LoRA & S-LoRA

Nicholas Satchanov

LoRA: Low-Rank Adaptation of Large Language Models

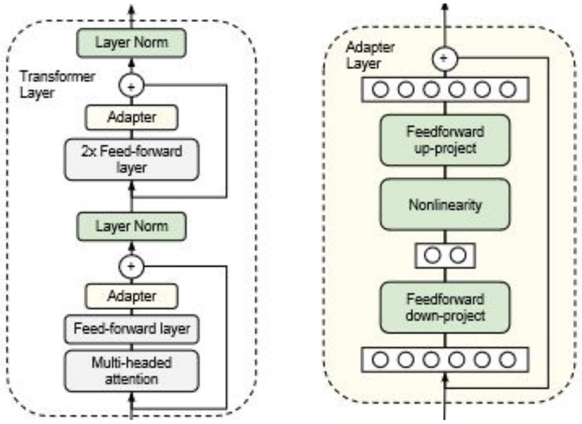
Background - Fine Tuning

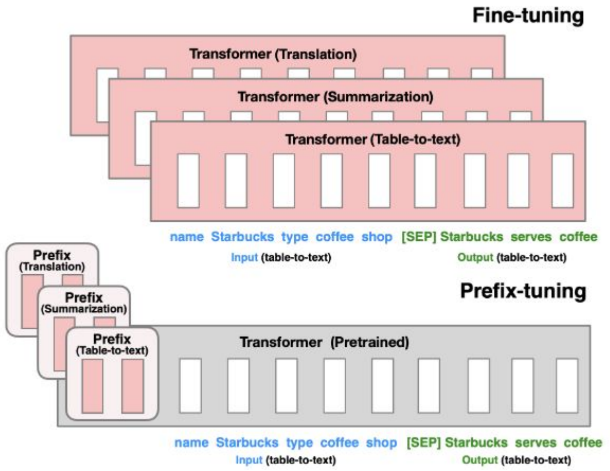
- Desire to create models for specific domains
  - Models are generally trained (pre-training)
  - Desire to fine-tune to become expert in other domain

- Models are large
  - Difficult to fine-tune
  - Need to readjust all parameters of large model
  - Fine tuners do not have access to pre-trainers fancy HW

Background - Past Works

- Two main approaches in fine-tuning optimization
- Adapter Layers
  - Add layers in transformer blocks
    - Small number of params, but must be computed sequentially
    - Increases inference latency
- Optimizing Prompts via Prefix Tuning
  - Update small task-specific vector (the prefix)
  - Reduces input sequence length
    - Leads to less effective prompts




Fine-tuning
Transformer (Translation)
Transformer (Summarization)
Transformer (Table-to-text)
name Starbucks type coffee shop [SEP] Starbucks serves coffee
Input (table-to-text)
Output (table-to-text)
Prefix-tuning
Prefix (Translation)
Prefix (Summarization)
Prefix (Table-to-text)
Transformer (Pretrained)
name Starbucks type coffee shop [SEP] Starbucks serves coffee
Input (table-to-text)
Output (table-to-text)

From works: Parameter-Efficient Transfer Learning for NLP
Prefix-Tuning: Optimizing Continuous Prompts for Generation

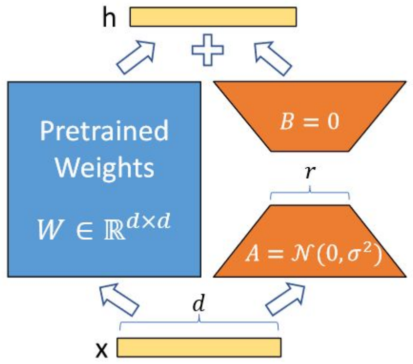
Key Insight: Intrinsic Dimension

- Key insight: Over-parameterized models reside on a low intrinsic dimension
  - Intrinsic Dimension: The dimension of parameter space where solutions to input first appear
    - Measurement of how difficult it is to answer a problem
    - Problem can be solved by navigating a subspace of problem space
- "200 parameters (randomly projected back into the full parameter space) are enough to represent the problem of tuning a RoBERTa model to within 90% of the performance of the full model."
  - OG param size = 125M
- Hypothesis: Change in weights during model fine-tuning also has low intrinsic rank
  - Can then encode update to weights as low dimension matrix

From works: Measuring the Intrinsic Dimension of Objective Landscapes
Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning

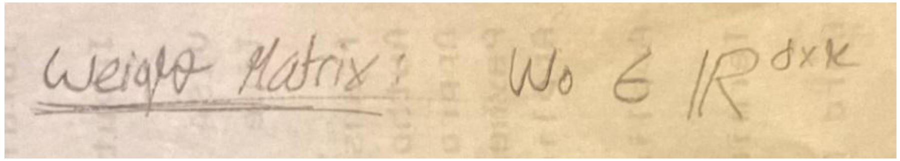
Approach, High Level Idea

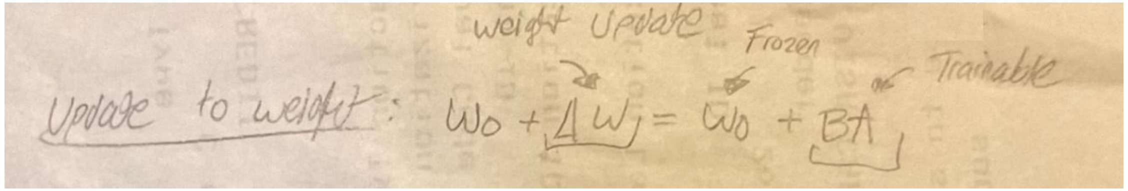
- Train some dense layers indirectly
  - Train on decomposition of these matrices instead
  - Do not alter the original pre-trained parameters


h
Pretrained Weights
$W \in \mathbb{R}^{d \times d}$
$B = 0$
r
$A = \mathcal{N}(0, \sigma^2)$
d
x

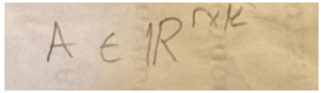
Technical Details - Weight Updates

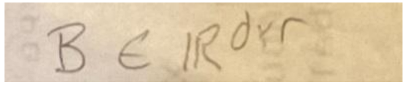
- Don't calculate new weights, calculate offset to weights

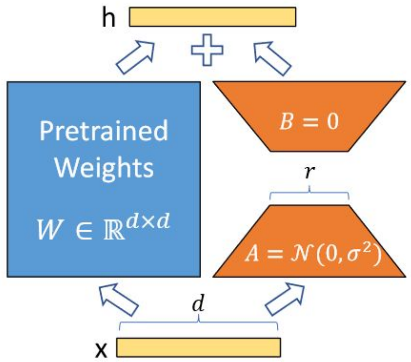

Weight Matrix: $W_0 \in \mathbb{R}^{d \times k}$

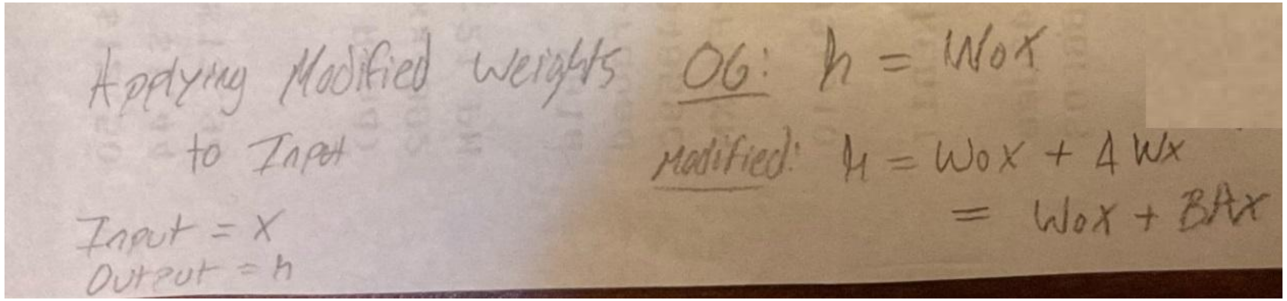

weight update Frozen Trainable

Update to weight: $W_0 + \underbrace{\Delta W} = W_0 + \underbrace{BA}$


$A \in \mathbb{R}^{r \times k}$


$B \in \mathbb{R}^{d \times r}$


h
Pretrained Weights
$W \in \mathbb{R}^{d \times d}$
$B = 0$
r
$A = \mathcal{N}(0, \sigma^2)$
d
x


Applying Modified weights to Input

Input = x
Output = h

OG: $h = W_0 x$

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

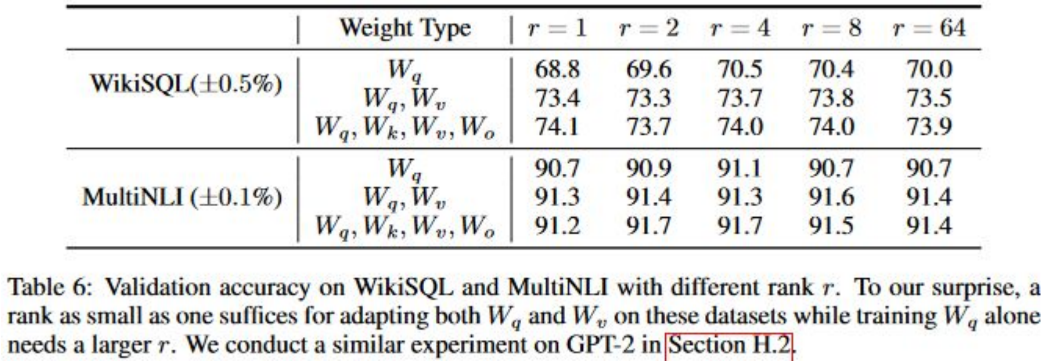
- A & B init values

Benefits of LoRA

- Model is now fine-tuned with a small number of parameters
  - Can reduce VRAM requirements by up to ⅔ if $r << d$
    - Easier to train
    - Easier to store
    - Faster to swap
- Do not need to calculate gradient onto most parameters
  - Since original weights are frozen
  - Speedup in training
    - 25% speedup during GPT-3 175B training compared to full fine-tuning
- Storage & compute efficient due to low amount of parameters
- Generalized sweeping range of fine-tuning performance
  - $r = d$ -> LoRA becomes full fine-tuning
- No inference latency increase
- Orthogonal to prior methods

Optimal LoRA Rank (r)

- Was the hypothesis correct?
- However, small r may not work
  For every case

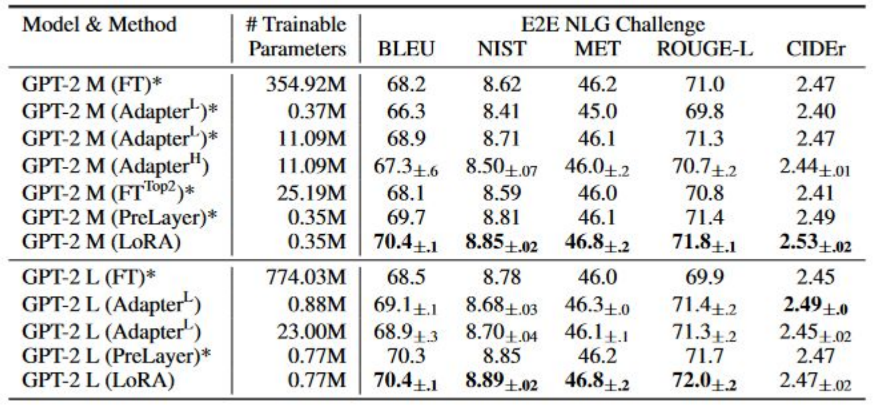


	Weight Type	$r = 1$	$r = 2$	$r = 4$	$r = 8$	$r = 64$
WikiSQL($\pm 0.5\%$)	W_q	68.8	69.6	70.5	70.4	70.0
	W_q, W_v	73.4	73.3	73.7	73.8	73.5
	W_q, W_k, W_v, W_o	74.1	73.7	74.0	74.0	73.9
MultiNLI ($\pm 0.1\%$)	W_q	90.7	90.9	91.1	90.7	90.7
	W_q, W_v	91.3	91.4	91.3	91.6	91.4
	W_q, W_k, W_v, W_o	91.2	91.7	91.7	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.

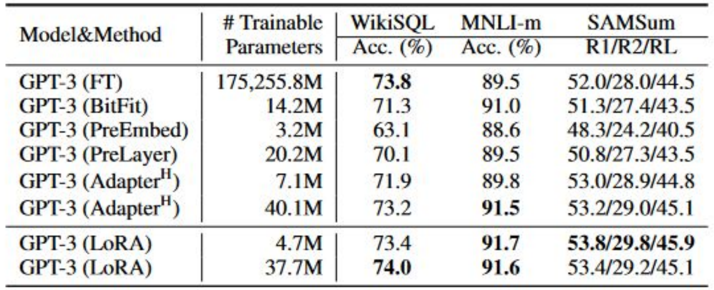
Evaluation

- E2E NLG = End-to-end natural language generation in restaurant domain
- WikiSQL = SQL query & natural language pairs
- MNLI-m = Multi-genre natural language inference (relationship between sentence pairs)
- SAMSum = Text conversations with summaries



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**$_{\pm.2}$	$2.47_{\pm.02}$

Table 3: GPT-2 medium (M) and large (L) with different adaptation methods on the E2E NLG Challenge. For all metrics, higher is better. LoRA outperforms several baselines with comparable or fewer trainable parameters. Confidence intervals are shown for experiments we ran. * indicates numbers published in prior works.



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.

Strengths, Weaknesses, and Limitations

- Strengths
  - Works in conjunction with other methods
  - Easy swapping and small memory footprint
- Weaknesses/Limitations
  - Only applied LoRA to attention weights
    - Limited design space exploration in this work
  - Cannot batch inputs to different tasks with different A and B in single forward pass
    - Limits training throughput

Directions for Future Research

- Combining LoRA with other fine-tuning methods
- Better ways to select LoRA’d weight matrices?
- Smaller exploring space for finding relationship between weight updates and pre-training weights

The End
(of LoRA)

S-LoRA: Serving Thousand of Concurrent LoRA Adapters

Background & Previous Work

- LoRA adapters
  - LoRA enables quick swapping and light storage of different fine-tunings
    - Swap adapters by adding and subtracting LoRA weights from the base model
  - How can we utilize this to efficiently swap tunings?
- KV cache causes decoding to be more memory intensive than compute intensive
- Orca - Iteration-level scheduling
  - Batch at token level instead of request level
  - Increases throughput by allowing new requests to join currently running batch
- vLLM
  - Optimize Orca memory efficiency with PagedAttention
  - Manage KV cache in paged fashion

Main Idea

- Store adapter weights in main memory
  - GPU memory too small to store there
- Issues
  - GPU memory too limited to store adapter weights
    - Dynamic fetching from elsewhere leads to memory fragmentation and I/O overhead
  - Hard to batch computation of adapters w/ unique ranks and in non-contiguous memory
  - Need to minimize communication and memory overhead between GPUs
- S-LoRA = Scalable LoRA
  - Unified Paging: Unified memory pool to reduce fragmentation and increase batch size
  - Heterogenous Batching: Batching in a way that allows concurrency of different adapters
  - S-LoRA Tensor Parallelism: Split up adapter weights for inference across GPUs

Batching & Scheduling

$$h = xW' = x(W + AB) \\ = xW + xAB.$$

- Don’t merge weights of base + adapter
  - Doesn’t support concurrent different adapters
  - Cannot batch different adapters together
- Instead, compute xAB on-the-fly
  - But poor HW utilization due to non-uniform rank dimension & sequence length
  - Use custom CUDA kernel to avoid this
- Schedule requests at the token level (Orca)
  - Can dump new requests into running batch where space is free
  - Reduces memory usage
  - More opportunity for concurrency
- Adapter clustering
  - Schedule together same adapters
  - Leads to less active adapters -> More memory space for KV cache

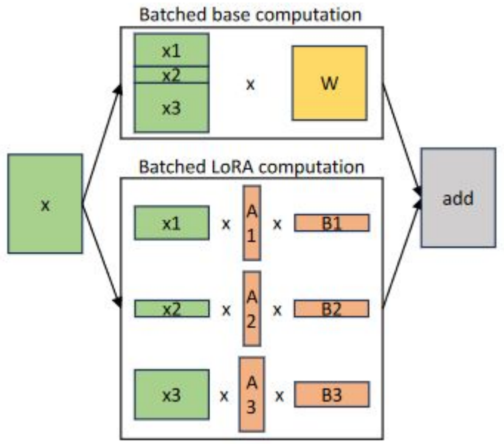


Figure 1. Separated batched computation for the base model and LoRA computation. The batched computation of the base model is implemented by GEMM. The batched computation for LoRA adapters is implemented by custom CUDA kernels which support batching various sequence lengths and adapter ranks.

Memory Management

Rank r gets r pages
- Adapter weights

Sequence length s gets s pages
- KV caches

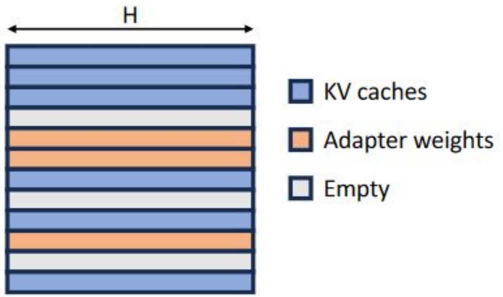
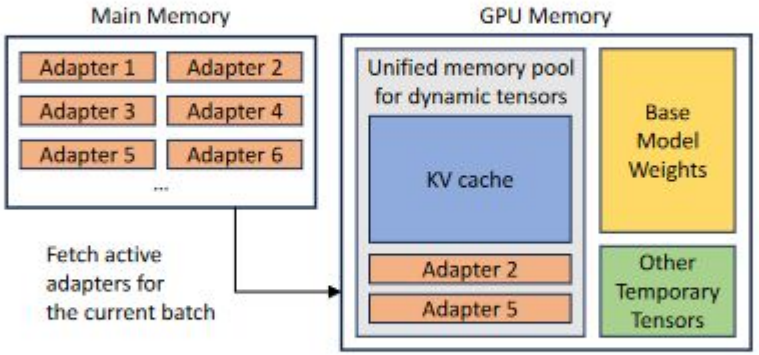


Figure 3. Unified memory pool. We use a unified memory pool to store both KV caches and adapter weights in a non-contiguous way to reduce memory fragmentation. The page size is H elements.

- Loading and unloading of active adapters
  - Inactive adapters -> Main memory
  - Active adapters -> GPU memory
- Naive loading and unloading problematic
  - Causes memory fragmentation
  - Latency overhead in switching data
- Unified paging - KV cache and adapter weights share same memory pool
  - Dynamic adapter weights act like dynamic KV caches
    - Ranks of active adapters depend on adapter selected to service request
    - KV cache size changes with sequence length
    - Both KV cache and adapter weights allocated on request input and deallocated on request completion
  - Interleaved storage to reduce fragmentation
- Prefetching and overlapping
  - Predict adapter weights needed for next batch when decoding current batch
    - Like mixture of experts?



Tensor Parallelism

- Base model already partitioned
- Need to partition adapter weights
  - Add blocks do the fine tuning from LoRA
- All weight matrices partitioned among all devices
  - No replication of weights
- Communication cost of LoRA computation dwarfed by base model

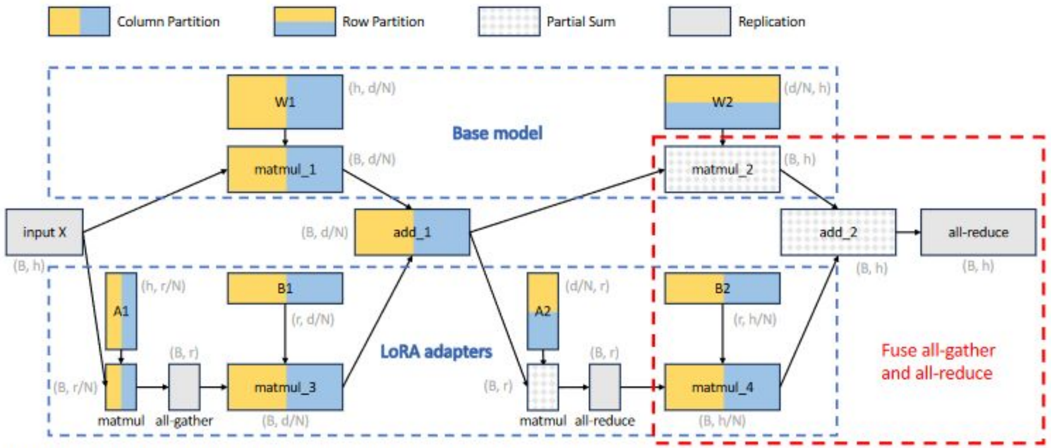


Figure 4. Tensor parallelism partition strategy for batched LoRA computation. This is a computational graph where nodes represent tensors/operators and the edges represent dependency. We use different colors to represent different partition strategies, which include column partition, row partition, partial sum, and replication. The per-GPU shape of each tensor is also annotated in gray. Note that B is the number of tokens, h is the input dimension, N is the number of devices, d is the hidden size, and r is the adapter rank.

Evaluation

- Results are from synthetically generated trace
- Bmm = No unified memory
  Or custom kernels

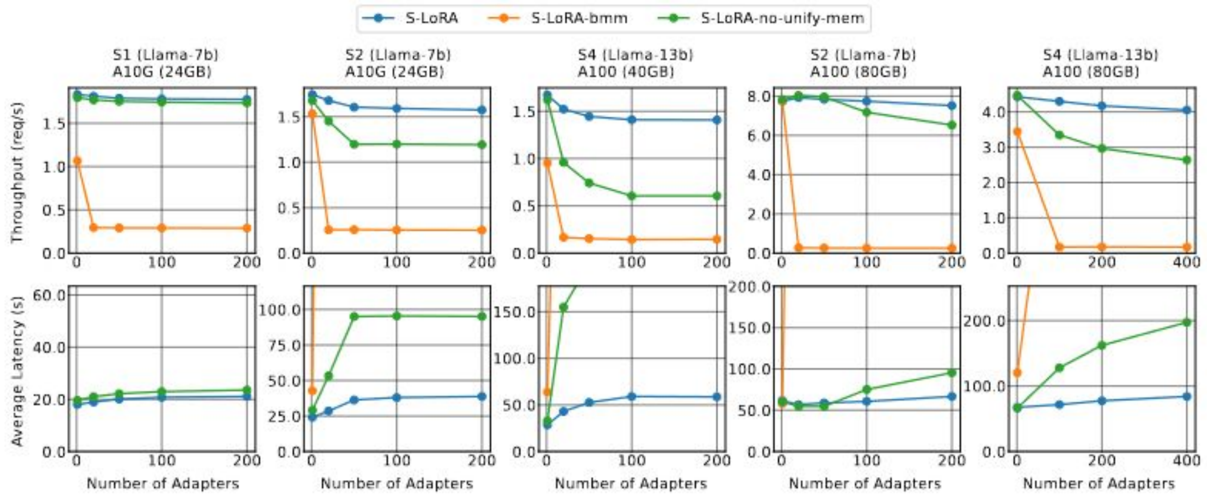


Figure 5. The throughput and average request latency of S-LoRA and its variants under different numbers of adapters. S-LoRA achieves significantly better performance and can scale to a large number of adapters. We run S-LoRA-bmm for a shorter duration since it has a significantly lower throughput. Some S-LoRA-bmm curves are omitted because it is out of the figures's scope.

Evaluation

- Traces generated from service of different base models
  - Diff model = Diff adapter
- SLO = service level objective
  - Desired latency of processing requests

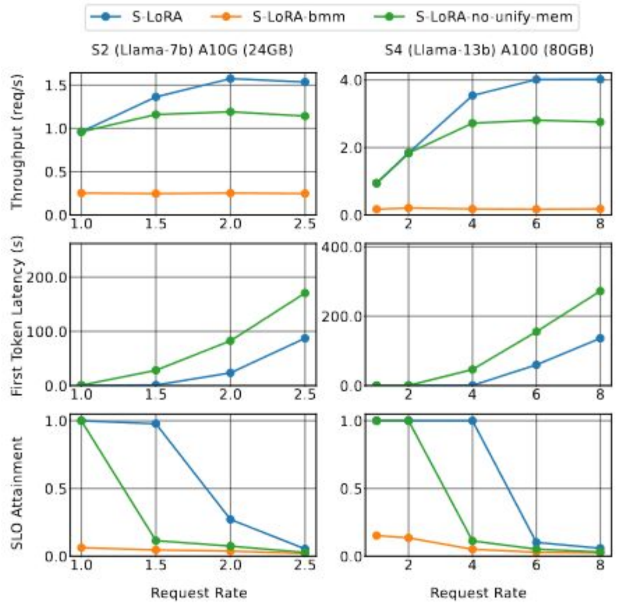


Figure 6. The throughput, first token latency, and SLO attainment of S-LoRA and its variants under different request rates. Note that in both settings the first token latency of S-LoRA-bmm is out of the figure's scope.

Strengths, Weaknesses, and Limitations

- Strengths
  - Able to support many LoRA adapters at same time (2,000)
- Weaknesses & Limitations
  - Main memory limits number of stored adapters
    - No study on if this is an issue given adapter and main memory sizes
  - Paper didn’t provide explanations for outcomes of actions
  - Study is limited to just Llama model
    - Is tensor partitioning also heavily based on model’s partition which isn’t general?
  - Evaluation focuses on comparing its own configs and not to other solutions

Future Work

- Optimizing the non-uniform kernel deployment
  - Batch different ranks & use non-contiguous memory
- Employ quantization and sparsification to LoRA adapter weights
- Evaluate this work on real hardware

The End