With the appearance of ChatGPT, the world recognized the powerful potential of large language models, which can understand natural language and respond to user requests with high accuracy. In the abbreviation of LLM, the first letter **L** stands for **Large**, reflecting the massive number of parameters these models typically have.

Modern LLMs often contain over a billion parameters. Now, imagine a situation where we want to adapt an LLM to a downstream task. A common approach consists of **fine-tuning**, which involves adjusting the model's existing weights on a new dataset. However, this process is extremely slow and resource-intensive—especially when run on a local machine with limited hardware.

The core principles of **LoRA (Low-Rank Adaptation)**, a popular technique for reducing the computational load during fine-tuning of large models. As a bonus, we'll also take a look at QLoRA, which builds on LoRA by incorporating quantization to further enhance efficiency.

Neural network representation

LoRA

The idea described in the previous section perfectly illustrates the core concept of LoRA. **LoRA** stands for **Low-Rank Adaptation**, where the term low-rank refers to the technique of **approximating a large weight matrix by factorizing it into the product of two smaller matrices with a much lower rank k**. This approach significantly reduces the number of trainable parameters while preserving most of the model's power.

Training

Let us assume we have an input vector x passed to a fully connected layer in a neural network, which before fine-tuning, is represented by a weight matrix W. To compute the output vector y, we simply multiply the matrix by the input: $y = Wx$.

During fine-tuning, the goal is to adjust the model for a downstream task by modifying the weights. This can be expressed as learning an additional matrix ΔW, such that: $y = (W + \Delta W)x = Wx + \Delta Wx$. As we saw the multiplication trick above, we can now replace ΔW by multiplication BA, so we ultimately get: $y = Wx + BAx$. As a result, we freeze the matrix $W$and solve the optimization task to find matrices A and B that totally contain much less parameters than ΔW!

However, direct calculation of multiplication ***(BA)x*** during each forward pass is very slow due to the the fact that matrix multiplication ***BA*** is a heavy operation. To avoid this, we can leverage associative property of matrix multiplication and rewrite the operation as ***B(Ax)***. The multiplication of ***A*** by ***x*** results in a vector that will be then multiplied by ***B*** which also ultimately produces a vector. This sequence of operations is much faster.

In terms of backpropagation, LoRA also offers several benefits. Despite the fact that a gradient for a single neuron still takes nearly the same amount of operations, we now deal with much fewer parameters in our network, which means:

- we need to compute far fewer gradients for ***A*** and ***B*** than would originally have been required for ***W***.
- we no longer need to store a giant matrix of gradients for ***W***.

Finally, to compute ***y***, we just need to add the already calculated ***Wx*** and ***BAx***. There are no difficulties here since matrix addition can be easily parallelized.

As a technical detail, before fine-tuning, matrix ***A*** *is initialized using a Gaussian distribution, and matrix* ***B*** *is initialized with zeros. Using a zero matrix for* ***B*** *at the beginning ensures that the model behaves exactly as before, because* ***BAx = 0 · Ax = 0****, so* ***y*** *remains equivalent to* ***Wx****.*

This makes the initial phase of fine-tuning more stable. Then, during backpropagation, the model gradually adapts its weights for ***A*** *and* ***B*** *to learn new knowledge.*

After training

After training, we have calculated the optimal matrices ***A*** and ***B***. All we have to do is multiply them to compute ***ΔW***, which we then add to the pretrained matrix ***W*** to obtain the final weights.

While the matrix multiplication ***BA*** *might seem like a heavy operation, we only perform it once, so it should not concern us too much! Moreover, after the addition, we no longer need to store* ***A****,* ***B****, or* ***ΔW****.*

Subtlety

While the idea of LoRA seems inspiring, a question might arise: during normal training of neural networks, why can't we directly represent y as ***BAx*** instead of using a heavy matrix ***W*** to calculate **y = Wx**?

The problem with just using ***BAx*** is that the model's capacity would be much lower and likely insufficient for the model to learn effectively. During training, a model needs to learn massive amounts of information, so it naturally requires a large number of parameters.

In LoRA optimization, we treat ***Wx*** as the prior knowledge of the large model and interpret ***ΔWx = BAx*** as task-specific knowledge introduced during fine-tuning. So, we still cannot deny the importance of ***W*** in the model's overall performance.

Adapter

Studying LLM theory, it is important to mention the term "**adapter**" that appears in many LLM papers.

*In the LoRA context, an **adapter** is a combination of matrices A and B that are used to solve a particular downstream task for a given matrix W.*

For example, let us suppose that we have trained a matrix ***W*** such that the model is able to understand natural language. We can then perform several independent LoRA optimizations to tune the model on different tasks. As a result, we obtain several pairs of matrices:

- ***(A_1, B_1)***—adapter used to perform question-answering tasks.
- ***(A_2, B_2)***—adapter used for text summarization problems.
- ***(A_3, B_3)***—adapter trained for chatbot development.

- **QLoRA**
- QLoRA is another popular term whose difference from LoRA is only in its first letter, **Q**, which stands for "**quantized**". The term "**quantization**" refers to the reduced number of bits that are used to store weights of neurons.
- For instance, we can represent neural network weights as floats requiring 32 bits for each individual weight. ***The idea of quantization consists of compressing neural network weights to a smaller precision without significant loss or impact on***

the model's performance. So, instead of using 32 bits, we can drop several bits to use, for instance, only 16 bits.

- **prefix-tuning**
- **Prefix-tuning** is an interesting alternative to LoRA. *The idea also consists of using adapters for different downstream tasks but this time adapters are integrated inside the attention layer of the Transformer*.
- More specifically, during training, all model layers become frozen except for those that are added as prefixes to some of the embeddings calculated inside attention layers. In comparison to LoRA, prefix tuning does not change model representation, and in general, it has much fewer trainable parameters. As previously, to account for the prefix adapter, we need to perform addition, but this time with fewer elements.
- *Unless given very limited computational and memory constraints, LoRA adapters are still preferred in many cases, compared to prefix tuning.*