Summary of “LoRA: Low-Rank Adaptation of Large Language Models”

Kalab Assefa(kassefa), Nicholas Mellon(nbmellon), Rishika Kalidindi(rishikak), Sitota Mersha(sitotaem)

Problem and Motivation

Fine-tuning large language models is highly resource-intensive, requiring significant hardware and storage, especially when managing multiple instances for different tasks. For each downstream task, a separate set of parameters must be learned, equivalent in size to the original model. For models like GPT-3 with 175 billion parameters, storing and deploying fine-tuned versions becomes impractical. LoRA addresses these issues by hypothesizing that the necessary changes for model adaptation can be captured through a low-rank update, reducing the overhead of fine-tuning while preserving the performance of the model.

Related Works

To solve this problem, there are two main approaches taken by existing works. The first is to **add additional adapter layers**. Works like the Houlsby et al. (2019) and Lin et al. (2020) add a fixed number of adapter layers per transformer block and Ruckl ¨ e et al., 2020; Pfeiffer et al., 2021 try to improve latency by pruning layers or using multi task settings. However, all of these works do not circumvent the additional compute. Significant additional latency is also unavoidable as adapter layers are processed sequentially and batching cannot be leveraged in the online inference setting. Additional depth also requires sharded models to synchronize more causing further latency. Another solution is **prompt optimization** as put forth by works such as Li & Liang, 2021 by prefix tuning(adding special characters or word embeddings to the prompt) but this is observed to be difficult to achieve as performance optima is hard to reach. Another issue of reserving a part of the sequence is the reduced length for downstream tasks causing worse performance.

Solution Overview

LoRA (Low-Rank Adaptation) introduces a method that significantly reduces the number of trainable parameters required for fine-tuning large language models. Instead of updating all the parameters in a pre-trained model, LoRA freezes the model's weights and injects trainable low-rank decomposition matrices into each layer of the Transformer architecture.

The original weights are frozen during training and the rank decomposition matrices at each layer are trained, reducing the compute significantly. Post training, the newly trained module is merged into the pre-trained weights and so there are no additional layers.

This approach reduces the number of trainable parameters by a factor of 10,000 and lowers the GPU memory requirement by a factor of three, all while maintaining high training throughput and avoiding additional inference latency.

The key advantages of LoRA include:

- The ability to use a single shared pre-trained model for multiple tasks by building small LoRA modules for each task. This reduces storage and task-switching overhead.
- LoRA lowers the hardware barrier to entry, making training more efficient, as gradients and optimizer states only need to be calculated for the injected low-rank matrices, not the entire model.
- The linear design allows LoRA to merge the trainable matrices with frozen weights during deployment, resulting in no additional inference latency.
- LoRA is compatible with many existing fine-tuning methods, such as prefix-tuning.

Additionally, LoRA dramatically reduces memory and storage usage during training. On a large model like GPT-3 with 175 billion parameters, it cuts VRAM consumption from 1.2 TB to 350 GB, enabling training with fewer GPUs and avoiding I/O bottlenecks. This reduction also allows for faster task-switching by swapping only the LoRA weights rather than the entire model, which is stored in VRAM. LoRA also speeds up training by 25% on GPT-3 175B compared to full fine-tuning, as it avoids calculating gradients for most parameters.

Limitations

One major limitation of LoRA is that due to merging a module of a particular task to the model, it is not possible to batch requests of different tasks. It is possible to keep the modules unmerged and apply it as a separate layer based on the request; however, this runs into the same additional latency issue as the previous solutions.

Future Research Directions

- Explore combining LoRA with other adaptation methods for enhanced performance.
- Investigate how pre-trained models adapt to downstream tasks, with LoRA potentially offering clearer insights than full fine-tuning.
- Develop more systematic approaches for selecting weight matrices, moving beyond current heuristic methods.
- Study the potential rank-deficiency of original model weights, inspired by the rank-deficiency observed in LoRA's weight updates.

Summary of Class Discussion

Q: Why is B set = 0 as a design choice?
A: Initially we want to start with the original model weights and so we want ΔW to be 0. Tus we make B to be 0 so BA=0. The authors posit that A may also be 0 instead of B. There was some discussion hypothesizing the possibility of gradient vanishing based on initialisation of A to 0 instead.

Q: If you do LoRA on a “wrong” part of the model, can you get worse values?
A: Yes, applying it to different parts will have different outcomes and a certain degree of experimentation is necessary.

Q: Why is LoRA faster (because isn’t it similar to adapters added to model)?
A: No, when you are doing backpropagation, Wo doesn’t matter so you just do it on BA, and size of BA is << than W. LoRA inference is faster than adapter inference as ΔW is absorbed into W.

Q: Is the evaluation of inference on throughput, latency too, or just accuracy?
A: Just accuracy, because latency (etc.) should be the same as the original model.

Q: Have they applied LoRA to MLP or other non-attention models?
A: No, just attention because it's most computationally expensive. It can be applied to other layers as well but the authors restricted it to attention layers for efficiency. It is a design decision taken.

Q: Can LoRA be used for pre-training?
A: LoRA rank decomposes matrices to lower number of parameters which makes sense for fine tuning as it is over parameterised. But for pre-training it causes the model to be at a lower parameter count which is pointless.

Q: Can LoRA be used for sparse upcycling and re-training MoE layers instead of deep copy?
Insu Note: Yes, can be further explored in the mixture of LoRA paper.

Q: Is there some way to choose optimally or automatise hyperparameter values like r and k?
A: There is no direct way. Experiments need to be conducted to decide. Grid search may help choose. This paper also focuses on preventing additional adapter inference latency and so are alright with experimentation to decide the right hyper parameters.

Summary of “Sparse Upcycling: Training Mixture-of-Experts from Dense Checkpoints”

Problem and Motivation

Training large-scale neural networks from scratch is often inefficient and computationally expensive, which limits their accessibility and reuse. Scaling up dense models—where all parameters are applied to every input—can lead to high computational costs. Similarly, upscaling mixture-of-experts (MoE) models from scratch is also inefficient and requires a larger computational budget. To address these issues, the authors propose upcycling dense models into sparse MoE models, aiming to achieve significant performance gains while reducing the computational burden of training large-scale models. This is especially relevant for fine-tuning models for different domains or tasks.

Related Works

The main related concept has to do with **Mixture of Experts (MoE) models** (Shazeer et al., 2017), which are models with trained “experts” that get certain tokens relevant to their expertise routed to them throughout the training and inference process. Further, sparse set-ups of these models (where tokens are routed to a subset of experts instead of every expert) is becoming more common to improve efficiency and quality in certain cases (Nie et al., 2021; Wu et al., 2022). This paper uses these concepts via *sparse upcycling*.

Other important ideas that fall under a similar umbrella are **Reuse of trained parameters**, where at the start of training one model, another already trained model’s parameters are used (Berner et al., 2019). As well as **Pruning**, which removes less important nodes from a model to decrease its size and computational cost (sparse upcycling is similar in that sparsity is introduced, but different in that dense models are grown into this larger sparse model).

Solution Overview

The paper proposes a new technique for efficiently training Mixture-of-Experts (MoE) models from pre-trained dense models. The paper duplicates each MLP in each transformer layer multiple times and adds a certain router network, which is trained to route tokens to the MLPs. The upcycled model is then re-trained (to what point depends on use-case).

This approach offers several advantages:

1) It allows for effective utilization of existing, publicly available dense models.
2) Sparse MoE models significantly reduce computational costs compared to upscaled fully dense models.

3) The approach offers some flexibility in choosing and mixing components and hyperparameters in “model surgery,” allowing for customization for different use cases.

Limitations

The main concern and limitation with this paper is that it only dives into performance with a couple of model architectures in two specific areas (vision, language). While it may be the case that the idea (and similar performance/output) generalizes to different architectures in other domains, that is not certain. Further, different kinds of architectures may pose a problem for the model “surgery” that is needed in having to change to the sparse MoE set-up.

One other limitation is that long-term performance was not discussed. It would be useful to see if the performance in the proposed strategy remains stable over time.

Future Research Directions

Three main future directions that build off the limitations of the paper are:

1) Scaling this technique to larger models and datasets and comparing stability on those over time.
2) Showing that the technique works in other niche domains
3) Combining some of the above ideas, providing a more general framework of this technique that can generalize to models with different architectures (ie, reduce implementation time/cost of model surgery).

Summary of Class Discussion

Q: How do they copy MLP layers? Do they actually double the size when they copy?
A: They do a deep copy. Having duplicates is necessary because they would need to back propagate on each of the matrices.

Q: What is the difference between initializing parameters randomly vs using the pre-trained weights?
A: Random initialization will converge eventually but will use a lot more compute so it is better to just start with pre-trained params.

> **A:** Follow-up: this only is worth it if you have a very small budget. If you have a big budget it's worth it to train from scratch.
> **Q:** Why is a cold-start even a consideration in this?
> **A:** Starting from scratch can optimize model more, but requires more compute

Q: Does it imply you are using the same token embeddings (if MLP layers are the same)?
A: Yes everything is the same, no modifications for token embeddings.

Insu Note: Top-K is the usual routing method, Expert Choice is a new method proposed by Google that allows experts to sample tokens to be balanced. There is discussion in the paper on how many MLP layers (and which ones) should be replaced with MoE.

Q: Why was MoE done on the backside of the model?
A: The authors decided starting MoE on backside was better because of less back propagation cost, even though doing at front may provide better results

Q: Are there any further works involving more general frameworks for automatically choosing hyperparams for these techniques?
A: Not yet, you still need to do a narrowing down yourself based on your hardware and budget

> **Q:** Follow up: But doing grid search on this increases computing cost by a lot?
> **A:** You spend this compute initially to make it more efficient in the long run and for deployment. It's a game of compute time vs storage complexity

Q: Why not train for specific tasks from the start (example math)?
A: Makes more sense to have a robust general model to build off at start given limited resources. Also quality and breadth of data for very specific cases may not be large enough.

Q: Where does the initial quality drop come from?
A: Dense is well trained to that point, but when we mess with it there will be a quality drop and also the router is also randomly initialized.