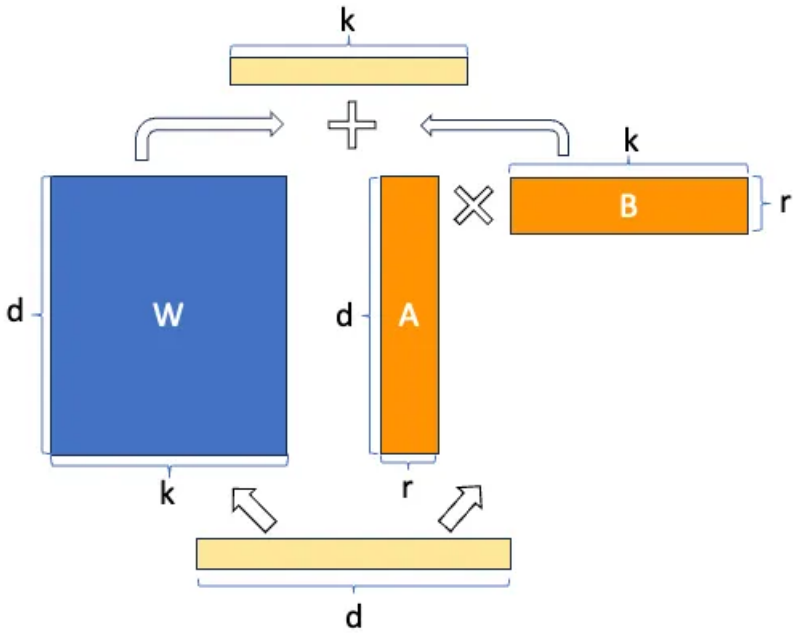
Fine-tuning a GPT — LoRA

Chris Kuo/Dr. Dataman · Follow
18 min read · 6 days ago



This post explains the proven fine-tuning method **LoRA**, the abbreviation for "**Lo**w-**R**ank **A**daptation of Large Language Models". In this post, I will walk you through the LoRA technique, its architecture, and its advantages. I will present related background knowledge, such as the concepts of "low-rank" and "adaptation" to help your understanding. Similar to "Fining-tune a GPT — Prefix-tuning", I cover a code example and will walk you through the code line by line. I will especially cover the GPU-consuming nature of fine-tuning a Large Language Model (LLM). Then I talk more about practical treatments which have been packaged as Python libraries "bitsandbytes" and "accelerate". After completing this article, you will be able to explain:

- When do we still need fine-tuning?
- The challenges in adding more layers — Inference Latency
- What is the "rank" in the "low-rank" of LoRA?
- The architecture of LoRA
- The advantages of LoRA

- Fine-tuning is still GPU-intensive
- The techniques to reduce the use of GPUs
- The code example

Why do we still need fine-tuning?

Pretrained Large Language Models (LLMs) are already trained with different types of data for various tasks such as text summarization, text generation, question and answering, etc. Why do we still need to fine-tune a LLM? The simple reason is to train the LLM to adapt to your domain data to do specific tasks. Just like a well-trained chef can do Italian cuisine or Chinese cuisine, the chef just needs to transfer his basic cooking knowledge and sharpen his skills to do other kinds of cuisines. The term "transfer learning" and "fine-tuning" are sometimes used interchangeably in NLP (for example, "Parameter-Efficient Transfer Learning for NLP" by Neil Houlsby et al. (2019)). With the arrival of LLMs, the term "fine-tuning" is used even more. (If you like to understand more about image learning, you can take a look at "Transfer Learning for Image Classification — (1) All Start Here" or the book "Transfer learning for image classification — with Python examples".)

The challenges in adding more layers — Inference Latency

Since the magnitudes of LLMs are growing unprecedentedly, we will try not to touch the trained parameters but to add layers to an LLM or add values to parameters. The added layers are often called the "adapters" and the fine-tuning technique is called "adapter-tuning". It involves adding small adapter modules to the pre-trained model and trains only the parameters in the adapter modules.

However, it has been found that the additional layers cause latency in production, which is called "Inference Latency". It is not pleasant if you need to wait for more than 20 seconds for an LLM to provide an answer. This issue appears unavoidable because the adapter layers are added sequentially to an LLM. They have to be processed sequentially and there is no way to parallel them. You may ask how about batching data for fast speed. It is a good idea. But in real-time use or inferencing, users typically input a prompt or a question one at a time, so the batch size is 1 and there is not much batching for the data. What's the solution? A worthy thing to highlight is that **LoRA does not add layers but adds values to parameters.** This solution will not result in inference latency. I'll go deep into the architecture of LoRA soon. At this moment, all I like to share with you is there are different fine-tuning strategies, and the adapt-tuning strategy can cause inference latency.

The title of LoRA is "Low-Rank Adaptation of Large Language Models". Now we know "adaptation" means fine-tuning the domain data and tasks. It is not called an "adapter" because it does not add adapters. It is called "adaption" to describe its adaptation process.

Now we are ready to talk about the architecture. Let me still break it does to talk about the “low-rank” in LoRA.

What is the “rank” in the “low-rank” of LoRA?

Let me bring up the definition of the “rank” of a matrix. It is the maximal number of linearly independent columns of a matrix. Below I show a 2 * 3 matrix V and its transpose matrix with 3 * 2 dimension. Notice that the second row of V is the exact opposite of the first row. It is not linearly independent. The only linear independent row is the first row. So the rank of V is 1. Likewise, the rank of its transpose is 1.

$$V = \begin{bmatrix} 1 & 2 & 1 \\ -1 & -2 & -1 \end{bmatrix} \qquad V^T = \begin{bmatrix} 1 & -1 \\ 2 & -2 \\ 1 & -1 \end{bmatrix}$$

Those additional rows or columns are not linearly independent and can be reconstructed by other rows or columns. With this property, although a matrix can be very large, its true information falls only on independent rows or columns.

Great, now we are ready to welcome the architecture of LoRA.

The architecture of LoRA

LoRA trains and stores the additional weight changes in a matrix while freezing all the pre-trained model weights. This design is in each layer of the Transformer architecture. You may immediately question that the number of weight changes is still the same as the number of parameters in the pre-trained model, so we are still training that vast number of parameters. Let me use notation to describe this more succinctly to explain LoRA’s solution. Let all the parameters of a LLM in the matrix W0 and the additional weight changes in the matrix ΔW, the final weights become W0 + ΔW. The authors of LoRA [1] proposed that the change in weight change matrix ΔW can be decomposed into two low-rank matrices A and B. LoRA does not train the parameters in ΔW directly, but the parameters in A and B. So the number of trainable parameters is much less. Hypothetically suppose the dimension of A is 100 * 1 and that of B is 1 * 100, the number of parameters in ΔW will be 100 * 100 = 10000. There are only 100 + 100 = 200 to train in A and B, instead of 10000 to train in ΔW.

Let me explain how it works. Figure 1 is a re-print of the image in the original paper [1]. Assume the pre-trained weight matrix is W0. Its dimension is d * k. It will be frozen during model training. The updated matrix is ΔW with dimension d * k as well. The update matrix ΔW can be decomposed into A and B. The dimension of A is r * k, and that of B is d * r. Upon the training completion, the W0 and ΔW will be stored separately. When a new input x enters the LoRA fine-tuned model, x will be multiplied with W and ΔW separately. Assume the dimension of x is 1 * d. So the dimension of x

multiplying with W becomes 1 * k, and the dimension of x multiplying with ΔW is also 1 * k. The two output vectors are summed coordinate-wise to become the final output h.

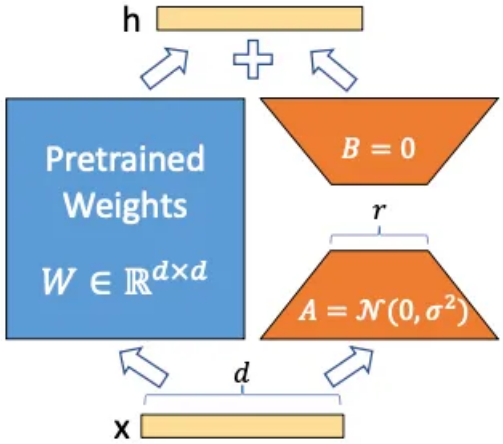


Figure 1: LoRA structure (image credit: [1])

I modify Figure 1 to Figure 2 to explain the low-rank solution better. The update matrix ΔW is the product of two low-rank matrices A and B.

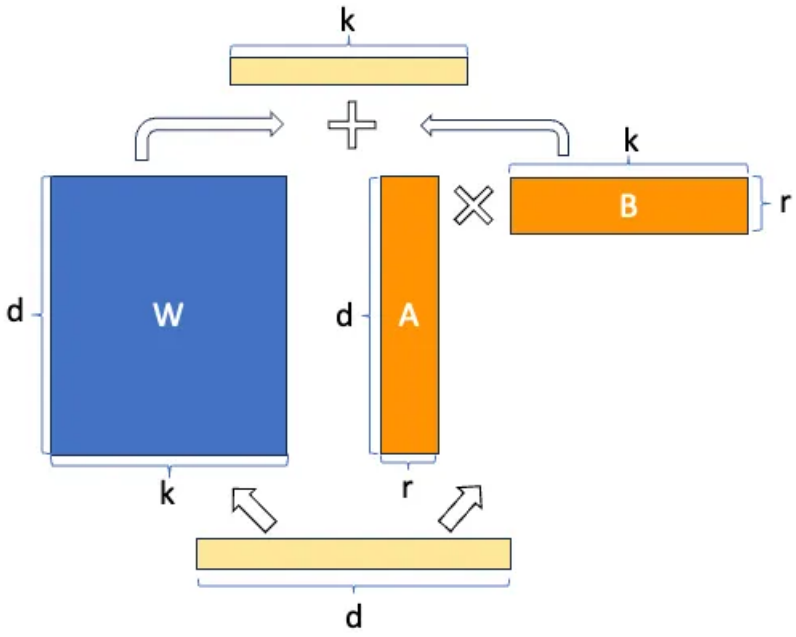


Figure 2: LoRA structure (image by author)

I will focus on the choice for the dimension *r*. The dimension *r* determines how small the rank of A and B can be. It is a hyper parameter. A small r will lead to fewer parameters to turn. While it will shorten the training time, it also could result in information loss and decrease the model performance as *r* becomes smaller.

Some of you may notice the above idea is a kind of matrix decomposition technique. Let me bring up the concept of Singular Vector Decomposition (SVD) here just for reference. It says a large matrix can be decomposed into three smaller matrices. The

three smaller matrices can “reconstruct” the large matrix without much information loss. Figure 3 illustrates the idea of SVD. Matrix A can be decomposed into three matrices. We can re-order the rows of matrix U in descending order. The blue area in matrix U represent the columns with values and the white area represents those with very small values. Although the dimension of matrix U is m by n, it can be truncated to be a m by k matrix (k<n). Similar matrix operations apply to the other two matrices. We will choose the value for k to be as small as possible but try not to lose too much information.

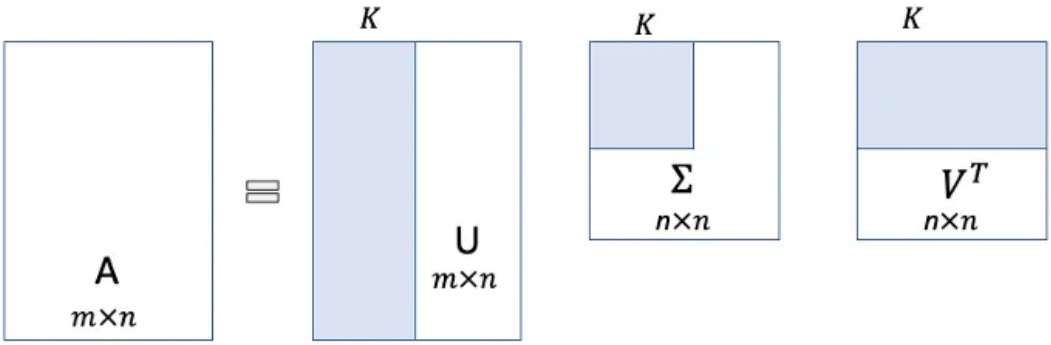


Figure 3: SVD (image by author)

Why do I bring in SVD? The choice for a small k makes SVD low-rank. Similarly, in LoRA fine-tuning we choose the value r to bring down the dimension of matrices A and B. A large r means more parameters to tune, i.e., a complex model, and a small r will not learn that much information. Therefore the choice of *r* needs to balance the model complexity and model performance.

The advantages of LoRA

Besides the no inference latency as explained above, let’s review more advantages of LoRA.

The first one is the reduction in the number of trainable parameters. The authors of [1] documented that “compared to GPT-3 175B fine-tuned with Adam, LoRA can reduce the number of trainable parameters by 10,000 times and the GPU memory requirement by 3 times.” A related advantage is its low-rank design. Because it only optimizes low-rank matrices, it results in the efficiency of model training.

The next advantage is modularity. You can build many small LoRA modules for different tasks. This advantage of modularity is similar to that of prefix-tuning. For example, you can build a LoRA module on a base LLM for text summarization, and another LoRA module on the same base LLM for question and answer. When the two fine-tuned models are deployed for real-time inferencing, you just need to load the same base model once. Given the physical size of an LLM at 100+ GB, this advantage cannot be ignored.

Maybe you are itching to work on LoRA now just like me. But wait, let me bring up a practical challenge for all fine-tuning tasks — memory and GPU constraints.

Fine-tuning is still GPU-intensive

As we are pleased with the strategy not to touch the billions of parameters in a pre-trained model, the challenges are not over yet. Fine-tuning LLMs also requires many GPUs. As this post documented, just to use (or called inference) the BLOOM-176B model would require 8x 80GB A100 GPUs. If we want to fine-tune BLOOM-176B, we will need 72 of these GPUs. Much larger models, like PaLM, would require even more resources.

There are a lot of collective efforts to reduce the use of many GPUs in model inferencing and model fine-tuning. One important development is the Int8 inference that can reduce the memory requirement by half without sacrificing model performance. So let me gently describe it.

The techniques to reduce the use of GPUs

We all know the size of a model is determined by the number of its parameters. We shall learn the fact that the precision for a parameter also critically determines the size of a model. A parameter is typically stored and presented in float32. Float32 (FP32) stands for the standardized IEEE 32-bit floating point representation. Figure 1 the 32-bit floating-point representation for the value 0.15625. The 32-bit representation for all parameters makes the model size huge.

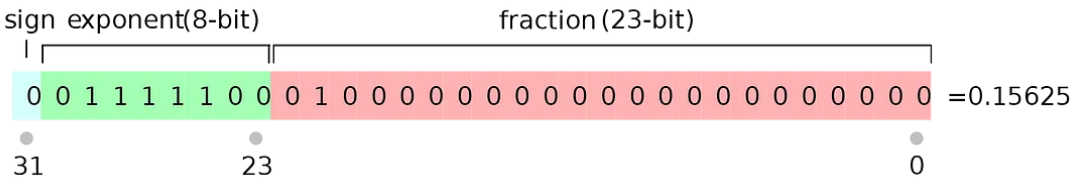


Figure 1: FP32 (Image credit: Here)

What can we do? The research community has come up with a new format, bfloat16 (BF16). The BF16 format requires 2 bytes in comparison with the FP32 format which requires 4 bytes. For this reason, the FP32 format is called full precision (4 bytes) and the BF16 format is half precision (2 bytes).

How can we maintain the precision of the parameters and reduce the requirement for GPUs? The strategy is to adopt a mixed precision approach. It keeps the parameters of a pre-trained model in FP32 as the main weights while performing the computation in BP16. In fine-tuning, the computation for the gradients is done in FP16, then used to update the main parameters.

However, the mixed use of the BP16 with the FP32 format still does not reduce the size of a model to a more manageable size. To remediate the challenge, this paper by Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer [3] introduces the 8-bit (Int8) quantization method. Because it is 8-bit, which is 1/4 of 32-bit, it potentially can reduce the size of a model to 1/4th. Obviously, it is not so easy, this will cause the degradation of the model and the model prediction suffers. What is the remedy? They discovered that in a matrix multiplication computation, the outliers are more important than the non-outliers. So the outliers can be stored in FP16 while the non-outliers in the 8-bit format or called int8. Then the outliers and non-outliers are put back together to receive the full result in FP16. This process has been developed in the `bitsandbytes` library.

All of the above management for GPUs have been included in the `accelerate` and `bitsandbytes` libraries. Later in the code we will do a pip install to install these libraries.

The code example

The base model and the dataset in the example are the same as those in the previous post "Fine-tune a GPT — Prefix-tuning". Let me give the description in case you visit this post first. This example will fine-tune the pre-trained model "bigscience/bloomz-7b1" with the special Twitter dataset that has the tweet comments and labels for "complaint", and "not a complaint". The entire Huggingface code example can be found here.

(1) Load the libraries

First, let's pip install the following libraries. As explained before, the `accelerate` and `bitsandbytes` libraries were developed to manage the use of GPUs.

```
!pip install -q bitsandbytes accelerate loralib datasets loralib
!pip install -q git+https://github.com/huggingface/transformers.git@main git+https
```

(2) The source model

The Open Pre-trained Transformers (OPT) were first released by Meta AI on May 3rd, 2022 in this repository. OPT is a suite of decoder-only pre-trained transformers ranging from 125M to 175B parameters. It was trained as a causal language model. OPT belongs to the same family of decoder-only models as GPT-3. We will use the 6.7 billion parameters "opt-6.7b" as the base model available here.

```
import os
os.environ["CUDA_VISIBLE_DEVICES"]="0"
import torch
import torch.nn as nn
import bitsandbytes as bnb
from transformers import AutoTokenizer, AutoConfig, AutoModelForCausalLM

model = AutoModelForCausalLM.from_pretrained(
    "facebook/opt-6.7b",
    load_in_8bit=True,
    device_map='auto',
)

tokenizer = AutoTokenizer.from_pretrained("facebook/opt-6.7b")
```

To use this base model, we still need to prepare it so we can do fine-tuning.

(3) Prepare the model for fine-tuning

The `left` library has the `prepare_model_for_kbit_training()` class to prepare the model for fine-tuning. (The class `prepare_model_for_kbit_training()` was expanded from the deprecated `prepare_model_for_int8_training()` class.)

```
from peft import prepare_model_for_kbit_training
model = prepare_model_for_kbit_training(model)
```

The above line of code actually does the following things. I show you the block of code just to explain. You still can use the above one line of code.

```
for param in model.parameters():
  param.requires_grad = False  # freeze the model - train adapters later
  if param.ndim == 1:
    # cast the small parameters (e.g. layernorm) to fp32 for stability
    param.data = param.data.to(torch.float32)

model.gradient_checkpointing_enable()  # reduce number of stored activations
model.enable_input_require_grads()

class CastOutputToFloat(nn.Sequential):
  def forward(self, x): return super().forward(x).to(torch.float32)
model.lm_head = CastOutputToFloat(model.lm_head)
```

First, it casts all the non-int8 modules to the full precision (`fp32`), as I have explained in the previous Section “The libraries to reduce the use of GPUs”. Then it adds a `forward_hook` to the input embedding layer. This enables gradient computation. It also

enables gradient checkpointing. The “forward hook” is a less-documented topic. This post provides an excellent overview.

The above descriptions require more background and may not appeal to you. Even so, it will not prevent you from the use of LoRA. Let’s continue.

(4) Apply LoRA

Let’s first understand how many parameters are trainable. The function below counts the number of trainable parameters.

```
def print_trainable_parameters(model):
    """
    Prints the number of trainable parameters in the model.
    """
    trainable_params = 0
    all_param = 0
    for _, param in model.named_parameters():
        all_param += param.numel()
        if param.requires_grad:
            trainable_params += param.numel()
    print(
        f"trainable params: {trainable_params} || all params: {all_param} || trai
    )
```

The `param.requires_grad` is a PyTorch function that indicates if a parameter is trainable or not. If it is True, a parameter is trainable. The function counts how many parameters are trainable. We are not using it yet. We will use this function together with LoRA to see the number of trainable and total parameters.

Huggingface has the “**P**arameter-**E**fficient **F**ine-**T**uning” (PEFT) library that includes Prefix-tuning and LoRA. The `LoraConfig` class will first define the LoRA technique. The `get_peft_model` class will wrap LoRA to the model and return the LoRA model. (You can find more details in the github).

```
from peft import LoraConfig, get_peft_model

config = LoraConfig(
    r=16,
    lora_alpha=32,
    target_modules=["q_proj", "v_proj"],
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM"
)
```

```
model = get_peft_model(model, config)
print_trainable_parameters(model)
```

Let me explain the above hyper-parameters in the `LoraConfig()` class.

- The `r` is the Lora attention dimension. It is set to 16. The default is 8.
- The `lora_alpha` is the alpha parameter for Lora scaling. It is set to 32. The default is 8.
- The `lora_dropout` is the dropout probability for Lora layers. The default is 0.0. Here is set to 0.05 or 5%.
- The `bias` is the type of bias. It can be "none", "all", or "lora_only". The default is "none".
- The `task_type` is the type of model. Here we are using a causal language model so the value is "CAUSAL_LM". If you are running a Sequence-Classification model, the value will be "SEQ_CLS". If it is a Sequence-to-Sequence-Language-Model, the value will be "SEQ_2_SEQ_LM". If it is a Token-classification model such as Named Entity Recognition (NER) and Part-of-Speech (PoS) tagging, it should be "TOKEN_CLS".
- The `target_modules` is the list of module names.

The output is: *trainable params: 8388608 || all params: 6666862592 || trainable%: 0.12582542214183376*. The LoRA technique lets us train a set of parameters that is only 12.5% of the original number of parameters.

Good. Up to now, we have specified LoRA and the model. Now let's turn to the data.

(5) Data source for fine-tuning

We will use the "Abirate/english_quotes" dataset available on this Huggingface page. that contains all the quotes retrieved from goodreads quotes. This dataset is ideal to train a model for text generation. Below is a snapshot of the data format. It has a quote column, an author column, and the tag column for multiple labels. Because of the tag column, the dataset is also ideal to train a model for multi-label text classification.

Split

train (2.51k rows)

quote (string)	author (string)	tags (sequence)
""Be yourself; everyone else is already taken.""	"Oscar Wilde"	["be-yourself", "gilbert-perreira", "honesty", "inspirational", "misattributed-oscar-wilde",…
""I'm selfish, impatient and a little insecure. I make mistakes, I am out of control and at times har…	"Marilyn Monroe"	["best", "life", "love", "mistakes", "out-of-control", "truth", "worst"]
""Two things are infinite: the universe and human stupidity; and I'm not sure about the universe.""	"Albert Einstein"	["human-nature", "humor", "infinity", "philosophy", "science", "stupidity", "universe"]
""So many books, so little time.""	"Frank Zappa"	["books", "humor"]

(6) Data pre-processing

Data will be tokenized for model training. We will use a very useful class `load_dataset` class in the `datasets` library. This library has many tools that can shape a dataset, create additional columns, or convert between features and formats. We will use its function `.map()` to apply to a dataset.

```
from datasets import load_dataset
data = load_dataset("Abirate/english_quotes")
data = data.map(lambda samples: tokenizer(samples['quote']), batched=True)
```

The above code applies tokenization to the samples. Notice that there is a batch processing parameter `batched=True`. It lets the operations (in our case, tokenization) be done in batches. If it is set to false, the data pre-processing operations will be done to each sample individually and will take more time.

(7) Model training

Model training requires many specifications. The Trainer class and the TrainingArguments class let you specify everything in model training. The `Trainer()` specifies

- the model to be trained in "model=",
- the data source in "train_dataset=",
- all important model specifications in "args=",
- and data processing in "data_collator=".

Let's read the code.

```
import transformers

trainer = transformers.Trainer(
    model=model,
```

```
        train_dataset=data['train'],
        args=transformers.TrainingArguments(
            per_device_train_batch_size=4,
            gradient_accumulation_steps=4,
            warmup_steps=100,
            max_steps=200,
            learning_rate=2e-4,
            fp16=True,
            logging_steps=1,
            output_dir='outputs'
        ),
        data_collator=transformers.DataCollatorForLanguageModeling(tokenizer, mlm=Fals
    )

    model.config.use_cache = False  # silence the warnings. Please re-enable for infe

    trainer.train()
```

The TrainingArguments is a big class that lets you specify the training process of a model. It contains more than 80 functions. The above hyparameters are all about speeding up the model training process.

The `per_device_train_batch_size` helps is an important hyperparameter that speed up the computation. The batch size is the number of samples to be passed through the model at once. Let me leverage an example in my book on image neural network training. Suppose there are 100,000 images bundled into 32 images per batch, there will be 100,000 / 32 = 3,125 batches. An epoch will go through all 3,125 batches. If you train the model without splitting data to batches, the model has to store all the error values together in the memory. To understand batch size better, you can take a look at "Transfer Learning for Image Classification — (6) Build and Fine-tune the Transfer Learning Model" or my book "Transfer learning for image classification".

The `gradient_accumulation_steps` help to speed up the computation. In a typical neural network model, the gradients are calculated for the whole batch at once thus is time-consuming. We can calculate the gradients iteratively in smaller batches by doing a forward and backward pass through the model, and accumulate the gradients. Once enough gradients are accumulated, we then run the optimization process. This helps to increase the overall batch size to speed up.

The `warmup_steps` specifies the number of steps in "warming up" an optimizer. In the first few steps of model optimization, a model is adjusted to learn the data. Here it is set at 100. This hyper-parameter informs the model that the first 100 steps are the warming-up steps. In the warmup steps, the learning rate is small. After the first 100 steps, the learning rate will go with any specified learning rate. This hyperparameter goes with `learning_rate`.

The `learning_rate` is a hyper-parameter to control the rate at which an algorithm updates the parameters. or learns the values of the parameters. Many readers are already familiar with the learning rate. If you like to get more explanation, you can take a look at "My lecture notes on random forest, gradient boosting, regularization, and H2O.ai".

The `fp16` is also called "half-precision". It is a floating-point computer number format that occupies 16 bits. When fp16 is true, it tells the model that we do not require ultra-precision and do not need to store so many decimal points. This can greatly reduce the size of a model.

Next, let's understand the DataCollator family. The data collators form batches for the data. The DataCollator family has many classes that prepare data for different types of models. These classes include "DefaultDataCollator()", "DataCollatorWithPadding()", "DataCollatorForTokenClassification()", "DataCollatorForSeq2Seq()", "DataCollatorForLanguageModeling()", "DataCollatorForWholeWordMask()", and so on. By their names, we know they are about forming data batches given different types of data and modeling tasks.

There is an interesting hyperparameter `mlm` for whether or not to use masked language modeling. If it is set to `False`, the target labels are the same as the inputs and the padding tokens are ignored (the padding tokens are set to -100). In our case, we do not train a mask language model, so it is set to `False`.

The `trainer.train()` executes the model building. This will be a time-consuming step. Let's just sit back and relax for the results.

(8) Save the model

Finally, we will save the model to be used next time. let's save the model by pushing it to Huggingface. You can also save the fine-tuned model locally.

```
from huggingface_hub import notebook_login
notebook_login()
model.push_to_hub("ybelkada/opt-6.7b-lora", use_auth_token=True)
```

(9) Load the model

When we load the model, we will load the base model and the LoRA parameters.

```
import torch
from peft import PeftModel, PeftConfig
from transformers import AutoModelForCausalLM, AutoTokenizer
```

```
peft_model_id = "ybelkada/opt-6.7b-lora"
config = PeftConfig.from_pretrained(peft_model_id)
model = AutoModelForCausalLM.from_pretrained(config.base_model_name_or_path, retu
tokenizer = AutoTokenizer.from_pretrained(config.base_model_name_or_path)

# Load the Lora model
model = PeftModel.from_pretrained(model, peft_model_id)
```

The code line `config=PeftConfig.from_pretrained(peft_model_id)` declares it is a LoRA fine-tuned model. The line `model=...` loads the base model and the line `tokenizer=...` loads the tokenizer. Finally, the line `model=PeftModel.from_pretrained(model, peft_model_id)` puts the base model and the LoRA parameters together.

(10) Model prediction

Now we can use the fine-tuned model in real-time production. Our model is fine-tuned by all quotes and can complete quotes. Suppose you type in “Two things are infinite: ” to ask the model to complete it.

```
batch = tokenizer("Two things are infinite: ", return_tensors='pt')

with torch.cuda.amp.autocast():
  output_tokens = model.generate(**batch, max_new_tokens=50)

print('\n\n', tokenizer.decode(output_tokens[0], skip_special_tokens=True))
```

The model returns:

“Two things are infinite: the universe and human stupidity; and I’m not sure about the universe. I’m not sure about the universe either.”

Not bad, right?

I hope this will help you to understand LoRA and how to apply LoRA to fine-tune a model.

Some caveats of LoRA over prefix-tuning

I have introduced prefix-tuning in “Fine-tune a GPT — Prefix-tuning”. You may be interested in the comparison between the two. The authors of [1] documented that prefix tuning is difficult to optimize. Its model training improvement is not expectable because the improvement is not monotonically increasing, as documented in its original paper. This means you have to experiment a wide range of parameters. Second,

in prefix-tuning some part of the sequence length will be reserved for the prefixes, which reduces the sequence length available. These minor disadvantages seem acceptable. I highlight them so we will be aware of them.

Summary

This post explained the case for fine-tuning, the architecture of LoRA, and its advantages. We learned why fine-tuning is still GPU-intensive and the most up-to-date solution. We went over a code example to build a LoRA fine-tuning model.

I believe this article can help your understanding of Fine-tuning a GPT. If you are interested in Large Language Models, you can find other articles in the series:

- Large Language Model Datasets
- Fine-tuning a GPT — Predix-tuning
- Fine-tuning a GPT — LoRA
- GenAI model evaluation metric — ROUGE

References

- [1] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, & Weizhu Chen. (2021). LoRA: Low-Rank Adaptation of Large Language Models. arXiv:2106.09685
- [2] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, & Sylvain Gelly. (2019). Parameter-Efficient Transfer Learning for NLP. arXiv:1902.00751
- [3] Tim Dettmers, Mike Lewis, Younes Belkada, & Luke Zettlemoyer. (2022). LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale. arXiv:2208.07339

Large Language Models | Gpt | Data Science | Python Programming | Artificial Intelligence