How to Fine-Tune an LLM Part 1: Preparing a Dataset for Instruction Tuning

Learn how to fine-tune an LLM on an instruction dataset! We'll cover how to format the data and train a model like Llama2, Mistral, etc. is this minimal example in (almost) pure PyTorch.

Thomas Capelle

Created on October 3 | Last edited on January 16

In this article, we will explore how to prepare your data to fine-tune your LLM on instructions (a.k.a. instruction tuning). We'll go step-by-step through how you need to format your data and apply the preprocessing techniques required to be able to fine-tune your model after. We'll do this as a tutorial and make every step simple to understand what is happening under the hood of those preprocessing pipelines. These are necessary steps if you want to be able to debug your fine-tuned large language model (LLM) afterward.

💡 NOTE: you can follow along with this notebook.

Llama meets Alpaca, courtesy of MidJourney and Morgan

What we'll be covering in this article.

Table of Contents

Framing The Project

An excellent repo (and video) to understand LLMs is Karpathy's nanoGPT. That's because it's a barebones PyTorch implementation of the original GPT-2 architecture, with a minimalistic training loop and everything you expect from a functional parallel compatible training script. That said, it's a completion example, where we feed the model a massive corpus of text and load the data with a "poor's man dataloader." How do we adapt this code to instruction training? Is it the same thing? How can we use a pre-trained Llama 2 or Mistral model?

There are multiple libraries out there that enable you to fine-tune the latest Open Source LLMs on your dataset. The HuggingFace transformers library is the most used to handle models, datasets, and training (and it also has a very nice W&B integration!). Another that builds on top of transformers is Axolotl. It's battle-tested by the OSS community and has all the bells and whistles you might expect. Under the hood, it uses multiple tricks and libraries, like transformers, peft, bitsandbytes, deepspeed, etc.

Can we implement a minimal fine-tuning pipeline to understand what's happening? Let's find out!

Choosing Your Instruction Dataset

An **instruction dataset** is a list of pairs: instruction and answer.

Instruction	Answer
Explain the concept of a bubble sort algorithm to a non-technical audience.	A bubble sort algorithm is a type of sorting algorithm that is used to sort elements in an array. It works by looking at each element of the array and comparing it to the next element. If the first element is bigger than the second element, they are
Instruction	Answer

Make the second sentence shorter. **Context**	Winter is the coldest season, often accompanied by snow.

💡 Some instructions require context to produce the output!

As you can imagine, constructing a high-quality dataset is expensive and time-consuming, but when we think about interacting with an LLM, this is probably how we want to use them most of the time (as ChatGPT does).

There are also many high-quality instruction datasets with different formats and lengths. Some are created manually, like the Flan Collection and Dolly15k dataset while others are made using LLMs like the Alpaca dataset. The open source community has actively curated and augmented datasets to fine-tune and create instruction models. Some of the recent datasets like OpenOrca, Platypus, OpenHermes produce very high-quality fine-tuned models that score high on the leaderboards and different evaluation tasks.

In this article, we'll use the Alpaca dataset and explore how to pre-process and format that dataset to train a LLama model.

What Is The Alpaca Dataset?

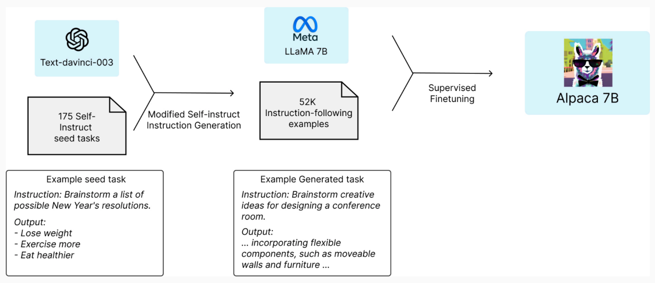
The Alpaca dataset is a synthetic dataset developed by Stanford researchers using the OpenAI davinci model to generate instruction/output pairs and fine-tuned Llama. The dataset covers a diverse list of user-oriented instructions, including email writing, social media, and productivity tools.

💡 This model is often referred to as Alpaca or Alpaca-GPT3.

In their words:

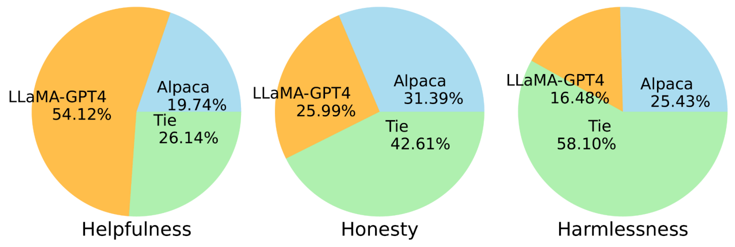
> *"We are releasing our findings about an instruction-following language model, dubbed Alpaca, which is fine-tuned from Meta's LLaMA 7B model. We train the Alpaca model on 52K instruction-following demonstrations generated in the style of self-instruct using text-davinci-003. On the self-instruct evaluation set, Alpaca shows many behaviors similar to OpenAI's text-davinci-003, but is also surprisingly small and easy/cheap to reproduce."*

See the pipeline to create the Alpaca dataset and fine-tuning below:

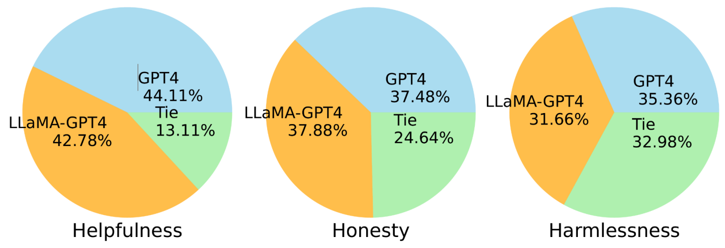

Text-davinci-003
175 Self-Instruct seed tasks
Modified Self-instruct Instruction Generation
Meta LLaMA 7B
52K Instruction-following examples
Supervised Finetuning
Alpaca 7B
Example seed task
Instruction: Brainstorm a list of possible New Year's resolutions.
Output:
- Lose weight
- Exercise more
- Eat healthier
Example Generated task
Instruction: Brainstorm creative ideas for designing a conference room.
Output:
... incorporating flexible components, such as moveable walls and furniture ...

The Alpaca dataset and Alpaca-Llama model pipeline from https://crfm.stanford.edu/2023/03/13/alpaca.html

We'll use an updated version that, instead of davinci-003 (GPT-3), uses GPT-4 to get an even better model! More details on the official dataset repo.



- LLaMA-GPT-4 performs substantially better than LLaMA-GPT-3 in the "Helpfulness" criteria.



- LLaMA-GPT-4 performs similarly to the original GPT-4 in all three criteria, suggesting a promising direction for developing state-of-the-art instruction-following LLMs.

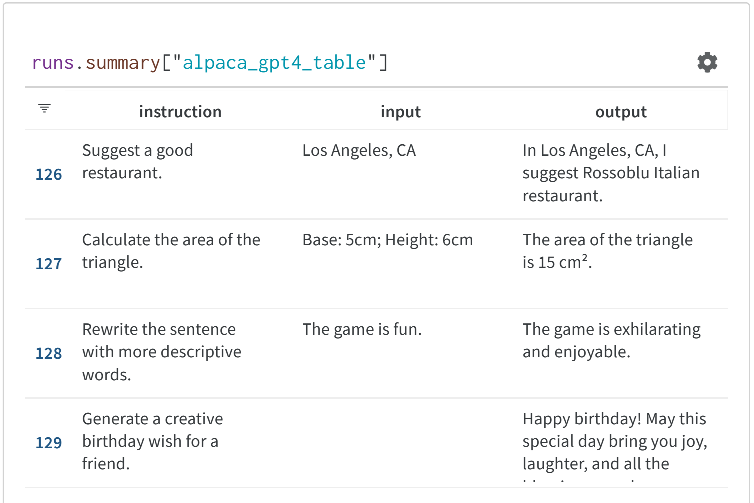
The Alpaca-GPT4 Dataset

The Alpaca-GPT4 dataset is just a single JSON file, alpaca_gpt4_data.json contains 52K instruction-following data generated by GPT-4 with prompts in Alpaca. This JSON file has the same format as Alpaca data, except the output is generated by GPT-4.

An example:

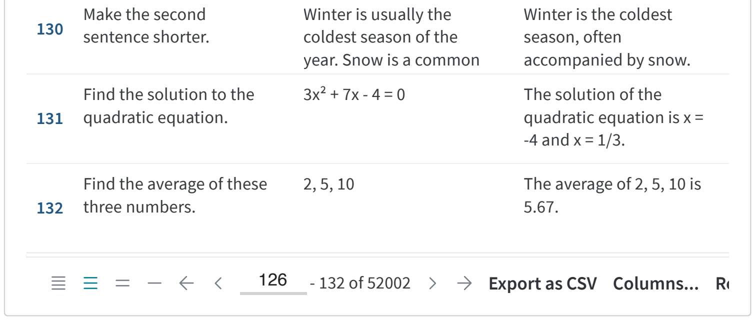
```
instruction: str, describes the task the model should perform.
                  Each of the 52K instructions is unique.
input:       str, optional context or input for the task.
output:      str, the answer to the instruction as generated by GPT-
```

We logged the dataset as a W&B Table to be able to inspect the different instruction/output pairs quickly. We will use Tables later on to inspect our model predictions and compare different training recipes. Check the table below using the arrows and hovering over the instruction/input/output tooltips:



runs.summary["alpaca_gpt4_table"]

	instruction	input	output
126	Suggest a good restaurant.	Los Angeles, CA	In Los Angeles, CA, I suggest Rossoblu Italian restaurant.
127	Calculate the area of the triangle.	Base: 5cm; Height: 6cm	The area of the triangle is 15 cm².
128	Rewrite the sentence with more descriptive words.	The game is fun.	The game is exhilarating and enjoyable.
129	Generate a creative birthday wish for a friend.		Happy birthday! May this special day bring you joy, laughter, and all the



130	Make the second sentence shorter.	Winter is usually the coldest season of the year. Snow is a common	Winter is the coldest season, often accompanied by snow.
131	Find the solution to the quadratic equation.	$3x^2 + 7x - 4 = 0$	The solution of the quadratic equation is x = -4 and x = 1/3.
132	Find the average of these three numbers.	2, 5, 10	The average of 2, 5, 10 is 5.67.

I encourage you to explore the dataset. Some tasks are simple, and others are, well, not so simple. Still: it's impressive how good GPT-4 generates these 🤯.

> As Gerard commented some of the outputs from GPT-4 are wrong, for instance example **131**, the solutions to the quadratic equation are not correct. Asking GPT-4 today, it actually solves the equation appropiately.

Log The Alpaca Dataset to W&B

See that code below. Also, as a reminder, all the code from this article can be found here.

```python
import json

with open("alpaca_data.json", "r") as f:
    alpaca = json.load(f)

with wandb.init(project="alpaca_ft"):
    at = wandb.Artifact(
        name="alpaca_gpt4",
        type="dataset",
        description="A GPT4 generated Alpaca like dataset for instru
        metadata={"url":"https://github.com/Instruction-Tuning-with-
    )
    at.add_file("alpaca_data.json")

    # table
    table = wandb.Table(columns=list(alpaca[0].keys()))
    for row in alpaca:
        table.add_data(*row.values())
```

Dataset preparation and tokenization

A row of the dataset (or one example) it's a dictionary with keys: `instruction, input`, and `output`.

```python
import json

with open("alpaca_data.json", "r") as f:
    alpaca = json.load(f)

len(alpaca)
>>  52002

one_row = alpaca[232]
one_row = {
   'instruction': 'What are the three primary colors?',
   'input': '',
   'output': 'The three primary colors are red, blue, and yellow.'
}
```

We need to do some preprocessing so we can feed the LLM with this data. Let's define some functions to format the instructions:

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

def prompt_input(row):
    return ("Below is an instruction that describes a task, paired w
            "Write a response that appropriately completes the reque
            "### Instruction:\n{instruction}\n\n### Input:\n{input}\
```

We have instructions with and without prompts, so we must deal with them separately. We could have concatenated the output simultaneously, but we will keep it separate as we will re-use these later on the instruction fine-tuning.

We get something that looks like this:

```
row = alpaca[232]
print(prompt_input(row))

>> Below is an instruction that describes a task, paired with an inp

### Instruction:
What are the three primary colors?

### Input:

### Response:
```

We can then merge both paths into:

```
def create_prompt(row):
    return prompt_no_input(row) if row["input"] == "" else prompt_in

prompts = [create_prompt(row) for row in alpaca]  # all LLM inputs a
```

End of String Token (EOS)

This token is essential because it tells the model when to stop producing text; for LLama models, `EOS_TOKEN = "</s>"`

We will append this token after each response:

```
EOS_TOKEN = "</s>"
outputs = [row['output'] + EOS_TOKEN for row in alpaca]
```

we explicitly add this token at the end of each response…

```
outputs[0]
# this is a oneliner split here for readability
>> 1.Eat a balanced diet and make sure to include plenty of fruits a
\n2. Exercise regularly to keep your body active and strong.
\n3. Get enough sleep and maintain a consistent sleep schedule.</s>'
```

We will also store the concatenation of both instruction and output:

```
dataset = [{"prompt":s, "output":t, "example": s+t} for s, t in zip(
```

💡 You could store this preprocessed dataset as a W&B Artifact and avoid re doing this every time 😎

Tokens, tokens everywhere: How to tokenize and organize text

We need to convert the dataset into tokens. You can quickly do this with the workhorse of the transformers library, the Tokenizer! This function does a lot of heavy lifting besides tokenizing the text.

- It tokenizes the text
- Converts the outputs to PyTorch tensors
- Pads the inputs to match the length
- and more!

Let's try that mighty tokenizer!

```
model_id = 'meta-llama/Llama-2-7b-hf'
tokenizer = AutoTokenizer.from_pretrained(model_id)
tokenizer.pad_token = tokenizer.eos_token
```

We have to tell the tokenizer what token to use to pad; in this case, it's the EOS token (that has id = 2). We can specify the length of the resulting padded sequence and complete it accordingly.

```
tokenizer.encode("My experiments are going strong!")
# >> [1, 1619, 15729, 526, 2675, 4549, 29991]

tokenizer.encode("My experiments are going strong!", padding='max_le
# >> [1, 1619, 15729, 526, 2675, 4549, 29991, 2, 2, 2]
```

We can also get PyTorch tensors directly:

```
tokenizer.encode("My experiments are going strong!",
                 padding='max_length',
                 max_length=10,
                 return_tensors="pt")
# >> tensor([[    1,  1619, 15729,   526,  2675,  4549, 29991,     2
```

The latter is beneficial as we can put the tokenizer inside the collate function! This way, we sample from the dataloader strings, and then the collate function tokenizes and converts them to PyTorch tensors.

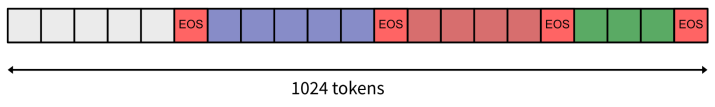
Creating a Train-Eval Split

Let's keep some samples to perform evaluation later on; we store the split as a W&B Table to be able to inspect the datasets.

```
import random
random.shuffle(dataset). # shuffle inplace

train_dataset = dataset[:-1000]
eval_dataset = dataset[-1000:]

train_table = wandb.Table(dataframe=pd.DataFrame(train_dataset))
eval_table  = wandb.Table(dataframe=pd.DataFrame(eval_dataset))

with wandb.init(project="alpaca_ft", job_type="split_data"):
    wandb.log({"train_dataset":train_table, "eval_dataset":eval_tabl
```

Packing: Combining multiple samples into a longer sequence

To make training more efficient and use the longer context of these LLMs we'll do something called "**packing**". We will combine multiple examples to fill the model's memory and make training more efficient instead of feeding examples individually. This way we avoid doing a lot of padding and dealing with different lengths.



💡 After discussing with Lewis Tunstall 🤗 (one of the author's of the NLP with Transformers book), he pointed me out the more efficient way of doing this by actually `packing` sequences until a desired lenght and then feeding the model the packed-batch without need to pad with tokens.

The main idea here is that the instruction/output samples are short, so let's concatenate a bunch of them together, separated by the EOS token. We can also pre-tokenize and pre-pack the dataset and make everything faster! If we define a `max_seq_len = 1024`, the code to pack would look something like this:

```
max_seq_len = 1024

def pack(dataset, max_seq_len=1024):
    tkds_ids = tokenizer([s["example"] for s in dataset])["input_ids

    all_token_ids = []
    for tokenized_input in tkds_ids:
        all_token_ids.extend(tokenized_input + [tokenizer.eos_token_

    packed_ds = []
    for i in range(0, len(all_token_ids), max_seq_len+1):
        input_ids = all_token_ids[i : i + max_seq_len+1]
        if len(input_ids) == (max_seq_len+1):
            packed_ds.append({"input_ids": input_ids[:-1], "labels":
            # if you use the model.output.loss you don't need to shi
    return packed_ds

train_ds_packed = pack(train_dataset)
eval_ds_packed = pack(eval_dataset)
```

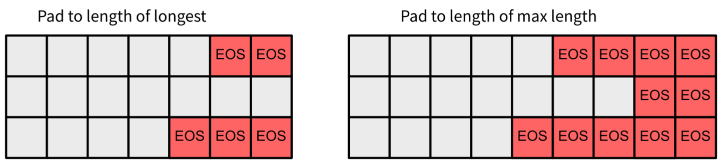
💡 The amazing `trl` library has this implemented for us here.

Doing so, we end **up** with more than 11k sequences of length 1024.

Second Option: Batching multiple sequences of different lengths

There is another technique to construct batches from lines of different sizes; it's by padding the sequences and making them longer so they can be batched together.

The tokenizer has a batching function that creates the batch from different samples and pads according to the desired strategy.



This can be done by calling the tokenized directly on the list of texts:

```
tokenizer(["My experiments are going strong!",
           "I love Llamas"],
          padding='longest',
          return_tensors="pt")

>> {'input_ids': tensor([[    1,  1619, 15729,   526,  2675,  4549,
                          [    1,   306,  5360,   365,  5288,   294,
    'attention_mask': tensor([[1, 1, 1, 1, 1, 1, 1],
                              [1, 1, 1, 1, 1, 1, 0]])}

tokenizer(["My experiments are going strong!",
           "I love Llamas"],
          # padding='max_length',
          padding='max_length',
          max_length=10,
          return_tensors="pt")

>> {'input_ids': tensor([[    1,  1619, 15729,   526,  2675,  4549,
                          [    1,   306,  5360,   365,  5288,   294,
    'attention_mask': tensor([[1, 1, 1, 1, 1, 1, 1, 0, 0, 0],
                              [1, 1, 1, 1, 1, 1, 0, 0, 0, 0]])}
```

So, we can use this function to create the final batch that we will pass to the model. Also note that this task could be done offline, preprocessing the full dataset once. This is done in most scenarios, and people stream from the tokenized dataset. The transformers library has even a `FastTokenizer` class implemented in Rust to make this step faster.

> This solution is not very performant, as every batch will have different length and contain tokens that don't teach anything to the model.

Storing our preprocessed datasets on W&B

Now that we have packed our datasets we can save them securely to train a model!

To get the model lineage and know precisely what dataset was used to fine-tune our models, it's good practice to version the data and keep everything organized. We'll log the dataset as a W&B Artifact.

We can store our data back into JSONL format, where each line corresponds to a dictionary object:

```
import json
def save_jsonl(data, filename):
    with open(filename, 'w') as file:
        for entry in data:
            json.dump(entry, file)
            file.write('\n')

# dump everything to jsonl files
save_jsonl(train_ds_packed, "train_packed_alpaca.jsonl")
save_jsonl(eval_ds_packed, "eval_packed_alpaca.jsonl")

# Create a W&B artifact
packed_at = wandb.Artifact(
    name="packed_alpaca",
    type="dataset",
    description="Alpaca dataset packed in sequences",
    metadata={"max_seq_len":1024, "model_id":model_id})

packed_at.add_file("train_packed_alpaca.jsonl")
packed_at.add_file("eval_packed_alpaca.jsonl")

# log the artifact to the project, we can give this run a job_type
with wandb.init(project="alpaca_ft", job_type="preprocess"):
    wandb.log_artifact(packed_at)
```

You can store relevant information from the dataset on the description and metadata arguments if needed.

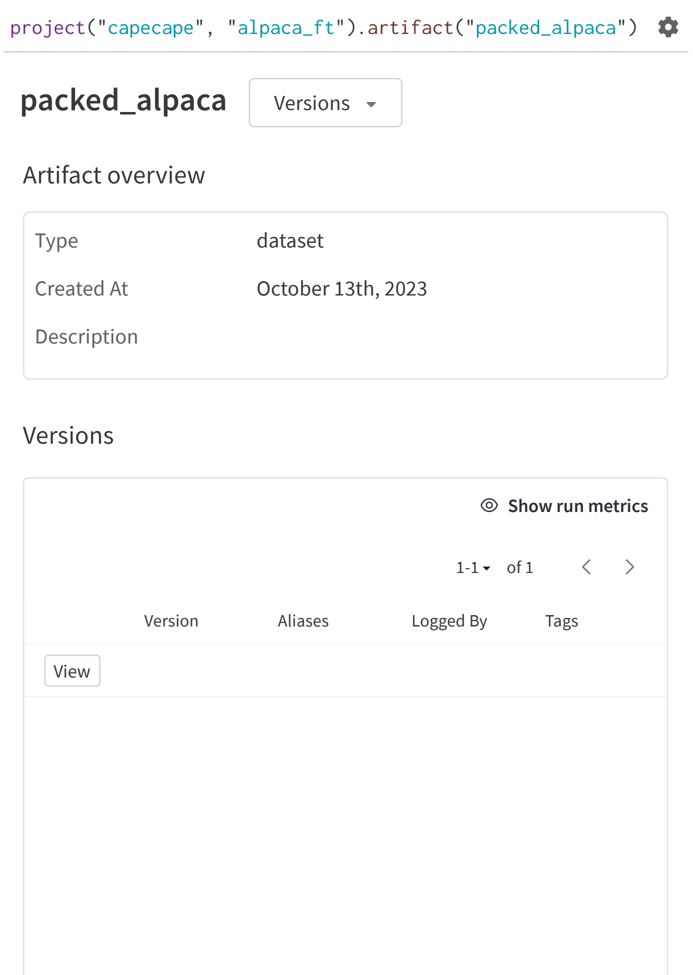
> The code for this article and the data pipeline can be found here

Conclusion and remarks

Good data fosters great models, and the formatting and preprocessing stages play a critical role in preparing your dataset for the fine-tuning task. Many minute details contribute to instructing a model effectively, along with countless engineering nuances and techniques that expedite data flow and optimize the use of GPUs.

I have grappled with understanding how the tokenizer interacts with text during the batching and sequence creation processes. I hope this article provides the essential insights needed, enabling you to fine-tune a model via a state-of-the-art script that masks complexity - with added confidence that everything should proceed smoothly.

With our preprocessed dataset ready and available in the project Artifacts panel, we can swiftly access it and begin fine-tuning our model.



```
project("capecape", "alpaca_ft").artifact("packed_alpaca")
```

packed_alpaca — Versions

Artifact overview

Type	dataset
Created At	October 13th, 2023
Description	

Versions

Show run metrics

1-1 of 1

	Version	Aliases	Logged By	Tags
View				

👉 Continue to Part 2: Training our LLM

How to Fine-Tun…
In part 1, we prepped ou…

How to Fine-tune an LL…
Exploring how to get the best o…

Also, check out these articles for some more reading on Training, Evaluating and Running LLMs

How to Run LL…
This article explores …

Training Tiny Llam…
Exploring how SoftMax …

Fine-Tuning …
A tutorial for fine-…

How to Evaluate, …
This article provides an …

Tags: Articles, NLP, GenAI, LLM, Tutorial, Experiment, Fine-tuning

Created with ❤️ on Weights & Biases.

https://wandb.ai/capecape/alpaca_ft/reports/How-to-Fine-Tune-an-LLM-Part-1-Preparing-a-Dataset-for-Instruction-Tuning--Vmlldzo1NTcxNzE2

Subscribe