alpaca-lora: Experimenting with home-cooked Large Language Model

Wei Zheng • 7 Oct 2023

Wei, Oct 2023

A screen shot of failed/succeeded training runs.

Introduction

Large language models (LLM) became the buzzword in software development since the release ChatGPT. Its ability to have natural conversation with humans is just the tip of an iceberg. Enhanced with tools like LangChain or Semantic Kernel, LLM has the potential to completely change how users interact with software. In another word, LLM can create synergies among functionalities and data sources, and provide a more efficient and intuitive user experience.

For example, many people are already using AI-based content creation tools for their next viral videos. A typical video production pipeline includes scripting, logistics, storyboarding, editing and marketing, just to name a few. To streamline the process, a LLM could help content creators to do research while writing scripts, purchase props for the shoot, generate storyboards based on the script (might need stable diffusion for image generation), facilitate the editing process

and write eye-catching title/video descriptions for attracting views on social media. LLM is the core that orchestrate all of these, but there could be several concerns when incorporating LLM into a software product:

1. If I use OpenAI's API, will I become too dependent on this service? What if they bump up the price? What if they change service availability?
2. I don't like how OpenAI censors contents or provides non-constructive feedback on certain user inputs. (Or the other way around: I don't like how OpenAI censorship ignores certain things that are sensitive in my use case.)
3. If my clients prefer private cloud or on-premises deployment, what ChatGPT alternatives do I have?
4. I just want to have control. I need to customize the LLM and I want it cheap.

It is for these concerns, that I wonder if there could be an open-sourced equivalent of OpenAI's GPT models. Luckily, the wonderful open-source communities are already sharing some very promising solutions. I decided to try out alpaca-lora, a parameter-efficient fine-tuning method for training your own LLM. This blog post discusses the process, the problems that I encountered, how I solved them and what could happen next. If you also want to use the technique for training your own LLM, I hope the information can help.

Let's begin!

What is LLaMA, alpaca and LoRA?

LLaMA is a collection of foundation language models from Meta AI. These autoregressive models have parameter count ranging from 7B to 65B, and they can only be used for research purposes. Meta recently released LLaMA 2, which can be used commercially with some limits. What makes the LLaMA model family special is its training corpus and performance. Meta states that LLaMA is trained on publicly available datasets. Its 13B variant achieves performance comparable to GPT-3 (a 175B model) on most benchmarks.

alpaca is Stanford's attempt to build an instruction following LLM based on LLaMA. It works like this:

1. Ask a language model to generate instruction-input-response triplets.
2. Use the generated data for fine-tuning LLaMA.

After fine-tuning, LLaMA is able to follow user's instruction and respond accordingly. Here is a comparison between vanilla LLaMA's response and fine-tuned LLaMA response:

```
{
    "instruction": "Answer user's question.",
    "input": "Can you put a medium voltage circuit
        breaker in a low voltage switchgear? Why?
        Explain in detail.",

    "vanilla LLaMA": {
        "answer": "Yes, you can put a medium voltage
        circuit breaker in a low voltage switchgear.
        The medium voltage circuit breaker can be put
        in a low voltage switchgear because the medium
        voltage circuit breaker can be put in a low
        voltage switchgear because the medium voltage
        circuit breaker can be put in a low voltage
        switchgear because the medium voltage circuit
        breaker can be put....(repetition continues)"
    },

    "fine-tuned LLaMA": {
        "answer": "No, it is not possible to put a
        medium voltage circuit breaker in a low
        voltage switchgear. This is because low
        voltage switchgear is designed to handle
        voltages up to 1 kV, while medium voltage
        switchgear is designed to handle voltages from
        1 kV to 38 kV. The circuit breakers, contacts,
        and other components in low voltage switchgear
        are not suitable for the higher voltages and
        currents of medium voltage switchgear. Using a
        medium voltage circuit breaker in a low
        voltage switchgear could result in damage to
        the components and could pose a safety
        hazard."
    }
}
```


As you can see, fine-tuning changes LLM behavior quite drastically. The vanilla model is stuck in a repetition loop. While the fine-tuned model did not yield a 100% correct response, at least its answer is a resounding “No”. Fine-tuning is a necessary step for producing usable LLM. In many cases, deploying an open-sourced fine-tuned LLM is sufficient. But in some tailored business use cases, it may be preferable to fine-tune models on domain-specific datasets.

Alpaca’s biggest drawback is its resource requirement. Its GitHub page states that

> Naively, fine-tuning a 7B model requires about 7 x 4 x 4 = 112 GB of VRAM.

This is more VRAM than a A100 80GB GPU can handle. We can bypass the VRAM requirement using LoRA. LoRA works like this:

1. Select some weights in a model, such as the query projection weight W_q in a transformer model. Add (yes, arithmetic addition) adapter weights to the selected weights.
2. Freeze the original model, only train the added weight.

The added weight has some special properties. Inspired by this paper, Edward Hu et al. showed that for an original model weight $W_o \in R^{d \times k}$, you can produce a fine-tuned weight $W_o' = W_o + BA$ for downstream tasks, where $B \in R^{d \times r}$, $A \in R^{r \times k}$ and $r \ll min(d, k)$ is the "intrinsic rank" of the adapter weight. It is important to set a proper r for the adapter weight, since a smaller r lowers model performance, and a larger r increases adapter weight size without proportional performance gain. This technique is similar to truncated SVD, which approximates a matrix by decomposing it into several smaller matrices and only keeping a few largest singular values. Assume $W_o \in R^{100 \times 100}$, a full fine-tuning would change 10,000 parameters. LoRA fine-tuning with $r = 8$ would decompose the fine-tuned weight into 2 parts, $B \in R^{100 \times 8}$ and $A \in R^{8 \times 100}$, each part contains 800 parameters (1600 parameters in total.) The number of trainable parameters is reduced 6.25 times.

After transforming the model with LoRA, we got a model that has only ~1% trainable weights, yet its performance is greatly improved in certain domain. This would allow us to train 7B or 13B models on more accessible hardware such as RTX 4090 or V100.

The fine-tuning experiment

I ran the experiment on Huawei Cloud with a GPU accelerated VM instance (`p2s.2xlarge`, 8vCPU, 64GB RAM, 1x V100 32GB VRAM.) It is known that V100 does not support bfloat16 data type, and its tensor core does not support int8 acceleration. These 2 limits can slow down mixed precision training and cause numerical overflow during mixed precision training. We will keep this in mind for later discussion.

Quickly scanning the source code

`finetune.py` and `generate.py` are the core of the project. The first script fine-tunes LLaMA models, the second script uses the fine-tuned model to chat with users. Let's first take a look at the main flow of `finetune.py`:

1. loading a pretrained large foundation model

```
model = LlamaForCausalLM.from_pretrained(
    base_model,  # name of a huggingface compatible
        LLaMA model
    load_in_8bit=True,
    torch_dtype=torch.float16,
    device_map=device_map,
)
```

2. load the model's tokenizer

```
tokenizer = LlamaTokenizer.from_pretrained(base_model)
tokenizer.pad_token_id = (
    0
        # unk. we want this to be different from the
        eos token
)
tokenizer.padding_side = "left"  # Allow batched
        inference
```

3. based on the training template, prepare model inputs with two functions, `tokenize` and `generate_and_tokenize_prompt`.
4. create a LoRA adapted model using huggingface's PEFT

```
config = LoraConfig(
    r=lora_r,  # the lora rank
    lora_alpha=lora_alpha,  # a weight scaling factor,
        think of it like learning rate
    target_modules=lora_target_modules,  # transformer
        modules to apply LoRA to
    lora_dropout=lora_dropout,
    bias="none",
    task_type="CAUSAL_LM",
)
model = get_peft_model(model, config)
```

5. create a trainer instance and start training

```
trainer = transformers.Trainer(
    model=model,
    train_dataset=train_data,
    eval_dataset=val_data,
    args=transformers.TrainingArguments(
        ...
```

This is pretty simple. At the end, the script produces a model folder with checkpoints, adapter weights and adapter configuration.

Next, let's look at the main flow of `generate.py`:

1. load model and adapter weights

```
model = LlamaForCausalLM.from_pretrained(
    base_model,
    device_map={"": device},
    torch_dtype=torch.float16,
)
model = PeftModel.from_pretrained(
    model,
    lora_weights,
    device_map={"": device},
    torch_dtype=torch.float16,
)
```

2. specify generation config

```
generation_config = GenerationConfig(
    temperature=temperature,
    top_p=top_p,
    top_k=top_k,
    num_beams=num_beams,
    **kwargs,
)

generate_params = {
    "input_ids": input_ids,
    "generation_config": generation_config,
    "return_dict_in_generate": True,
    "output_scores": True,
```

```
    "max_new_tokens": max_new_tokens,
}
```

3. define functions for streaming and non-streaming generation mode:

```
if stream_output:  # streaming
    ...

# Without streaming
with torch.no_grad():
    generation_output = model.generate(
        input_ids=input_ids,
        generation_config=generation_config,
        return_dict_in_generate=True,
        output_scores=True,
        max_new_tokens=max_new_tokens,
    )
s = generation_output.sequences[0]
output = tokenizer.decode(s)
yield prompter.get_response(output)
```

4. Start a Gradio server to test the model:

```
gr.Interface(
    ...
```

The first attempt

The project's `README.md` stated that the following fine-tune settings produces a LLaMA 7B with performance comparable to Stanford alpaca. An "official" alpaca-lora weight was shared on huggingface.

```
python finetune.py \
    --base_model='decapoda-research/llama-7b-hf' \
    --num_epochs=10 \
    --cutoff_len=512 \
    --group_by_length \
    --output_dir='./lora-alpaca' \
    --lora_target_modules='[q_proj,k_proj,v_proj,o_proj
        ]' \
    --lora_r=16 \
    --micro_batch_size=8
```

However, in my experience, it did not yield a usable model. Running it on a V100 will encounter the following show-stopping issues:

1. loading the model with `load_in_8bit` causes data type error.
2. a binding script will cause PEFT to produce an invalid adapter. The invalid adapter makes no change to the original LLaMA model and only produces gibberish.
3. the `decapoda-research/llama-7b-hf` model apparently used the wrong tokenizer. Its pad token, bos token and eos token are different from LLaMA's official tokenizer.
4. as previously mentioned, V100 lacks proper support for int8/fp16 mixed training. This causes unexpected behaviors, such as `training loss = 0.0` and `eval loss = NaN`.

After digging around and wasting numerous VM hours, I found the necessary changes to make training work on a single V100.

```
...
# do not use decapoda-research/llama-7b-hf as
        base_model. use a huggingface LLaMA model that
        was properly converted and has a correct
        tokenizer, e.g., yahma/llama-7b-hf or
        huggyllama/llama-7b.
# decapoda-research/llama-7b-hf is likely to cause
        overflow/underflow on V100. train loss goes to
        0 and eval loss becomes NaN. using yahma/
        llama-7b-hf or huggyllama/llama-7b somehow
        mitigates this issue
model = LlamaForCausalLM.from_pretrained(
    base_model,
    load_in_8bit=True,  # only work for 7B LLaMA. On a
        V100, set True to save some VRAM at the cost
        of slower training; set False to speed up
        training at the cost of more VRAM / smaller
        micro batch size
    torch_dtype=torch.float16,
    device_map=device_map,
)
...
# comment out the following line if load_in_8bit=False
model = prepare_model_for_int8_training(model)
...
# set legacy=False to avoid unexpected tokenizer
        behavior. make sure no tokenizer warning was
        raised during tokenizer instantiation
tokenizer = LlamaTokenizer.from_pretrained(base_model,
        legacy=False)
```

```
...
# the following binding script results in invalid
        adapter. simply comment them out
old_state_dict = model.state_dict
model.state_dict = (
    lambda self, *_, **__: get_peft_model_state_dict(
        self, old_state_dict()
    )
).__get__(model, type(model))
...
# if load_in_8bit=True, need to cast data type during
        training
with torch.autocast('cuda'):
    trainer.train(resume_from_checkpoint=resume_from_ch
        eckpoint)
```

After making these changes, these training arguments produce a usable model.

```
{
    "args": [
        "--base_model",
        "yahma/llama-7b-hf",
        "--num_epochs=10",
        "--data_path",
        "yahma/alpaca-cleaned",
        "--output_dir",
        "./your output dir",
        "--cutoff_len=512",
        "--
        lora_target_modules=[q_proj,k_proj,v_proj,o_pr
        oj]",
        "--lora_r=16",
        "--micro_batch_size=2",
        "--batch_size=16",
        "--wandb_project",
        "your wandb project name"
    ],
}
```

Note that the training script not only saves the “best” model (checkpoint with the lowest eval loss), but also a couple checkpoints. I used wandb to log training/ eval losses for further analysis.

The first observation

I throw some random questions at the fine-tuned model. It produces some reasonable response. Here is one example:

> Instruction: Answer user's question carefully.
>
> Input: What happens when you rack out a switchgear circuit breaker truck?
>
> Response: When a switchgear circuit breaker truck is "racked out," it means that all of the circuit breakers in the truck have been opened and the truck is in a non-energized state. This is typically done when the truck is being serviced, repaired, or inspected. Once the circuit breakers have been racked out, the truck is safe to work on and can be re-energized once the necessary maintenance or repairs have been completed.

This example queries the LLM's internal knowledge. The model seemed to know a lot about switchgears. However, when I tried to empirically test its reasoning ability, it produced a very weird result:

> Instruction:
>
> Given the information below about a project, answer questions. Chinese-Llama-2 project aims to enhance the understanding, generation, translation capabilities of the large language model Llama-2 in Chinese language. With the application of methods such as LoRA fine-tuning, full-parameter instruction fine-tuning, and secondary pre-training, we cordially invite you to download and utilize the associated datasets, training guides, and model parameters.
>
> Input: What methods are used in this project?
>
> Response: Chinese-Llama-2 project aims to enhance the understanding, generation, translation capabilities of the large language model Llama-2 in Chinese language. With the application of methods such as LoRA fine-tuning, full-parameter instruction fine-tuning, and secondary pre-training, we cordially invite you to download and utilize the associated datasets, training guides, and model parameters.

As you can see, the model likes to repeat the prompt. I call it *prompt-repeater* model, and other developers had also been reporting this behavior in the repo issues. Its ability to answer questions only gets better with more carefully

designed prompts. This behavior is not what we want in a production system, as we cannot guarantee prompt effectiveness across different models. The models have to be less sensitive to prompts. We want to somehow improve the performance of this LLM.

In the next session, I will discuss what caused this problem and how to improve fine-tuning result.

The second attempt and the (somewhat) success

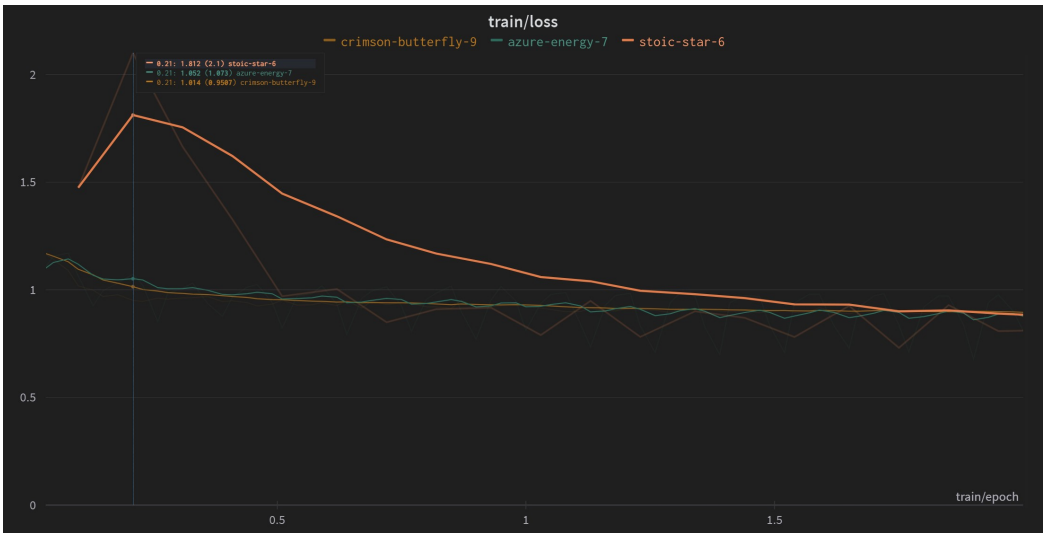
Here are 3 things I tried to improve fine-tuning result:

1. Mask out loss on prompts (helps avoiding prompt repetition)
2. Turn off `group-by-length` option (helps improving performance, makes loss curve look smoother)
3. Do not trust the eval loss curve. Use a checkpoint that has a lower training loss, even though its eval loss could be higher than the "best" checkpoint. (helps improving performance, since eval loss is not the best matrix here)

Let's explain these 3 points one by one.

Mask out loss on prompts

I was looking for the causes of prompt repetition until I found this post and the official lora weights commit message. They suggested that prompts should be excluded in loss calculation. Basically, you don't want to encourage the model to output prompt tokens. Masking out the prompts during training would not encourage the model to repeat prompt tokens. The chart below explains this: out of the 3 training runs, `stoic-star-6` is the only run that did not mask out prompts during training. Its training loss is thus higher at the beginning. I suspect that if **a)** prompts are not masked out when calculating loss, and **b)** training is insufficient, the model will be more likely to repeat prompts rather than following instructions.



masked loss comparison

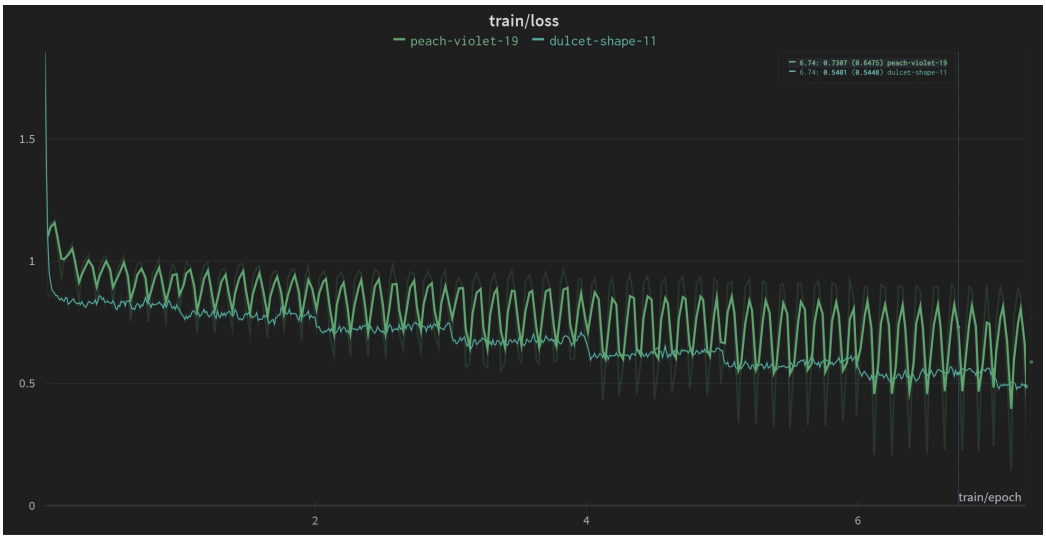
In the source code, loss masking is done by setting prompt tokens to -100:

> Tokens with indices set to `-100` are ignored (masked), the loss is only computed for the tokens with labels in `[0, ..., config.vocab_size]`.

Turn off `group-by-length` option

`group-by-length` option allows huggingface's `Trainer` to group inputs of similar length into batches. This helps to save VRAM usage when padding input sequences. However, it would greatly reduce sample variance within a single batch. During the training process, we generally prefer exposing the model to a variety of training samples. Setting `group-by-length` to `False` reduces sample variation. It also causes loss fluctuation during training (For example, two consecutive batches have padded lengths of 10 and 50. The shorter batch has lower loss, and the longer batch has higher loss. This results in an oscillating loss curve, as shown in the figure below).

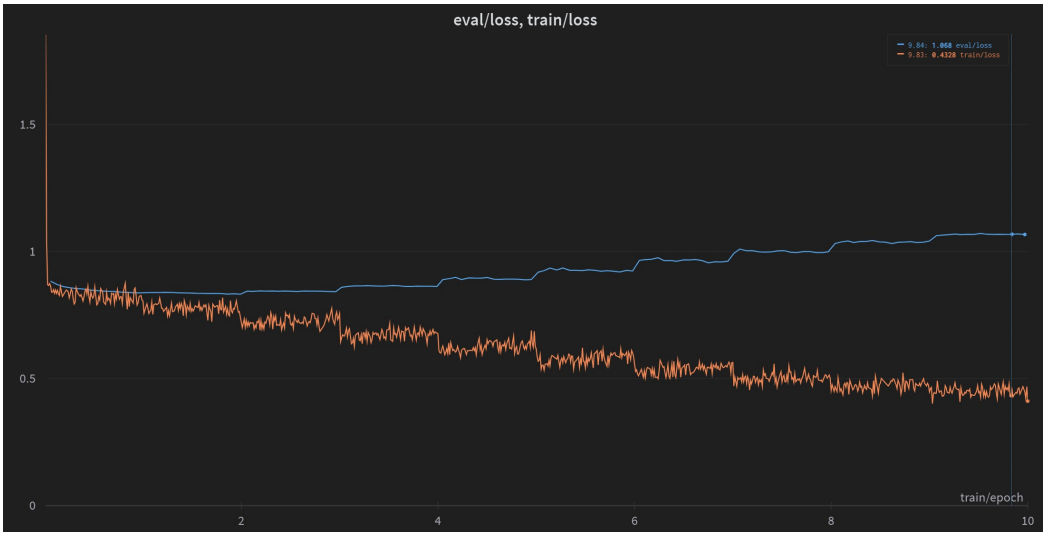
On the other hand, since `group-by-length` reduces in-batch sample variation, I suspect the model performance could also suffer from this. The figure below compares training loss with or without `group-by-length`. It is clear that the averaged loss is higher for run `peach-violet-19`, which has `group-by-length` enabled.



group_by_length_comparison

Do not trust the eval loss curve

I observed that training loss and eval loss are diverged in all of my runs. Here is an example:



llm_eval vs train loss

In this example, the optimum checkpoint occurs around `epoch=2`. If you load the optimum checkpoint as suggested by the original repo, you are most likely to get a *prompt-repeater* model as mentioned in the previous session.

By loading a model checkpoint near the end of training, I got a more performant model. I asked the same question as before and the model returned the correct answer:

> Instruction:
>
> Given the information below about a project, answer questions. Chinese-Llama-2 project aims to enhance the understanding, generation, translation capabilities of the large language model Llama-2 in Chinese language. With the application of methods such as LoRA fine-tuning,

> full-parameter instruction fine-tuning, and secondary pre-training, we cordially invite you to download and utilize the associated datasets, training guides, and model parameters.
>
> Input: What methods are used in this project?
>
> Response: The Chinese-Llama-2 project uses methods such as LoRA fine-tuning, full-parameter instruction fine-tuning, and secondary pre-training.

But judging by the evaluation loss, this model should perform worse. What causes the dilemma?

I think evaluation loss is not a good matrix for measuring large language model performance. LLaMA uses `CrossEntropyLoss` for training and evaluation loss:

```
# modelling_llama.py from transformers library
...
# forward function under LlamaForCausalLM class
if labels is not None:
    # Shift so that tokens < n predict n
    shift_logits = logits[..., :-1, :].contiguous()
    shift_labels = labels[..., 1:].contiguous()
    # Flatten the tokens
    loss_fct = CrossEntropyLoss()
    loss = loss_fct(shift_logits.view(-1, self.config.v
        ocab_size), shift_labels.view(-1))
```

When testing on evaluation set, a model could produce the same answer with different wording:

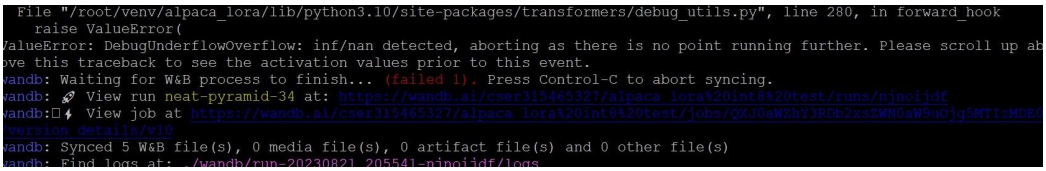
```
{
    "evaluation prompt": "What is 1 + 3?"
    "evaluation answer": "4."
    "prediction answer": "The answer is 4."
}
```

Both answers are correct, but if the prediction answer is not an exact match of the evaluation answer, evaluation loss will be high. In this case, we need a better evaluation matrix to measure the model's performance. We will worry about proper evaluation later. For now, let's assume the best model is the one with the lowest training loss.

Going beyond 7B

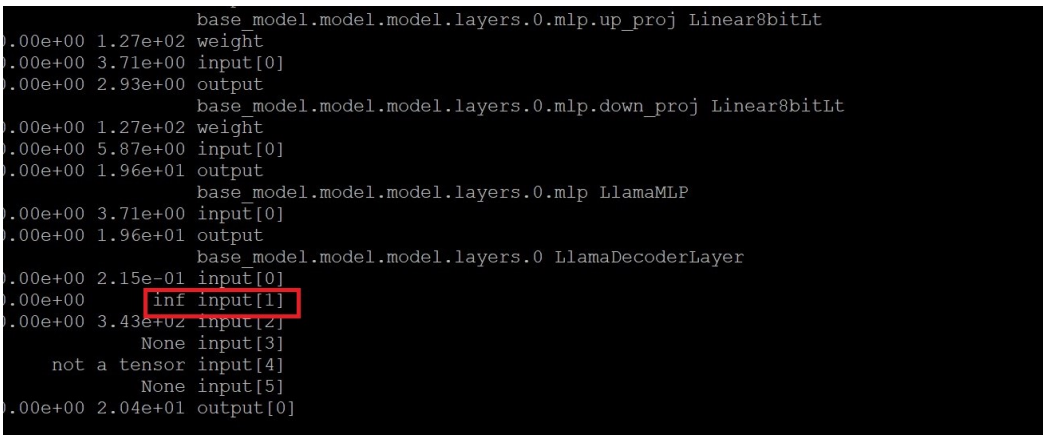
I tried fine-tuning a 13B model on V100. While V100 can handle both int8 and fp16 training on a 7B model, it simply cannot handle int8 training on 13B model. If `load_int_8bit = True`, the 13B model will produce `training_loss = 0.0`. We can use some debugging tools to understand why this happens (**spoiler alert:** it is caused by overflow/underflow).

I used huggingface's `DebugUnderflowOverflow` tool to inspect parameters during training. In the first forward pass, it detected inf/nan values:



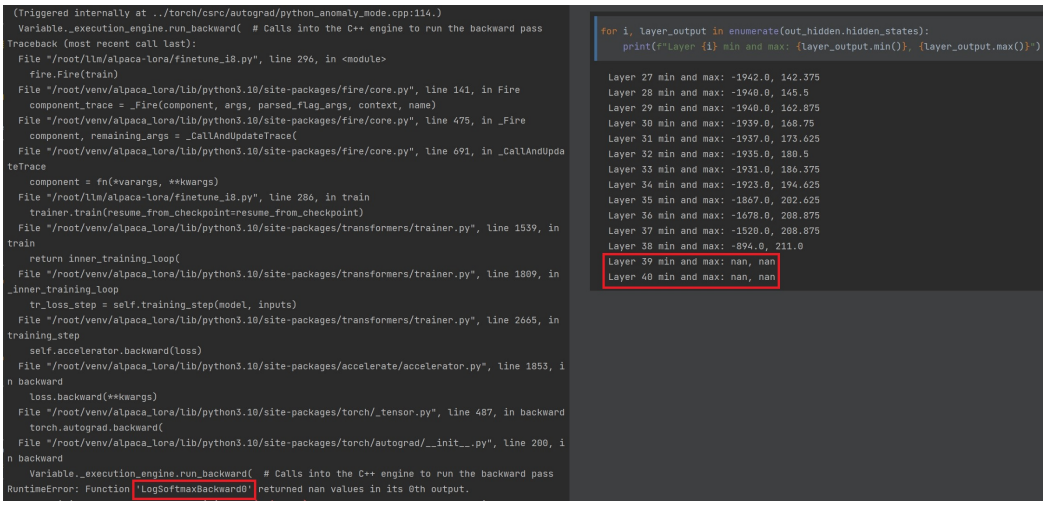
inf nan detected

More specifically, `DebugUnderflowOverflow` caught negative infinity values in the 2nd input of `LlamaDecoderLayer`, as shown in the figure below. The 2nd input is `attention_mask`. I dived a bit deeper and found out that the `attention_mask` is supposed to have very large negative values for padding elements. Coincidentally, the negative infinity values are at the beginning of every sequence. This observation leads me to believe that negative infinity values are supposed to occur at this layer. Further investigation also showed that the infinity values did not cause more infinity values in the next few layers. Therefore, overflow at `LlamaDecoderLayer` is most likely not the root cause of abnormal training loss.



inf decoder layer

Next, I inspected the outputs of each layer. It was very clear that the outputs of the final layers are overflowing, as shown in the figure below. I believe that this is caused by the limited precision of int-8 weights (or the limited range of `float16`. It is likely that `bfloat16` could avoid this problem).



layer output anomaly

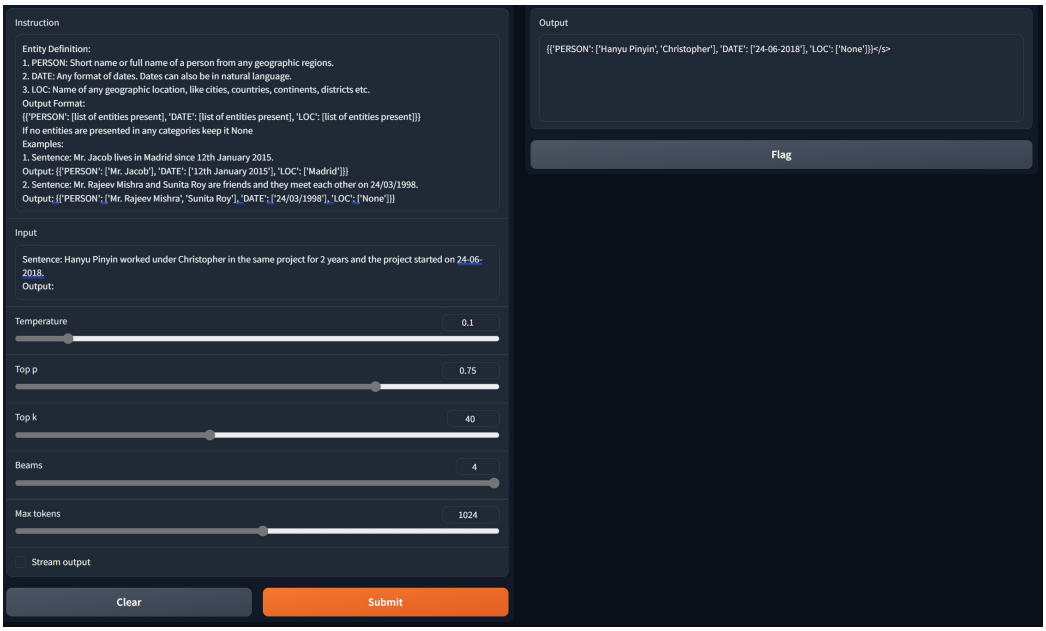
To solve the overflow problem, I used float16 during training. V100 does not have enough VRAM for training a 13B model unless some tricks were used. Hugging Face DeepSpeed provides several methods, such as CPU offloading, to reduce training VRAM usage. But the simplest trick is to enable gradient checkpointing by calling `model.gradient_checkpointing_enable()` before training starts.

Gradient checkpointing trades off training speed for less VRAM usage. Typically, during the forward pass, activations were computed and stored in memory for use during backward pass. This takes up additional memory. However, with gradient checkpointing, instead of storing activations during the forward pass, they are re-calculated during backward pass, thus saving VRAM. Here is a nice article about this technique.

I was able to train Llama 13B with float16 and gradient checkpointing enabled:

```
python finetune.py \
    --base_model=yahma/llama-13b-hf \
    --num_epochs=10 \
    --output_dir 'your/output/dir' \
    --
lora_target_modules='[q_proj,k_proj,v_proj,o_proj]' \
    --cutoff_len=1024 \
    --lora_r=16 \
    --micro_batch_size=4 \
    --batch_size=128 \
    --wandb_project 'alpaca_lora_13b' \
    --train_on_inputs=False
```

The 13B model can handle some advanced tasks such as name entity recognition. I use an example prompt for test and this is the 13B model's accurate response:



name entity recognition

All’s good! This is an exciting start. The model allows us to create complex applications with LangChain.

At this point, we are still missing tools for automatic model evaluation. We can use Language Model Evaluation Harness to evaluate our models on many test cases, or even create our own test cases. It is the same tool that Hugging Face uses for its Open LLM Leaderboard. While evaluation is a crucial aspect of LLM development, this article focuses solely on the training process. I may discuss evaluation in a future article.

Summary

In this article, we introduced the concept of large foundation models (LFMs) and several fine-tuning methods that make LFMs behave as desired. We then focused on LoRA, a parameter-efficient method for fine-tuning LFM, and explained the fine-tuning code as well as performance improvement techniques. Finally, we went a step further and successfully trained a Llama 13B model on a V100 GPU. Although the 13B model training ran into some problems, we found that these problems were imposed by hardware limitations and presented solutions. At the end, we got a fine-tuned LLM that works, but we have not yet quantitatively evaluated the LLM’s performance.

About the author

Hello there! My name is Wei. I am a dedicated problem solver, Senior AI Specialist & Analytics project lead at ABB and machine learning Google Developer Expert. I hold an M.S. in Mechanical Engineering from the University of Minnesota Twin Cities and a B.S. degree in Mechanical Engineering from the University of Illinois at Urbana-Champaign.

My tech stack focus on python / C# programming, computer vision, machine learning, algorithms, and micro-services, but I also have a wide span of interests such as game development (Unity), front/back-end development, technical leadership, tinkering with single board computers and robotics.

I hope this article can help people in some way. Thanks for reading, and happy problem-solving!