Efficient Pretraining and Finetuning Techniques

He He

NEW YORK UNIVERSITY

October 18, 2023

Logistics

- HW3 released: finetuning BERT! Due on Nov 3.

Introduction

Plan for today:

- How to train larger models on larger data with less compute
- How to finetune larger models with less compute
- Goal is to get an overview of the field

Introduction

Plan for today:

- How to train larger models on larger data with less compute
- How to finetune larger models with less compute
- Goal is to get an overview of the field

Why care about efficiency?

- Practical reasons: training and running these models are expensive!
- Methods that help scaling may eventually leads to *better* models (e.g., transformers)

> *"The bitter lesson is based on the historical observations that 1) AI researchers have often tried to build knowledge into their agents, 2) this always helps in the short term, and is personally satisfying to the researcher, but 3) in the long run it plateaus and even inhibits further progress, and 4) breakthrough progress eventually arrives by an opposing approach based on scaling computation by search and learning."* — Richard Sutton "The bitter lesson"

Table of Contents

Efficient pretraining

Efficient finetuning

Overview

Approaches to speed up pretraining

Overview

Approaches to speed up pretraining

- Reduce model size
- Design more sample-efficient learning objectives
- Improve efficiency of self-attention
- Improve system-level efficiency

Approach 1: Reduce model size

Idea 1: reduce the number of parameters

ALBERT (a lite BERT) [Lan et al., 2020]

- **Factorization**:
    - Recall that in Transformer, we first need to map the one-hot encoding (of size V) of a token to Q, K, V embeddings (of size H)
    - The number of parameters is $V \times H$
    - We can instead first map it to a lower-dim space (of size E) so that the number of params is $V \times E + E \times H$

Approach 1: Reduce model size

Idea 1: reduce the number of parameters

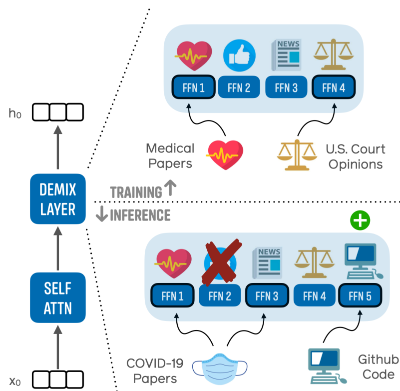
ALBERT (a lite BERT) [Lan et al., 2020]

- **Parameter sharing**:
  - Share feedforward network weights across layers
  - Share self-attention weights across layers
  - ALBERT: share all params across layers

Approach 1: Reduce model size

Idea 2: reduce interaction among parameters (sparse/modular architectures)

DEMix [Gururangan et al., 2022]



- Replace the FFN layer with an ensemble of n experts
- Route examples to experts corresponding to its domain determinstically

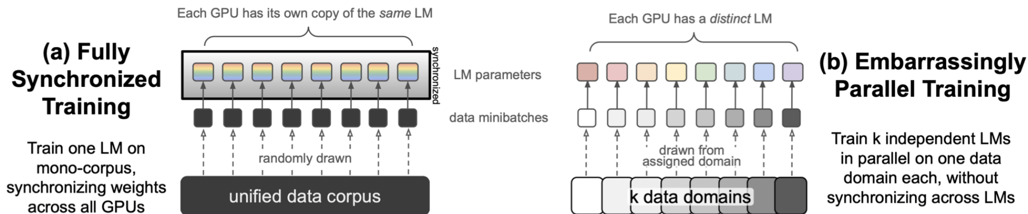
$$\mathrm{FFN}(h) = \sum_{i=1}^{n} \mathbb{I}[x \in \text{domain } i]\mathrm{FFN}_i(x)$$

- Only a subset of params are active for each example/batch

Approach 1: Reduce model size

Idea 2: reduce interaction among parameters (sparse/modular architectures)

Branch-Train-Merge [Li et al., 2022]



- Train domain experts in parallel and ensemble them (or take weighted average of their parameters)
- Reduce synchronization among GPUs at the cost of increased model size
- Easy to expand/remove domain experts

Approach 2: design sample-efficient learning objectives

ALBERT: Inter-sentence coherence loss

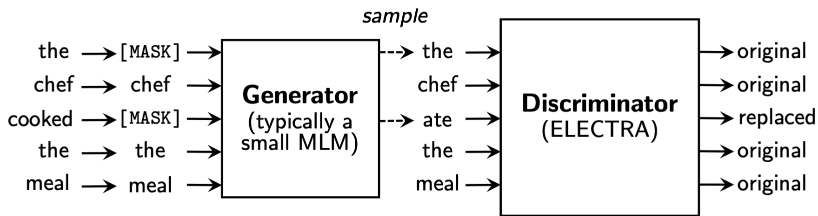
- Motivation: the next sentence prediction task is too easy
- Design hard negative examples
- Input: take two consecutive sentences, swap their order randomly
- Output: predict if they are in natural order

  I went home. SEP I slept. +1

  I slept. SEP I went home. -1
- Model needs to learn temporal order of events (commonsense, causality etc.)

Approach 2: design sample-efficient learning objectives

ELECTRA [Clark et al., 2020]: discriminate from true vs guessed tokens



- First train the generator for n steps using the MLM objective.
- Freeze generator weights. Train the discriminator using the sequence classification objective. Keep discriminator for finetuning.
- Comparison with MLM: predict at every position; hard negative examples.

Approach 2: design sample-efficient learning objectives

ELECTRA result:

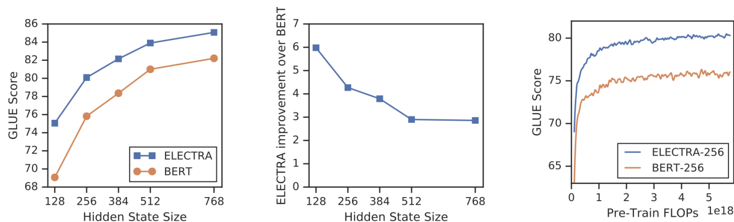


Figure: Finetuning result on the GLUE benchmark

- Larger improvement at smaller model sizes
- Faster training
- An effective approach if you don't have large compute for pretraining

Approach 3: alternatives to self-attention

Transformer recap

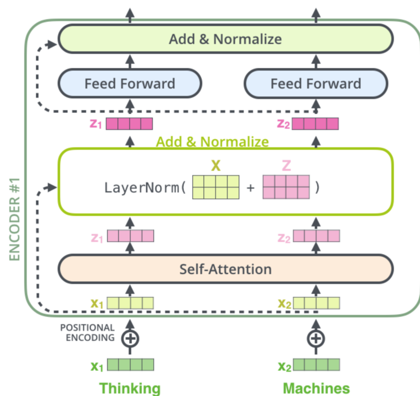


Figure: From The Illustrated Transformer

Which components require matrix multiplication?

Approach 3: alternatives to self-attention

Transformer recap

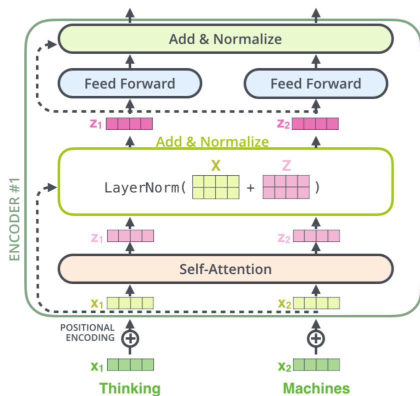


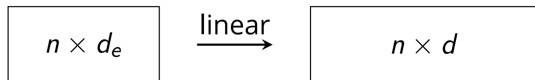
Figure: From The Illustrated Transformer

Which components require matrix multiplication?

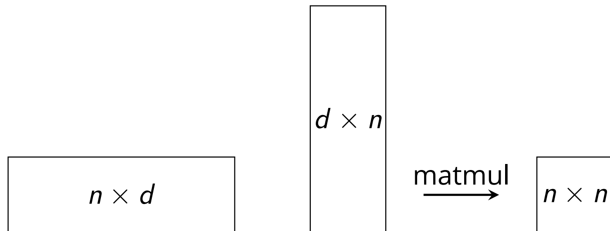
- Self-attention
  - Q,K,V projection
  - Scaled dot-product attention
- Feed-forward layer

Compute cost of transformers

Q, K, V projection:

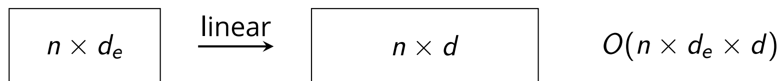


Scaled dot-product attention:

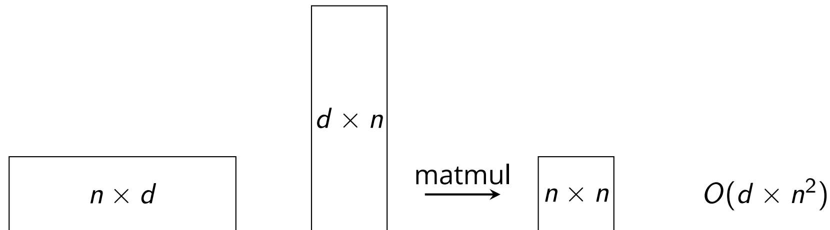


Compute cost of transformers

Q, K, V projection:

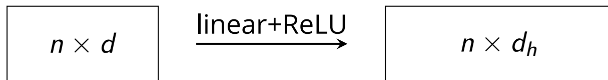


Scaled dot-product attention:



Compute cost of transformers

Feed-forward layer (GPT-2):

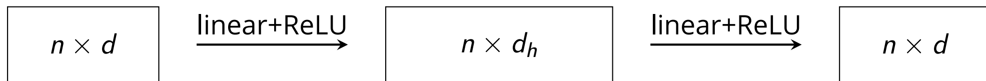


$O(n \times d \times d_h)$

- Two-layer FFN
- $d_h = 4d$ ($d > 1K$) by default in GPT-2
- Approximately half of the compute time

Compute cost of transformers

Feed-forward layer (GPT-2):



$O(n \times d \times d_h)$

- Two-layer FFN
- $d_h = 4d$ ($d > 1K$) by default in GPT-2
- Approximately half of the compute time

Improve efficiency of self-attention (for long sequences)

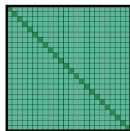
Improve efficiency of self-attention (for long sequences)

Key idea: reduce the $O(n^2)$ time and memory cost

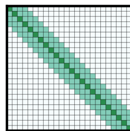
- Sparsify the attention matrix
  - Deterministic mask
  - Data-dependent mask (Reformer [Kitaev et al., 2020])
- Compress the key-value memory
  - Low-rank projection
  - Attention-based projection

Sparse attention

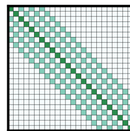
Longformer [Beltagy et al., 2020]: attention within a local window



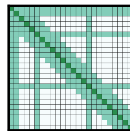
(a) Full n^2 attention



(b) Sliding window attention



(c) Dilated sliding window

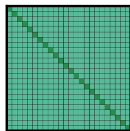


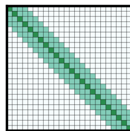
(d) Global+sliding window

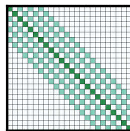
- Sliding window: attending to a *local* window of size w around each token $O(n \times w)$
- Dilated sliding window: reaching *longer range* with a larger window size with gaps
- Global window: *full attention* on specific tokens, e.g., `[CLS]` in BERT

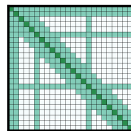
Sparse attention

Longformer [Beltagy et al., 2020]: attention within a local window


(a) Full n^2 attention


(b) Sliding window attention

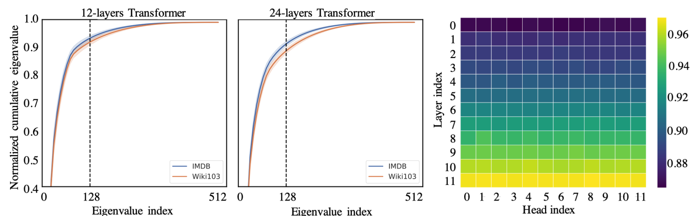

(c) Dilated sliding window


(d) Global+sliding window

- Sliding window: attending to a *local* window of size w around each token
  $O(n \times w)$
- Dilated sliding window: reaching *longer range* with a larger window size with gaps
- Global window: *full attention* on specific tokens, e.g., `[CLS]` in BERT
- Details: balancing efficiency and performance
  - Adding dilation on some heads
  - Using small window size on lower layers and larger ones on higher layers

Compresse the KV memory

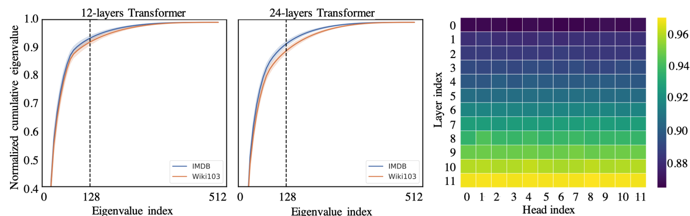
Self-attention is low rank [Wang et al., 2020]



- Left: cumulative eigenvalues of pretrained transformer with $n = 512$

Compresse the KV memory

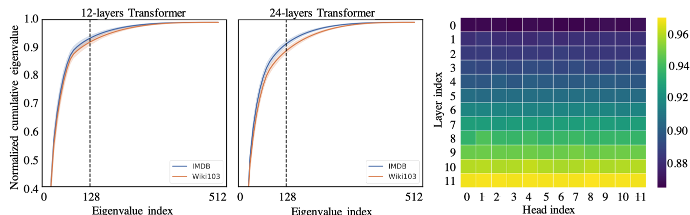
Self-attention is low rank [Wang et al., 2020]



- Left: cumulative eigenvalues of pretrained transformer with $n = 512$
  - Most information in the attention matrix can be recovered by the top 128 eigenvectors

Compresse the KV memory

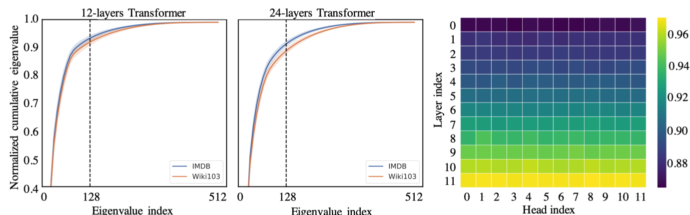
Self-attention is low rank [Wang et al., 2020]



- Left: cumulative eigenvalues of pretrained transformer with $n = 512$
  - Most information in the attention matrix can be recovered by the top 128 eigenvectors
- Right: cumulative eigenvalues of the top 128 eigenvalues across layers

Compresse the KV memory

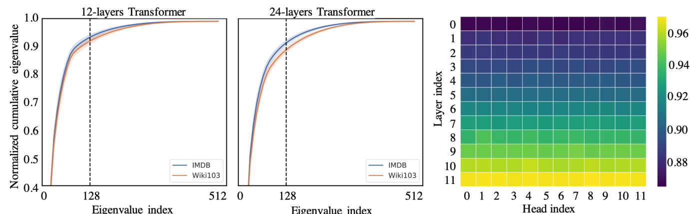
Self-attention is low rank [Wang et al., 2020]



- Left: cumulative eigenvalues of pretrained transformer with $n = 512$
  - Most information in the attention matrix can be recovered by the top 128 eigenvectors
- Right: cumulative eigenvalues of the top 128 eigenvalues across layers
  - Higher layers are more low-rank

Compresse the KV memory

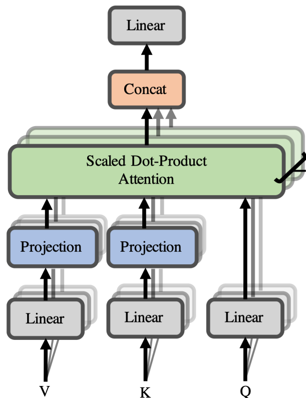
Self-attention is low rank [Wang et al., 2020]



- Left: cumulative eigenvalues of pretrained transformer with $n = 512$
  - Most information in the attention matrix can be recovered by the top 128 eigenvectors
- Right: cumulative eigenvalues of the top 128 eigenvalues across layers
  - Higher layers are more low-rank
- **Idea**: instead of attending to n tokens, attend to k principal components

Summarize the KV memory

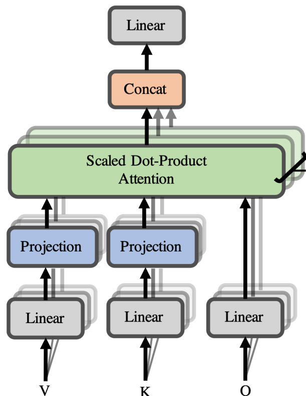
Linformer [Wang et al., 2020]: compute self-attention in a lower dimension



- Reduce dimensionality of the "memory": Map K, V from $n \times d$ to $k \times d$

Summarize the KV memory

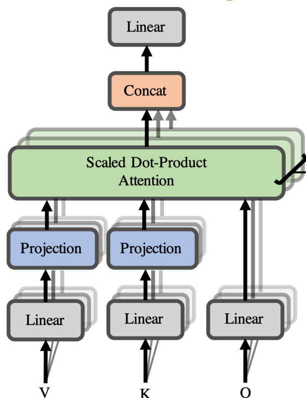
Linformer [Wang et al., 2020]: compute self-attention in a lower dimension



- Reduce dimensionality of the "memory": Map K, V from $n \times d$ to $k \times d$
- Attend to the lower-dimensional memory: $\mathrm{softmax}\left(Q_{n\times d} K^T_{k\times d} / \sqrt{d}\right)$

Summarize the KV memory

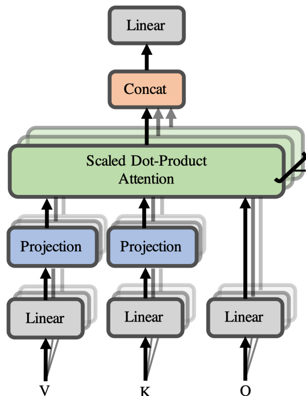
Linformer [Wang et al., 2020]: compute self-attention in a lower dimension



- Reduce dimensionality of the "memory": Map K, V from $n \times d$ to $k \times d$
- Attend to the lower-dimensional memory: $\mathrm{softmax}\left(Q_{n\times d} K^T_{k\times d} / \sqrt{d}\right)$
  - What's the dimension of the attention matrix?

Summarize the KV memory

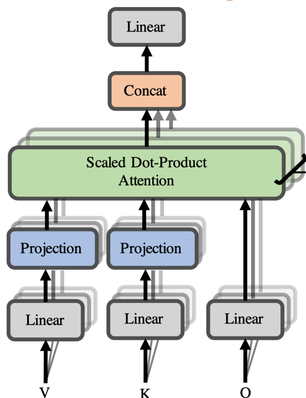
Linformer [Wang et al., 2020]: compute self-attention in a lower dimension



- Reduce dimensionality of the "memory": Map K, V from $n \times d$ to $k \times d$
- Attend to the lower-dimensional memory:
  $\mathrm{softmax}\left(Q_{n\times d} K^T_{k\times d} / \sqrt{d}\right)$
  - What's the dimension of the attention matrix?
  - What's the dimension of the self-attention output?

Summarize the KV memory

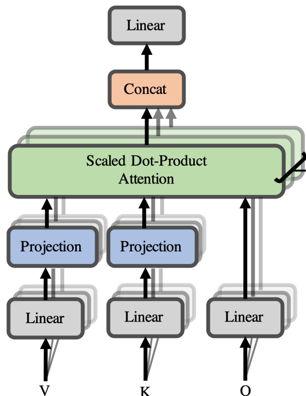
Linformer [Wang et al., 2020]: compute self-attention in a lower dimension



- Reduce dimensionality of the "memory": Map K, V from $n \times d$ to $k \times d$
- Attend to the lower-dimensional memory: $\mathrm{softmax}\left(Q_{n \times d} K_{k \times d}^T / \sqrt{d}\right)$
  - What's the dimension of the attention matrix?
  - What's the dimension of the self-attention output?
- Computation cost: $O(nk)$ (linear in n)

Summarize the KV memory

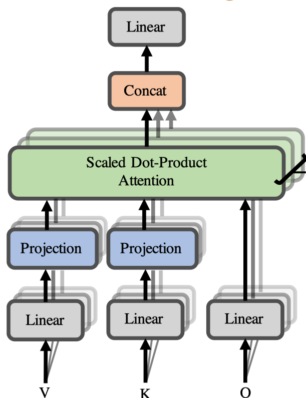
Linformer [Wang et al., 2020]: compute self-attention in a lower dimension



- Reduce dimensionality of the "memory": Map K, V from $n \times d$ to $k \times d$
- Attend to the lower-dimensional memory: $\mathrm{softmax}\left(Q_{n\times d} K^T_{k\times d} / \sqrt{d}\right)$
  - What's the dimension of the attention matrix?
  - What's the dimension of the self-attention output?
- Computation cost: $O(nk)$ (linear in n)
- Downside of uisng Linformer as a decoder?

Summarize the KV memory

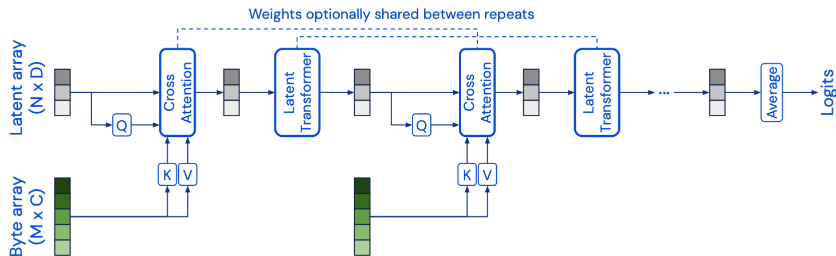
Linformer [Wang et al., 2020]: compute self-attention in a lower dimension



- Reduce dimensionality of the "memory": Map K, V from $n \times d$ to $k \times d$
- Attend to the lower-dimensional memory: $\mathrm{softmax}\left(Q_{n\times d} K^T_{k\times d} / \sqrt{d}\right)$
  - What's the dimension of the attention matrix?
  - What's the dimension of the self-attention output?
- Computation cost: $O(nk)$ (linear in n)
- Downside of uisng Linformer as a decoder?
  - Unclear how to mask: past and future are mixed

Compress the KV memory

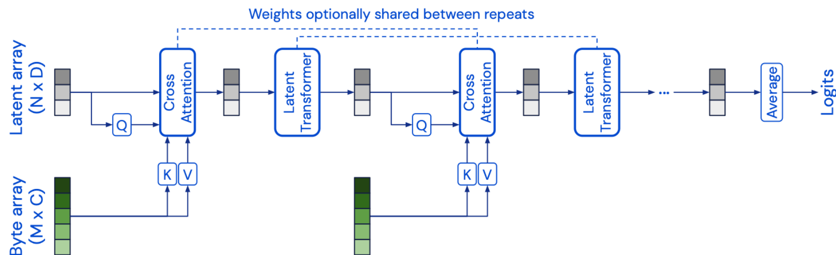
Perceiver [Jaegle et al., 2021]: use latent states to compress the KV memory



- Use latent states ($k \times d_s$) as queries to attend to K,V ($n \times d$) $\longrightarrow$ compress KV memory to lower dimensional states ($k \times d_s$)

Compress the KV memory

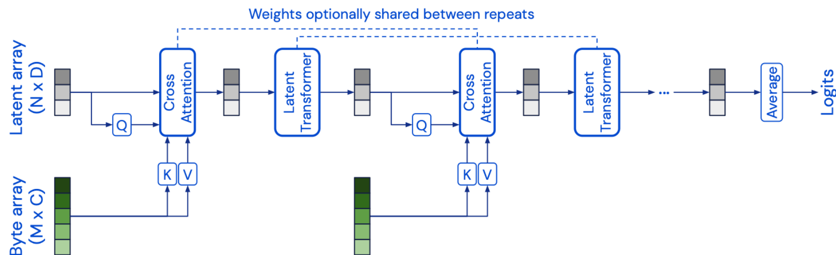
Perceiver [Jaegle et al., 2021]: use latent states to compress the KV memory



- Use latent states ($k \times d_s$) as queries to attend to K,V ($n \times d$) $\longrightarrow$ compress KV memory to lower dimensional states ($k \times d_s$)
- Map to latent states using cross attention: $O(nk)$

Compress the KV memory

Perceiver [Jaegle et al., 2021]: use latent states to compress the KV memory



- Use latent states ($k \times d_s$) as queries to attend to K,V ($n \times d$) $\longrightarrow$ compress KV memory to lower dimensional states ($k \times d_s$)
- Map to latent states using cross attention: $O(nk)$
- Self-attention layers on lower dimenstional *latent states* ($O(k^2)$)

Summary on efficient self-attention

Improve the quadratic time and space complexity of self-attention

- Sparsify the attention matrix
- Compress the KV memory

Summary on efficient self-attention

Improve the quadratic time and space complexity of self-attention

- Sparsify the attention matrix
- Compress the KV memory

Bad news: Most techniques are not widely used in large pretrained models now. Why?

- Improvement in time/space complexity doesn't always translate to real time/space savings
- These techniques often breaks structure and sacrifice the batching ability on GPUs
- Only see improvement on very long sequences

Approach 4: system-level approaches

- Operates at a lower abstraction level
- Often brings more direct impact on efficiency
- Example:
    - Gradient accumulation
    - Model and data parallelism (e.g., deepspeed)
    - Flash attention: exploit GPU memory asymmetry

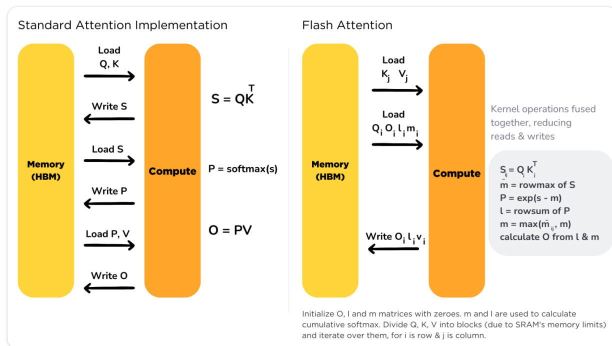


Table of Contents

Efficient pretraining

Efficient finetuning

Improve finetuning efficiency

Problem:

- In NLP, typically all parameters of the pretrained models (e.g., BERT) are finetuned, which is expensive for large models.
- Saving and loading finetuned models for different tasks is costly.

Improve finetuning efficiency

Problem:

- In NLP, typically all parameters of the pretrained models (e.g., BERT) are finetuned, which is expensive for large models.
- Saving and loading finetuned models for different tasks is costly.

Can we finetune a smaller number of parameters to achieve performance similar to full finetuning?

Improve finetuning efficiency

Problem:

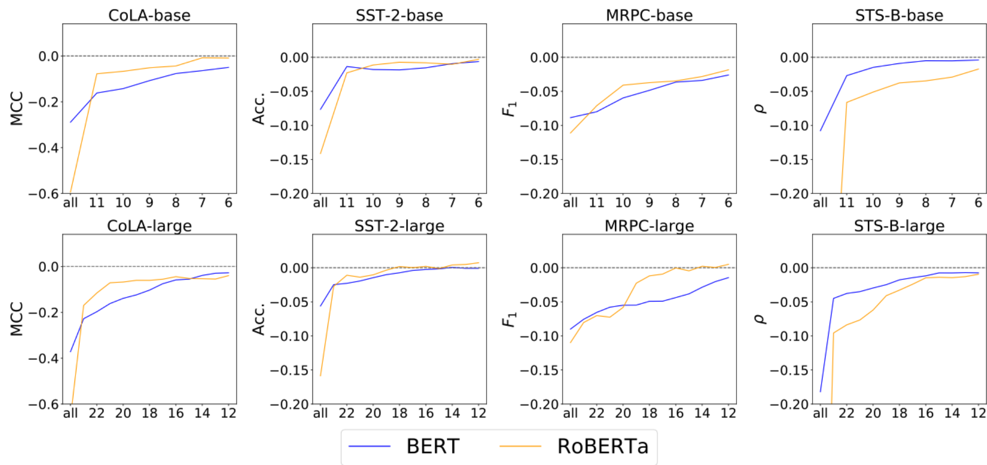
- In NLP, typically all parameters of the pretrained models (e.g., BERT) are finetuned, which is expensive for large models.
- Saving and loading finetuned models for different tasks is costly.

Can we finetune a smaller number of parameters to achieve performance similar to full finetuning?

- Select a subset of parameters from the pretrained weights to update
- Add a small number of parameters to adapte the (frozen) pretrained model

Finetune a subset of parameters

Freezing the first X layers [Lee et al., 2019]



A fourth of the layers need to be fine-tuned to obtain 90% of the performance.

Finetune a subset of parameters

BitFit [Ben-Zaken et al., 2022]: only finetune the bias term (0.1% of the parameters)

Bias terms in QKV projection

$$\mathbf{Q}^{m,\ell}(\mathbf{x}) = \mathbf{W}_q^{m,\ell}\mathbf{x} + \mathbf{b}_q^{m,\ell}$$
$$\mathbf{K}^{m,\ell}(\mathbf{x}) = \mathbf{W}_k^{m,\ell}\mathbf{x} + \mathbf{b}_k^{m,\ell}$$
$$\mathbf{V}^{m,\ell}(\mathbf{x}) = \mathbf{W}_v^{m,\ell}\mathbf{x} + \mathbf{b}_v^{m,\ell}$$

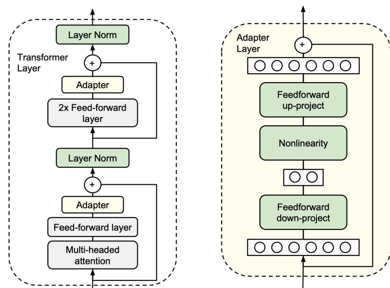
Bias terms in MLP layers

$$\mathbf{h}_2^\ell = \text{Dropout}\left(\mathbf{W}_{m_1}^\ell \cdot \mathbf{h}_1^\ell + \mathbf{b}_{m_1}^\ell\right)$$
$$\mathbf{h}_3^\ell = \mathbf{g}_{LN_1}^\ell \odot \frac{(\mathbf{h}_2^\ell + \mathbf{x}) - \mu}{\sigma} + \mathbf{b}_{LN_1}^\ell$$
$$\mathbf{h}_4^\ell = \text{GELU}\left(\mathbf{W}_{m_2}^\ell \cdot \mathbf{h}_3^\ell + \mathbf{b}_{m_2}^\ell\right)$$
$$\mathbf{h}_5^\ell = \text{Dropout}\left(\mathbf{W}_{m_3}^\ell \cdot \mathbf{h}_4^\ell + \mathbf{b}_{m_3}^\ell\right)$$
$$\text{out}^\ell = \mathbf{g}_{LN_2}^\ell \odot \frac{(\mathbf{h}_5^\ell + \mathbf{h}_3^\ell) - \mu}{\sigma} + \mathbf{b}_{LN_2}^\ell$$

Result: 80.9 (BitFit, 0.08% params) vs 81.8 (full finetuning) on GLUE

Adapt the frozen pretrained model

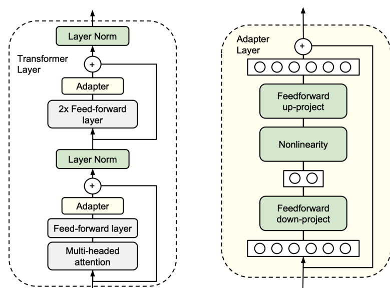
Adapter [Houlsby et al., 2019]: insert small networks to the pretrained model



- Insert learnable ”adapters” in-between layers
- Adapters uses a bottleneck structure to reduce parameters
- Adapters uses a skip-connection

Adapt the frozen pretrained model

Adapter [Houlsby et al., 2019]: insert small networks to the pretrained model

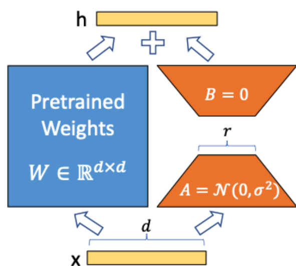


- Insert learnable ”adapters” in-between layers
- Adapters uses a bottleneck structure to reduce parameters
- Adapters uses a skip-connection such that it can be “reduced” to the original frozen model

Result: less than 0.4% performance drop with 3% more parameters on GLUE

Adapt the frozen pretrained model

LoRA [Hu et al., 2021]: add low-rank matrices as additional parameters



Hypothesis: weight matrices are low rank

Adapters: For any matrix multiplication $h = W_0 x$, we modify it to

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

- $W_0 \in \mathbb{R}^{d \times k}, B \in \mathbb{R}^{d \times r}, A \in \mathbb{R}^{r \times k} (r \ll k)$
- Initialization: $BA = 0$
- Can be applied to any weight matrices, e.g., QKV projection matrices

Adapt the frozen pretrained model

Compare LoRA and the original adapters:

- LoRA recovers full finetuning by increasing r

  Adapter recovers an MLP model with increasing params

Adapt the frozen pretrained model

Compare LoRA and the original adapters:

- LoRA recovers full finetuning by increasing r

  Adapter recovers an MLP model with increasing params
- LoRA has no additional inference cost

Adapt the frozen pretrained model

Compare LoRA and the original adapters:

- LoRA recovers full finetuning by increasing r

  Adapter recovers an MLP model with increasing params
- LoRA has no additional inference cost by setting $W_0 \leftarrow W_0 + BA$ (doesn't work for multiple tasks)

  Adapter incurs additional inference cost due to the added params

The most widely used efficient finetuning method on very large models ($>$100B).

Summary

Reduce finetuning cost by reducing the number of parameters to update

- Finetune a subset of parameters
- Finetune an additional adapters inserted to the model
- System approach: mixed-precision training (e.g., converting some or all params to fp16)

Other ways to adapt the model without parameter update: prompting, in-context learning (later)