philschmid.de

Efficient Large Language Model training with LoRA and Hugging Face

Philipp Schmid

13–16 minutes

In this blog, we are going to show you how to apply Low-Rank Adaptation of Large Language Models (LoRA) to fine-tune FLAN-T5 XXL (11 billion parameters) on a single GPU. We are going to leverage Hugging Face Transformers, Accelerate, and PEFT.

You will learn how to:

1. Setup Development Environment
2. Load and prepare the dataset
3. Fine-Tune T5 with LoRA and bnb int-8
4. Evaluate & run Inference with LoRA FLAN-T5

Quick intro: PEFT or Parameter Efficient Fine-tuning

PEFT, or Parameter Efficient Fine-tuning, is a new open-source library from Hugging Face to enable efficient adaptation of pre-trained language models (PLMs) to various downstream applications without fine-tuning all the model's parameters. PEFT currently includes techniques for:

- LoRA: LORA: LOW-RANK ADAPTATION OF LARGE LANGUAGE MODELS
- Prefix Tuning: P-Tuning v2: Prompt Tuning Can Be Comparable to Fine-tuning Universally Across Scales and Tasks
- P-Tuning: GPT Understands, Too
- Prompt Tuning: The Power of Scale for Parameter-Efficient Prompt Tuning

Note: This tutorial was created and run on a g5.2xlarge AWS EC2 Instance, including 1 NVIDIA A10G.

1. Setup Development Environment

In our example, we use the PyTorch Deep Learning AMI with already set up CUDA drivers and PyTorch installed. We still have to install the Hugging Face Libraries, including transformers and datasets. Running the following cell will install all the required packages.

```
# install Hugging Face Libraries
!pip install  git+https://github.com/huggingface/peft.git
!pip install "transformers==4.27.2" "datasets==2.9.0" "accelerate==0.17.1" "evaluate==0.4.0" "bitsandbytes==0.37.1" loralib --upgrade --quiet
# install additional dependencies needed for training
!pip install rouge-score tensorboard py7zr
```

2. Load and prepare the dataset

we will use the samsum dataset, a collection of about 16k messenger-like conversations with summaries. Conversations were created and written down by linguists fluent in English.

```
{
  "id": "13818513",
  "summary": "Amanda baked cookies and will bring Jerry some tomorrow.",
  "dialogue": "Amanda: I baked cookies. Do you want some?\r\nJerry: Sure!\r\nAmanda: I'll bring you tomorrow :-)"
}
```

To load the `samsum` dataset, we use the **`load_dataset()`** method from the 🤗 Datasets library.

```
from datasets import load_dataset

# Load dataset from the hub
dataset = load_dataset("samsum")

print(f"Train dataset size: {len(dataset['train'])}")
print(f"Test dataset size:
```

```
{len(dataset['test'])}")

# Train dataset size: 14732
# Test dataset size: 819
```

To train our model, we need to convert our inputs (text) to token IDs. This is done by a 🤗 Transformers Tokenizer. If you are not sure what this means, check out chapter 6 of the Hugging Face Course.

```
from transformers import AutoTokenizer,
AutoModelForSeq2SeqLM

model_id="google/flan-t5-xxl"

# Load tokenizer of FLAN-t5-XL
tokenizer =
AutoTokenizer.from_pretrained(model_id)
```

Before we can start training, we need to preprocess our data. Abstractive Summarization is a text-generation task. Our model will take a text as input and generate a summary as output. We want to understand how long our input and output will take to batch our data efficiently.

```
from datasets import concatenate_datasets
import numpy as np
# The maximum total input sequence length after
tokenization.
# Sequences longer than this will be truncated,
sequences shorter will be padded.
tokenized_inputs =
concatenate_datasets([dataset["train"],
dataset["test"]]).map(lambda x:
tokenizer(x["dialogue"], truncation=True),
batched=True, remove_columns=["dialogue",
"summary"])
input_lenghts = [len(x) for x in
tokenized_inputs["input_ids"]]
# take 85 percentile of max length for better
utilization
max_source_length =
int(np.percentile(input_lenghts, 85))
print(f"Max source length: {max_source_length}")
```

```
# The maximum total sequence length for target
text after tokenization.
# Sequences longer than this will be truncated,
sequences shorter will be padded."
tokenized_targets =
concatenate_datasets([dataset["train"],
dataset["test"]]).map(lambda x:
tokenizer(x["summary"], truncation=True),
batched=True, remove_columns=["dialogue",
"summary"])
target_lenghts = [len(x) for x in
tokenized_targets["input_ids"]]
# take 90 percentile of max length for better
utilization
max_target_length =
int(np.percentile(target_lenghts, 90))
print(f"Max target length: {max_target_length}")
```

We preprocess our dataset before training and save it to disk. You could run this step on your local machine or a CPU and upload it to the Hugging Face Hub.

```
def
preprocess_function(sample,padding="max_length"):
    # add prefix to the input for t5
    inputs = ["summarize: " + item for item in
sample["dialogue"]]

    # tokenize inputs
    model_inputs = tokenizer(inputs,
max_length=max_source_length, padding=padding,
truncation=True)

    # Tokenize targets with the `text_target`
keyword argument
    labels =
tokenizer(text_target=sample["summary"],
max_length=max_target_length, padding=padding,
truncation=True)

    # If we are padding here, replace all
tokenizer.pad_token_id in the labels by -100 when
```

```
we want to ignore
    # padding in the loss.
    if padding == "max_length":
        labels["input_ids"] = [
            [(l if l != tokenizer.pad_token_id
else -100) for l in label] for label in
labels["input_ids"]
        ]

    model_inputs["labels"] = labels["input_ids"]
    return model_inputs

tokenized_dataset =
dataset.map(preprocess_function, batched=True,
remove_columns=["dialogue", "summary", "id"])
print(f"Keys of tokenized dataset:
{list(tokenized_dataset['train'].features)}")

# save datasets to disk for later easy loading
tokenized_dataset["train"].save_to_disk("data/trai
tokenized_dataset["test"].save_to_disk("data
/eval")
```

3. Fine-Tune T5 with LoRA and bnb int-8

In addition to the LoRA technique, we will use bitsanbytes LLM.int8() to quantize out frozen LLM to int8. This allows us to reduce the needed memory for FLAN-T5 XXL ~4x.

The first step of our training is to load the model. We are going to use philschmid/flan-t5-xxl-sharded-fp16, which is a sharded version of google/flan-t5-xxl. The sharding will help us to not run off of memory when loading the model.

```
from transformers import AutoModelForSeq2SeqLM

# huggingface hub model id
model_id = "philschmid/flan-t5-xxl-sharded-fp16"

# load model from the hub
model =
AutoModelForSeq2SeqLM.from_pretrained(model_id,
load_in_8bit=True, device_map="auto")
```

Now, we can prepare our model for the LoRA int-8 training using

peft.

```
from peft import LoraConfig, get_peft_model,
prepare_model_for_int8_training, TaskType

# Define LoRA Config
lora_config = LoraConfig(
 r=16,
 lora_alpha=32,
 target_modules=["q", "v"],
 lora_dropout=0.05,
 bias="none",
 task_type=TaskType.SEQ_2_SEQ_LM
)
# prepare int-8 model for training
model = prepare_model_for_int8_training(model)

# add LoRA adaptor
model = get_peft_model(model, lora_config)
model.print_trainable_parameters()

# trainable params: 18874368 || all params:
11154206720 || trainable%: 0.16921300163961817
```

As you can see, here we are only training 0.16% of the parameters of the model! This huge memory gain will enable us to fine-tune the model without memory issues.

Next is to create a `DataCollator` that will take care of padding our inputs and labels. We will use the `DataCollatorForSeq2Seq` from the 🤗 Transformers library.

```
from transformers import DataCollatorForSeq2Seq

# we want to ignore tokenizer pad token in the
loss
label_pad_token_id = -100
# Data collator
data_collator = DataCollatorForSeq2Seq(
    tokenizer,
    model=model,
    label_pad_token_id=label_pad_token_id,
    pad_to_multiple_of=8
)
```

The last step is to define the hyperparameters (`TrainingArguments`) we want to use for our training.

```
from transformers import Seq2SeqTrainer,
Seq2SeqTrainingArguments

output_dir="lora-flan-t5-xxl"

# Define training args
training_args = Seq2SeqTrainingArguments(
    output_dir=output_dir,
                auto_find_batch_size=True,
    learning_rate=1e-3, # higher learning rate
    num_train_epochs=5,
    logging_dir=f"{output_dir}/logs",
    logging_strategy="steps",
    logging_steps=500,
    save_strategy="no",
    report_to="tensorboard",
)

# Create Trainer instance
trainer = Seq2SeqTrainer(
    model=model,
    args=training_args,
    data_collator=data_collator,
    train_dataset=tokenized_dataset["train"],
)
model.config.use_cache = False  # silence the
warnings. Please re-enable for inference!
```

Let's now train our model and run the cells below. Note that for T5, some layers are kept in `float32` for stability purposes.

```
# train model
trainer.train()
```

The training took ~10:36:00 and cost ~13.22$ for 10h of training. For comparison a full fine-tuning on FLAN-T5-XXL with the same duration (10h) requires 8x A100 40GBs and costs ~322$.

We can save our model to use it for inference and evaluate it. We will save it to disk for now, but you could also upload it to the Hugging Face Hub using the `model.push_to_hub` method.

```
# Save our LoRA model & tokenizer results
peft_model_id="results"
trainer.model.save_pretrained(peft_model_id)
tokenizer.save_pretrained(peft_model_id)
# if you want to save the base model to call
#
trainer.model.base_model.save_pretrained(peft_mode
```

Our LoRA checkpoint is only 84MB small and includes all of the learnt knowleddge for samsum.

4. Evaluate & run Inference with LoRA FLAN-T5

After the training is done we want to evaluate and test it. The most commonly used metric to evaluate summarization task is rogue_score short for Recall-Oriented Understudy for Gisting Evaluation). This metric does not behave like the standard accuracy: it will compare a generated summary against a set of reference summaries.

We are going to use `evaluate` library to evaluate the `rogue` score. We can run inference using PEFT and `transformers`. For our FLAN-T5 XXL model, we need at least 18GB of GPU memory.

```
import torch
from peft import PeftModel, PeftConfig
from transformers import AutoModelForSeq2SeqLM,
AutoTokenizer

# Load peft config for pre-trained checkpoint
etc.
peft_model_id = "results"
config =
PeftConfig.from_pretrained(peft_model_id)

# load base LLM model and tokenizer
model =
AutoModelForSeq2SeqLM.from_pretrained(config.base_
load_in_8bit=True,  device_map={"":0})
tokenizer =
AutoTokenizer.from_pretrained(config.base_model_na

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

```
peft_model_id, device_map={"":0})
model.eval()

print("Peft model loaded")
```

Let's load the dataset again with a random sample to try the summarization.

```
from datasets import load_dataset
from random import randrange


# Load dataset from the hub and get a sample
dataset = load_dataset("samsum")
sample = dataset['test']
[randrange(len(dataset["test"]))]

input_ids = tokenizer(sample["dialogue"],
return_tensors="pt",
truncation=True).input_ids.cuda()
# with torch.inference_mode():
outputs = model.generate(input_ids=input_ids,
max_new_tokens=10, do_sample=True, top_p=0.9)
print(f"input sentence:
{sample['dialogue']}\n{'---'* 20}")

print(f"summary:\n{tokenizer.batch_decode(outputs.
skip_special_tokens=True)[0]}")
```

Nice! our model works! Now, lets take a closer look and evaluate it against the `test` set of processed dataset from `samsum`. Therefore we need to use and create some utilities to generate the summaries and group them together. The most commonly used metrics to evaluate summarization task is rogue_score short for Recall-Oriented Understudy for Gisting Evaluation). This metric does not behave like the standard accuracy: it will compare a generated summary against a set of reference summaries.

```
import evaluate
import numpy as np
from datasets import load_from_disk
from tqdm import tqdm

# Metric
```

```
metric = evaluate.load("rouge")

def
evaluate_peft_model(sample,max_target_length=50):
    # generate summary
    outputs =
model.generate(input_ids=sample["input_ids"].unsqu
do_sample=True, top_p=0.9,
max_new_tokens=max_target_length)
    prediction =
tokenizer.decode(outputs[0].detach().cpu().numpy()
skip_special_tokens=True)
    # decode eval sample
    # Replace -100 in the labels as we can't
decode them.
    labels = np.where(sample['labels'] != -100,
sample['labels'], tokenizer.pad_token_id)
    labels = tokenizer.decode(labels,
skip_special_tokens=True)

    # Some simple post-processing
    return prediction, labels

# load test dataset from distk
test_dataset = load_from_disk("data/eval
/").with_format("torch")

# run predictions
# this can take ~45 minutes
predictions, references = [] , []
for sample in tqdm(test_dataset):
    p,l = evaluate_peft_model(sample)
    predictions.append(p)
    references.append(l)

# compute metric
rogue = metric.compute(predictions=predictions,
references=references, use_stemmer=True)

# print results
print(f"Rogue1: {rogue['rouge1']* 100:2f}%")
print(f"rouge2: {rogue['rouge2']* 100:2f}%")
```

```
print(f"rougeL: {rogue['rougeL']* 100:2f}%")
print(f"rougeLsum: {rogue['rougeLsum']*
100:2f}%")

# Rogue1: 50.386161%
# rouge2: 24.842412%
# rougeL: 41.370130%
# rougeLsum: 41.394230%
```

Our PEFT fine-tuned FLAN-T5-XXL achieved a rogue1 score of 50.38% on the test dataset. For comparison a full fine-tuning of flan-t5-base achieved a rouge1 score of 47.23. That is a 3% improvements.

It is incredible to see that our LoRA checkpoint is only 84MB small and model achieves better performance than a smaller fully fine-tuned model.

Thanks for reading! If you have any questions, feel free to contact me on Twitter or LinkedIn.