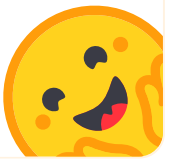


LoRA

This conceptual guide gives a brief overview of LoRA, a technique that accelerates the fine-tuning of large models while consuming less memory.

To make fine-tuning more efficient, LoRA's approach is to represent the weight updates with two smaller matrices (called **update matrices**) through low-rank decomposition. These new matrices can be trained to adapt to the new data while keeping the overall number of changes low. The original weight matrix remains frozen and doesn't receive any further adjustments. To produce the final results, both the original and the adapted weights are combined.

This approach has a number of advantages:

- LoRA makes fine-tuning more efficient by drastically reducing the number of trainable parameters.
- The original pre-trained weights are kept frozen, which means you can have multiple lightweight and portable LoRA models for various downstream tasks built on top of them.
- LoRA is orthogonal to many other parameter-efficient methods and can be combined with many of them.
- Performance of models fine-tuned using LoRA is comparable to the performance of fully

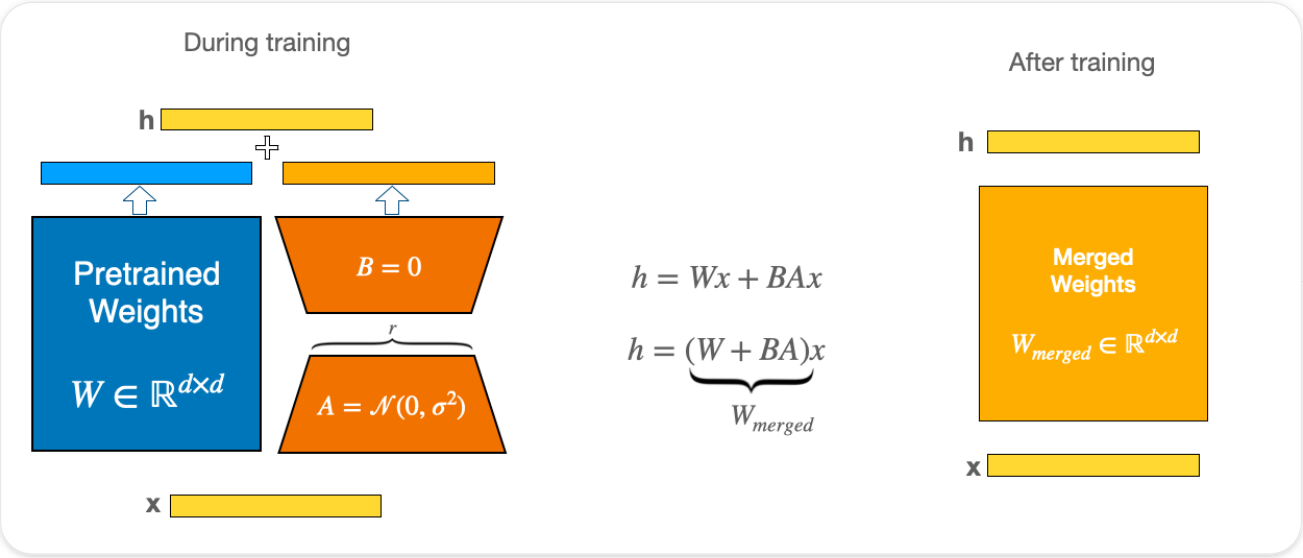
fine-tuned models.

- LoRA does not add any inference latency because adapter weights can be merged with the base model.

In principle, LoRA can be applied to any subset of weight matrices in a neural network to reduce the number of trainable parameters. However, for simplicity and further parameter efficiency, in Transformer models LoRA is typically applied to attention blocks only. The resulting number of trainable parameters in a LoRA model depends on the size of the low-rank update matrices, which is determined mainly by the rank `r` and the shape of the original weight matrix.

Merge LoRA weights into the base model

While LoRA is significantly smaller and faster to train, you may encounter latency issues during inference due to separately loading the base model and the LoRA model. To eliminate latency, use the merge_and_unload() function to merge the adapter weights with the base model which allows you to effectively use the newly merged model as a standalone model.



This works because during training, the smaller weight matrices (*A* and *B* in the diagram above) are separate. But once training is complete, the weights can actually be merged into a new weight matrix that is identical.

Utils for LoRA

Use `merge_adapter()` to merge the LoRa layers into the base model while retaining the PeftModel. This will help in later unmerging, deleting, loading different adapters and so on.

Use `unmerge_adapter()` to unmerge the LoRa layers from the base model while retaining the PeftModel. This will help in later merging, deleting, loading different adapters and so on.

Use unload() to get back the base model without the merging of the active lora modules. This will help when you want to get back the pretrained base model in some applications when you want to reset the model to its original state. For example, in Stable Diffusion WebUi, when the user wants to infer with base model post trying out LoRAs.

Use delete_adapter() to delete an existing adapter.

Use add_weighted_adapter() to combine multiple LoRAs into a new adapter based on the user provided weighing scheme.

Common LoRA parameters in PEFT

As with other methods supported by PEFT, to fine-tune a model using LoRA, you need to:

1. Instantiate a base model.
2. Create a configuration (`LoraConfig`) where you define LoRA-specific parameters.
3. Wrap the base model with `get_peft_model()` to get a trainable `PeftModel`.
4. Train the `PeftModel` as you normally would train the base model.

`LoraConfig` allows you to control how LoRA is applied to the base model through the following parameters:

- `r`: the rank of the update matrices, expressed in `int`. Lower rank results in smaller update matrices with fewer trainable parameters.
- `target_modules`: The modules (for example, attention blocks) to apply the LoRA update matrices.
- `alpha`: LoRA scaling factor.

- `bias`: Specifies if the `bias` parameters should be trained. Can be `'none'`, `'all'` or `'lora_only'`.
- `modules_to_save`: List of modules apart from LoRA layers to be set as trainable and saved in the final checkpoint. These typically include model's custom head that is randomly initialized for the fine-tuning task.
- `layers_to_transform`: List of layers to be transformed by LoRA. If not specified, all layers in `target_modules` are transformed.
- `layers_pattern`: Pattern to match layer names in `target_modules`, if `layers_to_transform` is specified. By default `PeftModel` will look at common layer pattern (`layers`, `h`, `blocks`, etc.), use it for exotic and custom models.
- `rank_pattern`: The mapping from layer names or regexp expression to ranks which are different from the default rank specified by `r`.
- `alpha_pattern`: The mapping from layer names or regexp expression to alphas which are different from the default alpha specified by `lora_alpha`.

LoRA examples

For an example of LoRA method application to various downstream tasks, please refer to the following guides:

- Image classification using LoRA
- Semantic segmentation

While the original paper focuses on language models, the technique can be applied to any dense layers in deep learning models. As such, you can leverage this technique with diffusion models. See Dreambooth fine-tuning with LoRA task guide for an example.

← Fully Sharded Data Parallel Prompting →