A Simple Guide to Fine-Tuning GPT-2 using QLoRa

Hamza Fatnaoui

Introduction

In this guide, you will dive into the process of fine-tuning GPT-2 on your own dataset to use it for specific tasks. We will define GPT-2, explore how it works, and discuss the benefits of fine-tuning this model for various applications. Additionally, we will focus on fine-tuning GPT-2 using QLoRA, a technique that enhances the model's performance while optimizing resource usage. By the end of this guide, you'll have a comprehensive understanding of how to tailor GPT-2 to meet your unique needs.

1. Overview of GPT-2

1. **Architecture**: GPT-2 is based on the Transformer architecture, which uses self-attention mechanisms to process input sequences efficiently.
2. **Pre-training**: The model is pre-trained on a diverse dataset containing a large portion of the internet's text, allowing it to learn a wide range of language patterns and knowledge.
3. **Generative Capabilities**: GPT-2 can generate human-like text based on a given prompt, making it useful for various applications, including content creation, chatbots, translation, and more.
4. **Fine-tuning**: take a look at the next

2. Benefits of Fine-Tuning GPT-2

Fine-tuning in deep learning is a form of transfer learning. It involves taking a pre-trained model, which has been trained on a large dataset for a general task such as image recognition or natural language understanding, and making minor adjustments to its internal parameters. The goal is to optimize the model's performance on a new, related task without starting the training process from scratch.

3. QLoRa

QLoRA is the extended version of LoRA which works by quantizing the precision of the weight parameters in the pre trained LLM to 4-bit precision. Typically, parameters of trained models are stored in a 32-bit format, but QLoRA compresses them to a 4-bit format. This reduces the memory footprint of the LLM, making it possible to finetune it on a single GPU. This method significantly reduces the memory footprint, making it feasible to run LLM models on less powerful hardware, including consumer GPUs.

Steps to Fine-Tune GPT-2

1. Preparing Your Dataset

1.1. Data Collection:

- Gather a dataset that is relevant to the specific task you want to fine-tune GPT-2 on. We are going to use The WikiQA corpus, which is a publicly available set of question and sentence pairs, collected and annotated for research on open-domain question answering. This is the link to the provided dataset on Hugging Face: WikiQA on Hugging Face.

1.2. Data Preprocessing:

- First we need to load the dataset using Hugging Face datasets library.

```
# Step 1: Install the Hugging Face datasets library
!pip install datasets

# Step 2: Load the dataset
from datasets import load_dataset

# Loading the 'glue' dataset
dataset = load_dataset("microsoft/wiki_qa")
```

- The data provided contains additional features, but we are only interested in the questions and the responses. We will extract these two features and store them in two files, `Q_A_trai.txt` and `Q_A_test.txt`, to fine-tune and test our model.

```
with open('Q_A_train.txt', 'w') as file:
    for row in dataset['train']:
        question = row['question']
        answer = row['answer']
        file.write(f"Question: {question}\nAnswer: {answer}\n\n")

with open('Q_A_test.txt', 'w') as file:
    for row in dataset['test']:
        question = row['question']
        answer = row['answer']
        file.write(f"Question: {question}\nAnswer: {answer}\n\n")
```

2. Setting Up the Environment

2.1. Install Required Libraries:

- Ensure you have the necessary libraries installed,

```
!pip install -q -U bitsandbytes
!pip install -q -U transformers
!pip install -q -U peft
!pip install -q -U accelerate
!pip install -q datasets
```

pip install transformers: HuggingFace's Transformers library is a popular open-source library for natural language processing (NLP) tasks. It provides a unified API for various transformer models such as BERT, GPT-2

pip install accelerate -U: to make the training faster and more efficient.

bitsandbytes: helps manage the resource demands of the model

peft: to minimize computational calculations during the fine-tuning process

2.2. Environment Setup:

- Using Google Colab, we faced countless session timeouts, so we switched to Lightning AI, which provides 22 free GPU hours monthly. You can try out the platform at Lightning AI.

3. Loading the Pre-trained GPT-2 Model And QLoRa

3.1. Code:

```
from transformers import GPT2LMHeadModel, GPT2Tokenizer
model = GPT2LMHeadModel.from_pretrained('gpt2')
tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
```

GPT2LMHeadModel: loads the architecture and pre-trained parameters of the GPT-2 model, facilitating language modeling tasks by predicting the next word in a sequence based on context.

GPT2Tokenizer: prepares text inputs for the model by converting them into tokens

3.2. QLoRa

```
from peft import prepare_model_for_kbit_training

model.gradient_checkpointing_enable()
model = prepare_model_for_kbit_training(model)
```

```python
from peft import LoraConfig, get_peft_model

config = LoraConfig(
    r=8,
    lora_alpha=32,
    target_modules=["attn.c_attn", "attn.c_proj", "mlp.c_fc", "mlp.c_proj"],
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM"
)

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

4.Preparing the Dataset

4.1. Loading Data:

- We have already loaded the dataset and converted it into a suitable format (questions and answers).

```
Question: how are glacier caves formed?
Answer: A partly submerged glacier cave on Perito Moreno Glacier .

Question: how are glacier caves formed?
Answer: The ice facade is approximately 60 m high

Question: how are glacier caves formed?
Answer: Ice formations in the Titlis glacier cave

Question: how are glacier caves formed?
Answer: A glacier cave is a cave formed within the ice of a glacier .
```

4.2. Tokenizing Data:

```
from transformers import TextDataset, DataCollatorForLanguageModeling
def load_dataset(file_path, tokenizer, block_size = 128):
    dataset = TextDataset(
        tokenizer = tokenizer,
        file_path = file_path,
        block_size = block_size,
    )
    return dataset
```

→Load_dataet function: tokenizes each piece of text using the tokenizer `GPT2Tokenizer` , and formats them into a dataset suitable for further processing or training in natural language processing tasks.

4.3. Handling the padding:

```
def load_data_collator(tokenizer, mlm = False):
    data_collator = DataCollatorForLanguageModeling(
        tokenizer=tokenizer,
        mlm=mlm,
    )
    return data_collator
```

→Load_data_collator: It takes care of the intricate details of data batching, padding

5. Training

5.1. Training function:

```
def train(train_file_path,model_name,
          output_dir,
          overwrite_output_dir,
          per_device_train_batch_size,
          num_train_epochs,
          save_steps,model,tokenizer):
  train_dataset = load_dataset(train_file_path, tokenizer)
  data_collator = load_data_collator(tokenizer)

  tokenizer.save_pretrained(output_dir)

  model.save_pretrained(output_dir)

  training_args = TrainingArguments(
        output_dir=output_dir,
        overwrite_output_dir=overwrite_output_dir,
        per_device_train_batch_size=per_device_train_batch_size,
        num_train_epochs=num_train_epochs,
        gradient_accumulation_steps=4,
        save_steps=save_steps,
        fp16=True,
        logging_steps=1,
        optim="paged_adamw_8bit"
      )

  trainer = Trainer(
          model=model,
          args=training_args,
          data_collator=data_collator,
          train_dataset=train_dataset,
  )
```

→overwrite_output_dir: setting this variable to 'false' allows you to ensure that any existing models or checkpoints in the specified output directory are not overwritten. This way, you can keep multiple versions of your models for comparison or backup purposes.

→per_device_train_batch_size: This means the model will process 4 sentences (or training examples) in parallel on each device (GPU) during training. This helps to speed up the training process and make it more efficient.

→output_dir: where the checkpoint and the architecture is going to be stored

5.2. Training the model:

- Initilizing our variables

```
train_file_path = "Q_A_train.txt"
model_name = 'gpt2'
overwrite_output_dir = False
per_device_train_batch_size = 8
num_train_epochs = 50.0
save_steps = 50000
output_dir = 'Chat_Model/'
```

- training

```
train(
    train_file_path=train_file_path,
    model_name=model_name,
    output_dir=output_dir,
    overwrite_output_dir=overwrite_output_dir,
    per_device_train_batch_size=per_device_train_batch_size,
    num_train_epochs=num_train_epochs,
    save_steps=save_steps,
    model=model,
    tokenizer=tokenizer)
```

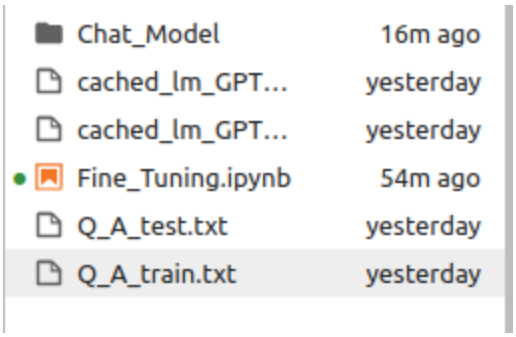
- noticing the epochs

Step	Training Loss
1	3.527000
2	3.555900
3	3.647500
4	3.536500

- the completion of the fine tuning with training loss equal to 0.2

6445	2.473100
6446	2.410900
6447	2.464200
6448	2.437000
6449	2.548500
6450	2.391600

- So finally, your bar will look like this, containing these files



6. Inference

- Now, after fine tuning the model we are going to use these learned patterns to predict outcomes for new data.

6.1. code

```python
from transformers import GPT2LMHeadModel, GPT2Tokenizer
```

```python
def load_model(model_path):
    model = GPT2LMHeadModel.from_pretrained(model_path)
    return model


def load_tokenizer(tokenizer_path):
    tokenizer = GPT2Tokenizer.from_pretrained(tokenizer_path)
    return tokenizer

def generate_text(model_path, sequence, max_length):

    model = load_model(model_path)
    tokenizer = load_tokenizer(model_path)
    ids = tokenizer.encode(f'{sequence}', return_tensors='pt')
    final_outputs = model.generate(
        ids,
        do_sample=True,
        max_length=max_length,
        pad_token_id=model.config.eos_token_id,
        top_k=50,
        top_p=0.95,
    )
    print(tokenizer.decode(final_outputs[0], skip_special_tokens=True))
```

→max_length: Once the sequence reaches this length, generation stops.

→pad_token_id: Specifies the token ID used for padding the sequence.

→sampling:

- **Greedy Decoding**: Always picking the most likely next word.
- **Sampling**: Randomly picking from a group of likely words

→top_k: Limits the sampling pool to the top `k` most likely next tokens.

→top_p: Enables nucleus sampling, also known as top-p sampling.

6.2. Example

- The example code with the generated output

```
model_path = "Chat_Model"
sequence2 = "Question: how a water pump works"
max_len = 50
print(generate_text(model_path, sequence2, max_len))
```

```
The installed version of bitsandbytes was compiled without GPU support. 8-bit optimizers, 8-bit multiplication, and GPU quantization are unava
ilable.
/home/zeus/miniconda3/envs/cloudspace/lib/python3.10/site-packages/peft/tuners/lora/layer.py:1119: UserWarning: fan_in_fan_out is set to False
but the target module is `Conv1D`. Setting fan_in_fan_out to True.
  warnings.warn(
The attention mask is not set and cannot be inferred from input because pad token is same as eos token.As a consequence, you may observe unexp
ected behavior. Please pass your input's `attention_mask` to obtain reliable results.
Answer: It consists of three main components: the pump, which pumps water into the tanks through a narrow channel to allow for quick disposal,
and the hose, a vent, which releases water directly from the tanks
```

6.3 Your guide

- Here is the entire code

Fine_Tuning_GBT2_QLORA.ipynb