ChartQuery-VLM Fine-Tuning

Below is a deep explanation of the project along with a cell-by-cell walkthrough of the code. This project demonstrates how to fine-tune a multimodal Vision-Language Model (VLM)—specifically, the **Qwen2-VL-7B-Instruct** model—on a chart question-answering task using techniques that include low-bit quantization and parameter-efficient fine-tuning via LoRA. I'll break down each markdown/code cell and explain what is happening.

---

1. Installing Dependencies and Setting Environment Variables

```
!pip3 install bitsandbytes peft trl
import os
os.environ["WANDB_DISABLED"] = "true"
```

- **Installation of Libraries:**
  - **bitsandbytes:** A library that supports 4-bit quantization. Quantization reduces the model's memory footprint by representing weights in lower precision. This is essential for handling large models on GPUs with limited memory.
  - **peft:** Stands for "Parameter-Efficient Fine-Tuning." It provides utilities to use methods like LoRA, which update only a small set of parameters during fine-tuning.
  - **trl:** A library from Hugging Face designed for training (and fine-tuning) large language models with reinforcement learning or supervised fine-tuning methods.
- **Disabling Weights & Biases (wandb):**
  - The environment variable `"WANDB_DISABLED"` is set to `"true"`, which turns off logging to the wandb platform. This can be useful if you want to avoid remote logging during local experiments.

---

2. Importing Modules and Setting Up the Environment

```
from datasets import load_dataset
import torch
from transformers import Qwen2VLForConditionalGeneration, Qwen2VLProcessor, BitsAndBytesConfig
from peft import LoraConfig, get_peft_model
from trl import SFTConfig, SFTTrainer

import warnings
warnings.filterwarnings("ignore")
```

- **Dataset and Torch:**
  - `load_dataset` is used for fetching datasets from Hugging Face's hub.
  - `torch` is imported for tensor operations and device management.
- **Model and Processor Imports:**
  - **Qwen2VLForConditionalGeneration:** The model class for the Qwen2-VL, which is designed for conditional generation tasks (e.g., generating text based on both an image and text prompt).
  - **Qwen2VLProcessor:** A helper to preprocess both images and text, combining them into the proper input format.
  - **BitsAndBytesConfig:** Used to specify quantization parameters when loading the model.
- **LoRA and Trainer:**
  - **LoraConfig & get_peft_model:** These functions from the `peft` package set up and apply LoRA, which adds trainable adapter layers that let you fine-tune the model without updating all its parameters.
  - **SFTConfig & SFTTrainer:** Provided by the `trl` library, these are used to configure and run supervised fine-tuning.
- **Warning Suppression:**

    - Any warnings are suppressed for cleaner output during training and evaluation.

- **Device Setup:**

  ```
  device = "cuda" if torch.cuda.is_available() else "cpu"
  print(f"Using device: {device}")
  ```

    - This snippet checks if a CUDA-enabled GPU is available. If not, the model will run on the CPU.

---

3. Defining Hyperparameters and Model Settings

```
MODEL_ID = "Qwen/Qwen2-VL-7B-Instruct"
EPOCHS = 1
BATCH_SIZE = 1
GRADIENT_CHECKPOINTING = True,  # Tradeoff between memory efficiency and computation time.
USE_REENTRANT = False,
OPTIM = "paged_adamw_32bit"
LEARNING_RATE = 2e-5
LOGGING_STEPS = 50
EVAL_STEPS = 50
SAVE_STEPS = 50
EVAL_STRATEGY = "steps"
SAVE_STRATEGY = "steps"
METRIC_FOR_BEST_MODEL="eval_loss"
LOAD_BEST_MODEL_AT_END=True
MAX_GRAD_NORM = 1
WARMUP_STEPS = 0
DATASET_KWARGS={"skip_prepare_dataset": True} # We have to put for VLMs
REMOVE_UNUSED_COLUMNS = False # VLM thing
MAX_SEQ_LEN=128
```

```
NUM_STEPS = (283 // BATCH_SIZE) * EPOCHS
print(f"NUM_STEPS: {NUM_STEPS}")
```

- **MODEL_ID:** Identifies the specific pretrained model on Hugging Face.
- **Training Hyperparameters:** These include the number of epochs, batch size, learning rate, and various steps for logging, evaluation, and saving.
- **Gradient Checkpointing:** Enabled to save GPU memory at the cost of extra computation during backpropagation.
- **Dataset Settings:**
    - `DATASET_KWARGS` with `"skip_prepare_dataset": True` and `REMOVE_UNUSED_COLUMNS = False` are options that help when preparing multimodal data for VLMs.
- **NUM_STEPS Calculation:**
    - Here, the total number of training steps is computed based on the dataset size and batch size.

---

4. Formatting the Data for a Chat-based Multimodal Setup

```
system_message = """You are a highly advanced Vision Language Model (VL
M), specialized in analyzing, describing, and interpreting visual data.
Your task is to process and extract meaningful insights from images, videos, a
nd visual patterns,
leveraging multimodal understanding to provide accurate and contextually rel
evant information."""
```

- **System Message:**
    - This message provides context to the model about its role and the type of input it will handle. It is the “persona” given to the model in the conversation.

```
def format_data(sample):
    return [
        {
            "role": "system",
            "content": [{"type": "text", "text": system_message}],
        },
        {
            "role": "user",
            "content": [
                {
                    "type": "image",
                    "image": sample["image"],
                },
                {
                    "type": "text",
                    "text": sample["query"],
                },
            ],
        },
        {
            "role": "assistant",
            "content": [{"type": "text", "text": sample["label"][0]}],
        },
    ]
```

- **Function Explanation:**
    - **Input:** A single sample from the dataset that contains an image, a query (user prompt), and a label (expected answer).
    - **Output:** A list of dictionaries mimicking a dialogue:
        - **System Message:** Sets the context.
        - **User Message:** Contains both the image (as an image-type content) and a text query.
        - **Assistant Message:** Contains the target answer.

This structure helps the processor and model to work with the dialogue format that is commonly used in instruction-tuning.

5. Loading and Preparing the Dataset

```
train_dataset, eval_dataset, test_dataset = load_dataset("HuggingFaceM4/ChartQA",
                                                         split=["train[:1%]", "val[:1%]", "test[:1%]"])

print(len(train_dataset))
print("-"*30)
print(train_dataset)
print("-"*30)
print(train_dataset[0])
print("-"*30)

train_dataset = [format_data(sample) for sample in train_dataset]
eval_dataset = [format_data(sample) for sample in eval_dataset]
test_dataset = [format_data(sample) for sample in test_dataset]
```

- **Dataset Loading:**
  - The **ChartQA** dataset is loaded using the Hugging Face Datasets library. Only 1% of each split (train, validation, test) is used, likely for a proof-of-concept or debugging purposes.
- **Printing Samples:**
  - The code prints dataset length and examples to inspect the raw data.
- **Data Formatting:**
  - Each sample is transformed using the `format_data` function so that the data follows the expected dialogue format.

---

6. Extracting and Inspecting a Sample for Inference

```
sample_data = test_dataset[0]
sample_question = test_dataset[0][1]["content"][1]["text"]
sample_answer = test_dataset[0][2]["content"][0]["text"]
sample_image = test_dataset[0][1]["content"][0]["image"]
```

```
print(sample_question)
print(sample_answer)
sample_image
```

- **Purpose:**
  - This section extracts one sample from the test set to inspect its components:
    - **Question:** The text query from the user.
    - **Answer:** The expected answer provided by the assistant.
    - **Image:** The visual input corresponding to the sample.

This check ensures that the data has been formatted correctly before proceeding to model inference and training.

---

7. Loading the Pretrained Model with Quantization

```
if device == "cuda":
    bnb_config = BitsAndBytesConfig(
        load_in_4bit=True,
        bnb_4bit_use_double_quant=True,
        bnb_4bit_quant_type="nf4",
        bnb_4bit_compute_dtype=torch.bfloat16
    )
    model = Qwen2VLForConditionalGeneration.from_pretrained(
        MODEL_ID,
        device_map="auto",
        quantization_config=bnb_config,
        use_cache=False
    )
else:
    model = Qwen2VLForConditionalGeneration.from_pretrained(
        MODEL_ID,
        use_cache=False
    )

processor = Qwen2VLProcessor.from_pretrained(MODEL_ID)
processor.tokenizer.padding_side = "right"
```

- **Quantization:**
  - When a CUDA device is available, the model is loaded using a 4-bit quantization configuration. The parameters:
    - **load_in_4bit=True:** Loads the model weights in 4-bit precision.
    - **Double Quantization & NF4:** Specific settings that help maintain a good balance between compression and model performance.
    - **Compute Dtype:** Uses `torch.bfloat16` for computations.
- **Device Map:**
  - The `device_map="auto"` argument helps automatically place model components across available GPUs.
- **Processor Setup:**
  - The processor is also loaded, and the tokenizer's padding side is set to "right" so that tokenization correctly handles the input sequences.

---

8. Creating a Text Generation Function

```
def text_generator(sample_data):
    text = processor.apply_chat_template(
        sample_data[0:2], tokenize=False, add_generation_prompt=True
    )

    print(f"Prompt: {text}")
    print("-"*30)

    image_inputs = sample_data[1]["content"][0]["image"]

    inputs = processor(
        text=[text],
        images=image_inputs,
        return_tensors="pt"
    )
    inputs = inputs.to(device)

    generated_ids = model.generate(**inputs, max_new_tokens=MAX_SEQ_LE
N)

    output_text = processor.batch_decode(
        generated_ids, skip_special_tokens=True
    )
    del inputs
    actual_answer = sample_data[2]["content"][0]["text"]
    return output_text[0], actual_answer
```

- **Building the Prompt:**
  - The function uses the processor's `apply_chat_template` method to combine the system and user messages into a single prompt. The flag `add_generation_prompt=True` likely appends special tokens or formatting that cues the model to generate an answer.
- **Preparing Inputs:**
  - The text prompt and image are both processed by the `processor` to produce tensors suitable for the model.
- **Generating Output:**
  - The model's `.generate()` method is then called to produce an output sequence, which is decoded back into text.
- **Return Values:**
  - The function returns both the generated answer and the actual (target) answer for comparison.

---

9. Testing the Text Generation Function

```
generated_text, actual_answer = text_generator(sample_data)
print(f"Generated Answer: {generated_text}")
print(f"Actual Answer: {actual_answer}")
```

- **Purpose:**
  - This cell calls the text generation function on a sample from the test set to verify that the model can produce an answer from the given image and text prompt.
  - It prints both the generated answer and the ground truth answer to help in evaluation.

---

10. Configuring LoRA for Parameter-Efficient Fine-Tuning

```
peft_config = LoraConfig(
    lora_alpha=16,
    lora_dropout=0.1,
    r=8,
    bias="none",
    target_modules=["q_proj", "v_proj"],
    task_type="CAUSAL_LM",
)

print(f"Before adapter parameters: {model.num_parameters()}")
peft_model = get_peft_model(model, peft_config)
peft_model.print_trainable_parameters()  # After LoRA trainable parameters in
creases. Since we add adapter.
```

- **LoRA Configuration:**
  - **lora_alpha, lora_dropout, and r:** These hyperparameters control the scaling, dropout, and rank (size of the low-rank updates) for the adapter layers.

    - **target_modules:** Specifies which layers of the model (here, “q_proj” for the text part and “v_proj” for the vision part) will be modified using LoRA.
    - **Task Type:** Set to `"CAUSAL_LM"` indicating a causal (autoregressive) language modeling objective.
- **Parameter Reporting:**
    - The code prints the total number of parameters before and after applying the LoRA adapters. This demonstrates that only a small subset of parameters are made trainable, making the fine-tuning process much more efficient.

---

11. Setting Up the Training Configuration with SFTTrainer

```
training_args = SFTConfig(
    output_dir="./output",
    num_train_epochs=EPOCHS,
    per_device_train_batch_size=BATCH_SIZE,
    per_device_eval_batch_size=BATCH_SIZE,
    gradient_checkpointing=GRADIENT_CHECKPOINTING,
    learning_rate=LEARNING_RATE,
    logging_steps=LOGGING_STEPS,
    eval_steps=EVAL_STEPS,
    eval_strategy=EVAL_STRATEGY,
    save_strategy=SAVE_STRATEGY,
    save_steps=SAVE_STEPS,
    metric_for_best_model=METRIC_FOR_BEST_MODEL,
    load_best_model_at_end=LOAD_BEST_MODEL_AT_END,
    max_grad_norm=MAX_GRAD_NORM,
    warmup_steps=WARMUP_STEPS,
    dataset_kwargs=DATASET_KWARGS,
    max_seq_length=MAX_SEQ_LEN,
    remove_unused_columns=REMOVE_UNUSED_COLUMNS,
    optim=OPTIM,
)
```

- **Training Arguments:**
    - This cell defines the configuration for supervised fine-tuning (SFT) using the `SFTConfig` class.
    - It covers aspects like output directory, batch sizes, learning rate, checkpointing, and evaluation strategies.
    - The use of gradient checkpointing and low-level optimizers (like `"paged_adamw_32bit"`) is tailored to balance efficiency with performance when working with a large model.

---

12. Preparing the Data Collation Function

```
collate_sample = [train_dataset[0], train_dataset[1]]  # for batch size 2.

def collate_fn(examples):
    texts = [processor.apply_chat_template(example, tokenize=False) for examp
le in examples]
    image_inputs = [example[1]["content"][0]["image"] for example in example
s]

    batch = processor(
        text=texts, images=image_inputs, return_tensors="pt", padding=True
    )
    labels = batch["input_ids"].clone()
    labels[labels == processor.tokenizer.pad_token_id] = -100
    batch["labels"] = batch["input_ids"]

    return batch

collated_data = collate_fn(collate_sample)
```

```
print(collated_data.keys())  # dict_keys(['input_ids', 'attention_mask', 'pixel_val
ues', 'labels'])
```

- **Collate Function:**
    - This function takes a list of formatted samples (each following the dialogue structure) and processes them into a single batch.
    - **Tokenization and Padding:**
        - The text parts are combined into one prompt per example using the processor.
        - The images are collected from the user message.
    - **Label Creation:**
        - The token IDs are copied to create labels for the language model loss computation.
        - Padding tokens are replaced with -100 (a common practice so that they are ignored in the loss calculation).
- **Purpose:**
    - This ensures that the training loop receives correctly formatted inputs, including both image and text data, along with corresponding labels.

---

13. Instantiating the Trainer and Running Training

```
trainer = SFTTrainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    data_collator=collate_fn,
    peft_config=peft_config,
    processing_class=processor.tokenizer,
)
```

```
print("-"*30)
print("Initial Evaluation")
metric = trainer.evaluate()
print(metric)
print("-"*30)

print("Training")
trainer.train()
print("-"*30)

trainer.save_model(training_args.output_dir)
```

- **Trainer Setup:**
    - **SFTTrainer:** This class is responsible for managing the training loop. It receives the model, the training/evaluation datasets, the collate function, and the PEFT (LoRA) configuration.
    - **Processing Class:** The tokenizer is provided so that the trainer can correctly handle input sequences.
- **Evaluation and Training:**
    - An initial evaluation is performed on the evaluation set to establish a baseline metric (using `eval_loss` as the metric).
    - The training loop is executed via `trainer.train()`.
    - After training, the fine-tuned model (including the adapter weights) is saved to the output directory.

---

14. Memory Cleanup and Reloading the Fine-tuned Model

```
import gc
import time

def clear_memory():
    if "inputs" in globals():
        del globals()["inputs"]
    if "model" in globals():
        del globals()["model"]
    if "processor" in globals():
        del globals()["processor"]
    if "trainer" in globals():
        del globals()["trainer"]
    if "peft_model" in globals():
        del globals()["peft_model"]
    if "bnb_config" in globals():
        del globals()["bnb_config"]
    time.sleep(2)

    gc.collect()
    time.sleep(2)
    torch.cuda.empty_cache()
    torch.cuda.synchronize()
    time.sleep(2)
    gc.collect()
    time.sleep(2)

    print(f"GPU allocated memory: {torch.cuda.memory_allocated() / 1024**3:.2f} GB")
    print(f"GPU reserved memory: {torch.cuda.memory_reserved() / 1024**3:.2f} GB")

clear_memory()
```

- **Purpose:**
    - Memory cleanup is critical when working with large models on GPU. This function deletes global references to large objects, calls Python’s garbage collector, and empties the GPU cache.
    - It then prints out the current GPU memory usage to confirm that resources have been freed.

15. Reloading the Model and Loading the Fine-tuned Adapter

```
if device == "cuda":
    bnb_config = BitsAndBytesConfig(
        load_in_4bit=True,
        bnb_4bit_use_double_quant=True,
        bnb_4bit_quant_type="nf4",
        bnb_4bit_compute_dtype=torch.bfloat16
    )
    model = Qwen2VLForConditionalGeneration.from_pretrained(
        MODEL_ID,
        device_map="auto",
        quantization_config=bnb_config,
        use_cache=True
    )
else:
    model = Qwen2VLForConditionalGeneration.from_pretrained(
        MODEL_ID,
        use_cache=True
    )

processor = Qwen2VLProcessor.from_pretrained(MODEL_ID)
processor.tokenizer.padding_side = "right"

print(f"Before adapter parameters: {model.num_parameters()}")
model.load_adapter("./output")
print(f"After adapter parameters: {model.num_parameters()}")
```

- **Reloading the Base Model:**
    - The model is reloaded (again with quantization if on GPU) but now with `use_cache=True` for more efficient inference.
- **Loading the Adapter:**

    - The previously saved LoRA adapter is loaded using `model.load_adapter("./output")`.
    - Comparing the number of parameters before and after loading shows that while the overall model size remains huge, only a few extra parameters (the adapters) have been added to the trainable set.

---

16. Running Final Inference with the Fine-tuned Model

```
generated_text, actual_answer = text_generator(sample_data)
print(f"Generated Answer: {generated_text}")
print(f"Actual Answer: {actual_answer}")
```

- **Final Test:**
    - The same `text_generator` function is used on the sample data to generate an answer with the fine-tuned model.
    - Printing both the generated answer and the actual answer allows for a quick qualitative assessment of the model's performance after fine-tuning.

Overall Summary

- **Multimodal Setup:** The project uses both image and text data. The data is formatted into a conversational structure (system, user, assistant) that guides the model during inference.
- **Efficiency Techniques:**
    - **4-bit Quantization:** Reduces the memory footprint of the model so that it can run on GPUs with less memory.
    - **LoRA (via peft):** Applies parameter-efficient fine-tuning so that only a small subset of parameters (the adapters) is updated during training.
- **Training Workflow:**
    - Data is preprocessed, collated, and fed into an SFT (Supervised Fine-Tuning) Trainer from the `trl` library.

    - The model is evaluated both before and after training to monitor improvements.
    - After training, the adapter weights are saved and later reloaded for inference.
- **Resource Management:**
    - Special care is taken to free GPU memory after training, ensuring that subsequent operations (like inference) run smoothly.

This detailed walkthrough shows how modern techniques (quantization, LoRA) can be combined to fine-tune large, multimodal models efficiently—even when hardware resources are limited.

References

ṆciteÖturn0search0Ộ

(For details on parameter-efficient fine-tuning and LoRA, you can refer to the Hugging Face documentation on PEFT and the TRL library.)

ṆciteÖturn0search1Ộ

(For more on quantization with bitsandbytes and working with multimodal models, check related Hugging Face blog posts and research articles.)