PEFT Integration

TRL supports PEFT (Parameter-Efficient Fine-Tuning) methods for memory-efficient model training. PEFT enables fine-tuning large language models by training only a small number of additional parameters while keeping the base model frozen, significantly reducing computational costs and memory requirements.

This guide covers how to use PEFT with different TRL trainers, including LoRA, QLoRA, and prompt tuning techniques.

For a complete working example, see the SFT with LoRA/QLoRA notebook.

Installation

To use PEFT with TRL, install the required dependencies:

```
pip install trl[peft]
```

For QLoRA support (4-bit and 8-bit quantization), also install:

```
pip install bitsandbytes
```

Quick Start

All TRL trainers support PEFT through the `peft_config` argument. The simplest way to enable PEFT is by using the command-line interface with the `--use_peft` flag:

```
python trl/scripts/sft.py \
    --model_name_or_path Qwen/Qwen2-0.5B \
    --dataset_name trl-lib/Capybara \
    --use_peft \
    --lora_r 32 \
    --lora_alpha 16 \
    --output_dir Qwen2-0.5B-SFT-LoRA
```

Alternatively, you can pass a PEFT config directly in your Python code:

```python
from peft import LoraConfig
from trl import SFTTrainer

# Configure LoRA
peft_config = LoraConfig(
    r=32,
    lora_alpha=16,
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM",
)

# Configure training - note the higher learning rate for LoRA (10x base rate)
training_args = SFTConfig(
    learning_rate=2.0e-4,  # 10x the base rate (2.0e-5) for LoRA
    ...
)

# Create trainer with PEFT
trainer = SFTTrainer(
    model=model,
    train_dataset=dataset,
    peft_config=peft_config,
)
```

Three Ways to Configure PEFT

TRL provides three different methods to configure PEFT, each suited for different use cases:

1. Using CLI Flags (Simplest)

The easiest way to enable PEFT is to use the `--use_peft` flag with the command-line interface. This method is ideal for quick experiments and standard configurations:

```bash
python trl/scripts/sft.py \
    --model_name_or_path Qwen/Qwen2-0.5B \
    --dataset_name trl-lib/Capybara \
    --use_peft \
    --lora_r 32 \
    --lora_alpha 16 \
    --lora_dropout 0.05 \
    --output_dir Qwen2-0.5B-SFT-LoRA
```

Pros: Quick setup, no code required

Cons: Limited to LoRA, fewer customization options

2. Passing peft_config to Trainer (Recommended)

For more control, pass a PEFT configuration directly to the trainer. This is the recommended approach for most use cases:

```python
from peft import LoraConfig
from trl import SFTConfig, SFTTrainer

peft_config = LoraConfig(
    r=32,
    lora_alpha=16,
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM",
    target_modules=["q_proj", "v_proj", "k_proj", "o_proj"],
)

trainer = SFTTrainer(
    model=model,
    args=training_args,
    train_dataset=dataset,
    peft_config=peft_config,  # Pass config here
)
```

Pros: Full control, supports all PEFT methods (LoRA, Prompt Tuning, etc.)

Cons: Requires Python code

3. Applying PEFT to Model Directly (Advanced)

For maximum flexibility, you can apply PEFT to your model before passing it to the trainer:

```python
from peft import LoraConfig, get_peft_model
from transformers import AutoModelForCausalLM
from trl import SFTConfig, SFTTrainer

# Load base model
model = AutoModelForCausalLM.from_pretrained("Qwen/Qwen2-0.5B")

# Apply PEFT configuration
peft_config = LoraConfig(
    r=32,
    lora_alpha=16,
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM",
)
model = get_peft_model(model, peft_config)

# Pass PEFT-wrapped model to trainer
trainer = SFTTrainer(
    model=model,  # Already has PEFT applied
    args=training_args,
    train_dataset=dataset,
    # Note: no peft_config needed here
)
```

Pros: Maximum control, useful for custom model architectures or complex setups

Cons: More verbose, requires understanding of PEFT internals

Learning Rate Considerations

When using LoRA or other PEFT methods, you typically need to use a **higher learning rate** (approximately 10x) compared to full fine-tuning. This is because PEFT methods train only a small fraction of parameters, requiring a larger learning rate to achieve similar parameter updates.

Recommended learning rates:

Trainer	Full Fine-Tuning	With LoRA (10x)
SFT	`2.0e-5`	`2.0e-4`
DPO	`5.0e-7`	`5.0e-6`
GRPO	`1.0e-6`	`1.0e-5`
Prompt Tuning	N/A	`1.0e-2` to `3.0e-2`

> *Why 10x? LoRA adapters have significantly fewer trainable parameters than the full model. A higher learning rate compensates for this reduced parameter count, ensuring effective training. For detailed explanation, see this blog post.*

For additional best practices on using LoRA effectively, refer to the LoRA Without Regret documentation.

PEFT with Different Trainers

TRL's trainers support PEFT configurations for various training paradigms. Below are detailed examples for each major trainer.

Supervised Fine-Tuning (SFT)

The `SFTTrainer` is used for supervised fine-tuning on instruction datasets.

With LoRA

```
python trl/scripts/sft.py \
    --model_name_or_path Qwen/Qwen2-0.5B \
    --dataset_name trl-lib/Capybara \
    --learning_rate 2.0e-4 \
    --num_train_epochs 1 \
    --per_device_train_batch_size 2 \
    --gradient_accumulation_steps 8 \
    --use_peft \
    --lora_r 32 \
    --lora_alpha 16 \
    --output_dir Qwen2-0.5B-SFT-LoRA
```

Python Example

```python
from peft import LoraConfig
from trl import SFTConfig, SFTTrainer

# Configure LoRA
peft_config = LoraConfig(
    r=32,
    lora_alpha=16,
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM",
    target_modules=["q_proj", "v_proj"],  # Optional: specify target modules
)

# Configure training with higher learning rate for LoRA
training_args = SFTConfig(
    learning_rate=2.0e-4,  # 10x the base rate for LoRA
    ...
)

# Create trainer with PEFT config
trainer = SFTTrainer(
    model=model,
    args=training_args,
    train_dataset=dataset,
    peft_config=peft_config,  # Pass PEFT config here
)

trainer.train()
```

Proximal Policy Optimization (PPO)

Multi-Adapter RL Training

You can use a single base model with multiple PEFT adapters for the entire PPO algorithm - including retrieving reference logits, computing active logits, and calculating rewards. This approach is useful for memory-efficient RL training.

> This feature is experimental and convergence has not been extensively tested. We encourage the community to share feedback and report any issues.

Requirements

Install PEFT and optionally bitsandbytes for 8-bit models:

```
pip install peft bitsandbytes
```

Training Workflow

The multi-adapter approach requires three stages:

1. **Supervised Fine-Tuning (SFT)**: Train a base model on your target domain (e.g., IMDB dataset) using `SFTTrainer`
2. **Reward Model Training**: Train a reward model adapter using PEFT and `RewardTrainer` (see reward modeling example)
3. **PPO Training**: Fine-tune new adapters using PPO with the reward adapter

> Use the same base model (architecture and weights) for stages 2 & 3.

Basic Usage

After training your reward adapter and pushing it to the Hub:

```
from peft import LoraConfig
from trl.experimental.ppo import PPOTrainer, AutoModelForCausalLMWithValueHead

model_name = "huggyllama/llama-7b"
rm_adapter_id = "trl-lib/llama-7b-hh-rm-adapter"

# Configure PPO adapter
lora_config = LoraConfig(
    r=16,
    lora_alpha=32,
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM",
)

# Load model with reward adapter
model = AutoModelForCausalLMWithValueHead.from_pretrained(
    model_name,
    peft_config=lora_config,
    reward_adapter=rm_adapter_id,
)

trainer = PPOTrainer(model=model, ...)
```

In your training loop, compute rewards using:

```
rewards = trainer.model.compute_reward_score(**inputs)
```

Advanced Features

Quantized Base Models

For memory-efficient training, load the base model in 8-bit or 4-bit while keeping adapters in float32:

```
from transformers import BitsAndBytesConfig

model = AutoModelForCausalLMWithValueHead.from_pretrained(
    model_name,
    peft_config=lora_config,
    reward_adapter=rm_adapter_id,
    quantization_config=BitsAndBytesConfig(load_in_8bit=True),
)
```

QLoRA: Quantized Low-Rank Adaptation

QLoRA combines 4-bit quantization with LoRA to enable fine-tuning of very large models on consumer hardware. This technique can reduce memory requirements by up to 4x compared to standard LoRA.

How QLoRA Works

1. **4-bit Quantization**: The base model is loaded in 4-bit precision using `bitsandbytes`
2. **Frozen Weights**: The quantized model weights remain frozen during training
3. **LoRA Adapters**: Only the LoRA adapter parameters are trained in higher precision
4. **Memory Efficiency**: Enables fine-tuning of models like Llama-70B on a single consumer GPU

Using QLoRA with TRL

Simply combine `load_in_4bit=True` with PEFT configuration:

Command Line

```
python trl/scripts/sft.py \
    --model_name_or_path meta-llama/Llama-2-7b-hf \
    --dataset_name trl-lib/Capybara \
    --load_in_4bit \
    --use_peft \
    --lora_r 32 \
    --lora_alpha 16 \
    --per_device_train_batch_size 1 \
    --gradient_accumulation_steps 16 \
    --output_dir Llama-2-7b-QLoRA
```

Python Example

```
import torch

from peft import LoraConfig
from transformers import AutoModelForCausalLM, BitsAndBytesConfig
from trl import SFTConfig, SFTTrainer

# Configure 4-bit quantization
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.bfloat16,
    bnb_4bit_use_double_quant=True,
)

# Load model with quantization
model = AutoModelForCausalLM.from_pretrained(
    "meta-llama/Llama-2-7b-hf",
    quantization_config=bnb_config,
    device_map="auto",
)

# Configure LoRA
peft_config = LoraConfig(
    r=32,
    lora_alpha=16,
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM",
)

# Configure training with higher learning rate for LoRA
training_args = SFTConfig(
    learning_rate=2.0e-4,  # 10x the base rate for QLoRA
    ...
)

# Create trainer with PEFT config
trainer = SFTTrainer(
    model=model,
    args=training_args,
    train_dataset=dataset,
    peft_config=peft_config,
)

trainer.train()
```

QLoRA Configuration Options

The `BitsAndBytesConfig` provides several options to optimize memory and performance:

```
import torch

from transformers import BitsAndBytesConfig

bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",  # or "fp4"
    bnb_4bit_compute_dtype=torch.bfloat16,  # Compute dtype for 4-bit base models
    bnb_4bit_use_double_quant=True,  # Nested quantization for additional memory
)
```

Configuration Parameters:

- `bnb_4bit_quant_type`: Quantization data type (`"nf4"` or `"fp4"`). NF4 is recommended.
- `bnb_4bit_compute_dtype`: The dtype used for computation. Use `bfloat16` for better training stability.
- `bnb_4bit_use_double_quant`: Enable nested quantization to save additional ~0.4 bits per parameter.

8-bit Quantization

For slightly higher precision with reduced memory savings, you can use 8-bit quantization:

```
from transformers import BitsAndBytesConfig, AutoModelForCausalLM

bnb_config = BitsAndBytesConfig(load_in_8bit=True)

model = AutoModelForCausalLM.from_pretrained(
    "meta-llama/Llama-2-7b-hf",
    quantization_config=bnb_config,
    device_map="auto",
)
```

Or via command line:

```
python trl/scripts/sft.py \
    --model_name_or_path meta-llama/Llama-2-7b-hf \
    --load_in_8bit \
    --use_peft \
    --lora_r 32 \
    --lora_alpha 16
```

Prompt Tuning

Prompt tuning is another PEFT technique that learns soft prompts (continuous embeddings) prepended to the input, while keeping the entire model frozen. This is particularly effective for large models.

How Prompt Tuning Works

1. **Virtual Tokens**: Adds learnable continuous embeddings (virtual tokens) to the input
2. **Frozen Model**: The entire base model remains frozen
3. **Task-Specific Prompts**: Each task learns its own prompt embeddings
4. **Extreme Efficiency**: Only the prompt embeddings are trained (typically 8-20 tokens)

Using Prompt Tuning with TRL

```
from peft import PromptTuningConfig, PromptTuningInit, TaskType
from trl import SFTConfig, SFTTrainer

# Configure Prompt Tuning
peft_config = PromptTuningConfig(
    task_type=TaskType.CAUSAL_LM,
    prompt_tuning_init=PromptTuningInit.TEXT,
    num_virtual_tokens=8,
    prompt_tuning_init_text="Classify if the tweet is a complaint or not:",
    tokenizer_name_or_path="Qwen/Qwen2-0.5B",
)

# Configure training with higher learning rate for Prompt Tuning
training_args = SFTConfig(
    learning_rate=2.0e-2,  # Prompt Tuning typically uses 1e-2 to 3e-2
    ...
)

# Create trainer with PEFT config
trainer = SFTTrainer(
    model=model,
    args=training_args,
    train_dataset=dataset,
    peft_config=peft_config,  # Pass PEFT config here
)

trainer.train()
```

Prompt Tuning Configuration

```
from peft import PromptTuningConfig, PromptTuningInit, TaskType

peft_config = PromptTuningConfig(
    task_type=TaskType.CAUSAL_LM,  # Task type
    prompt_tuning_init=PromptTuningInit.TEXT,  # Initialize from text
    num_virtual_tokens=8,  # Number of virtual tokens
    prompt_tuning_init_text="Your initialization text here",
    tokenizer_name_or_path="model_name",
)
```

Configuration Parameters:

- `task_type`: The task type (`TaskType.CAUSAL_LM` for language modeling)
- `prompt_tuning_init`: Initialization method (`TEXT`, `RANDOM`)
- `num_virtual_tokens`: Number of virtual tokens to prepend (typically 8-20)
- `prompt_tuning_init_text`: Text to initialize the virtual tokens (when using `TEXT` init)
- `tokenizer_name_or_path`: Tokenizer for initializing from text

Prompt Tuning vs LoRA

Feature	Prompt Tuning	LoRA
Parameters Trained	~0.001%	~0.1-1%
Memory Usage	Minimal	Low
Training Speed	Fastest	Fast
Model Modification	None	Adapter layers
Best For	Large models, many tasks	General fine-tuning
Learning Rate	Higher (1e-2 to 3e-2)	Standard (1e-4 to 3e-4)

Advanced PEFT Configurations

LoRA Configuration Parameters

```
from peft import LoraConfig

peft_config = LoraConfig(
    r=16,  # LoRA rank
    lora_alpha=32,  # LoRA scaling factor
    lora_dropout=0.05,  # Dropout probability
    bias="none",  # Bias training strategy
    task_type="CAUSAL_LM",  # Task type
    target_modules=["q_proj", "v_proj"],  # Modules to apply LoRA
    modules_to_save=None,  # Additional modules to train
)
```

Key Parameters:

- `r`: LoRA rank (typical values: 8, 16, 32, 64). Higher rank = more parameters but potentially better performance.
- `lora_alpha`: Scaling factor (typically 2x the rank). Controls the magnitude of LoRA updates.
- `lora_dropout`: Dropout probability for LoRA layers (typical: 0.05-0.1).
- `target_modules`: Which modules to apply LoRA to. Common choices:
    - `["q_proj", "v_proj"]`: Attention query and value (memory efficient)

    - `["q_proj", "k_proj", "v_proj", "o_proj"]`: All attention projections
    - `["q_proj", "k_proj", "v_proj", "o_proj", "gate_proj", "up_proj", "down_proj"]`: All linear layers
- `modules_to_save`: Additional modules to fully train (e.g., `["embed_tokens", "lm_head"]`)

Target Module Selection

You can specify which modules to apply LoRA to. Common patterns:

```
# Minimal (most memory efficient)
target_modules=["q_proj", "v_proj"]

# Attention only
target_modules=["q_proj", "k_proj", "v_proj", "o_proj"]

# All linear layers (best performance, more memory)
target_modules=["q_proj", "k_proj", "v_proj", "o_proj", "gate_proj", "up_proj", "
```

Using Command-Line Arguments

TRL scripts accept PEFT parameters via command line:

```
python trl/scripts/sft.py \
    --model_name_or_path Qwen/Qwen2-0.5B \
    --dataset_name trl-lib/Capybara \
    --use_peft \
    --lora_r 32 \
    --lora_alpha 16 \
    --lora_dropout 0.05 \
    --lora_target_modules q_proj v_proj \
    --output_dir output
```

Available flags:

- `--use_peft`: Enable PEFT
- `--lora_r`: LoRA rank (default: 16)
- `--lora_alpha`: LoRA alpha (default: 32)

- `--lora_dropout`: LoRA dropout (default: 0.05)
- `--lora_target_modules`: Target modules (space-separated)
- `--lora_modules_to_save`: Additional modules to train
- `--use_rslora`: Enable Rank-Stabilized LoRA
- `--use_dora`: Enable Weight-Decomposed LoRA (DoRA)
- `--load_in_4bit`: Enable 4-bit quantization (QLoRA)
- `--load_in_8bit`: Enable 8-bit quantization

Saving and Loading PEFT Models

Saving

After training, save your PEFT adapters:

```python
# Save the adapters
trainer.save_model("path/to/adapters")

# Or manually
model.save_pretrained("path/to/adapters")
```

This saves only the adapter weights (~few MB) rather than the full model (~several GB).

Loading

Load a PEFT model for inference:

```python
from transformers import AutoModelForCausalLM
from peft import PeftModel

# Load base model
base_model = AutoModelForCausalLM.from_pretrained("Qwen/Qwen2-0.5B")

# Load PEFT adapters
model = PeftModel.from_pretrained(base_model, "path/to/adapters")
```

```python
# Optionally merge adapters into base model for faster inference
model = model.merge_and_unload()
```

Pushing to Hub

You can easily share your PEFT adapters on the Hugging Face Hub:

```python
# Push adapters to Hub
model.push_to_hub("username/model-name-lora")

# Load from Hub
from peft import PeftModel
model = PeftModel.from_pretrained(base_model, "username/model-name-lora")
```

Multi-GPU Training

PEFT works seamlessly with TRL's multi-GPU support through `accelerate`:

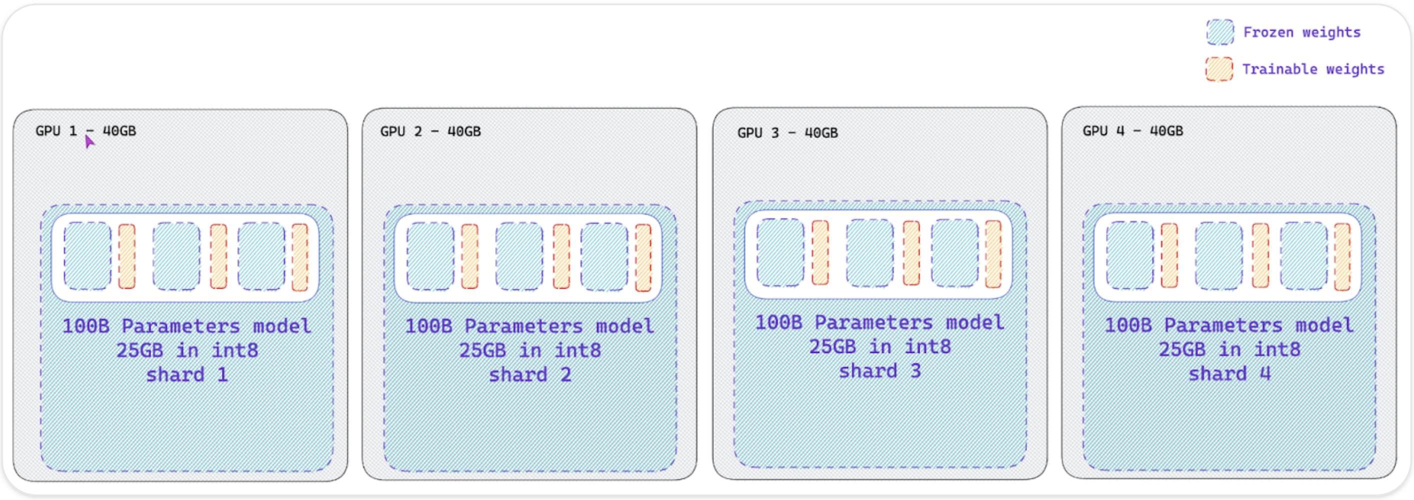
```bash
# Configure accelerate
accelerate config

# Launch training
accelerate launch trl/scripts/sft.py \
    --model_name_or_path Qwen/Qwen2-0.5B \
    --dataset_name trl-lib/Capybara \
    --use_peft \
    --lora_r 32 \
    --lora_alpha 16
```

For QLoRA with multiple GPUs, the base model is automatically sharded:

```bash
accelerate launch trl/scripts/sft.py \
    --model_name_or_path meta-llama/Llama-2-70b-hf \
    --load_in_4bit \
    --use_peft \
    --lora_r 32
```

Naive Pipeline Parallelism (NPP) for Large Models

For very large models (>60B parameters), TRL supports Naive Pipeline Parallelism (NPP), which distributes the model and adapters across multiple GPUs. The activations and gradients are communicated across GPUs, supporting both `int8` and other data types.



How to Use NPP

Load your model with a custom `device_map` to split it across multiple devices:

```python
from transformers import AutoModelForCausalLM
from peft import LoraConfig

# Create custom device map (see accelerate documentation)
device_map = {
    "model.embed_tokens": 0,
    "model.layers.0": 0,
    # ... distribute layers across GPUs
    "lm_head": 0,  # Must be on GPU 0
}

model = AutoModelForCausalLM.from_pretrained(
    "meta-llama/Llama-2-70b-hf",
    device_map=device_map,
    peft_config=lora_config,
)
```

> - Keep the `lm_head` module on the first GPU (device 0) to avoid errors
> - See this tutorial on device maps for proper configuration
> - Run training scripts directly (not with `accelerate launch`): `python script.py`
> - Data Parallelism is not yet supported with NPP

Resources

TRL Examples and Notebooks

- SFT with LoRA/QLoRA Notebook - Complete working example showing both LoRA and QLoRA implementations
- TRL Examples Directory - Collection of training scripts demonstrating PEFT with different trainers
- TRL Cookbook Recipes - Step-by-step guides for common PEFT training scenarios

Documentation

- PEFT Documentation - Official PEFT library documentation
- TRL Documentation - Complete TRL documentation with trainer guides
- LoRA Without Regret - Best practices for using LoRA effectively

Research Papers

- LoRA Paper - Original LoRA methodology and results
- QLoRA Paper - Efficient finetuning with 4-bit quantization
- Prompt Tuning Paper - The Power of Scale for Parameter-Efficient Prompt Tuning