**Quantization**

**What is Model Quantization?**

Quantization is a technique used to reduce the precision of numerical values in a model. Instead of using high-precision data types, such as 32-bit floating-point numbers, quantization represents values using lower-precision data types, such as 8-bit integers. This process significantly reduces memory usage and can speed up model execution while maintaining acceptable accuracy.

**Hugging Face and Bitsandbytes Uses**

Hugging Face’s Transformers library is a go-to choice for working with pre-trained language models. To make the process of model quantization more accessible, Hugging Face has seamlessly integrated with the Bitsandbytes library. This integration simplifies the quantization process and empowers users to achieve efficient models with just a few lines of code.

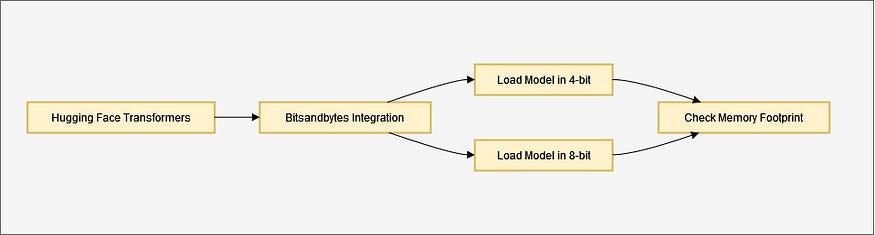
Install latest accelerate from source:

pip install git+<https://github.com/huggingface/accelerate.git>

Install latest transformers from source and bitsandbytes:

pip install git+https://github.com/huggingface/transformers.git

pip install bitsandbytes



Hugging Face and Bitsandbytes Integration Uses

## **Loading a Model in 4-bit Quantization**

One of the key features of this integration is the ability to load models in 4-bit quantization. This can be done by setting the load\_in\_4bit=True argument when calling the .from\_pretrained method. By doing so, you can reduce memory usage by approximately fourfold.

from transformers import AutoModelForCausalLM, AutoTokenizer

model\_id = "bigscience/bloom-1b7"

tokenizer = AutoTokenizer.from\_pretrained(model\_id)

model = AutoModelForCausalLM.from\_pretrained(model\_id, device\_map="auto", load\_in\_4bit=True)

## **Loading a Model in 8-bit Quantization**

For further memory optimization, you can load a model in 8-bit quantization. This can be achieved by using the load\_in\_8bit=True argument when calling .from\_pretrained. This reduces the memory footprint by approximately half.

from transformers import AutoModelForCausalLM, AutoTokenizer

model\_id = "bigscience/bloom-1b7"

tokenizer = AutoTokenizer.from\_pretrained(model\_id)

model = AutoModelForCausalLM.from\_pretrained(model\_id, device\_map="auto", load\_in\_8bit=True)

You can even check the memory footprint of your model using the get\_memory\_footprint method:

print(model.get\_memory\_footprint())

**Other Use cases:**

The Hugging Face and Bitsandbytes integration goes beyond basic quantization techniques. Here are some use cases you can explore:

**Changing the Compute Data Type**

You can modify the data type used during computation by setting the bnb\_4bit\_compute\_dtype to a different value, such as torch.bfloat16. This can result in speed improvements in specific scenarios. Here's an example:

from transformers import BitsAndBytesConfig

quantization\_config = BitsAndBytesConfig(load\_in\_4bit=True, bnb\_4bit\_compute\_dtype=torch.bfloat16)

**Using NF4 Data Type**

The NF4 data type is designed for weights initialized using a normal distribution. You can use it by specifying bnb\_4bit\_quant\_type="nf4":

from transformers import BitsAndBytesConfig

nf4\_config = BitsAndBytesConfig(load\_in\_4bit=True, bnb\_4bit\_quant\_type="nf4")

model\_nf4 = AutoModelForCausalLM.from\_pretrained(model\_id, quantization\_config=nf4\_config)

## **Nested Quantization for Memory Efficiency**

The integration also recommends using the nested quantization technique for even greater memory efficiency without sacrificing performance. This technique has proven beneficial, especially when fine-tuning large models:

from transformers import BitsAndBytesConfig

double\_quant\_config = BitsAndBytesConfig(load\_in\_4bit=True, bnb\_4bit\_use\_double\_quant=True)

model\_double\_quant = AutoModelForCausalLM.from\_pretrained(model\_id, quantization\_config=double\_quant\_config)

## **Loading a Quantized Model from the Hub**

A quantized model can be loaded with ease using the from\_pretrained method. Make sure the saved weights are quantized by checking the quantization\_config attribute in the model configuration:

model = AutoModelForCausalLM.from\_pretrained("model\_name", device\_map="auto")

In this case, you don’t need to specify the load\_in\_8bit=True argument, but you must have both Bitsandbytes and Accelerate library installed.

**Exploring Advanced techniques and configuration**

There are additional techniques and configurations to consider:

**Offloading Between CPU and GPU**

One advanced use case involves loading a model and distributing weights between the CPU and GPU. This can be achieved by setting llm\_int8\_enable\_fp32\_cpu\_offload=True. This feature is beneficial for users who need to fit large models and distribute them between the GPU and CPU.

**Adjusting Outlier Threshold**

Experiment with the llm\_int8\_threshold argument to change the threshold for outliers. This parameter impacts inference speed and can be fine-tuned to suit your specific use case.

**Skipping the Conversion of Some Modules**

In certain situations, you may want to skip the conversion of specific modules to 8-bit. You can do this using the llm\_int8\_skip\_modules argument.

**Fine-Tuning a Model Loaded in 8-bit**

With the support of adapters in the Hugging Face ecosystem, can fine-tune models loaded in 8-bit quantization, enabling the fine-tuning of large models with ease.

**Conclusion**

Quantization is a powerful technique for optimizing machine learning models. The integration of Hugging Face’s Transformers library with the Bitsandbytes library makes this technique accessible to a broader audience. Whether you’re looking to reduce memory usage, speed up model execution, or share quantized models with the community, this integration provides the tools and flexibility you need to do so. It’s a significant step towards making efficient machine learning models available to all.

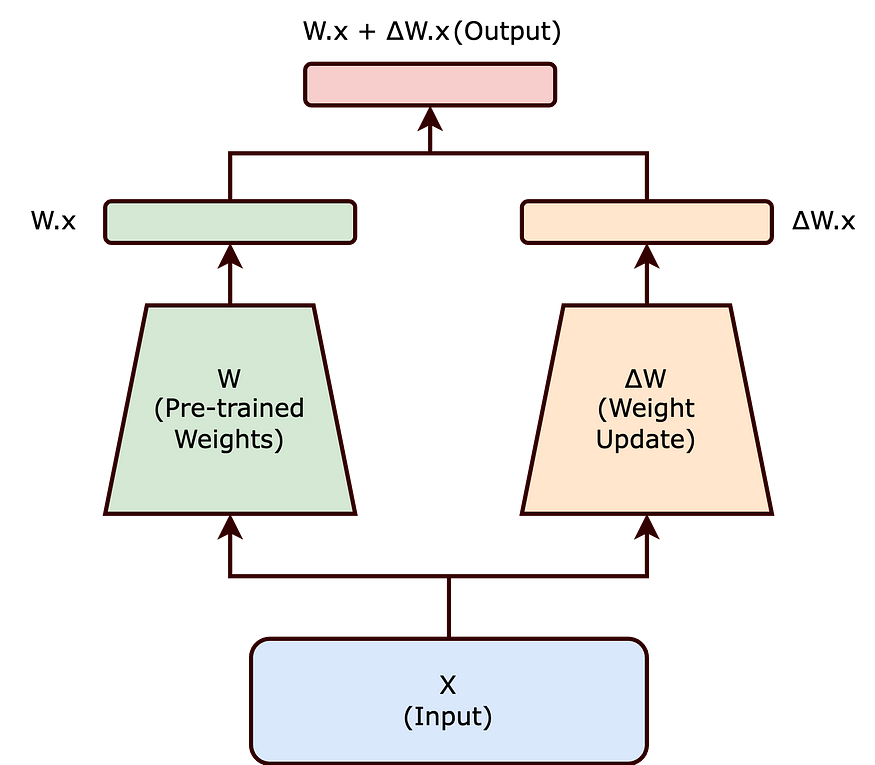
**LoRA (Low Rank Adaptation)**

Fine-tuning large pre-trained models is computationally challenging, often involving adjustment of millions of parameters. This traditional fine-tuning approach, while effective, demands substantial computational resources and time, posing a bottleneck for adapting these models to specific tasks. LoRA presented an effective solution to this problem by decomposing the update matrix during finetuing. To study LoRA, let us start by first revisiting traditional finetuing.

**Decomposition of ( Δ W )**

In traditional fine-tuning, we modify a pre-trained neural network’s weights to adapt to a new task. This adjustment involves altering the original weight matrix ( W ) of the network. The changes made to ( W ) during fine-tuning are collectively represented by ( Δ W ), such that the updated weights can be expressed as ( W + Δ W ).

Now, rather than modifying ( W ) directly, the LoRA approach seeks to decompose ( Δ W ). This decomposition is a crucial step in reducing the computational overhead associated with fine-tuning large models.



Traditional finetuning can be reimagined us above. Here W is frozen where as ΔW is trainable (Image by the blog author)

**The Intrinsic Rank Hypothesis**

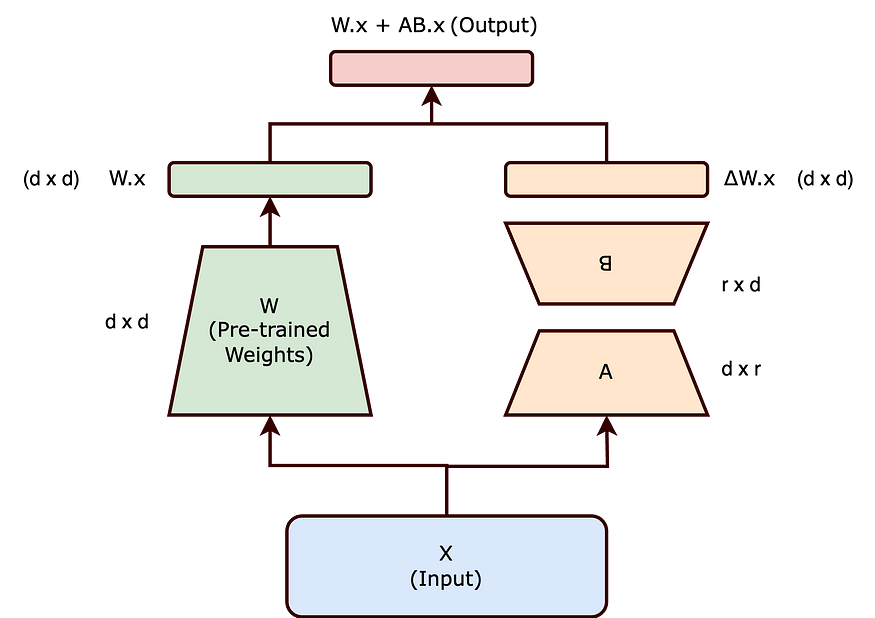
The intrinsic rank hypothesis suggests that significant changes to the neural network can be captured using a lower-dimensional representation. Essentially, it posits that not all elements of ( Δ W ) are equally important; instead, a smaller subset of these changes can effectively encapsulate the necessary adjustments.

**Introducing Matrices ( A ) and ( B )**

Building on this hypothesis, LoRA proposes representing ( Δ W ) as the product of two smaller matrices, ( A ) and ( B ), with a lower rank. The updated weight matrix ( W’ ) thus becomes:

[ W’ = W + BA ]

In this equation, ( W ) remains frozen (i.e., it is not updated during training). The matrices ( B ) and ( A ) are of lower dimensionality, with their product ( BA ) representing a low-rank approximation of ( Δ W ).



ΔW is decomposed into two matrices A and B where both have lower dimensionality then d x d. (Image by the blog author)

**Impact of Lower Rank on Trainable Parameters**

By choosing matrices ( A ) and ( B ) to have a lower rank ( r ), the number of trainable parameters is significantly reduced. For example, if ( W ) is a ( d x d ) matrix, traditionally, updating ( W ) would involve ( d² ) parameters. However, with ( B ) and ( A ) of sizes ( d x r ) and ( r x d ) respectively, the total number of parameters reduces to ( 2dr ), which is much smaller when ( r << d ).

The reduction in the number of trainable parameters, as achieved through the Low-Rank Adaptation (LoRA) method, offers several significant benefits, particularly when fine-tuning large-scale neural networks:

1. Reduced Memory Footprint: LoRA decreases memory needs by lowering the number of parameters to update, aiding in the management of large-scale models.
2. Faster Training and Adaptation: By simplifying computational demands, LoRA accelerates the training and fine-tuning of large models for new tasks.
3. Feasibility for Smaller Hardware: LoRA’s lower parameter count enables the fine-tuning of substantial models on less powerful hardware, like modest GPUs or CPUs.
4. Scaling to Larger Models: LoRA facilitates the expansion of AI models without a corresponding increase in computational resources, making the management of growing model sizes more practical.

In the context of LoRA, the concept of rank plays a pivotal role in determining the efficiency and effectiveness of the adaptation process. Remarkably, the paper highlights that the rank of the matrices *A* and *B* can be astonishingly low, sometimes as low as one.

Although the LoRA paper predominantly showcases experiments within the realm of Natural Language Processing (NLP), the underlying approach of low-rank adaptation holds broad applicability and could be effectively employed in training various types of neural networks across different domains.

**Conclusion**

LoRA’s approach to decomposing ( Δ W ) into a product of lower rank matrices effectively balances the need to adapt large pre-trained models to new tasks while maintaining computational efficiency. The intrinsic rank concept is key to this balance, ensuring that the essence of the model’s learning capability is preserved with significantly fewer parameters.

**QLoRA (Quantization and Low-Rank Adapters (LoRA))**

**Quantization in the context of deep learning is the process of reducing the numerical precision of a model's tensors, making the model more compact and the operations faster in execution. Quantization maps high-precision floating points to lower-precision values, usually 8- or even 4-bit fixed-point representation integers. This reduce tensor sizes by 75% or even 87.5%, but usually results in a significant reduction in model accuracy; however, it may also help the model generalize better, so experimentation may be worthwhile.**

**LoRA, which stands for "Low Rank Adaptation," is a method designed to fine-tune large pre-trained language models more efficiently by reducing the number of trainable parameters.**

First, some definitions. **A low-rank approximation of a matrix aims to approximate the original matrix as closely as possible, but with a lower rank. The rank of a matrix is a value that gives you an idea of the matrix’s complexity**; a lower-rank matrix reduces computational complexity, and thus increases efficiency of matrix multiplications. Low-rank decomposition refers to the process of effectively approximating a matrix A by deriving low-rank approximations of A. Singular Value Decomposition (SVD) is a common method for low-rank decomposition.

More details for the math people; feel free to skip this. The rank of a matrix is the linear space spanned by its rows or columns; to be precise, it is the maximal number of linearly independent columns (or rows) in a matrix. Low-rank decomposition approximates a given matrix A with lower-rank matrices by a product of two matrices U and V, which can then be further decomposed into matrices of lower dimensions.

**Now, onto LoRA. Fine-tuning usually involves updating the entire set of parameters in the model, which can be computationally expensive and time-consuming for large language models. LoRA makes this process more efficient by creating and updating low-rank approximations of the original weight matrices (called update matrices) which are formed using low-rank decomposition on the original weight matrix. Only these matrices are updated during fine-tuning - the original model weights remain the same - and thus LoRA’s total number of trainable parameters is equal to the size of the low-rank update matrices.** Since the low-rank matrices are being updated instead of all of the larger matrices with far more parameters, we can do this on a smaller, cheaper GPU.

Here's a simplified explanation of how LoRA works:

* Initial Model: Start with a large pre-trained model (e.g., Llama 2, Mistral, etc.).
* Low-Rank Matrix: Introduce low-rank approximations for the matrices that will be used to adapt the model for the specific task at hand. Low-rank matrices (adapters) are typically formed for all linear layers of the model, but this can vary based on the model architecture and the task.
* Transform Layers: Instead of directly modifying the original weights of the model, LoRA applies a transformation using the low-rank matrices to the outputs of affected layers.
* Fine-tuning: During the fine-tuning process, only the parameters in the low-rank matrices are updated. The rest of the model's parameters are kept fixed. Again, updating only low-rank matrices allows for fine-tuning on smaller, cheaper GPUs.
* Prediction: For making predictions, the adapted layers are used in conjunction with the original pre-trained model. The low-rank adapted layers act as a kind of "add-on" to the existing architecture, adjusting its behavior for the specific task.

The benefits of LoRA include:

* Efficiency: Because it only updates a small subset of parameters, fine-tuning is faster and requires less computational power. According to the LoRA paper, it can reduce the number of trainable parameters by 10,000 times and the GPU memory requirement by 3 times.
* Specialization: The low-rank adaptation allows the model to specialize in a particular task without a complete overhaul of the original weights.
* Preservation: By keeping the bulk of the model fixed, LoRA helps preserve the generalization capabilities of the original pre-trained model while still enabling task-specific adaptations.

Quantization + LoRA = QLoRA

In this method, the original model's parameters are first quantized to lower-bit values based on a user-defined quantization configuration. This makes the model more compact. Subsequently, LoRA is applied to the model's layers to further optimize for the specific task. This combination in QLoRA allows for fine-tuning on significantly less computational power, which essentially democratizes the ability to fine-tune models.



**TL;DR**

**QLoRA, a method which combines Quantization and Low-Rank Adaptation (LoRA), presents a groundbreaking approach to fine-tuning large pre-trained models. By applying quantization, it efficiently compresses the original model, and through LoRA, it drastically reduces the number of trainable parameters. This synergistic combination democratizes the fine-tuning process, making it feasible to perform on smaller, more accessible GPUs. By democratizing access to sophisticated fine-tuning methods, QLoRA stands as a significant advancement in the field of machine learning, promising a more inclusive future for both researchers and practitioners alike.**