CS 696 Applied Large Language Models
Spring Semester, 2025
Doc 23 Exam
Apr 22, 2025

%%capture

!pip install transformers accelerate datasets bitsandbytes trl vllm llm-blender unsloth rich tqdm

But what is the base?

Before running the notebook, the following packages need to be installed on a terminal tab

- pip install transformers bitsandbytes 'accelerate>=0.26.0' datasets trl evaluate

Also, the following options were chosen when opening this notebook

- Large - 8 CPUs & 16 GM RAM
- 1 GPU
- PyTorch notebook

Question 7

Use bitsandbytes to quantize the model you used in assignment two. Compare the memory (CPU and GPU) and run time required by a FP16 quantized model, an 8-bit quantized model, and a 4-bit model. How does the 8-bit model output compare with the full model?

JupyterHub Settings

Setting	Value
CPU & RAM	8 Cores & 16GB RAM
GPU	1 GPU
Notebook Type	PyTorch Notebook

Installed Packages

package	version
torch	2.5.1
torchaudio	2.5.1
torchvision	0.20.1
transformers	4.48.3
bitsandbyte	0.45.2
accelerate	1.3.0
trl	0.15.2
peft	0.14.0
vllm	0.7.3
unsloth	2025.3.14
datasets	3.4.0
xformers	0.0.27.post2

Requirments.txt

langchain
langgraph
langsmith
langchain-openai
langchain-text-splitters
langchain-community

References

LoRA and QLoRA- Effective methods to Fine-tune your LLMs in detail.
https://medium.com/@levxn/lora-and-qlora-effective-methods-to-fine-tune-your-llms-in-detail-6e56a2a13f3c

Preference Tuning LLMs: PPO, DPO, GRPO — A Simple Guide
https://anukriti-ranjan.medium.com/preference-tuning-llms-ppo-dpo-grpo-a-simple-guide-135765c87090

Overcoming Catastrophic Forgetting: A Simple Guide to Elastic Weight Consolidation
https://towardsai.net/p/l/overcoming-catastrophic-forgetting-a-simple-guide-to-elastic-weight-consolidation

Original
VRAM used: 7.117321014404297 GB
RAM used: 1.5459480285644531 GB
Run time required: 0:00:22.535928

1. **Compare the advantages and disadvantages of fine-tuning a pretrained LLM versus continuous pretraining. Provide examples of how each approach is more beneficial.**

Fine-tuning: Taking an existing, fully pretrained model and training it further on a task-specific (or domain-specific) dataset. Used to adapt a pretrained LLM for optimal performance on a targeted domain or task. It is cheaper, faster, and tends to yield larger gains for specific use cases.

Advantages

- **Task specific**: Fine-tuning makes the model perform extremely well on a particular task (classification, QA, summarization, etc.).
- **Lower computational cost**: Requires far fewer compute resources than large-scale pretraining on billions of tokens. Fine-tuning can be done with smaller datasets and fewer steps.
- **Faster turnaround**: Because of the smaller datasets and lower cost, training can be completed quickly (hours to days, depending on hardware and model size).
- **Highly effective for specialized domains**: Fine-tuning with domain-specific data (e.g., legal, medical) often gives substantial performance gains.

Disadvantages

- **Narrowed scope**: While it excels at the specific task, the model may lose some of its versatility or broad knowledge if not done carefully.
- **Potential catastrophic forgetting**: Excessively tuning on a small dataset may cause the model to "forget" parts of its general knowledge.
- **Ongoing maintenance**: If the underlying knowledge shifts (e.g., new research breakthrough on a more optimized LLM like DeepSeek), then you often need repeated fine-tunes or a strategy to preserve old knowledge plus incorporate new information.

Scenarios where Fine-Tuning is More Beneficial

Task-centric scenarios: For a clear downstream use case (e.g., a chatbot for customer support) and need maximum accuracy or reliability on that single task.

Domain-specific adaptation: Specialized data that is not covered well by the model's original pretraining (e.g., finance, law, medicine).

Resource constraints: To keep compute and data requirements low and complete training quickly.

Continuous pretraining: Continuing the large-scale pretraining process on additional unlabeled or broader-domain data to update or expand the model's general knowledge before (optionally) moving on to fine-tuning. Used to refresh or expand the model's general capabilities by exposing it to new or more diverse data. This keeps it up to date annd prepares it for a wide range of downstream tasks.

Advantages

- **Expanded and updated knowledge**: Overcomes "staleness" by training on more recent or diverse text, thus keeping the model's internal representations current.
- **Better general capabilities**: With more data, the model can learn additional linguistic patterns, styles, and factual information. This would increase its overall robustness.
- **Improved performance on multiple tasks**: When new pretraining data is broad, the model may see gains in tasks it was never specifically fine-tuned on.

Disadvantages

- **High computational cost**: Continuing large-scale pretraining can be extremely expensive, often requiring specialized hardware and large budgets.
- **Risk of catastrophic forgetting**: When new data differs significantly from the original pretraining set, older knowledge may degrade if the new data distribution is not carefully balanced.
- **Less targeted**: Continuous pretraining does not specifically tune the model to a single well-defined task. You still may need an additional fine-tuning step for best performance on specialized tasks.
- **Complex data sourcing**: Curating large additional datasets that are both relevant and high-quality can be difficult.

Scenarios where Continuous Pretraining is More Beneficial

Maintaining an up-to-date general model: If your product or service relies on having the latest information (e.g., real-world events, scientific breakthroughs). Therefore, it would be beneficial to continuously pretrained models stay more relevant.

Broadening coverage: If you want your LLM to gain stronger multilingual capabilities or cover emerging domains not present in the original pretraining set (e.g., new coding languages).

Multiple downstream use cases: If your organization requires the same base model to serve many tasks or domains. By refreshing the base model with continuous pretraining this can raise the performance across the board.

The best choice or combination depends on factors like budget, timeframe, domain specificity, and how critical it is for the LLM to stay current on new knowledge.

Which Question?

Code Notebook Python 3 (ipykernel)

[2]:
```
!pip install transformers
```

```
Collecting transformers
  Using cached transformers-4.49.0-py3-none-any.whl.metadata (44 kB)
Requirement already satisfied: filelock in /opt/conda/lib/python3.11/site-packages (from transf
ormers) (3.13.1)
Collecting huggingface-hub<1.0,>=0.26.0 (from transformers)
  Downloading huggingface_hub-0.29.3-py3-none-any.whl.metadata (13 kB)
Requirement already satisfied: numpy>=1.17 in /opt/conda/lib/python3.11/site-packages (from tra
nsformers) (1.26.4)
Requirement already satisfied: packaging>=20.0 in /opt/conda/lib/python3.11/site-packages (from
transformers) (24.1)
Requirement already satisfied: pyyaml>=5.1 in /opt/conda/lib/python3.11/site-packages (from tra
nsformers) (6.0.1)
Collecting regex!=2019.12.17 (from transformers)
  Using cached regex-2024.11.6-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metad
ata (40 kB)
Requirement already satisfied: requests in /opt/conda/lib/python3.11/site-packages (from transf
ormers) (2.32.3)
Collecting tokenizers<0.22,>=0.21 (from transformers)
  Downloading tokenizers-0.21.1-cp39-abi3-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metada
ta (6.8 kB)
```

4. What types of tasks would GRPO be preferable over DPO and vice versa?

   Use GPO for summarization, content moderation, translation and creating writing.
   Use GRPO for math and game tasks

4) GRPO vs DPO Task
    a) GRPO should be used on tasks that require complex reasoning.

Not true -5

    b) DPO should be used on simple tasks. DPO allows for direct control over the LLM. It allows for the LLM's behavior and preferences to be changed easily.

1. Compare the advantages and disadvantages of finetuning a pretrained LLM versus continuous pretraining. Provide examples of how each approach is more beneficial.

Continuous pretraining is first training the model on a large dataset like the internet to understand text. Then continuing to pretraining the model on a specific subset of data like github public repos to make LLM have a better understanding of specific domains like coding. Now the LLM could be finetuned for a task like unit test generation.

Fine tuning an LLM is getting it to become better at certain tasks like giving it coding prompts and the corresponding code output. The llm gets fine tuned on code generation.

Computational costs, differenece in data sizes? -2

Explain Error

Collapse Output

```
=== 8-bit Model ===
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
Cell In[7], line 83
     80 input_text = "Generate creative project ideas for a fine-tuning approach for cancer ris
k prediction."
     82 # Compare models
---> 83 compare_models("facebook/bart-large-cnn", input_text)

Cell In[7], line 56, in compare_models(model_name, input_text)
     54 print("\n=== 8-bit Model ===")
     55 start_time = time.time()
---> 56 model_8bit, tokenizer_8bit = load_and_quantize_model(model_name, "8bit")
     57 print_memory_usage("8-bit Model Loaded")
     58 output_8bit, time_8bit = generate_output(model_8bit, tokenizer_8bit, input_text, devic
e)

Cell In[7], line 21, in load_and_quantize_model(model_name, quantization_level)
     19     model = AutoModelForCausalLM.from_pretrained(model_name, torch_dtype=torch.float16)
     20 elif quantization_level == "8bit":
---> 21     model = AutoModelForCausalLM.from_pretrained(model_name, load_in_8bit=True, device_
map="auto")
     22 elif quantization_level == "4bit":
     23     model = AutoModelForCausalLM.from_pretrained(model_name, load_in_4bit=True, device_
map="auto")

File /opt/conda/lib/python3.11/site-packages/transformers/models/auto/auto_factory.py:564, in _
BaseAutoModelClass.from_pretrained(cls, pretrained_model_name_or_path, *model_args, **kwargs)
```

Q7

FULL SIZE Model

Initial RAM usage: 647.1484375 MB
Final RAM usage: 1426.49609375 MB
RAM usage: 779.34765625 MB

Initial VRAM usage: 0.0 MB
Final VRAM usage: 7296.52490234375 MB
VRAM usage: 7296.52490234375 MB

Start Time: 1741991392.2954392 s
End Time: 1741991417.075232 s
Time Elapsed: 24.77979278564453 s

Q7

16 Bit Model

Initial RAM usage: 1555.57421875 MB
Final RAM usage: 2900.71875 MB
RAM usage: 1345.14453125 MB

Initial VRAM usage: 2332.759765625 MB
Final VRAM usage: 9244.77490234375 MB
VRAM usage: 6912.01513671875 MB

Start Time: 1741995458.8022563 s
End Time: 1741995486.3418183 s
Time Elapsed: 27.539561986923218 s

Q7

8 Bit Model

Initial RAM usage: 647.203125 MB
Final RAM usage: 1555.109375 MB
RAM usage: 907.90625 MB

Initial VRAM usage: 0.0 MB
Final VRAM usage: 2332.759765625 MB
VRAM usage: 2332.759765625 MB

Start Time: 1741992199.4445302 s
End Time: 1741992426.3292723 s
Time Elapsed: 226.88474202156067 s

QLoRA performs even better than LoRA as it quantizes the weights produced by LoRA to use less bits while offering similar performance. However, it runs slower than LoRA does.

From QLORA: Efficient Finetuning of Quantized LLMs

> Our method, QLORA, uses a novel high-precision technique to quantize a pretrained model to 4-bit, then adds a small set of learnable Low-rank Adapter weights that are tuned by backpropagating gradients through the quantized weight

Comparison of performance between Phi-3 FP16, 8-bit and 4-bit quantization

Model Type	Memory Utilization	VRAM Utilization	Trainable Params	Average Generation Time
FP16-bit Quantized	984.21 MB	7288.38 MB	3.80 B	8.53 s
8-bit Quantized	1097.97 MB	3868.25 MB	3.80 B	13.84 s
4-bit Quantized	1138.34 MB	2324.63 MB	3.80 B	13.22 s

Figure 4: Comparison of results between each model

Quantization Mode	Memory Usage (GB)	Average Runtime (s)	Output Quality
FP16	6.52 GB	3.32 Seconds	High
8-bit	6.52 GB	5.55 Seconds	Slightly lower
4-bit	4.15 GB	5.89 Seconds	Noticeably lower

	Load Time Seconds	CPU Memory MB	GPU Memory	Inference seconds	Output size char
FP16	9.2	2,207	7,290	13.5	2,400
8-bit	8.8	2,500	3,876	27	2,500
4-bit	9.6	2,558	2,333	14	2,200

Model	Total Time Elapsed	Model Loading Time	Pipeline Time	Tokenizer Prompt Time	Tokenizer Decode Time	CPU Memory Usage	GPU Memory Usage
FP16 Model	20.00 sec	3.58 sec	0.0009 sec	0.17 sec	16.2 sec	1.37 GB	7.13 GB
8-bit Quantized Model	64.62 sec	3.82 sec	0.0009 sec	0.15 sec	60.6 sec	1.44 GB	3.79 GB
4-bit Quantized Model	18.44 sec	3.89 sec	0.001 sec	0.36 sec	14.2 sec	1.39 GB	2.28 GB

Configuration	Average Time (s)	Average Max RAM Used (Bytes)	Average Max VRAM Used (Bytes)
Default	27.13	1,497,801,523	15,293,378,560
FP16-Bit	26.91	2,087,607,500	22,935,783,424
8-Bit	48.07	20,88,942,796	22,935,783,424
4-Bit	29.16	20,89,299,968	22,935,783,424

```
quantized_8bit_config = BitsAndBytesConfig(load_in_8bit=True, load_in_4bit=False, bnb_4bit_compute_dtype=torch.float16)

phi_model = AutoModelForCausalLM.from_pretrained(
    phi_version,
    attn_implementation='eager',
    trust_remote_code=True,
    device_map=device,
    quantization_config=quantized_8bit_config
)
```

8& 9

Model	Training Time	CPU Memory Usage	GPU Memory Usage
GPT-2 (DPO - Direct Preference Optimization)	272.88 sec	1.59 GB	1.91 GB
GPT-2 (Unsloth)	260.24 sec	1.87 GB	0.18 GB

Table 1-1

Model	Memory Used
GPT-2 with DPO	2.39 GB

Training Details Original CPU Memory Usage: 1823.50 MB Original GPU Memory Usage: 1955.29 MB Execution time: 49.63 Step Training Loss 10 4.070600 20 2.350000 30 1.630800 40 0.709200 50 1.432200 60 0.732100 70 0.346300 80 0.158900 90 0.004600 100 0.025000 110 0.079400 120 0.001300 130 0.008400 140 0.010200

I provided 5 simple python questions for code generation task, and either of the models generated valid answers. One obvious different outcomes I can see is that the model with unsloth is generating more python code than the model in Q8 which generated more text instead of code. None of the code being generated was quite correct, but I would like to conclude that the model with unsloth would generate better code if I can feed larger datasets.

🔷 DPO Preference Dataset Creation 🔷

Loading dataset: google-research-datasets/mbpp (split: train)...

Dataset loaded! 374 examples available.

🔶 Loading Quantized Model 🔶

Model and tokenizer loaded successfully!

After model load Memory Usage

CPU Memory (RSS):	906.27 MiB
GPU Memory Allocated:	2340.13 MiB
GPU Memory Reserved:	2362.00 MiB
GPU Total Memory:	45515.00 MiB

🔷 Generating Preference Pairs 🔷

Output()

Progress: Collected **95** valid pairs after **128** examples.

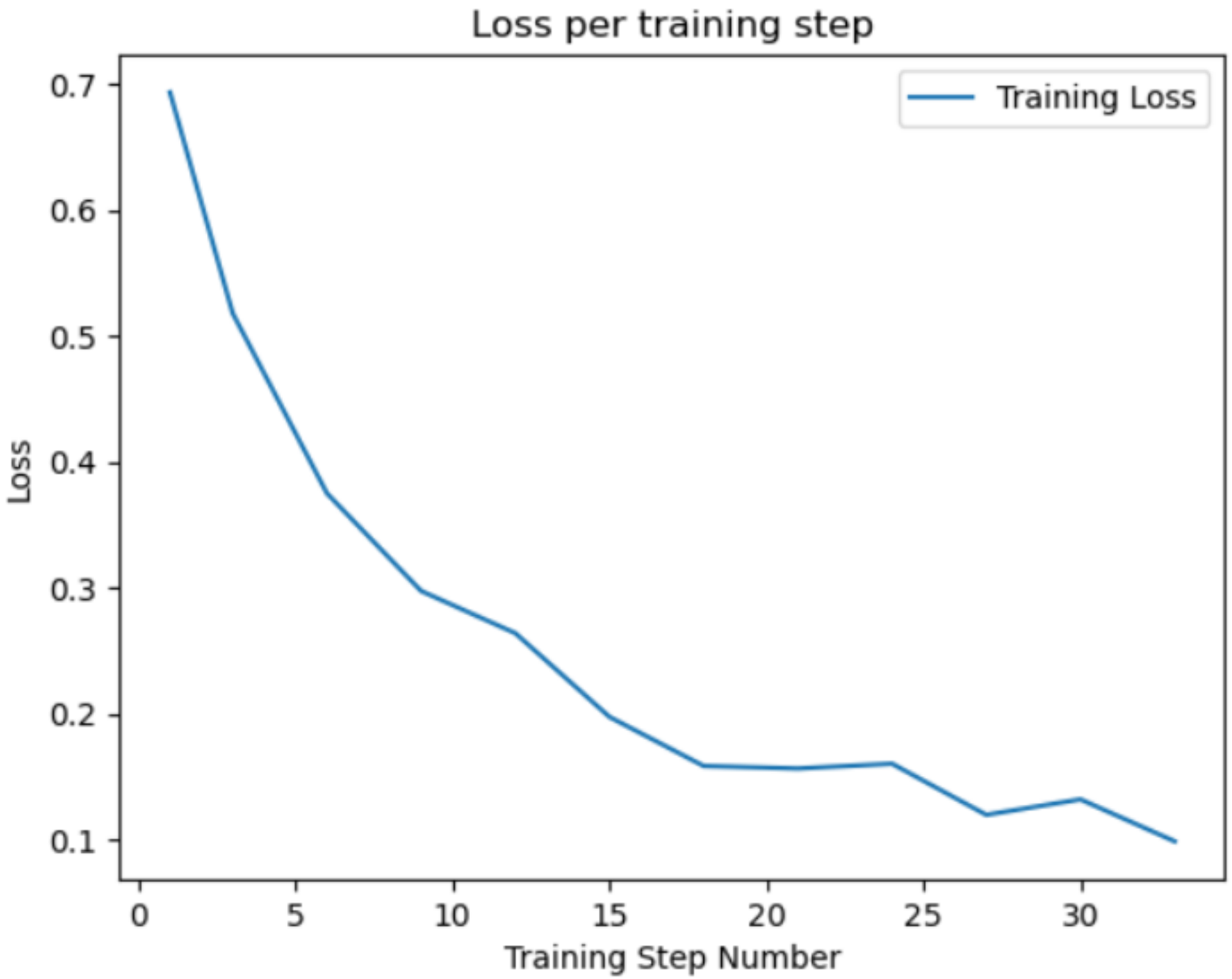
After 95 pairs Memory Usage

⏱ **Elapsed Time: 100.92s**

CPU Memory (RSS):	1595.86 MiB
GPU Memory Allocated:	2348.58 MiB
GPU Memory Reserved:	17122.00 MiB
GPU Total Memory:	45515.00 MiB

Q10

Step	Training Loss	reward	reward_std	completion_length
1	135.431800	-962.375000	228.641304	256.000000
2	6590.512700	-896.250000	184.137276	235.437500
3	1058.562000	-857.750000	286.585052	239.125000
4	159302.859400	-825.000000	181.026787	248.375000
10	1228.916100	-846.312500	258.462738	256.000000
11	118.637000	-934.500000	224.726295	256.000000
12	986.192800	-979.125000	144.186363	252.562500
13	2.797300	-728.125000	290.010208	220.500000
14	3709.835900	-883.312500	312.059906	221.375000
15	647285440.000000	-889.250000	129.812836	246.937500
80	0.033800	-818.625000	305.275803	214.687500
81	0.037600	-627.562500	414.162659	180.187500
82	0.039900	-736.625000	276.836624	207.375000
83	0.050300	-689.437500	244.548012	222.250000
84	0.026600	-671.187500	400.857178	182.250000
85	0.031000	-766.125000	313.988708	198.562500
86	0.026000	-844.375000	268.409988	219.750000
87	0.036000	-615.000000	349.350861	180.562500
88	1.529000	-648.125000	341.696182	169.562500
89	0.035600	-784.812500	295.637573	222.500000


Loss per training step
Training Loss
Loss
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
5
10
15
20
25
30
Training Step Number

```
# Convert dataset format
def mbpp2trainer_format(sample):
    # Convert the labels of the MBPP to the ones used in DPOTrainer
    # (prompt, chosen and rejected). For now, we will ignore the
    # rejected column since MBPP does not provide rejected answers.
    return {
        "prompt": sample["text"],
        "chosen": sample["code"],
        "rejected": ""
    }

# Convert the label names to usable DPOTrainer/GRPOTrainer names. Remove the old columns in the set
train_dataset = train_dataset.map(mbpp2trainer_format, remove_columns=train_dataset.column_names)
half_dataset  = half_dataset.map(mbpp2trainer_format, remove_columns=half_dataset.column_names)
```

```
train_dataset = load_dataset("google-research-datasets/mbpp", split="train")
train_dataset = train_dataset.rename_column('text', 'prompt')
train_dataset = train_dataset.rename_column('code', 'chosen')
train_dataset = train_dataset.add_column('rejected', [''] * train_dataset.shape[0])
train_dataset = train_dataset.remove_columns(['task_id', 'test_list', 'test_setup_code',
```

Test the model. Run 5 times to record the average run-time. Print the final result.¶

Training Type	Memory Utilization	VRAM Utilization	Trainable Params	Epoch Training Time For 3 Epochs	Final Training Loss	Average Generation Time
DPO, Default DPOTrainer	1394.02 MB	487.46 MB	124 M	2 min 57 sec	0.098	1.47 s
DPO, Unsloth LoRA Rank 16	1232.89 MB	137.77 MB	1.62 M	1 min 28 sec	0.001	4.39 s
DPO, Default DPOTrainer vLLM	7743.07 MB	7743.07 MB	124 M	2 min 57 sec	0.098	0.62 s
GRPO, Unsloth LoRA Rank 16	1260.32 MB	134.67 MB	1.62 M	2 hr 07 min	121.18	4.47 s

Q10

```
max_seq_length = 2048

model, tokenizer = FastLanguageModel.from_pretrained(
    model_name = "openai-community/gpt2",
    max_seq_length = max_seq_length,
    dtype = torch.bfloat16,
    load_in_4bit = False,
)

if tokenizer.pad_token is None:
    tokenizer.pad_token    = tokenizer.eos_token # Set pad_token to eos_token if missing

# Do model patching and add fast LoRA weights
model = FastLanguageModel.get_peft_model(
    model,
    r = 16, # Rank
    target_modules = ["c_attn", "c_proj"],
    lora_alpha = 64,
    lora_dropout = 0, # Supports any, but = 0 is optimized
    bias = "none",    # Supports any, but = "none" is optimized
    # [NEW] "unsloth" uses 30% less VRAM, fits 2x larger batch sizes!
    use_gradient_checkpointing = "unsloth", # True or "unsloth" for very long context
    random_state = 3407,
    max_seq_length = max_seq_length,
    task_type="CAUSAL_LM",
).to(dtype=torch.bfloat16)
```

Q10

```
# Configure the GRPOTrainer config and set the padding token ID
# Ensure we log the loss so we can plot over time
training_args = GRPOConfig(
    output_dir="GPT2-124M-GRPO",
    save_steps=1, # Save the model every step
    save_total_limit=3, # Limit saves to 3 most recent
    # padding_value = tokenizer.pad_token_id,
    logging_dir="./logs",
    logging_steps=1,
    report_to=["tensorboard"],
    logging_first_step=True,
    per_device_train_batch_size=2, # So we don't have too much memory allocation
    gradient_accumulation_steps=16,
    learning_rate=1e-5, # default 1e-4
    num_generations=2,
    num_train_epochs=3,
    max_grad_norm=1.0,
    # use_vllm=True,
)
```

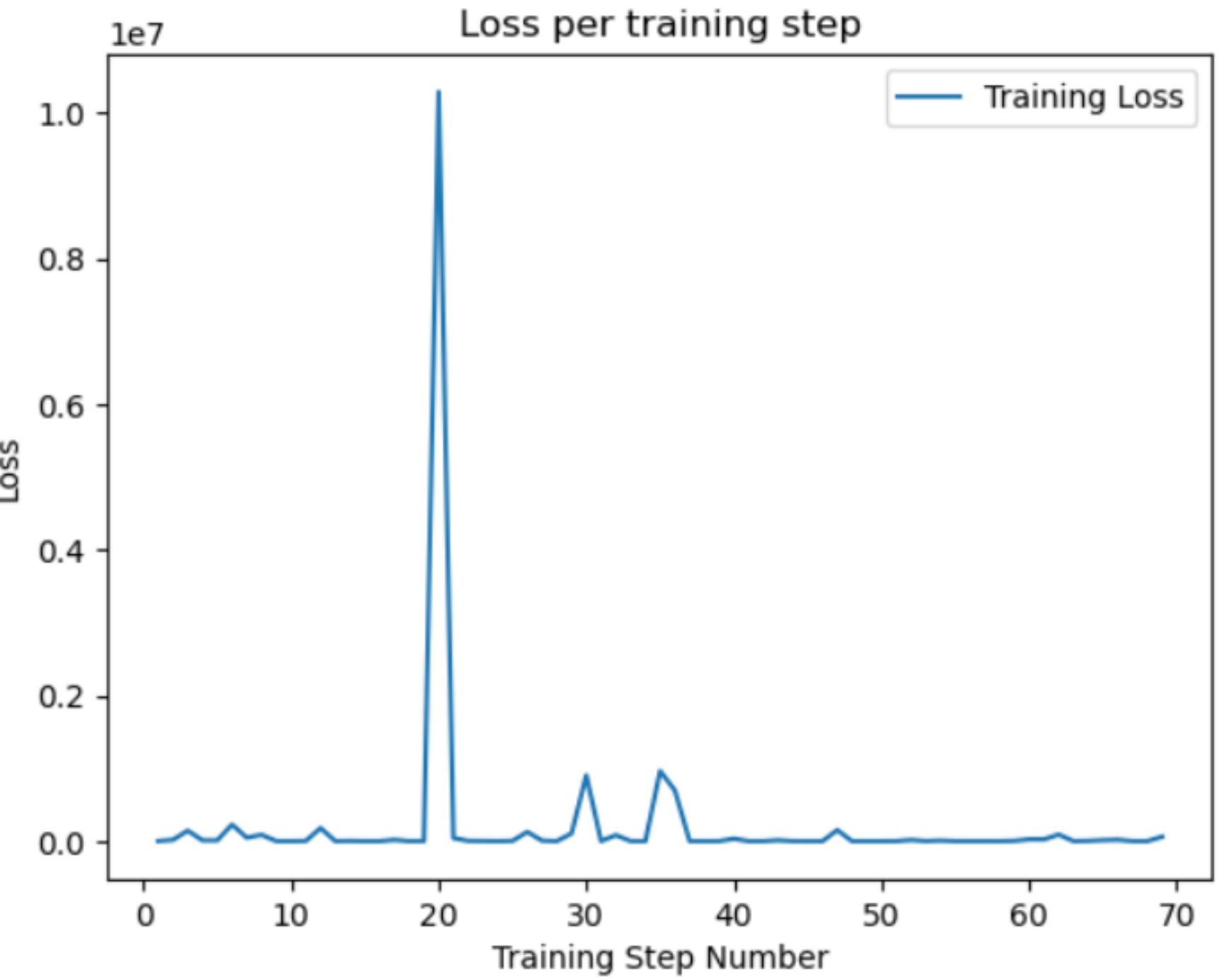
Q10

```
def reward_len(completions, **kwargs):
    return [-abs(100 - len(completion)) for completion in completions]

# Generate the GRPOTrainer helper function
grpo_trainer = GRPOTrainer(
    model,
    args=training_args,
    train_dataset=train_dataset,
    # tokenizer=tokenizer,
    reward_funcs=reward_len,
    processing_class=tokenizer,
    # num_generations_per_prompt=2,
)

grpo_trainer.train()
```

Q10



Step	Training Loss
1	1,649.7
2	17,721.6
3	144,172.7
4	11,415.1
5	10,959.0
6	227,587.8
7	47,801.8
8	87,213.0
9	672.3
10	125.6
11	2,916.7
12	179,805.8
13	441.1
14	3,222.9
15	63.5
16	321.9
17	17,864.5
18	515.4
19	1,644.8
20	10,290,129.0

```
def run_experiment(model, tokenizer, input_text, model_variant=""):

    torch.cuda.empty_cache()

    process = psutil.Process(os.getpid())
    cpu_mem_before = process.memory_info().rss / (1024 * 1024)
    gpu_mem_before = torch.cuda.memory_allocated(torch.device("cuda:0")) / (1024**2)

    print(f"Starting experiment for {model_variant}")
    print("CPU Memory before tokenization:", cpu_mem_before, "MB")
    print("GPU Memory before tokenization:", gpu_mem_before, "MB")

    start = time.time()
    inputs = tokenizer(input_text, return_tensors="pt").to("cuda")
    tokenization_time = time.time() - start

    start_gen = time.time()
    outputs = model.generate(
        inputs["input_ids"],
        max_length=200,
        do_sample=True,
        pad_token_id=tokenizer.eos_token_id,
        attention_mask=inputs["attention_mask"],
    )
    generation_time = time.time() - start_gen

    start_dec = time.time()
    generated_text = tokenizer.decode(outputs[0], skip_special_tokens=True)
    decoding_time = time.time() - start_dec

    cpu_mem_after = process.memory_info().rss / (1024 * 1024)
    gpu_mem_after = torch.cuda.memory_allocated(torch.device("cuda:0")) / (1024**2)
```

```
model, tokenizer = FastLanguageModel.from_pretrained(
    model_name = "openai-community/gpt2",
    max_seq_length = 150,
    fast_inference = True, # Enable vLLM
)
```

Q8

model = AutoModelForCausalLM.from_pretrained("openai-community/gpt2", device_map='auto')

Output

function findChains(n) return -1;\n\n

For simplicity: if you\'re interested in the set of all the known pairs where i can be zero, and each iteration of the chain consists of the given set of sets, and then you add (and subtract) it from the other set of sets, you get\n\n

[n+1] + [n+2] + [n +3] = 1,\n\n

Q9

model_name = "unsloth/zephyr-sft-bnb-4bit",

Output

```
def longest_chain(arr,n): \r\n
    arr.sort() \r\n
    ans = 1 \r\n
    for i in range(1,n):\r\n
        if arr[i] == arr[i-1]:\r\n
            ans = max(ans,i-1):\r\n
            ans = max(ans,i)\r\n
```