Fine-Tuning LLaMA-3.1-8B(Quantized) with DoRA

A comprehensive overview of the process for fine-tuning the LLaMA-3.1-8B-Instruct model using DoRA (Differentiated Optimization of Representations and Adapters)

Objective

The goal is to fine-tune the quantized `**LLaMA-3.1-8B-Instruct**` model using the DoRA technique on the `**FineTome-100k**` dataset to improve its performance for specific tasks while maintaining computational efficiency. The fine-tuned model and its adapters are then saved locally and uploaded to the Hugging Face Hub for sharing and deployment.

Prerequisites

- Hardware: Access to a GPU (T4 GPU on Google Colab).

- Dependencies Installed :

- `torch`
- `transformers`
- `datasets`
- `peft`
- `trl`
- `bitsandbytes`
- `huggingface_hub`

Process:

Environment Setup

The notebook begins by installing the required Python packages to ensure the environment is ready for fine-tuning and model management.

```
!pip install torch transformers datasets peft trl bitsandbytes
!pip install git+https://github.com/huggingface/peft.git
!pip install --upgrade trl
!pip install huggingface_hub
```

- Purpose: Installs core libraries (`torch`, `transformers`, `datasets`) for model training, `peft` for parameter-efficient fine-tuning (including DoRA), `trl` for supervised fine-tuning, `bitsandbytes` for quantization, and `huggingface_hub` for interacting with the Hugging Face Hub.

- Note: The `peft` library is installed from the Hugging Face GitHub repository to ensure the latest DoRA implementation is available.

Import Libraries and Configure Environment

The notebook imports necessary libraries and configures the environment to optimize GPU memory usage.

```
import torch
import os
os.environ["PYTORCH_CUDA_ALLOC_CONF"] = "expandable_segments:True"  # Reduce memory fragmentation
from transformers import AutoTokenizer, AutoModelForCausalLM, TrainingArguments
from peft import LoraConfig, get_peft_model, PeftModel
from trl import SFTTrainer
from datasets import load_dataset
```

- Memory Optimization: The environment variable `PYTORCH_CUDA_ALLOC_CONF` is set to `expandable_segments:True` to reduce CUDA memory fragmentation, critical for handling large models on limited GPU resources.

- Libraries:

 - `torch`: For tensor operations and GPU support.

 - `transformers`: For loading models and tokenizers.

 - `peft`: For parameter-efficient fine-tuning with DoRA/LoRA.

 - `trl`: For supervised fine-tuning using `SFTTrainer`.

 - `datasets`: For loading and processing datasets.

Model and Dataset Configuration

The model and dataset are defined for fine-tuning.

- Model: The `devatar/quantized_Llama-3.1-8B-Instruct` is a quantized version of the LLaMA-3.1-8B model, optimized for lower memory usage.

- Dataset: The `FineTome-100k` dataset from Hugging Face contains 100,000 conversation samples for fine-tuning.

- Output Directory: The fine-tuned model and adapters are saved to `./llama-3.1-8b-dora-finetuned`.

Load Model and Tokenizer

The model and tokenizer are loaded with configurations to optimize for memory efficiency.

```

```

```
tokenizer = AutoTokenizer.from_pretrained(model_id)
model = AutoModelForCausalLM.from_pretrained(
    model_id,
    device_map="auto",
    torch_dtype=torch.bfloat16,
    use_cache=False  # Disable cache to save memory
)
tokenizer.pad_token = tokenizer.eos_token
model.config.pad_token_id = tokenizer.pad_token_id
```

```

**Configure DoRA Fine-Tuning**

The model is prepared for DoRA fine-tuning using the `peft` library.

```

```
peft_config = LoraConfig(
    r=8,  # Low rank to save memory
    lora_alpha=16,
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM",
    target_modules=["q_proj", "k_proj", "v_proj", "o_proj"],  # Fewer modules
    use_dora=True  # Enable DoRA
)
model = get_peft_model(model, peft_config)
for name, param in model.named_parameters():
    if "lora" in name:  # Target DoRA/LoRA parameters
        param.requires_grad = True
```

```

- **DoRA Configuration:**

  - `r=8`: Low rank for the LoRA adapters to minimize memory usage.

  - `lora_alpha=16`: Scaling factor for LoRA updates.

  - `lora_dropout=0.05`: Dropout rate to prevent overfitting.

  - `bias="none"`: No bias terms in LoRA layers.

  - `task_type="CAUSAL_LM"`: Specifies the model is a causal language model.

  - `target_modules`: Limits fine-tuning to specific transformer modules (`q_proj`, `k_proj`, `v_proj`, `o_proj`) to reduce memory footprint.
```

- `use_dora=True`: Enables DoRA, which differentiates optimization for better performance.

Define Training Arguments

Training arguments are set to optimize the fine-tuning process for memory and performance.

```
training_args = TrainingArguments(
    output_dir=output_dir,
    per_device_train_batch_size=1,  # Minimize VRAM
    gradient_accumulation_steps=8,  # Effective batch size of 8
    optim="paged_adamw_8bit",
    learning_rate=2e-5,
    lr_scheduler_type="cosine",
    max_steps=500,
    logging_steps=10,
    save_strategy="steps",
    save_steps=100,
    fp16=True,
    report_to="none",
    overwrite_output_dir=True
)
```

Final Output

The fine-tuned model and adapters are available at `https://huggingface.com/avinashhm/llama-3.1-8b-dora-finetuned`.

Key Features of DoRA

- Parameter Efficiency: DoRA optimizes a smaller set of parameters compared to full fine-tuning, reducing memory and computational requirements.

- Performance: Combines the benefits of LoRA with differentiated optimization to achieve better performance on specific tasks.

- Memory Management: Configured to minimize VRAM usage, making it suitable for environments like Google Colab with a T4 GPU.

Future Improvements

- Hyperparameter Tuning: Experiment with different values for `r`, `lora_alpha`, and `learning_rate` to optimize performance.

- Larger Dataset: Use more samples from the `FineTome-100k` dataset if additional computational resources are available.

- Evaluation Metrics: Add evaluation steps to measure model performance post-fine-tuning.

Conclusion

A robust pipeline for fine-tuning the LLaMA-3.1-8B-Instruct model using DoRA, optimized for memory efficiency on a T4 GPU. The process includes environment setup, model and dataset configuration, fine-tuning, merging adapters, and uploading to the Hugging Face Hub. The resulting model is ready for deployment and can be accessed at the specified repository.

Reinforcement Fine-Tuning (GRPO) for Llama-3.1-8B-Instruct Model(Quantized)

Overview

This document outlines the process of reinforcement fine-tuning a quantized Llama-3.1-8B-Instruct model using the Unsloth library and GRPO (Generalized Reward-augmented Policy Optimization) in a Google Colab environment. The goal is to enhance the model's conversational performance by fine-tuning it on the Anthropic HH-RLHF dataset, leveraging 4-bit quantization and LoRA (Low-Rank Adaptation) for efficient training on limited hardware.

Objectives

- **Model Optimization**: Fine-tune a quantized Llama-3.1-8B-Instruct model to improve response quality for conversational tasks.
- **Efficiency**: Utilize 4-bit quantization and LoRA adapters to reduce memory usage and enable training on a single NVIDIA L4 GPU (22.161 GB memory).
- **Reinforcement Learning**: Apply GRPO with custom reward functions to align model outputs with desired conversational qualities.
- **Reproducibility**: Ensure the process is well-documented and repeatable in a Colab environment.

Methodology

The fine-tuning process involves:

1. **Environment Setup**: Install and configure necessary libraries, including bitsandbytes, unsloth, transformers, datasets, trl, and torch.
2. **Model Loading**: Load a pre-quantized Llama-3.1-8B-Instruct model with 4-bit quantization to optimize memory usage.
3. **LoRA Configuration**: Apply LoRA adapters to specific model modules for efficient fine-tuning.
4. **Dataset Preparation**: Preprocess the Anthropic HH-RLHF dataset to extract prompts, chosen, and rejected responses.
5. **Reward Functions**: Define two custom reward functions to evaluate model outputs based on response length and similarity to preferred responses.
6. **Training**: Use GRPOTrainer to fine-tune the model with specified hyperparameters.

7. **Model Saving**: Save the fine-tuned model for future use.

Code Implementation

1. Environment Setup

The script begins by ensuring the latest version of bitsandbytes is installed, followed by other dependencies. A runtime restart is required after installing bitsandbytes to ensure compatibility.

```
!pip uninstall bitsandbytes -y

!pip install bitsandbytes>=0.43.3 --extra-index-url https://download.pytorch.org/whl/cu12

# Verify bitsandbytes version

!pip show bitsandbytes

# Install other dependencies for Colab environment

if "COLAB_" in " ".join(os.environ.keys()):

   !pip install unsloth==2025.6.5 transformers datasets trl torch

else:

   !pip install unsloth==2025.6.5 transformers datasets trl torch
```

2. Library Imports

The necessary Python libraries are imported for model handling, dataset processing, and training.

```
import os

import re

import torch

from unsloth import FastLanguageModel

from datasets import load_dataset

from transformers import AutoTokenizer

from trl import GRPOConfig, GRPOTrainer
```

3. Model and Tokenizer Loading

The quantized Llama-3.1-8B-Instruct model is loaded with 4-bit quantization to reduce memory footprint. The tokenizer is configured with explicit padding settings.

```
model_name = "devatar/quantized_Llama-3.1-8B-Instruct"

model, tokenizer = FastLanguageModel.from_pretrained(

   model_name,

   load_in_4bit=True,

   device_map="auto",

)

# Set padding token explicitly

tokenizer.pad_token = tokenizer.eos_token if tokenizer.pad_token is None else
tokenizer.pad_token

tokenizer.padding_side = "right"

# Enable inference optimizations

FastLanguageModel.for_inference(model)
```

4. LoRA Configuration

LoRA adapters are added to specific transformer modules to enable efficient fine-tuning. Gradient checkpointing is enabled via Unsloth for memory efficiency.

```
model = FastLanguageModel.get_peft_model(

   model,

   r=16,  # LoRA rank

   target_modules=["q_proj", "k_proj", "v_proj", "o_proj", "gate_proj", "up_proj"],

   lora_alpha=16,

   lora_dropout=0,

   bias="none",

   use_gradient_checkpointing="unsloth",
```

```
    random_state=3407,
    use_rslora=False,
    loftq_config=None,
)
```

5. Dataset Preparation

The Anthropic HH-RLHF dataset is loaded and preprocessed to extract prompts and responses. A subset of 1,000 examples is used to manage computational resources.

```
dataset = load_dataset("Anthropic/hh-rlhf", split="train")
def format_hh_rlhf(example):
    chosen = example["chosen"]
    rejected = example["rejected"]
    chosen_parts = chosen.rsplit("Assistant:", 1)
    rejected_parts = rejected.rsplit("Assistant:", 1)
    prompt = chosen_parts[0].strip() if len(chosen_parts) > 1 else ""
    chosen_response = chosen_parts[-1].strip() if len(chosen_parts) > 1 else ""
    rejected_response = rejected_parts[-1].strip() if len(rejected_parts) > 1 else ""
    messages = [{"role": "user", "content": prompt}] if prompt else []
    formatted_prompt = tokenizer.apply_chat_template(messages, tokenize=False,
add_generation_prompt=True)
    return {
        "prompt": formatted_prompt,
        "tokens": tokenizer(formatted_prompt).input_ids,
        "chosen": chosen_response,
        "rejected": rejected_response
    }
dataset = dataset.select(range(1000)).map(format_hh_rlhf, batched=False)
```

```
max_prompt_length = max(len(x["tokens"]) for x in dataset) + 1
```

6. Reward Functions

Two reward functions are defined to evaluate model outputs:

- **match_format_exactly**: Rewards non-empty responses longer than 10 characters.
- **conversational_quality**: Rewards responses more similar to the "chosen" response than the "rejected" response, using sequence similarity (updated to use difflib).

```
import difflib

def match_format_exactly(prompts, completions, completion_ids=None, **kwargs):
    scores = []
    for completion in completions:
        response = completion
        score = 1.0 if response.strip() and len(response) > 10 else -1.0
        scores.append(score)
    return scores

def conversational_quality(prompts, completions, completion_ids=None, chosen=None, rejected=None):
    scores = []
    for completion, chosen_answer, rejected_answer in zip(completions, chosen, rejected):
        response = completion
        chosen_similarity = difflib.SequenceMatcher(None, response, chosen_answer).ratio()
        rejected_similarity = difflib.SequenceMatcher(None, response, rejected_answer).ratio()
        score = 1.5 if chosen_similarity > rejected_similarity else -0.5
        scores.append(score)
    return scores
```

7. GRPO Trainer Configuration

The GRPO trainer is configured with hyperparameters optimized for memory efficiency and quick training.

```
training_args = GRPOConfig(
    learning_rate=5e-6,
    weight_decay=0.1,
    warmup_ratio=0.1,
    lr_scheduler_type="cosine",
    optim="adamw_8bit",
    logging_steps=1,
    per_device_train_batch_size=1,
    gradient_accumulation_steps=16,
    num_generations=2,
    max_completion_length=512,
    max_steps=50,
    save_steps=25,
    max_grad_norm=1.0,
    report_to="none",
    output_dir="./outputs",
)
```

8. Training and Saving

The trainer is initialized and executed, followed by saving the fine-tuned model.

```
trainer = GRPOTrainer(
    model=model,
    processing_class=tokenizer,
```

```
    reward_funcs=[
        match_format_exactly,
        conversational_quality,
    ],
    args=training_args,
    train_dataset=dataset,
)


trainer.train()


model.save_pretrained_merged("fine_tuned_llama31_8b", tokenizer,
save_method="merged_16bit")
```

Fine-Tuning Dolly-V2-3B Using LoRA on LaMini-Instruction Dataset

Overview

This project demonstrates how to fine-tune the **Dolly-V2-3B**, a large language model developed by Databricks, using PEFT (Parameter-Efficient Fine-Tuning) with LoRA (Low-Rank Adaptation). The goal is to adapt the pre-trained model to follow instructions better by training it on the **LaMini-instruction dataset**.

Objectives

- Fine-tuning of a large language model using LoRA.

- Improve instruction-following capability on the LaMini-instruction dataset.

- Showcase deployment and inference capabilities post-training.

- Provide a reusable pipeline for future instruction tuning tasks.

Installation & Setup

```

! pip install torch transformers datasets peft bitsandbytes ipython

```

Key libraries:

- `transformers`: For loading and fine-tuning models.

- `peft`: To apply LoRA efficiently.

- `bitsandbytes`: For 8-bit quantized training to reduce memory usage.

- `datasets`: To handle data loading and preprocessing.

Data Loading and Preprocessing

Dataset Used:

- LaMini-instruction: A collection of instruction-response pairs for fine-tuning.

```

dataset = load_dataset("MBZUAI/LaMini-instruction", split='train')

small_dataset = dataset.select([i for i in range(200)])

```

Prompt Template:

```

prompt_template = """Below is an instruction that describes a task. Write a response that appropriately completes the request..."""

answer_template = """{response}"""

```

Each example is formatted with:

- `"prompt"`: Instruction template.

- `"answer"`: Ground truth response.

- `"text"`: Combined prompt + answer for training.

Tokenization:

- Model used: `databricks/dolly-v2-3b`

- Tokenizer: AutoTokenizer with `pad_token = eos_token`.

Model Configuration

Base Model:

- Name: `databricks/dolly-v2-3b` (~3 billion parameters)

- Precision: `float16`

- Device: GPU (with fallback to CPU)

Quantization:

- 8-bit loading enabled via `load_in_8bit=True`

LoRA Parameters:

Value	Parameter
256	Rank (`r`)
512	Alpha (`α`)
0.05	Dropout
`query_key_value`	Target Modules
Causal Language Modeling	Task Type

Total trainable parameters: ~83 million

Percentage of total model: ~2.93%

Training Setup

Training Arguments:

Value	Parameter
`dolly-3b-lora`	Output directory
1	Batch size (train/eval)
1e-4	Learning rate
3	Epochs
Per epoch	Evaluation strategy
Per epoch	Logging strategy
Per epoch	Save strategy
Yes (if GPU is available)	FP16 enabled

Training Results

Loss Metrics:

Validation Loss	Epoch	Training Loss
0.5622	1	0.5877
0.5807	2	0.3219
0.6278	3	0.1990

Observation: Training loss decreases steadily, but validation loss increases slightly in later epochs—indicating potential overfitting or limited dataset size.

Conclusion

- Efficient fine-tuning of a large LLM using LoRA.

- Practical application for instruction-following tasks.

The trained model is ready for deployment in real-world applications such as chatbots, customer service agents, or content generation tools.