3. PPO Training with Reward Model

RL_ppo_train_with_reward.py implements **Proximal Policy Optimization (PPO)** for fine-tuning a causal language model using a pre-trained reward model. The goal is to optimize the language model to produce outputs that maximize predicted rewards.

Key Parts of the Implementation

1. Quantized Policy Model Setup

The script uses **4-bit quantization with bf16 compute** for memory-efficient training:

```
bnb_cfg = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_use_double_quant=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.bfloat16
)
```

- `nf4` quantization reduces memory usage.
- `bf16` ensures numerical stability for logits and gradients, preventing NaN values during PPO updates.

---

2. Tokenizer

The tokenizer is loaded from the base model:

```
policy_model_name = "mistralai/Mistral-7B-Instruct-v0.1"
tokenizer = AutoTokenizer.from_pretrained(policy_model_name)
if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token
```

- Ensures padding token exists, necessary for batching.

---

3. Policy and Reference Models with Value Head

The **policy model** and a **reference model** are both causal LMs with a value head:

```
policy_model = AutoModelForCausalLMWithValueHead.from_pretrained(
    policy_model_name,
    quantization_config=bnb_cfg,
    device_map={"": 0},
    torch_dtype=torch.bfloat16
)
ref_model = AutoModelForCausalLMWithValueHead.from_pretrained(
    policy_model_name,
    quantization_config=bnb_cfg,
    device_map={"": 0}
)
```

- The **value head** predicts the expected reward for PPO.
- Reference model is used for **KL penalty** during PPO updates.

---

4. Reward Model

A pre-trained reward model scores generated outputs:

```
reward_model_name = "output/reward_model_pairwise"
reward_tokenizer = AutoTokenizer.from_pretrained(reward_model_name)
reward_model =
AutoModelForSequenceClassification.from_pretrained(reward_model_name)
reward_model.to(device)
```

- Reward model can output 1 or more labels.

- Supports scalar or probability-based rewards.

---

5. PPO Trainer Setup

The PPO trainer handles optimization of the policy:

```
ppo_config = PPOConfig(
    batch_size=2,
    mini_batch_size=1,
    learning_rate=1e-5,
    log_with=None
)
ppo_trainer = PPOTrainer(
    config=ppo_config,
    model=policy_model,
    ref_model=ref_model,
    tokenizer=tokenizer
)
```

- `batch_size` defines how many prompts are processed per PPO step.
- `mini_batch_size` is used internally by PPO for gradient computation.

---

6. Loading Prompts

Prompts are read from the JSONL file:

```
prompts = []
with open("output/pairwise_prefs_part_1.jsonl", "r") as f:
    for line in f:
        obj = json.loads(line)
        prompts.append(obj["prompt"])
```

- These prompts are inputs for the policy model to generate responses.

---

7. PPO Training Loop

The main training loop generates outputs, computes rewards, and updates the policy:

```
loader = DataLoader(prompts, batch_size=ppo_config.batch_size,
shuffle=True)

for step, prompt_batch in enumerate(loader):
    # Tokenize prompts
    batch = reward_tokenizer(prompt_batch, return_tensors="pt",
padding=True, truncation=True).to(device)

    # Generate responses from policy
    response_ids = policy_model.generate(
        input_ids=batch["input_ids"],
        attention_mask=batch.get("attention_mask"),
        max_new_tokens=50,
        do_sample=True,
        pad_token_id=tokenizer.pad_token_id
    )

    responses = [tokenizer.decode(r, skip_special_tokens=True) for r
in response_ids]

    # Compute rewards
    with torch.no_grad():
        reward_inputs = reward_tokenizer(responses,
return_tensors="pt", padding=True, truncation=True).to(device)
        reward_logits = reward_model(**reward_inputs).logits
        if reward_model.config.num_labels == 1:
            rewards = reward_logits.squeeze(-1)
        else:
            rewards = torch.softmax(reward_logits, dim=-1)[:, 1]

    # PPO step
```

```
    stats = ppo_trainer.step(
        queries=[batch["input_ids"][i] for i in
range(batch["input_ids"].size(0))],
        responses_list=[response_ids[i] for i in
range(response_ids.size(0))],
        rewards_list=[rewards[i] for i in range(rewards.size(0))]
    )

    print(f"[STEP {step}] Rewards: {rewards.tolist()} | PPO Stats:
{stats}")
```

- **Step 1:** Tokenize batch of prompts.
- **Step 2:** Policy generates new sequences.
- **Step 3:** Reward model scores responses.
- **Step 4:** PPO updates policy based on rewards and KL penalty against reference model.

---

8. Notes on Numerical Stability

- `torch_dtype=torch.bfloat16` ensures stable gradients.
- Prevents `NaN` issues during PPO training with 4-bit quantized models.

---

9. Summary

This implementation provides a **memory-efficient, stable PPO training pipeline**:

- Uses **4-bit quantized policy model** with bf16 computation.
- Integrates a **pre-trained reward model** for feedback.
- Uses **PPOTrainer** from `trl` to fine-tune policy safely.
- Handles prompt batching, reward computation, and PPO updates end-to-end.