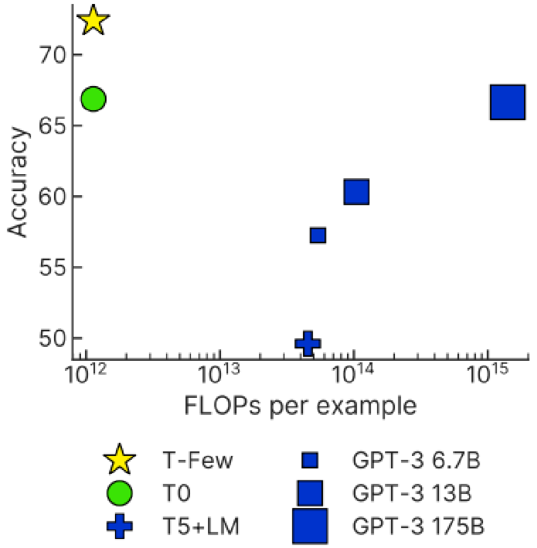


Figure 3: Accuracy of different few-shot learning methods. T-Few uses $(IA)^3$ for PEFT methods of T0, T0 uses zero-shot learning, and T5+LM and the GPT-3 variants use few-shot ICL. The x-axis corresponds to inference costs; details are provided in section 4.2.

04. 실험 결과

Results: PEFT methods

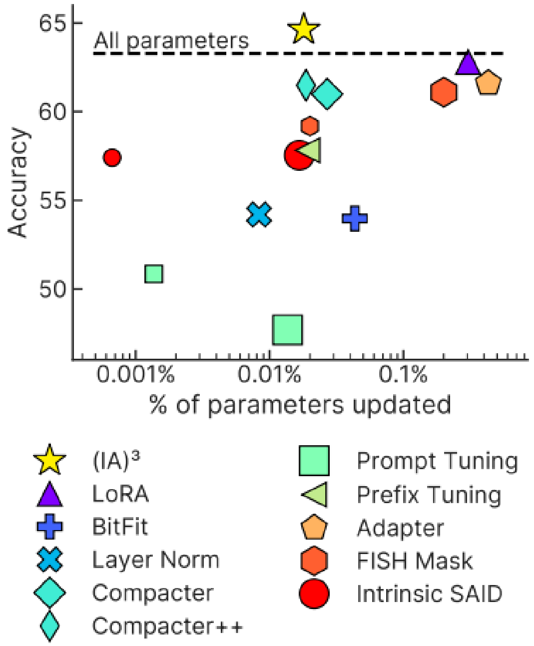


Figure 2: Accuracy of PEFT methods with L_{UL} and L_{LN} when applied to T0-3B. Methods that with variable parameter budgets are represented with larger and smaller markers for more or less parameters.

05. 코드 분석

```json
{
    "lora_scaling_rank": 1,
    "lora_rank": 0,
    "lora_init_scale": 0.0,
    "lora_modules": ".*SelfAttention|.*EncDecAttention|.*DenseReluDense",
    "lora_layers": "k|v|wi_1.*",
    "trainable_param_names": ".*lora_b.*",
    "model_modifier": "lora",
    "lr": 3e-3,
    "num_steps": 1000
}
```

```python
class LoRALinear(nn.Module):
    def __init__(self, linear_layer, rank, scaling_rank, init_scale):
        super().__init__()
        self.in_features = linear_layer.in_features
        self.out_features = linear_layer.out_features
        self.rank = rank
        self.scaling_rank = scaling_rank
        self.weight = linear_layer.weight
        self.bias = linear_layer.bias
        if self.rank > 0:
            self.lora_a = nn.Parameter(torch.randn(rank, linear_layer.in_features) * init_scale)
            if init_scale < 0:
                self.lora_b = nn.Parameter(torch.randn(linear_layer.out_features, rank) * init_scale)
            else:
                self.lora_b = nn.Parameter(torch.zeros(linear_layer.out_features, rank))
        if self.scaling_rank:
            self.multi_lora_a = nn.Parameter(
                torch.ones(self.scaling_rank, linear_layer.in_features)
                + torch.randn(self.scaling_rank, linear_layer.in_features) * init_scale
            )
            if init_scale < 0:
                self.multi_lora_b = nn.Parameter(
                    torch.ones(linear_layer.out_features, self.scaling_rank)
                    + torch.randn(linear_layer.out_features, self.scaling_rank) * init_scale
                )
            else:
                self.multi_lora_b = nn.Parameter(torch.ones(linear_layer.out_features, self.scaling_rank))

    def forward(self, input):
        if self.scaling_rank == 1 and self.rank == 0:
            # parsimonious implementation for ia3 and lora scaling
            if self.multi_lora_a.requires_grad:
                hidden = F.linear((input * self.multi_lora_a.flatten()), self.weight, self.bias)
            else:
                hidden = F.linear(input, self.weight, self.bias)
            if self.multi_lora_b.requires_grad:
                hidden = hidden * self.multi_lora_b.flatten()
            return hidden
        else:
            # general implementation for lora (adding and scaling)
            weight = self.weight
            if self.scaling_rank:
                weight = weight * torch.matmul(self.multi_lora_b, self.multi_lora_a) / self.scaling_rank
            if self.rank:
                weight = weight + torch.matmul(self.lora_b, self.lora_a) / self.rank
            return F.linear(input, weight, self.bias)

    def extra_repr(self):
        return "in_features={}, out_features={}, bias={}, rank={}, scaling_rank={}".format(
            self.in_features, self.out_features, self.bias is not None, self.rank, self.scaling_rank
        )
```

06. Conclusion

- T-few recipe
  - Base Model(T0) + IA3 + Loss Function

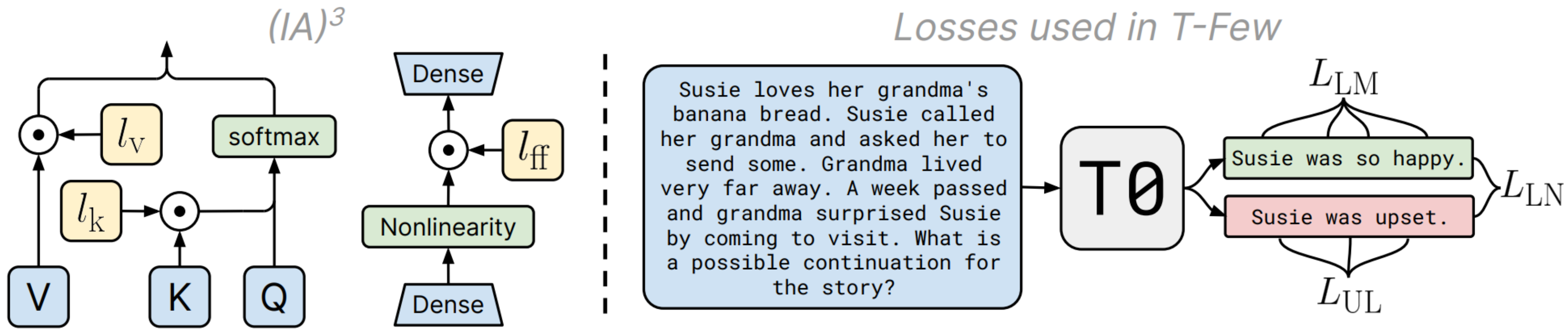


Figure 1: Diagram of (IA)^3 and the loss terms used in the T-Few recipe. *Left:* (IA)^3 introduces the learned vectors l_k, l_v, and l_{ff} which respectively rescale (via element-wise multiplication, visualized as $\odot$) the keys and values in attention mechanisms and the inner activations in position-wise feed-forward networks. *Right:* In addition to a standard cross-entropy loss L_{LM}, we introduce an unlikelihood loss L_{UL} that lowers the probability of incorrect outputs and a length-normalized loss L_{LN} that applies a standard $\mathrm{softmax}$ cross-entropy loss to length-normalized log-probabilities of all output choices.

QLoRA: Efficient Finetunin of Quantized LLMs

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, Luke Zettlemoyer
University of Washington

발제자: 윤예준

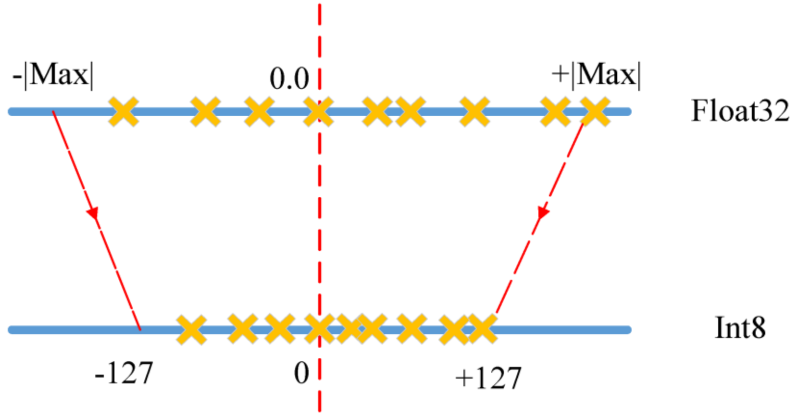
https://arxiv.org/abs/2205.05638

01. Introduction

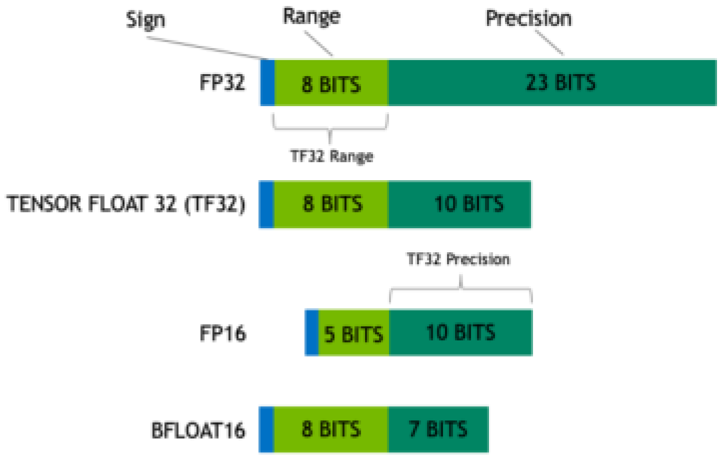
- LLM을 fine-tuning의 효과
  - 성능 개선
  - add desirable or remove undesirable behaviors

- LLM fine-tuning의 단점
  - fine-tuning하는데 매우 큰 비용 소요
  - LLaMA 65B 16비트 fine-tuning시 780GB 이상 필요

- 이를 해결하기 위한 방법: 양자화 =>기존 양자화 방법은 추론에만 동작(훈련 중 x)



8-bit Quantization



https://blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format/

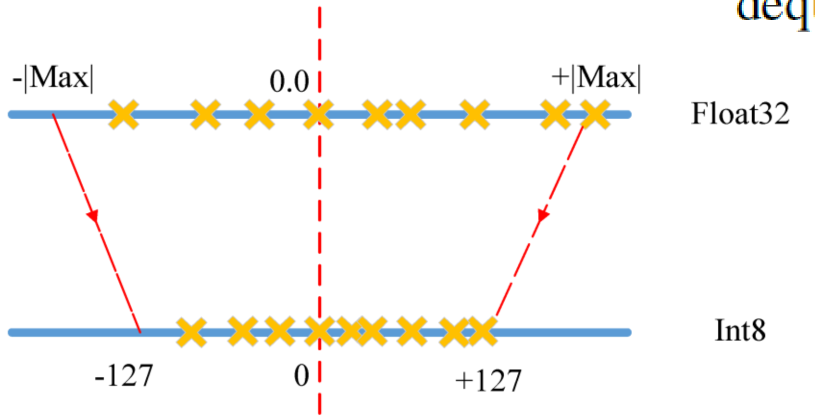
01. Introduction

Background

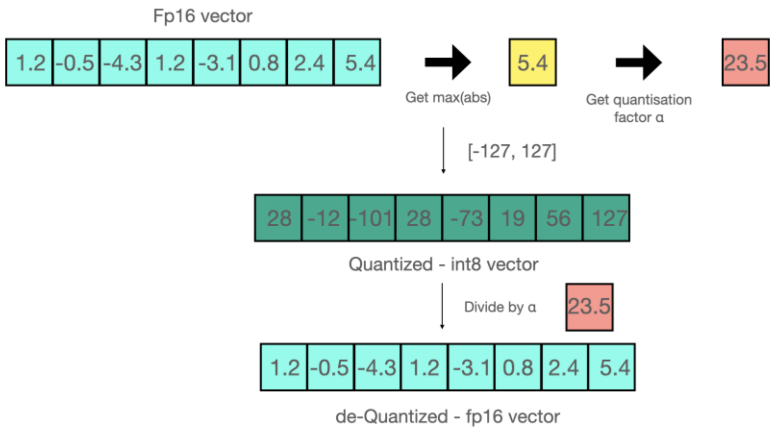
- Block-wise k-bit Quantization
  - 문제점: 입력 텐서에서 outlier 발생시 특정 bit 조합이 잘 나타나지 않는 경우 존재
  - 해결방안: 입력 텐서를 블록 n개로 chunk하여 n개의 양자화 상수 c_i생성

$$\mathbf{X}^{\text{Int8}} = \text{round}\left(\frac{127}{\text{absmax}(\mathbf{X}^{\text{FP32}})}\mathbf{X}^{\text{FP32}}\right) = \text{round}(c^{\text{FP32}} \cdot \mathbf{X}^{\text{FP32}}),$$

$$\text{dequant}(c^{\text{FP32}}, \mathbf{X}^{\text{Int8}}) = \frac{\mathbf{X}^{\text{Int8}}}{c^{\text{FP32}}} = \mathbf{X}^{\text{FP32}}$$



8-bit Quantization



01. Introduction

- This work
  - 양자화된 4비트 모델을 fine-tuning하는 방법을 제시
  - 성능 저하 없이 메모리 사용량을 줄일 수 있음을 입증

- 주요 특징
  - 4-bit NormalFloat (NF4): 정규 분포 가중치에 이론적으로 최적화된 새로운 데이터 유형
  - Double Quantization: 양자화 상수를 양자화하여 평균 메모리 사용량을 줄임
  - Paged Optimizers: 메모리 스파이크 관리하는데 사용

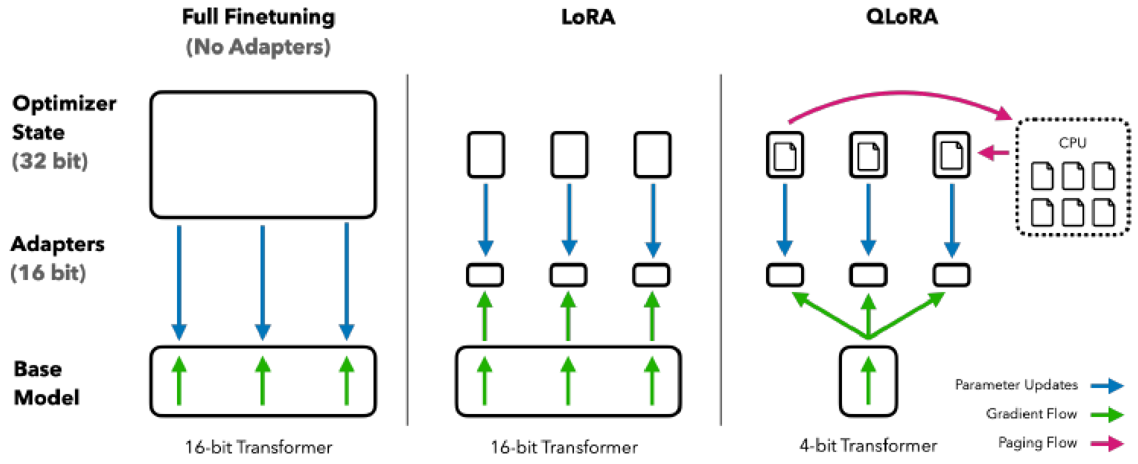


Figure 1: Different finetuning methods and their memory requirements. QLoRA improves over LoRA by quantizing the transformer model to 4-bit precision and using paged optimizers to handle memory spikes.

02. 제안 방법

4-bit NormalFloat Quantization

- NF data type builds on Quantile Quantization
- NF: Quantization bin이 input tensor로부터 할당된 값과 같도록 보장하는 정보 이론적 최적의 data type
- Quantile quantization은 input tensor의 quantile을 추정하는 경험적 누적 분포 함수로 동작
  간단히 말하면 분위수를 통한 양자화 => 분포 함수를 통해 input tensor에 대한 분위 추정하는 방식

- 문제점: 양자화 느림(분위수 추정 프로세스 비용이 많이 듦)
  - SRAM 양자화 같은 빠른 Quantile Quantization 방법 사용 가능
  - 그러나 이 또한 outlier에서 발생하는 오류를 해결하지 못함

- 해결방안:input tensor가 양자화 상수까지 고정된 분포에서 나오게 설정하면 됨
  - 사전 학습된 가중치는 일반적으로 zero-mean 정규 분포 형태를 띄므로 weights를 [-1, 1]의 범위에 맞게 표준편차를 이용하여 스케일 진행

02. 제안 방법

4-bit NormalFloat Quantization

- (1) k-bit 분위수 양자화 데이터 유형을 얻기 위해 이론적으로 N(0, 1) 분포를 띄는 $2^k + 1$ 분위수를 추정
- (2) 이 데이터 유형을 가지고 그 값을 [-1, 1] 범위로 정규화
- (3) input weight tensor를 absolute maximum rescaling를 통해 [-1, 1] 범위로 정규화하여 양자화

$$q_i = \frac{1}{2}\left(Q_X\left(\frac{i}{2^k+1}\right) + Q_X\left(\frac{i+1}{2^k+1}\right)\right),$$

- Qx is the quantile function of the standard normal distribution N(0, 1)

- 대칭적인 k-bit 양자화는 0의 정확한 표현을 갖지 않음.
  => 0의 개별 영점을 보장하고 k-bit 데이터 유형에 대한 2^k비트를 모두 사용하기 위해
  음수부분 양수부분 사분위수 추정하여 비대칭 데이터 유형을 생성한 다음 두 집합에서 발생하는 두 0 중 하나 제거

02. 제안 방법

Double Quantization

- 양자화 상수를 양자화하여 메모리를 줄이는 방법

- 32-bits의 상수와 64 blocksize를 통해 양자화 상수를 만든다고 가정하면 파라미터당 32/64 즉 0.5bits를 평균적으로 추가 사용함

- Double Quantization은 첫 번째 양자화 상수를 두 번째 양자화의 input으로 사용
- c_2^{FP32}는 c_2^{FP8}와 c_1^{FP32} 로 변함
- 수로 변환하면 8/64 + 32/(64*256) => 0.127
  (256 추가된 이유는 c1으로 한 번 더 양자화 할 때 성능 저하를 줄이기 위해 8-bits quantization에 사용된 256 block size를 사용하기 때문)
- 결과적으로 기존보다 약 0.373bit 줄일 수 있게 됨

02. 제안 방법

Paged Optimizers

- OOM이 발생할 때 CPU RAM or Disk 메모리를 사용하여 메모리 한계를 늘리는 방식
- NVIDIA 통합 메모리 기능 사용하여 CPU와 GPU간에 page 자동 전송 수행 (https://docs.nvidia.com/cuda/cuda-c-programming-guide/)

02. 제안 방법

QLoRA

- W: NF4, 64 block size
- c2: FP8, 256 block size
- parameter update는 only 어댑터에 대한 error에 대한 기울기만 업데이트

$$\mathbf{Y}^{\mathrm{BF16}} = \mathbf{X}^{\mathrm{BF16}}\mathrm{doubleDequant}(c_1^{\mathrm{FP32}}, c_2^{\mathrm{k\text{-}bit}}, \mathbf{W}^{\mathrm{NF4}}) + \mathbf{X}^{\mathrm{BF16}}\mathbf{L}_1^{\mathrm{BF16}}\mathbf{L}_2^{\mathrm{BF16}},$$

$$\mathrm{doubleDequant}(c_1^{\mathrm{FP32}}, c_2^{\mathrm{k\text{-}bit}}, \mathbf{W}^{\mathrm{k\text{-}bit}}) = \mathrm{dequant}(\mathrm{dequant}(c_1^{\mathrm{FP32}}, c_2^{\mathrm{k\text{-}bit}}), \mathbf{W}^{\mathrm{4bit}}) = \mathbf{W}^{\mathrm{BF16}},$$

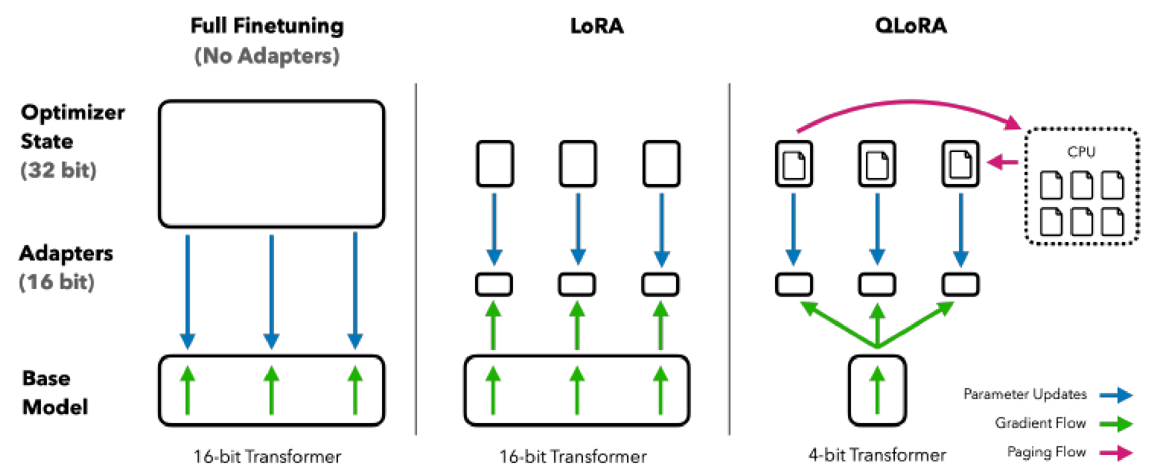


Figure 1: Different finetuning methods and their memory requirements. QLoRA improves over LoRA by quantizing the transformer model to 4-bit precision and using paged optimizers to handle memory spikes.

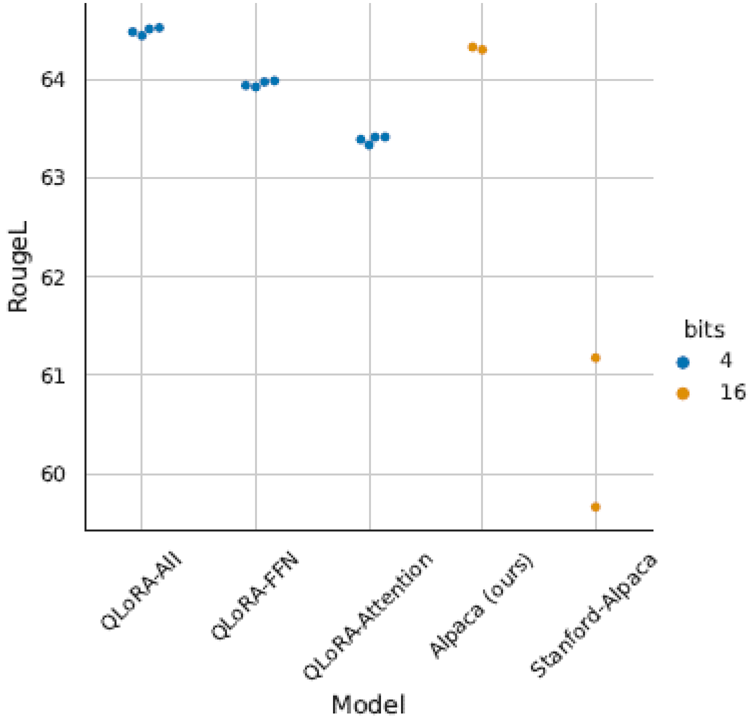


Figure 2: RougeL for LLaMA 7B models on the Alpaca dataset. Each point represents a run with a different random seed. We improve on the Stanford Alpaca fully finetuned default hyperparameters to construct a strong 16-bit baseline for comparisons. Using LoRA on all transformer layers is critical to match 16-bit performance.

04. 실험 결과

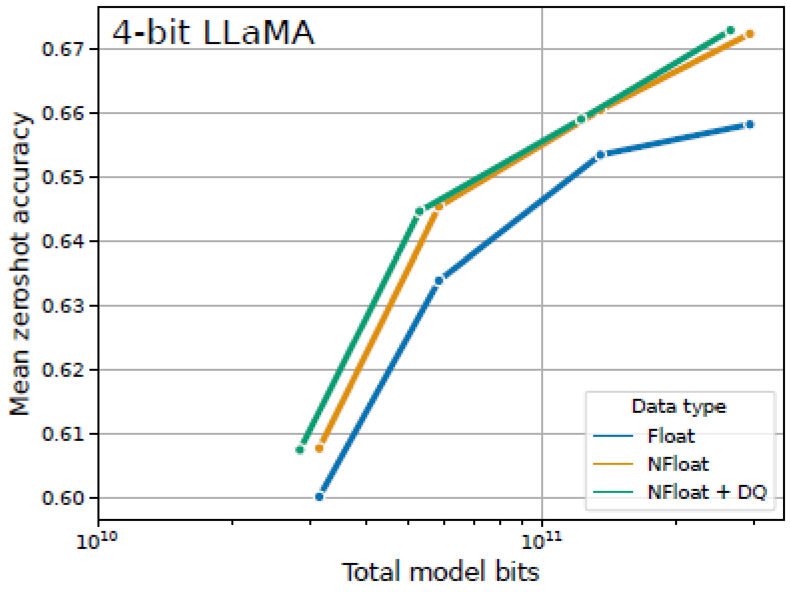


Figure 3: Mean zero-shot accuracy over Winogrande, HellaSwag, PiQA, Arc-Easy, and Arc-Challenge using LLaMA models with different 4-bit data types. The NormalFloat data type significantly improves the bit-for-bit accuracy gains compared to regular 4-bit Floats. While Double Quantization (DQ) only leads to minor gains, it allows for a more fine-grained control over the memory footprint to fit models of certain size (33B/65B) into certain GPUs (24/48GB).

04. 실험 결과

Table 3: Experiments comparing 16-bit BrainFloat (BF16), 8-bit Integer (Int8), 4-bit Float (FP4), and 4-bit NormalFloat (NF4) on GLUE and Super-NaturalInstructions. QLoRA replicates 16-bit LoRA and full-finetuning.

Dataset	GLUE (Acc.)	Super-NaturalInstructions (RougeL)				
Model	RoBERTa-large	T5-80M	T5-250M	T5-780M	T5-3B	T5-11B
BF16	88.6	40.1	42.1	48.0	54.3	62.0
BF16 replication	88.6	40.0	42.2	47.3	54.9	-
LoRA BF16	88.8	40.5	42.6	47.1	55.4	60.7
QLoRA Int8	88.8	40.4	42.9	45.4	56.5	60.7
QLoRA FP4	88.6	40.3	42.4	47.5	55.6	60.9
QLoRA NF4 + DQ	-	40.4	42.7	47.7	55.3	60.9

05. 코드 분석

```python
compute_dtype = (torch.float16 if args.fp16 else (torch.bfloat16 if args.bf16 else torch.float32))
model = AutoModelForCausalLM.from_pretrained(
    args.model_name_or_path,
    cache_dir=args.cache_dir,
    load_in_4bit=args.bits == 4,
    load_in_8bit=args.bits == 8,
    device_map=device_map,
    max_memory=max_memory,
    quantization_config=BitsAndBytesConfig(
        load_in_4bit=args.bits == 4,
        load_in_8bit=args.bits == 8,
        llm_int8_threshold=6.0,
        llm_int8_has_fp16_weight=False,
        bnb_4bit_compute_dtype=compute_dtype,
        bnb_4bit_use_double_quant=args.double_quant,
        bnb_4bit_quant_type=args.quant_type,
    ),
    torch_dtype=(torch.float32 if args.fp16 else (torch.bfloat16 if args.bf16 else torch.float32)),
    trust_remote_code=args.trust_remote_code,
    use_auth_token=args.use_auth_token
)
```

```python
if not args.full_finetune:
    model = prepare_model_for_kbit_training(model, use_gradient_checkpointing=args.gradient_checkpointing)

if not args.full_finetune:
    if checkpoint_dir is not None:
        print("Loading adapters from checkpoint.")
        model = PeftModel.from_pretrained(model, join(checkpoint_dir, 'adapter_model'), is_trainable=True)
    else:
        print(f'adding LoRA modules...')
        modules = find_all_linear_names(args, model)
        config = LoraConfig(
            r=args.lora_r,
            lora_alpha=args.lora_alpha,
            target_modules=modules,
            lora_dropout=args.lora_dropout,
            bias="none",
            task_type="CAUSAL_LM",
        )
        model = get_peft_model(model, config)

for name, module in model.named_modules():
    if isinstance(module, LoraLayer):
        if args.bf16:
            module = module.to(torch.bfloat16)
    if 'norm' in name:
        module = module.to(torch.float32)
    if 'lm_head' in name or 'embed_tokens' in name:
        if hasattr(module, 'weight'):
            if args.bf16 and module.weight.dtype == torch.float32:
                module = module.to(torch.bfloat16)
```

06. Conclusion

- QLoRA
  - 4-bit NormalFloat (NF4)
  - Double Quantization
  - Paged Optimizers

- 양자화된 4비트 모델을 fine-tuning하는 방법을 제시
- 65B Model을 48GB GPU로 학습할 수 있음을 보임

00. Open Questions

- task에 적합한 학습을 위해 PEFT 방법들을 어떻게 사용할 수 있을까?

00. 참고문헌

http://dsba.korea.ac.kr/seminar/?mod=document&uid=2733
https://velog.io/@nellcome/QLoRA%EB%9E%80

감사합니다.