LoRA: Low Rank Adaptation of LLMs
16 Oct 2021

Abstract
- LoW Rank Adaptation: freezes the pretrained model weights & injects trainable rank decomposition matrices into each layer of the Transformer architecture.
- Greatly reduces # of trainable parameters for down-stream tasks
- LoRA can reduce # of trainable params by 1,000 times & GPU memory requirement by 3 times.

Introduction
- Existing similar techniques increase inference latency & fail to match the fine-tuning baselines.
- Over-parameterized models reside on a low intrinsic dimension.

- We hypothesize that the change in weights during model adaptation also has a low "intrinsic rank"
- LoRA allows us to train some dense layers in a neural network indirectly by optimizing rank ~~[illegible]~~ decomposition matrices of the dense layer's change during adaptation instead, all while keeping the pretrained model weights frozen.
- A very low rank suffices even when the full rank is as high as 12,288, making LoRA storage & compute efficient.

- LoRA advantages:
  - A pre-trained model can be used to build many small LoRA modules for different tasks. Can freeze the model & efficiently switch tasks by replacing matrices A & B. Reduces memory requirement & ~~task~~ task-switching overhead
  - Makes training more efficient.
  - The simple linear design makes it easy to merge the trainable matrices w/ the frozen weights when deployed, introducing no inference latency
  - LoRA is orthogonal & combinable to many current methods

Problem Statement:

- Suppose we are given a pre-trained auto-regressive LM $P_\Phi(y|x)$, parameterized by Φ.

- During Full Fine-Tuning the model is initialized to pretrained weights Φ_0 & updated to $\Phi_0 + \Delta\Phi$ by repeatedly following the gradient to maximize the conditional LM objective:

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

- Problem: For each downstream task we want to optimize for, we learn a different set of $\Delta\Phi$ with $\dim(\Delta\Phi) = \dim(\Phi_0)$.

- Storing & deploying many independent instances of fine-tuned models can be challenging if not impossible.

- We adopt a more parameter efficient approach, where the task-specific parameter increment $\Delta\Phi = \Delta\Phi(\Theta)$ is further encoded by a much smaller-sized set of parameters Θ with $|\Theta| \ll |\Phi_0|$. The task of finding $\Delta\Phi$ thus becomes optimizing over Θ:

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

- We propose a low-rank representation to encode $\Delta\Phi$.

Aren't Existing Solutions Good Enough?
- Two current prominent strategies:
1. Adding adapter layers
2. Optimizing some forms of the input layer activations
- Both strats have their limitations
- Adapter Layers Introduce Inference Latency
- Directly optimizing the prompt is hard.
- Prefix tuning is difficult to optimize

Our Method

Low Rank Parametrized Update Matrices:

- Rank → # of linearly independent columns.
- NNs contain many dense layers which perform matrix multiplication
- The weight matrices in these layers usually have full rank

- When adapting to a specific task, the pre-trained LMs have a low "intrinsic dimension" and can still learn efficiently despite a random projection to a ~~smaller~~ subspace.

- We hypothesize the updates to the weights also have a low "intrinsic rank" during adaptation
- For pre-trained weight matrix $W_0 \in \mathbb{R}^{d \times k}$
- Constrain the update by representing the update with a low-rank decomposition:

  $W_0 + \Delta W = W_0 + BA$, where $B \in \mathbb{R}^{d \times r}$
  $A \in \mathbb{R}^{r \times k}$

  and rank $r \ll \min(d, k)$.

- During training, W_0 is frozen & doesn't receive gradient updates
- A & B contain trainable parameters.
- Modified forward pass:

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

- Use Gaussian init for A, zeros for B. - $\Delta W = BA = 0$ at initialization.
- Scale $\Delta W x$ by $\frac{\alpha}{r}$ where α is a constant in r.

- Generalization of Full Fine-Tuning:
- We roughly recover the expressiveness of full fine-tuning by setting the LoRA rank r to the rank of the pretrained weight matrices
- Ie, as we increase the # of trainable params, training LoRA roughly converges to training the original model

No additional Inference Latency:
- When deployed in production, we can explicitly compute & store $W = W_0 + BA$ & perform inference as usual.
- When we need to switch to another task, subtract BA from W & add $B'A'$.

Applying LoRA to Transformer:
- We can apply LoRA to any subset of weight matrices.
- In transformer, we have four weight matrices: (W_q, W_k, W_v, W_o) & two in the MLP module.

- We treat W_q (or W_k, W_v) as a single matrix of dimension $d_{model} \times d_{model}$, even though the output dimension is usually sliced into attention heads.
- Limit our study to only adapting the attention weights & freeze MLP modules, for simplicity & parameter efficiency.

Understanding the Low-Rank Updates:
Which Weight Matrices should we apply LoRA to (in the Transformer)?
- We only consider weight matrices in self-attention module.
- Set param budget of 18M (35MB if stored in FP16) on GPT-3 175B.
This corresponds to $r=8$ if we adapt one type of attention weights, or $r=4$ if we adapt two. For all 96 layers.