Background

Many applications in natural language processing rely on adapting one large-scale, pre-trained language model to multiple downstream applications. Such adaptation is usually done via fine-tuning, which updates all the parameters of the pre-trained model. The major downside of fine-tuning is that the new model contains as many parameters as in the original model.

Approach

Problem statement

During full fine-tuning, the model is initialized to pre-trained weights Φ_0 and updated to $\Phi_0 + \Delta\Phi$ by repeatedly following the gradient to maximize the conditional language modeling objective:

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

One of the main drawbacks for full fine-tuning is that for *each* downstream task, we learn a *different* set of parameters $\Delta\Phi$ whose dimension $|\Delta\Phi|$ equals $|\Phi_0|$. Thus, if the pre-trained model is large (such as GPT-3 with $|\Phi_0| \approx 175$ Billion), storing and deploying many independent instances of fine-tuned models can be challenging, if at all feasible.

In this paper, 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:

$$\max_{\Theta} \sum_{(x,y)\in\mathcal{Z}} \sum_{t=1}^{|y|} \log\left(p_{\Phi_0+\Delta\Phi(\Theta)}(y_t|x, y_{<t})\right) \tag{2}$$

In the subsequent sections, we propose to use a low-rank representation to encode $\Delta\Phi$ that is both compute- and memory-efficient. When the pre-trained model is GPT-3 175B, the number of trainable parameters $|\Theta|$ can be as small as 0.01% of $|\Phi_0|$.

Existing methods

Two major methods to tackle this problem:

1. Use Adaptor Layer: introduce latency. (large neural networks rely on hardware parallelism to keep the latency low, and adapter layers have to be processed sequentially.)
2. Directly Optimizing the Prompt is Hard. Use Prefix tuning. (Prefix tuning is difficult to optimize and that its performance changes non-monotonically in trainable parameters, confirming similar observations in the original paper. More)

Method

1. LOW-RANK-PARAMETRIZED UPDATE MATRICES

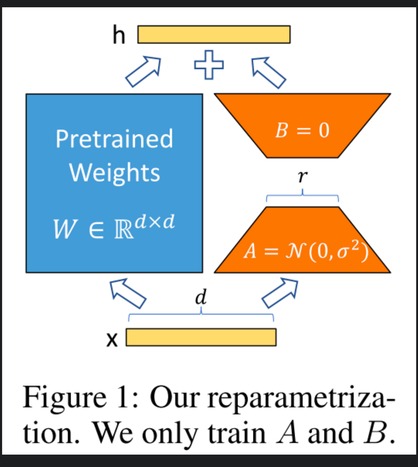

Figure 1: Our reparametrization. We only train A and B.

For a pre-trained weight matrix $W_0 \in R_d \times k$, we constrain its update by representing the latter with a low-rank decompostion:

$$W_0 + \Delta W = W_0 + BA$$

where

$$B \in R_d \times r, A \in R_r \times k.$$

And the rank $r \ll min(d, k)$. During training, W_0 is frozen and does not receive gradient updates, while A and B contain trainable parameters.

A Generalization of Full Fine-tuning.

No Additional Inference Latency.

2. APPLYING LORA TO TRANSFORMER

Apply LoRA to any subset of weight matrices in a neural network to reduce the number of trainable parameters.
In the Transformer architecture, there are four weight matrices in the self-attention module (W_q, W_k, W_v, W_o) and two in the MLP module. We treat W_q (or W_k, W_v) into attention heads. We limit our study to **only adapting the attention weights** for downstream as a single matrix of dimension $d_model \times d_model$, even though the output dimension is usually sliced tasks and freeze the MLP modules.

Practical Benefits and Limitations.
The most significant benefit comes from the reduction in memory and storage usage.

It is not straightforward to batch inputs to different tasks with different A and B in a single forward pass, if one chooses to absorb A and B intoW to eliminate additional inference latency. Though

优点：

1. 方便灵活，大模型冻住且能在不同任务里分享，切换任务时只对插入的低阶参数矩阵优化。方便部署。
2. 训练省时省力，不需要对绝大部分参数计算梯度和维护优化器参数，比起adaptive optimizer省3倍空间。
3. 相对于full fine tune模型，没有inference latency
4. LORA与其他算法可并行使用。