Generative Supervised Fine-tuning of GPT-2

Now that we have our GPT-2 model all trained up - we need a way we can get it to generate what we want.

In the following notebook, we're going to use an approach called "Supervised Fine-tuning" to achieve our goals today.

In essence, we're going to use each example as a self-contained unit (with potential for something called "packing") and this is going to allow us to build "labeled" data.

For this notebook, we're going to be flying quite high up in the levels of abstraction. Take extra care to look into the libraries we're using today!

Let's start by grabbing our dependencies, as always:

```
!pip install transformers accelerate datasets trl bitsandbytes -qU
```

```
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7.9/7.9 MB 57.7 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 261.4/261.4 kB 21.7 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 493.7/493.7 kB 34.8 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 133.9/133.9 kB 14.7 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 92.6/92.6 MB 9.5 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 311.1/311.1 kB 35.2 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3.8/3.8 MB 97.7 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.3/1.3 MB 71.9 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 115.3/115.3 kB 14.1 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 134.8/134.8 kB 16.5 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 100.0/100.0 kB 12.2 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 295.0/295.0 kB 29.6 MB/s eta 0:00:00
```

Dataset Curation

We're going to be fine-tuning our model on SQL generation today.

First thing we'll need is a dataset to train on!

We'll use this dataset today!

First up, let's load it and take a look at what we've got.

- load_dataset

```
from datasets import load_dataset

sql_dataset = load_dataset(
    "b-mc2/sql-create-context"
)
```

```
sql_dataset
```

```
    DatasetDict({
        train: Dataset({
            features: ['context', 'answer', 'question'],
            num_rows: 78577
        })
    })
```

```
type(sql_dataset['train'])
```

```
    datasets.arrow_dataset.Dataset
```

```
sql_dataset['train'][0]
```

```
    {'context': 'CREATE TABLE head (age INTEGER)',
     'answer': 'SELECT COUNT(*) FROM head WHERE age > 56',
     'question': 'How many heads of the departments are older than 56 ?'}
```

```
sql_dataset['train'][1]
```

```
    {'context': 'CREATE TABLE head (name VARCHAR, born_state VARCHAR, age VARCHAR)',
     'answer': 'SELECT name, born_state, age FROM head ORDER BY age',
     'question': 'List the name, born state and age of the heads of departments ordered by age.'}
```

So, we've got ~78.5K rows of:

- question - a natural language query about
- context - the `CREATE TABLE` statement - which gives us important context about the table
- answer - a SQL query that is aligned with both the question and the context.

Let's split our data into `train`, `val`, and `test` datasets.

We can use our `train` and `val` sets to train and evaluate our model during training - and our `test` set to ultimately benchmark the generations of our model!

```
split_sql_train_test = sql_dataset["train"].train_test_split(test_size=0.2)
split_sql_test_val = split_sql_train_test["test"].train_test_split(test_size=0.5)
```

```
from datasets import DatasetDict

split_sql_dataset = DatasetDict(
    {
        "train":split_sql_train_test["train"],
        "test":split_sql_test_val["train"],
        "val":split_sql_test_val["test"],
    }
)
```

```
split_sql_dataset
```

```
    DatasetDict({
        train: Dataset({
            features: ['context', 'answer', 'question'],
            num_rows: 62861
        })
        test: Dataset({
            features: ['context', 'answer', 'question'],
            num_rows: 7858
        })
        val: Dataset({
            features: ['context', 'answer', 'question'],
            num_rows: 7858
        })
    })
```

```
DatasetDict({
    train: Dataset({
        features: ['question', 'context', 'answer'],
        num_rows: 62861
    })
    val: Dataset({
        features: ['question', 'context', 'answer'],
        num_rows: 7858
    })
    test: Dataset({
        features: ['question', 'context', 'answer'],
        num_rows: 7858
    })
})
```

Creating a "Prompt"

Now we need to create a prompt that's going to allow us to interact with our model when we desired the trained behaviour.

Think of this as a pattern that aligns the model with our desired outputs.

We need a single text prompt, as that is what the `SFTTrainer` we're going to use to fine-tune our model expects.

The basic idea is that we're going to merge the `question`, `context`, and `answer` into a single block of text that shows the model our desired outputs.

Let's look at what that block needs to look like:

```
{bos_token}### Instruction:
{system_message}

### Input:
{input}

### Context:
{context}

### Response:
{response}{eos_token}
```

Let's look at that from a completed prompt perspective to get a bit more information:

```
<|startoftext|>### Instruction:
You are a powerful text-to-SQL model. Your job is to answer questions about a database. You are given a question and context regardin
You must output the SQL query that answers the question.

### Input:
How many locations did the team play at on week 7?

### Context:
CREATE TABLE table_24123547_2 (location VARCHAR, week VARCHAR)

### Response:\nSELECT COUNT(location) FROM table_24123547_2 WHERE week = 7<|endoftext|>
```

As you can see, our prompt contains completed examples of our task. We're going to show our model many of these examples over and over again to teach it to produce outputs that are aligned with our goals!

First step, let's create a template we can use to call `.format()` on while constructing our prompts.

```
TEXT2SQL_TRAINING_PROMPT_TEMPLATE = """
{bos_token}### Instruction:
{system_message}

### Input:
{input}

### Context:
{context}

### Response:
{response}{eos_token}
"""


TEXT2SQL_INFERENCE_PROMPT_TEMPLATE = """\
{bos_token}### Instruction:
{system_message}

### Input:
{input}

### Context:
{context}

### Response:
"""
```

Now let's create a function we can map over our dataset to create the full prompt text block.

```
def create_sql_prompt(sample):
  SYSTEM_MESSAGE = f"""You are a powerful text-to-SQL model. Your job is to answer questions about a database. You are given a qu
  You must output the SQL query that answers the question."""

  full_prompt = TEXT2SQL_TRAINING_PROMPT_TEMPLATE.format(
      bos_token = "<|startoftext|>",
      eos_token = "<|endoftext|>",
      system_message = SYSTEM_MESSAGE,
```

```
      input = sample["question"],
      context = sample["context"],
      response = sample["answer"],
  )

  return {"text" : full_prompt}
```

Helper Function Begin.

I've created this helper-function to be able to see how our model is doing visibly, rather than only through metrics.

```
def create_sql_prompt_and_response(sample):
  SYSTEM_MESSAGE = f"""You are a powerful text-to-SQL model. Your job is to answer questions about a database. You are given a qu
  You must output the SQL query that answers the question."""

  full_prompt = TEXT2SQL_INFERENCE_PROMPT_TEMPLATE.format(
      bos_token = "<|startoftext|>",
      system_message = SYSTEM_MESSAGE,
      input = sample["question"],
      context = sample["context"]
  )

  ground_truth = sample["answer"]

  return {"full_prompt" : full_prompt, "ground_truth" : ground_truth}
```

Helper Function End.

Let's look at an example of a formatted prompt.

```
create_sql_prompt(split_sql_dataset["train"][0])
```

```
{'text': '\n<|startoftext|>### Instruction:\nYou are a powerful text-to-SQL model. Your job is to answer questions about a
database. You are given a question and context regarding one or more tables.\n  You must output the SQL query that answers
the question.\n\n### Input:\nHow many alt names does 1964-011a have?\n\n### Context:\nCREATE TABLE table_12141496_1
(alt_name VARCHAR, id VARCHAR)\n\n### Response:\nSELECT COUNT(alt_name) FROM table_12141496_1 WHERE id = "1964-011A"
<|endoftext|>\n'}
```

Great!

Now we can map this over our dataset!

- DatasetDict.map()

```
split_sql_dataset = split_sql_dataset.map(create_sql_prompt)
```

```
Map: 100%  62861/62861 [00:12<00:00, 5961.21 examples/s]
Map: 100%  7858/7858 [00:01<00:00, 6307.78 examples/s]
Map: 100%  7858/7858 [00:01<00:00, 6007.33 examples/s]
```

Load the Model And Preproccessing

Now for the moment we've all been waiting for...

Loading our model!

Let's use the `AutoModelForCausalLM` and `AutoTokenzier` classes from `transformers` to see just how easy this is.

- AutoModelForCausalLM
- AutoTokenizer
- GPT-2 Model Card

```
from transformers import AutoModelForCausalLM, AutoTokenizer

model_id = "gpt2"

gpt2_base_model = AutoModelForCausalLM.from_pretrained(model_id, device_map="auto")
```

```
gpt2_tokenizer = AutoTokenizer.from_pretrained(model_id)
```

```
Downloading (…)lve/main/config.json: 100%  665/665 [00:00<00:00, 18.8kB/s]
Downloading model.safetensors: 100%  548M/548M [00:04<00:00, 161MB/s]
Downloading (…)neration_config.json: 100%  124/124 [00:00<00:00, 6.39kB/s]
Downloading (…)olve/main/vocab.json: 100%  1.04M/1.04M [00:00<00:00, 5.47MB/s]
Downloading (…)olve/main/merges.txt: 100%  456k/456k [00:00<00:00, 14.4MB/s]
Downloading (…)/main/tokenizer.json: 100%  1.36M/1.36M [00:00<00:00, 21.7MB/s]
```

We need to make sure our tokenizer has a `pad_token` in order to be able to pad sequences so they're all the same length.

We'll use a little trick here to set our padding token to our eos (end of sequence) token to make training go a little smoother.

```
gpt2_tokenizer.pad_token = gpt2_tokenizer.eos_token
```

We also need to make sure we resize our model to be aligned with the token embeddings. If we didn't do this - we'd face a shape error while training!

```
gpt2_base_model.resize_token_embeddings(len(gpt2_tokenizer))
```

```
Embedding(50257, 768)
```

Now let's use the Hugging Face `pipeline` to see what generation looks like for our untrained model.

```
from transformers import pipeline, set_seed, GenerationConfig
generator = pipeline('text-generation', model=gpt2_base_model, tokenizer=gpt2_tokenizer)
set_seed(42)

def generate_sample(sample):
  prompt_package = create_sql_prompt_and_response(sample)

  generation_config = GenerationConfig(
      max_new_tokens=50,
      do_sample=True,
      top_k=50,
      temperature=1e-4,
      eos_token_id=gpt2_base_model.config.eos_token_id,
  )

  generation = generator(prompt_package["full_prompt"], generation_config=generation_config)
  print("---------------")
  print("Model Response:")
  print(generation[0]["generated_text"].replace(prompt_package["full_prompt"], ""))
  print("+++++++++++++++")
  print("Ground Truth")
  print(prompt_package["ground_truth"])


print(split_sql_dataset["test"][0]["question"])
```

```
what is the position is 2012 when the last title is n/a and the first season is 2011?
```

```
print(split_sql_dataset["test"][0]["answer"])
```

```
SELECT position_in_2012 FROM table_name_93 WHERE last_title = "n/a" AND first_season = "2011"
```

```
generate_sample(split_sql_dataset["test"][0]) # it is really bad...
```

```
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
---------------
Model Response:

SELECT * FROM table_name_93 WHERE position_in_2012 = 2012

### Query:

SELECT * FROM table_name_93 WHERE position_in_2012 = 2011
```

```
### Query:

SELECT * FROM
+++++++++++++++
Ground Truth
SELECT position_in_2012 FROM table_name_93 WHERE last_title = "n/a" AND first_season = "2011"
```

Training the Model

Now that we have our model set up, our tokenizer set up, we can finally begin training!

Let's look at our Trainer, and set some hyper-parameters:

- `per_device_train_batch_size` - this is a batch size that accomodates distributed training - a default we could use is `4`
- `gradient_accumulation_steps` - this is exactly the same as the previous notebook, it's a way to "simulate" a large batch size by collecting losses over multiple iterations - scaling them - and then combining them together. - a default we could use is `4`
- `gradient_checkpointing` - I'll let the authors speak for themselves here. In essence: This saves memory at the cost of computational time. - let's set this to `True`
- `max_grad_norm` - this is the value used for gradient clipping, which is a method of reducing vanishing gradient potential - let's use `0.3`
- `max_steps` - how many steps will we train for? - this is up to you
- `learning_rate` - how fast should we learn? - lets use `2e-4`
- `save_total_limit` - how many versions of the model will we save? - the default of `3` should work well
- `logging_steps` - how often we should log - up to you
- `output_dir` - where to save our checkpoints - up to you
- `optim` - which optimizer to use, you'll notice we're using a full precision paged optimizer - this is a performative and stable optimizer - but it uses extra memory - we should use `paged_adamw_32bit`
- `lr_scheduler_type` - we are once again using a cosine scheduler! - we should use `cosine`
- `evaluation_strategy` - we have an evaluation dataset, this defines when we should leverage it during training - we should use `steps`
- `eval_steps` - how many steps we should evaluate for - up to you
- `warmup_ration` - how many "warmup" steps we take to reach our full learning rate before we start decaying. This is a ration of our max_steps - the default value of `0.3` should work!
- `TrainingArguments`

```
from transformers import TrainingArguments
from trl import SFTTrainer

training_args = TrainingArguments(
    per_device_train_batch_size=4,
    gradient_accumulation_steps=4,
    gradient_checkpointing=True,
    max_grad_norm=0.3,
    max_steps=500,
    learning_rate=2e-4,
    save_total_limit=3,
    logging_steps=10,
    output_dir="sql_gpt2",
    optim="paged_adamw_32bit",
    lr_scheduler_type="cosine",
    evaluation_strategy="steps",
    eval_steps=50,
    warmup_ratio=0.05
)
```

Now, for our `SFTTrainer` AKA "Where the magic happens".

This `SFTTrainer` is going to take our above training arguments, our data, our model and our tokenizer, and train it all for us!

Notice that we're setting `max_seq_length` to the maximum context window of our model - this ensures we do not exceed our maximum context window, and will pad our examples up to the maximum context window!

❓ QUESTION ❓

What is the maximum input sequence length for GPT-2?

```
trainer = SFTTrainer(
 gpt2_base_model,
 dataset_text_field="text",
 train_dataset=split_sql_dataset["train"],
 eval_dataset=split_sql_dataset["val"],
 tokenizer=gpt2_tokenizer,
 max_seq_length=1024,
 args=training_args
)
```

```
Map: 100%  62861/62861 [00:38<00:00, 1825.45 examples/s]
Map: 100%  7858/7858 [00:03<00:00, 2665.61 examples/s]
```

Finally, we can call our `.train()` method and watch it go!

```
trainer.train()
```

```
You're using a GPT2TokenizerFast tokenizer. Please note that with a fast tokenizer, using the `__call__` method is faster th
`use_cache=True` is incompatible with gradient checkpointing. Setting `use_cache=False`...
/usr/local/lib/python3.10/dist-packages/torch/utils/checkpoint.py:429: UserWarning: torch.utils.checkpoint: please pass in u
  warnings.warn(
```

[500/500 27:41, Epoch 0/1]

Step	Training Loss	Validation Loss
50	0.699800	0.663562
100	0.658100	0.612232
150	0.610700	0.586239
200	0.590800	0.564680
250	0.575500	0.554606
300	0.577200	0.541493
350	0.596200	0.531490
400	0.522600	0.526507
450	0.529800	0.524419
500	0.539700	0.523870

```
TrainOutput(global_step=500, training_loss=0.6663816547393799, metrics={'train_runtime': 1664.7095,
'train_samples_per_second': 4.806, 'train_steps_per_second': 0.3, 'total_flos': 661370959872000.0, 'train_loss':
0.6663816547393799, 'epoch': 0.13})
```

Let's save our fine-tuned model!

```
trainer.save_model()
```

Testing our Model

Now that we have a fine-tuned model, let's see how it did

```
ft_gpt2_model = AutoModelForCausalLM.from_pretrained("sql_gpt2")
```

```
generator = pipeline('text-generation', model=ft_gpt2_model, tokenizer=gpt2_tokenizer, )
```

```
generate_sample(split_sql_dataset["test"][0])
```

```
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
---------------
Model Response:
SELECT position_in_2012 FROM table_name_93 WHERE last_title = "n/a" AND first_season = "2011" AND first_season = "2011" AND
+++++++++++++++
Ground Truth
SELECT position_in_2012 FROM table_name_93 WHERE last_title = "n/a" AND first_season = "2011"
```

That is *significantly* better.

❓ QUESTION ❓

What methods could we use to validate our SQL outputs?

`ANSWER` : we could use humans or we could try to run the model responses and ground truth queries directly on data and verify if they match

❓ QUESTION ❓

How would you extend this notebook to another use-case?

`ANSWER` : We could use it similarly for instance to generate recommendation letters for students by giving some context over the student education and achievements and training examples