

[W&B Fully Connected](#) > [Articles](#)

How to Fine-tune an LLM

Part 3: The HuggingFace Trainer

Exploring how to get the best out of the Hugging Face Trainer and subclasses

[Thomas Capelle](#)

Created on November 9 | Last edited on February 2

In [our previous article](#), we fine-tuned a pre-trained Llama2 model on the Alpaca instruction dataset using almost pure PyTorch. This was an excellent exercise to understand what is happening under the hood but there are more straightforward ways to achieve this.

For example, the HuggingFace ecosystem has a specific library to help us with fine-tuning instruction models: [trl](#). Initially developed for [Reinforcement Learning](#) techniques like DPO, it has most of what we need to perform instruction tuning. We can also leverage this library with [peft](#) and [bitsandbytes](#) to achieve efficient parameter fine-tuning.

If you missed the prior two installments in this series, we recommend checking those out before you jump in:

**How to
Fine-Tun...**

Learn how to
fine-tune an...

**How to
Fine-Tun...**

In part 1, we
prepped ou...

Let's get into it.



Llamas and Alpacas love HuggingFace

Today, we're going to perform instruction tuning leveraging the HuggingFace ecosystem. These set of tools, combined with [W&B experiment tracking](#), will make our fine-tuning much easier!

More specifically, we'll fine-tune a Llama2 model on a variant of the Alpaca dataset. We'll also compare different fine-tuning approaches like LoRA and freezing the model. Finally, we'll evaluate the fine-tuned model by leveraging GPT-4.

What we'll be covering

[Instruction Fine-Tuning Using the trl Library](#)

[Preprocessing: ConstantLengthDataset](#)

[\(Optional\) Preprocessing: Masking Instructions by Using the](#)

[DataCollatorForCompletionOnlyLM](#)

[A Trainer for Fine-tuning: SFTTrainer](#)

[Sampling From the Model During Training](#)

[Parameter Efficient Fine-Tuning \(PEFT\)](#)

[Using LoRA with HuggingFace PEFT Library](#)

[Results of the LoRA Training](#)

[Fitting Big Models on Small GPUs: A Memory Comparison](#)

[Making Forward Pass Faster: 8-Bit/4-Bit Forward](#)

[Conclusions and Final Remarks](#)

Instruction Fine-Tuning Using the trl Library

Preparing the data for instruction, fine-tuning, and managing to [train an LLM](#) can be very tricky. The [trl library](#) from the HuggingFace ecosystem makes this task much more accessible with



The previously implemented fine-tuning with packing could be done in just a couple of lines by leveraging the `SFTTrainer`, a thin wrapper around `transformers.Trainer`. Under the hood, the `SFTTrainer` will prepare the dataset by applying the prompt formatting function, and pack the data to the desired `max_seq_length`.

```
from trl.train import SFTTrainer, TrainingArguments

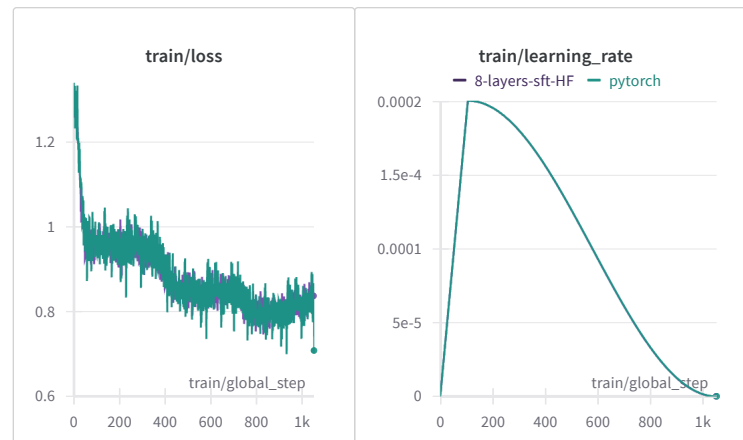
os.environ["WANDB_PROJECT"] = "alpaca_ft" # name your W&B project
os.environ["WANDB_LOG_MODEL"] = "checkpoint" # log all model checkpoints

training_args = TrainingArguments(
    report_to="wandb", # enables logging to W&B 🤖
    per_device_train_batch_size=16,
    learning_rate=2e-4,
    lr_scheduler_type="cosine",
    num_train_epochs=3,
    gradient_accumulation_steps=2, # simulate larger batch sizes
)

trainer = SFTTrainer(
    model,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    packing=True, # pack samples together for efficient training
    max_seq_length=1024, # maximum packed length
    args=training_args,
    formatting_func=formatting_func, # format samples with a model specific
)

trainer.train()
```

We get almost identical results as the ones obtained with the "pure-PyTorch" training loop but without (almost) any coding!



👉 We have the training loss from the HF Trainer implementation compared to our Part 2 pure PyTorch training loop

👉 We also log some generations



2
3
<div> <div>≡ ≡ = - < <</div> <div>1</div> <div>- 3 of 10 > ></div> </div> <div>Export as CSV Columns... Reset t</div>

Next, let's dive under the hood of the `SFTTrainer` class:

Preprocessing: ConstantLengthDataset

As we are training the model on instructions of different lengths, an excellent technique to optimize the training performance and reduce padding is to concatenate multiple sequences together. This enables the efficient batch of the tokens together and maximizes the model's throughput. You can read more about how this is implemented in the [first article of this series](#).

When you pass the flag `packing=True` to the trainer, what is happening under the hood is that a `ConstantLengthDataset` is being created so we can iterate over the dataset on fixed-length sequences.

The class is very similar to the packing we implemented in [Part 1](#) but has good compatibility with large datasets and is lazy, creating the sequences on the fly. You can use this class as a standalone tool and pass this to the `SFTTrainer` or let the trainer create the packed datasets for you. Depending on your use case, you may want to pre-compute the dataset and maybe store the results; another reason you may want to use the dataset class is to have control over the `DataLoader` creation. This may be useful if heavy pre-processing or pre-tokenization is necessary.

```
def prompt_no_input(row):
    return ("Below is an instruction that describes a task. "
           "Write a response that appropriately completes the request.\n\n"
           "### Instruction:\n{instruction}\n\n### Response:\n{output}")

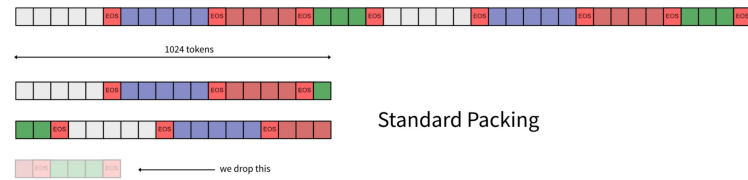
def prompt_input(row):
    return ("Below is an instruction that describes a task, paired with an input. "
           "Write a response that appropriately completes the request.\n\n"
           "### Instruction:\n{instruction}\n\n### Input:\n{input}\n\n### Response:\n{output}")

def create_alpaca_prompt(row):
    return prompt_no_input(row) if row["input"] == "" else prompt_input(row)

train_dataset = ConstantLengthDataset(
    tokenizer,
```



You can also pass a `concat_token_id` to be used between consecutive instructions. By default, the `eos_token` from the tokenizer will be used, but depending on the model, you may want to change this. Also, you can use this class to pack a raw text dataset, not necessarily for instruction training.



When you pass the `packing=True` argument to the `SFTTrainer`, under the hood it creates a `ConstantLengthDataset` like this and then creates the `DataLoaders`. It saves you the need to manually preprocess your dataset.

(Optional) Preprocessing: Masking Instructions by Using the `DataCollatorForCompletionOnlyLM`

Something we didn't do in our [previous article](#) is masking the tokens of the instruction (sometimes called instruction masking). This technique consists of setting the token labels of the instructions to -100 (the default value that the CrossEntropy PyTorch function ignores). This way, you're only back-propagating on the completions. Implementing this is not very complicated, but it can be tricky. Let's play with a simple example to understand what this means:

```
sample = {"instruction": "### Instruction: List three fruits\n### Re",
          "output": "- Apple\n- Orange\n- Strawberry"}

instruction = tokenizer(sample["instruction"])[ "input_ids" ]
output = tokenizer(sample["output"], add_special_tokens=False)[ "input_ids" ]

def printl(mylist):
    return print(*mylist, sep=' ')

printl(instruction)
printl(instruction + output)
# 1 835 2799 4080 29901 2391 2211 285 21211 13 2277 29937 13291 29901
# 1 835 2799 4080 29901 2391 2211 285 21211 13 2277 29937 13291 29901
```

- The first print statement is the **tokenized instruction**
- The second line is the **tokenized instruction + output**

Usually, the labels to train our model would be just the same as the inputs (to compute the loss, the model internally shifts labels by one before calling the Cross-Entropy loss. This is standard in autoregressive classification tasks like next token prediction), so a preprocessed sample would be:

```
{"input_ids": instruction + output, "labels": instruction + output} #
```



on the instruction. Remember, this is an auto-regressive model; completing a partial instruction without context is just guessing.

The trl library can enable this using a particular data collator:
[DataCollatorForCompletionOnlyLM](#).

The data collator function is responsible to combine multiple samples from the dataset and create a batch. In this case, the batch is created by the default transformers `DataCollatorForLanguageModeling` and then this will mask (set to -100) the tokens that belong to the instruction. The clever thing about this technique is that they tokenize the full sequences and then they mask on a token level. They can do this efficiently using a function like `np.where`



💡 Currently Packing is not supported with Instruction Masking. This is supported in other libraries like [Axolotl](#). From recent released models like Zephyr, it appears that using just packing without instruction masking gives good results already. More research on this probably will be available on the next couple of months. For the moment, I would just use packing without instruction masking if using trl as it gives a nice speed boost.

A Trainer for Fine-tuning: SFTTrainer

The transformers [Trainer](#) class is packed with features and parameters! This class is responsible for packing your model and datasets together so you can train a model. It also supports evaluation and callbacks.

It can sometimes be hard to find what exact parameters you need to input to get the most out of your training, so let's break it down. The Trainer can infer a lot from the model you are using, so most of the parameters you need to set up for a specific model are pulled from the hub when you select a specific pre-trained model. For instance, if you use the `'meta-llama/Llama-2-7b-hf'` model, the Trainer will pull the appropriate tokenizer and model capabilities. The minimal setup you need to do is passing an instantiated model or model name and a dataset.

For our specific use case: **Instruction fine-tuning**, HuggingFace provides a sub-class of the trainer, the `SFTTrainer` (Supervised Fine-Tuning Trainer), in the [trl library](#). This is what we will use from now on 📌

This tool makes your instruction fine-tuning life much easier. It enables you to format your dataset for instruction training on the fly, pass the appropriate formatting function, and the batches will be formatted accordingly. Let's take a moment and admire how simple the code becomes by using this `SFTTrainer`:

```
trainer = SFTTrainer(
```



```
trainer.train()
```

It's a good idea to provide parameters in the form of `TrainingArgs`; this way, one can control how the optimization process will be set up. There are many training arguments available that are well documented in the [Transformers library](#). Most of the optimization techniques underlying PyTorch are available here so that we can enable the same tricks as before, such as:

- Gradient checkpointing: Save memory by storing activations
- Gradient accumulation: Simulates larger batch sizes
- Learning rate schedulers: cosine, linear, etc.
- Mixed precision using **bfloat16**

To start, let's configure the training as we did before:

```
batch_size = 16
gradient_accumulation_steps = 2
num_train_epochs = 3

os.environ["WANDB_LOG_MODEL"] = "end" # the model will be uploaded to wandb

training_args = TrainingArguments(
    output_dir="./output/",
    report_to="wandb", # this tells the Trainer to log the metrics
    per_device_train_batch_size=batch_size,
    per_device_eval_batch_size=batch_size//2,
    bf16=True,
    learning_rate=2e-4,
    lr_scheduler_type="cosine",
    warmup_ratio = 0.1,
    gradient_accumulation_steps=gradient_accumulation_steps,
    gradient_checkpointing=True,
    evaluation_strategy="epoch",
    num_train_epochs=num_train_epochs,
    # logging strategies
    logging_strategy="steps",
    logging_steps=1,
    save_strategy="epoch", # saving is done at the end of each epoch
)
```

The Trainer supports logging to Weights & Biases out of the box! Pass the argument `report_to="wandb"` or start a run before calling `trainer.train()`. This is very handy as plenty of training libraries and codebases build on top of the Trainer, so **we get logging for free** on plenty of training libraries and codebases.

On my codebase, I prefer manually creating a run before calling train (by calling `wandb.init()` before calling `train`); this way I can store other parameters on the config. Also, for more extended training sessions, it is a good idea to sample from the model regularly to get a sense of how the training is going and debug potential issues. This can be achieved by passing a callback to the trainer.

Sampling From the Model During Training



model not learning as expected. To do this, we can use the `model.generate` method and store the prediction in a [W&B Table](#). We can achieve this behavior by subclassing the `WandbCallback` and creating the table when calling `evaluate`.

```
class LLMSampleCB(WandbCallback):
    def __init__(self, trainer, test_dataset, num_samples=10, max_new_tokens=10):
        """A CallBack to log samples a wandb.Table during training"""
        super().__init__()
        self._log_model = log_model
        self.sample_dataset = test_dataset.select(range(num_samples))
        self.model, self.tokenizer = trainer.model, trainer.tokenizer
        self.gen_config = GenerationConfig.from_pretrained(trainer.pretrained_model_name_or_path,
                                                           max_new_tokens=max_new_tokens)

    def generate(self, prompt):
        tokenized_prompt = self.tokenizer(prompt, return_tensors='pt')
        with torch.inference_mode():
            output = self.model.generate(tokenized_prompt, generation_config=self.gen_config)
        return self.tokenizer.decode(output[0][len(tokenized_prompt['input_ids']):])

    def samples_table(self, examples):
        """Create a wandb.Table to store the generations"""
        records_table = wandb.Table(columns=["prompt", "generation"])
        for example in tqdm(examples, leave=False):
            prompt = example["text"]
            generation = self.generate(prompt=prompt)
            records_table.add_data(prompt, generation, *list(self.gen_config.get_vocab().values()))
        return records_table

    def on_evaluate(self, args, state, control, **kwargs):
        """Log the wandb.Table after calling trainer.evaluate"""
        super().on_evaluate(args, state, control, **kwargs)
        records_table = self.samples_table(self.sample_dataset)
        self._wandb.log({"sample_predictions": records_table})
```

You'll also need a `test_dataset` formatted without the completions (answers) so the model is prompted to complete with an answer. The general format of the training script should look like this:

```
trainer = SFTTrainer(...)

wandb.init(project=..., config=config)

# we instantiate the W&B callback with the trainer object and the dataset
wandb_callback = LLMSampleCB(trainer, test_dataset, num_samples=10, max_new_tokens=10)
trainer.add_callback(wandb_callback)

# we then continue as regular
trainer.train()

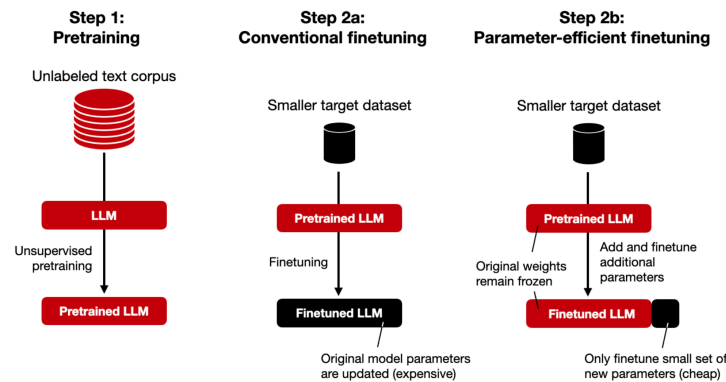
# good practice to end your run
wandb.finish()
```

💡 You can check the code for this [experiments here](#)



Remember that we only trained the last 8 model layers (around 1.7b parameters), but to do a full fine-tune and train the whole model, you would need to reduce the batch size, which would barely fit the 40GB. We already used a potent trick to save memory: [Gradient Checkpointing](#). This technique saves a ton of memory and enables you to train these big models without using [Model parallelism](#). The Trainer already has multiple memory-saving methods built-in; you can read more on this [great article](#) from the HuggingFace team.

If one wants to fit larger models on smaller GPUs, the ideal is combining multiple tricks but understanding what trade-off one is paying. For instance, **Gradient Checkpointing** saves memory, but training becomes slower.



Illustrating different types of fine-tune by [Sebastian Raschka](#)

The most powerful way of saving memory is by making the model smaller. Building on this idea and currently used extensively for model fine-tunings is **Parameter Efficient Fine-Tuning** or **PEFT**. This technique consists of only training a minimal subset (around 1%~2%) of strategically chosen model parameters, thus saving a ton of memory on the gradient computations.

What we have done so far by freezing the model and only training a subset of layers is an efficient fine-tuning strategy, but with these techniques, we trim the number of parameters to train way more aggressively.

TLDR: We freeze the whole model, then we inject some parameters at certain layers (mostly at the attention projections) and then only train those. The original model stays frozen and the newly added parameters (often called [adapters](#)) align the model to our new task.

By freezing the model, we save on the backward pass, as now we have fewer parameters to train and fewer gradients to compute. Roughly there is one partial derivative per parameter, so we can theoretically save 50% of memory minus the adapters by using PEFT.

One of the most widely used techniques is [LoRA](#) (Low-Rank Adaptation). This technique decomposes the linear transforms of the



dimensions of 4096 x 16 and 16 x 4096, totaling 524,000 parameters, only 3% of the original size. Once this matrix is trained, you can multiply them and add them back to the original model, so no architectural change happens on the original pre-trained model.

Another free goodie of this technique is that you only need to save the weights of the adapters, as the original pre-trained model remains unchanged during training! Sharing these fine-tuned LoRA models becomes easy and lightweight compared to the huge checkpoint files from the original pre-trained architecture.

Okay, but how is the quality of the fine-tuned model? The answer is **pretty good** 😊.

The results are on par with a complete model fine-tuned for a fraction of the trainable parameters. Some tasks show that this technique could help preserve the model's capabilities and perform better (see the right image).

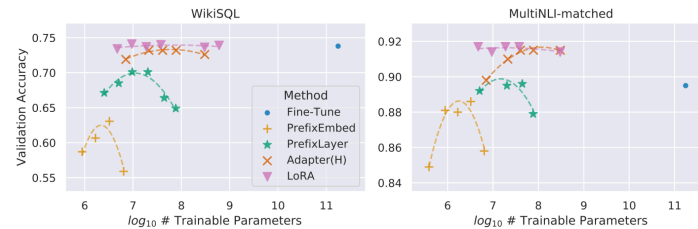
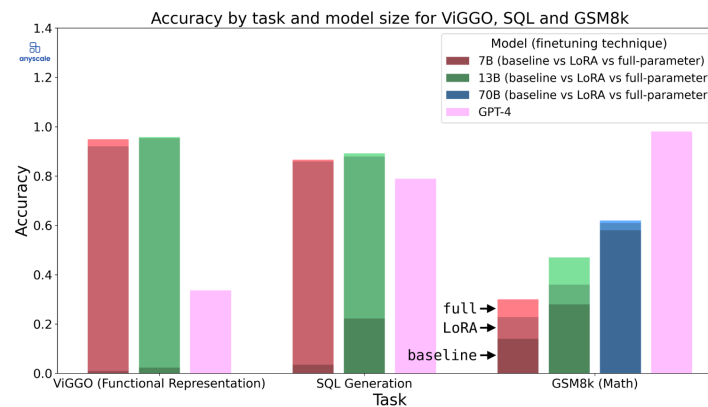


Figure 2: GPT-3 175B validation accuracy vs. number of trainable parameters of several adaptation methods on WikiSQL and MNL-matched. LoRA exhibits better scalability and task performance. See Section F.2 for more details on the plotted data points.

ref: Original LoRA paper results

Our friends at Anyscale also did an [in-depth analysis of full fine-tune vs LoRA for LLama models](#). The difference in accuracy between full fine-tuning and LoRA will vary depending on the tasks they evaluated. For some tasks, the performance, like ViGGO and SQL, difference is negligible, and in other more complex like GSM8k, full fine-tuning provides better results.

It's crucial to emphasize that LoRA serves as a low-rank approximation of the ideal weights when fine-tuning. This effectively limits the network's "adaptation capacity." - Anyscale



Differences between the pre-trained model, LoRA fine-tune, and Full fine-tune



Using LoRA with HuggingFace PEFT Library

Of course, HuggingFace has got you covered for LoRA! HuggingFace developed the [PEFT](#) library to implement LoRA and integrate it with the Trainer quickly. The library provides the necessary tools to inject the LoRA layers into the model and, after training, save and/or merge the layers back into the model.

```
from peft import LoraConfig

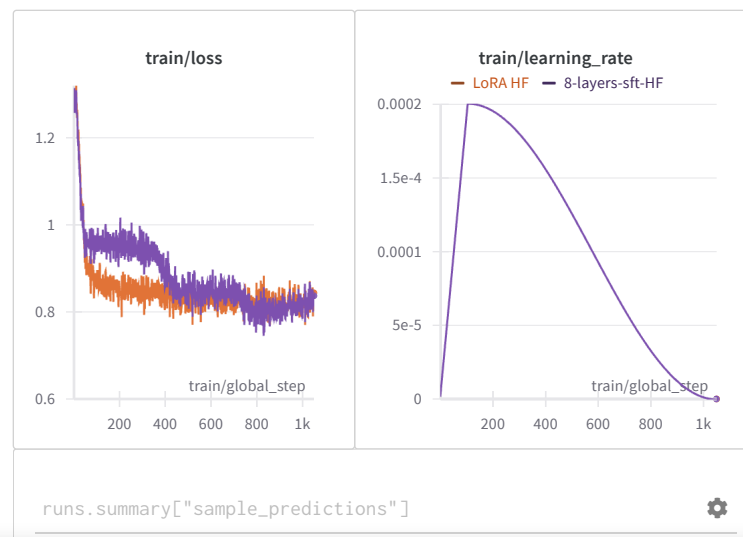
peft_config = LoraConfig(
    r=64, # the rank of the LoRA matrices
    lora_alpha=16, # the weight
    lora_dropout=0.1, # dropout to add to the LoRA layers
    bias="none", # add bias to the nn.Linear layers?
    task_type="CAUSAL_LM",
    target_modules=["q_proj", "k_proj", "v_proj", "o_proj"], # the names of the modules to target
    modules_to_save=None, # layers to unfreeze and train from the original model
)

trainer = SFTTrainer(
    ...
    peft_config = peft_config,
    ...
)
```

When using the `SFTTrainer` and passing the `peft_config`, under the hood, the trainers apply this config to wrap the model around `PeftModel` and add the `PeftSavingCallback` so we can store those fine-tuned LoRA layers safely.

Results of the LoRA Training

The training of LoRA works as expected, and the loss goes down very smoothly. We also show the same generations as before to inspect model generations at a glance. Loss-wise, we obtain similar values with both techniques.



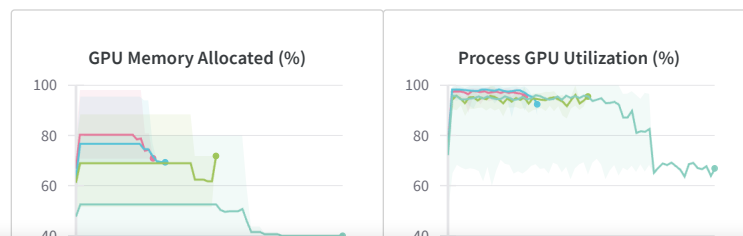
Without formal evaluation and just by checking our hand-picked examples, it looks like LoRA does a similar job to our partially fine-tuned model.

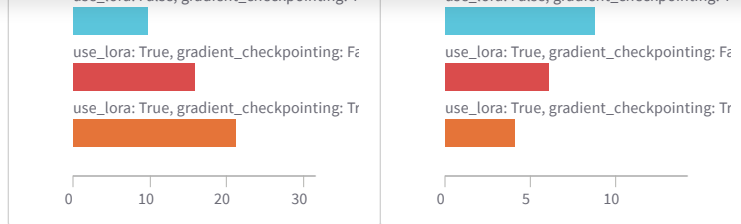
💡 This is the same [recipe](#) used to train [Zephyr](#), a state-of-the-art model instruction fine-tune done by the H4 HuggingFace research team.

Fitting Big Models on Small GPUs: A Memory Comparison

We have seen a lot of people running these LLMs on consumer-grade GPUs. The limitation of doing this is that the cards have a maximum of 24GB of memory compared to the professional workstation and AI-oriented ones with up to 80GB! We have multiple tricks to reduce the memory footprint of the training with different trade-offs. Let's compare what LoRA can bring to the table to train the last eight layers of the model as [we did in our previous article](#). This technique has proven to work well on fine-tuning and is what Jeremy Howard recommends trying first. Also, it is a good tradeoff of memory usage and speed, you can play with this parameter by adjusting the `n_freeze` param in the training script.

- We explore SFT with and without gradient accumulation
- LoRA with and without gradient accumulation
- We fit the largest possible batch size in every case. To maintain an ef batch size of 32, we only perform increments of 2. In this case, **effective batch size = batch size x gradient accumulation steps**.





☒
A100 - 40GB 48

In the charts above, we report the results of the different combinations of parameters and fine-tuning techniques. We report for GPUs: **A100 - 40GB** and **A10 - 24GB**

- Gradient checkpointing reduces memory footprint (larger batch sizes) but slows training.
- LoRA is relatively slower, but the memory usage is reduced even further
- We can combine both tricks above and get even lower memory usage.

I kept the failing runs so that you can verify that I tried fitting a larger batch size without success...

Making Forward Pass Faster: 8-Bit/4-Bit Forward

When using LoRA, we still have to compute a complete forward pass, which can be pretty costly for these large models. [By quantizing the model](#), we can improve our training performance and reduce the memory footprint. As the model is already frozen, one can load the model straight in 8-bit (or even 4-bit!). We will cover this in a follow-up article as it requires other technical pieces and analysis.

This requires changing both the model and the LoRA layers to support quantization. The library `bitsandbytes` makes this easy and the `trl`/`peft` libraries are ready to support this feature.

The only change you need to load the model in 4-bit precision is passing the flag `load_in_4bit`:

```
model = AutoModelForCausalLM.from_pretrained('meta-llama/Llama-2-7b', load_in_4bit=True)
```

This is a very simple example, but it shows the basic idea of how to load the model in 4-bit precision.



power, but there are multiple tricks to get you training on smaller GPUs with minimal downside besides speed.

How to Fine-Tun...

Learn how to fine-tune an...

How to Fine-Tun...

In part 1, we prepped ou...

Tags: Articles, LLM, NLP, GenAI, HuggingFace, Experiment, Fine-tuning, Intermediate

New course
RAG++ : From POC to Production

Weights & Biases

ENROLL NOW →

Created with  on Weights & Biases.


https://wandb.ai/capecape/alpaca_ft/reports/How-to-Fine-tune-an-LLM-Part-3-The-HuggingFace-Trainer--Vmldzo1OTEyNjMy

Made with Weights & Biases. [Sign up](#) or [log in](#) to create reports like this one.

Never lose track of another ML
project. **Try W&B today.**

[SIGN UP](#)

[TRY W&B NOW](#)

Weights & Biases
Get weekly updates with the latest ML news.

[Subscribe](#)

PRODUCTS

[Dashboard](#) [Sweeps](#) [Artifacts](#) [Reports](#) [Tables](#)

QUICKSTART

[Documentation](#)

RESOURCES

[Courses](#) [Forum](#) [Tutorials](#) [Benchmarks](#)

By clicking "Accept All Cookies", you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

