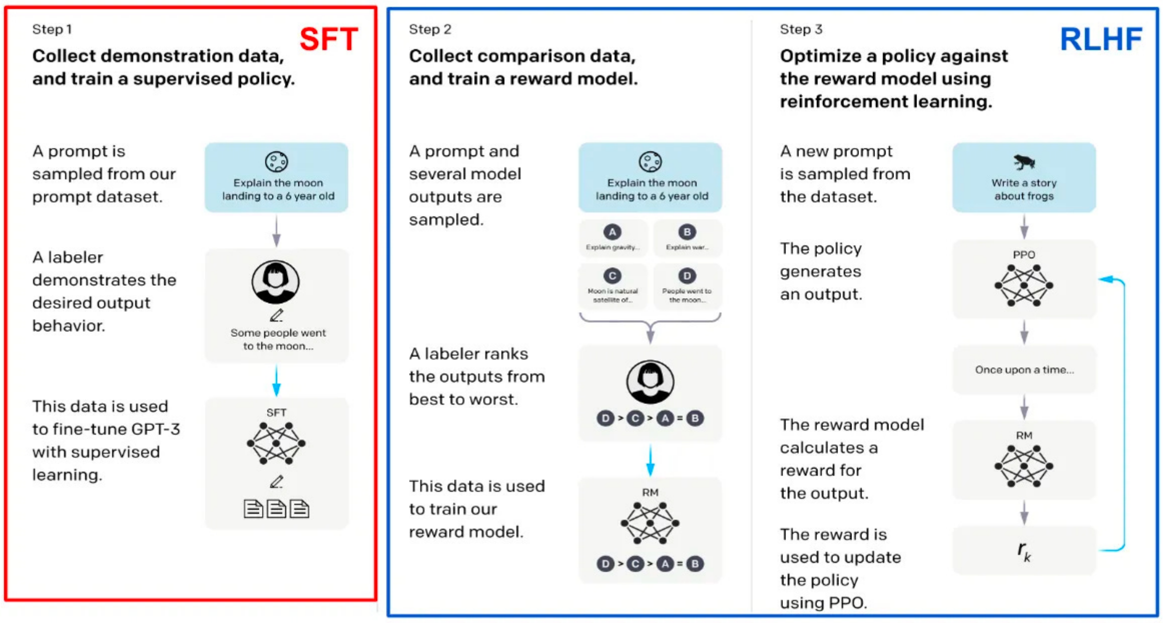
Step 1
SFT
Collect demonstration data, and train a supervised policy.
A prompt is sampled from our prompt dataset.
Explain the moon landing to a 6 year old
A labeler demonstrates the desired output behavior.
Some people went to the moon...
This data is used to fine-tune GPT-3 with supervised learning.
SFT
Step 2
Collect comparison data, and train a reward model.
A prompt and several model outputs are sampled.
Explain the moon landing to a 6 year old
Explain gravity...
Explain war...
Moon is natural satellite of...
People went to the moon...
A labeler ranks the outputs from best to worst.
D > C > A = B
This data is used to train our reward model.
RM
D > C > A = B
Step 3
RLHF
Optimize a policy against the reward model using reinforcement learning.
A new prompt is sampled from the dataset.
Write a story about frogs
The policy generates an output.
PPO
Once upon a time...
The reward model calculates a reward for the output.
RM
The reward is used to update the policy using PPO.
r_k

Reward Modeling (RM), the critical step for training a model to predict human preferences, which is later used to fine-tune the SFT model via **RLHF (PPO)**. Below is a detailed, step-by-step guide.

Step 1: Understand the Goal of Reward Modeling

Purpose:

- Train a **reward model (RM)** to score responses based on **human preferences** (e.g., helpfulness, safety, accuracy).
- The RM outputs a scalar value where **higher scores = better responses**.

Key Difference from SFT:

- **SFT** learns from *demonstration data* (single ideal response per prompt).
- **RM** learns from *comparison data* (ranked responses to the same prompt).

Step 2: Collect Comparison Data

Data Format:

Each training example consists of:

- A **prompt**.
- **Two or more responses** (ranked by humans: `chosen` > `rejected`).

Example Dataset (JSON):

```json
[
  {
    "prompt": "Explain quantum computing.",
    "chosen": "Quantum computing uses qubits to perform calculations exponentially faster than classical computers for certain problems...",
    "rejected": "Quantum computing is like magic but with science stuff."
  },
  {
    "prompt": "Write a Python function to reverse a string.",
    "chosen": "def reverse_string(s):\n    return s[::-1]",
    "rejected": "Just use print('reverse me')[::-1]"
  }
]
```

How to Collect This Data?

1. **Option A: Human Annotators**
   - Give annotators a prompt and 2-4 model responses (from SFT model).
   - Ask them to rank responses by **quality, accuracy, safety**.
2. **Option B: Public Datasets**
   - **Anthropic HH-RLHF**: 160k human-ranked dialogues.
   - **OpenAI Summarization Comparisons**: 93k ranked summaries.
   - **Custom Data**: Use GPT-4 to generate candidate responses, then filter/rank them.

Step 3: Preprocess Data

Tokenization:

- Use the same tokenizer as the SFT model (e.g., `cl100k_base` for GPT-3.5-style models).
- Format each `(prompt, response)` pair into token IDs:

```
tokenizer(prompt + response, return_tensors="pt", padding="max_length", truncation=True,
max_length=1024)
```

Dataset Structure:

- Each batch contains:
  - `input_ids`: Tokenized `(prompt + chosen_response)`.
  - `attention_mask`: Mask for padding tokens.
  - `labels`: Binary labels (`1` for chosen, `0` for rejected).

Step 4: Train the Reward Model

Model Architecture:

- Start with the **SFT model** (or a smaller version for efficiency).
- Replace the **language modeling head** with a **regression head** (outputs a scalar score).

Loss Function:

Use **pairwise ranking loss** (Bradley-Terry model):

$$\mathcal{L}(\theta) = -\mathbb{E}\left[\log\left(\sigma\left(R_\theta(x, y_c) - R_\theta(x, y_r)\right)\right)\right]$$

- R_θ = reward model with parameters θ.
- y_c = chosen response, y_r = rejected response.
- σ = sigmoid function.

Training Code (PyTorch):

```python
from transformers import AutoModelForSequenceClassification, AutoTokenizer, Trainer
import torch

# Load SFT model and tokenizer
model = AutoModelForSequenceClassification.from_pretrained("your_sft_model", num_labels=1)
tokenizer = AutoTokenizer.from_pretrained("your_sft_model")

# Dataset class
class RewardDataset(torch.utils.data.Dataset):
    def __init__(self, comparisons):
        self.comparisons = comparisons

    def __getitem__(self, idx):
        item = self.comparisons[idx]
        chosen = tokenizer(item["prompt"] + item["chosen"], truncation=True,
max_length=1024)
        rejected = tokenizer(item["prompt"] + item["rejected"], truncation=True,
max_length=1024)
        return {
            "input_ids_chosen": chosen["input_ids"],
            "attention_mask_chosen": chosen["attention_mask"],
            "input_ids_rejected": rejected["input_ids"],
            "attention_mask_rejected": rejected["attention_mask"]
        }

# Loss function
def compute_loss(model, batch):
    chosen_scores = model(
        input_ids=batch["input_ids_chosen"],
        attention_mask=batch["attention_mask_chosen"]
    ).logits.squeeze()
    rejected_scores = model(
        input_ids=batch["input_ids_rejected"],
        attention_mask=batch["attention_mask_rejected"]
    ).logits.squeeze()
    loss = -torch.log(torch.sigmoid(chosen_scores - rejected_scores)).mean()
    return loss

# Trainer
trainer = Trainer(
    model=model,
    args=TrainingArguments(
        output_dir="./reward_model",
        per_device_train_batch_size=8,
        learning_rate=1e-5,
        num_train_epochs=1
    ),
    train_dataset=RewardDataset(comparisons),
    compute_loss=compute_loss
)
trainer.train()
```

Key Hyperparameters:

Hyperparameter	Recommended Value	Notes
Batch Size	8-32	Depends on GPU memory.
Learning Rate	1e-5 to 5e-5	Lower than SFT (stable training).
Epochs	1-2	Overfitting risks noisy rankings.
Model Size	6B params or less	Smaller than SFT for efficiency.

Step 5: Evaluate the Reward Model

Quantitative Metrics:

1. **Accuracy**: % of time RM scores `chosen` > `rejected` on held-out data.
2. **Kendall's Tau**: Rank correlation between RM and human rankings.

Qualitative Checks:

- Manually inspect if:
  - High-quality responses score higher.
  - Harmful/nonsensical responses score low.

Example Evaluation Code:

```python
def evaluate_rm(model, eval_dataset):
    correct = 0
    for item in eval_dataset:
        chosen_score = model(item["input_ids_chosen"], item["attention_mask_chosen"]).logits.item()
        rejected_score = model(item["input_ids_rejected"], item["attention_mask_rejected"]).logits.item()
        if chosen_score > rejected_score:
            correct += 1
    accuracy = correct / len(eval_dataset)
    print(f"Accuracy: {accuracy:.2f}")
```

Step 6: Save & Prepare for RLHF

Export the Reward Model:

```python
model.save_pretrained("./final_reward_model")
tokenizer.save_pretrained("./final_reward_model")
```

Next Step: RLHF with PPO

- Use the RM to guide **reinforcement learning** (e.g., PPO) on the SFT model.

Common Pitfalls & Fixes

Issue	Solution
Overfitting	Use smaller RM, fewer epochs.
Noisy Rankings	Filter low-agreement comparisons.
Short Responses Win	Add length normalization to scores.

Full Pipeline Summary

Step	Key Actions
1. Collect Data	Ranked `(prompt, chosen, rejected)` pairs (human or GPT-4 + filtering).
2. Preprocess	Tokenize pairs, pad to fixed length.
3. Train RM	Fit model with pairwise ranking loss (LR=1e-5, 1-2 epochs).
4. Evaluate	Check accuracy/Kendall's Tau on held-out data.
5. Deploy	Save RM for use in RLHF (PPO).

Example: Training on Anthropic HH-RLHF

```
python train_reward_model.py \
  --model_name "EleutherAI/gpt-neo-2.7B" \
  --dataset "anthropic/hh-rlhf" \
  --epochs 1 \
  --lr 1e-5 \
  --batch_size 16
```

Output: A reward model that scores responses like:

- Good: *"Photosynthesis converts sunlight to energy."* → **+5.2**
- Bad: *"Plants eat sunlight."* → **-1.3**

Next Steps

1. **Proximal Policy Optimization (PPO)**: Use the RM to optimize the SFT model.
2. **Iterative Refinement**: Continuously collect new comparisons to improve RM.

RLHF and PPO

Proximal Policy Optimization (PPO) for **Reinforcement Learning from Human Feedback (RLHF)**, the final step to align your model with human preferences. This is how ChatGPT 3/3.5 was fine-tuned after SFT and Reward Modeling.

PPO for RLHF: Step-by-Step Guide

1. Understand the Goal

What PPO Does:

- Uses the **Reward Model (RM)** to train the **SFT model** via reinforcement learning.
- Optimizes responses to maximize **human-preferred outputs** while avoiding drastic changes from the original SFT model (via KL divergence penalty).

Key Components:

Component	Role
Policy Model (πϕ)	The SFT model being optimized (e.g., GPT-3 after SFT).
Reward Model (Rθ)	Predicts scalar rewards for responses (from Step 2: Reward Modeling).
Reference Model	Frozen copy of SFT model to compute KL divergence.
PPO Algorithm	Updates the policy to maximize rewards while staying near the reference.

2. Data Flow in PPO RLHF

Training Loop:

1. **Sample a batch of prompts** (e.g., from your SFT dataset).
2. **Generate responses** using the current policy model (πφ).
3. **Score responses** with the reward model (`Rθ`).
4. **Compute KL divergence** between πφ and the reference model.
5. **Update policy** using PPO to maximize:

$$\text{Total Reward} = R_\theta(x, y) - \beta \cdot \text{KL}(\pi_\phi(y|x) \| \pi_{\text{ref}}(y|x))$$

   - β = KL penalty weight (e.g., 0.1).

3. Implement PPO RLHF

Key Libraries:

```
pip install transformers torch accelerate peft trl
```

- `trl`: HuggingFace's library for RLHF (PPO, DPO).
- `peft`: For efficient fine-tuning (LoRA, QLoRA).

Code Implementation:

```python
from transformers import AutoModelForCausalLM, AutoTokenizer
from trl import PPOTrainer, PPOConfig
import torch

# Load models
model = AutoModelForCausalLM.from_pretrained("your_sft_model")
ref_model = AutoModelForCausalLM.from_pretrained("your_sft_model")  # Frozen reference
tokenizer = AutoTokenizer.from_pretrained("your_sft_model")
reward_model = AutoModelForSequenceClassification.from_pretrained("your_reward_model")

# PPO Config
ppo_config = PPOConfig(
    batch_size=32,
    learning_rate=1e-5,
    kl_penalty="kl",  # Use KL divergence penalty
    kl_coef=0.1,      # β (KL penalty weight)
    cliprange=0.2,    # PPO clipping range
    steps=10000,      # Total training steps
)

# Initialize PPOTrainer
ppo_trainer = PPOTrainer(
    model=model,
    ref_model=ref_model,
    tokenizer=tokenizer,
    config=ppo_config,
)

# Training loop
for epoch in range(3):
    for batch in prompt_dataset:  # Batch of prompts
        # Generate responses
        inputs = tokenizer(batch["prompt"], return_tensors="pt", padding=True)
        outputs = model.generate(**inputs, max_length=1024)
        response = tokenizer.decode(outputs[0], skip_special_tokens=True)

        # Compute reward
        reward_inputs = tokenizer(batch["prompt"] + response, return_tensors="pt")
        reward = reward_model(**reward_inputs).logits[0].item()

        # Compute KL divergence (automatically handled by PPOTrainer)
        # Update policy
        ppo_trainer.step([inputs["input_ids"]], [outputs], [reward])
```

---

4. Hyperparameters & Optimization

Critical PPO Settings:

Hyperparameter	Recommended Value	Notes
Batch Size	16-64	Smaller for stability.
Learning Rate	1e-6 to 1e-5	Very low to avoid divergence.
KL Coefficient (β)	0.05-0.2	Balances reward vs. divergence.
Clip Range	0.1-0.3	Clips policy updates for stability.
GAE (λ)	0.9-1.0	For advantage estimation.

Advanced Tricks:

1. **Mixed-Precision Training** (`fp16=True`):
   - Speeds up training with minimal accuracy loss.
2. **LoRA/QLoRA**:
   - Fine-tune only low-rank adapters to save memory.

   ```
   from peft import LoraConfig
   lora_config = LoraConfig(r=8, lora_alpha=16, target_modules=["q_proj", "v_proj"])
   model.add_adapter(lora_config)
   ```
3. **Gradient Clipping**:
   - Prevents exploding gradients (`max_grad_norm=1.0`).

---

5. Evaluating PPO Performance

Quantitative Metrics:

1. **Mean Reward**:
   - Average reward per batch (should increase over time).
2. **KL Divergence**:
   - Should stay small (e.g., < 10 nats).

3. **Perplexity**:
    - Compare to the reference model (should not degrade drastically).

Qualitative Checks:

- Manually inspect if:
    - Responses are **more helpful** than SFT.
    - No **reward hacking** (e.g., gibberish with high scores).
    - Avoids **safety violations**.

Example Evaluation Code:

```python
def evaluate_ppo(model, eval_prompts):
    for prompt in eval_prompts:
        inputs = tokenizer(prompt, return_tensors="pt")
        output = model.generate(**inputs, max_length=512)
        print(f"Prompt: {prompt}\nResponse: {tokenizer.decode(output[0])}\n---")
```

---

6. Common Pitfalls & Fixes

Issue	Solution
Reward Hacking	Increase KL penalty (β), clip rewards.
Unstable Training	Lower learning rate, smaller batches.
Overoptimization	Early stopping based on KL divergence.
Catastrophic Forgetting	Regularize with reference model.

Full RLHF Pipeline Summary

Step	Key Actions
1. SFT Model	Fine-tune base model on high-quality `(prompt, response)` pairs.
2. Reward Model	Train on ranked `(prompt, chosen, rejected)` pairs.

Step	Key Actions
3. PPO RLHF	Optimize SFT model using RM + KL penalty (this guide).
4. Evaluation	Check reward/KL trends + human A/B tests.

Example: Training on Anthropic HH-RLHF

```
python train_ppo.py \
  --sft_model "your_sft_model" \
  --reward_model "your_reward_model" \
  --dataset "anthropic/hh-rlhf" \
  --kl_coef 0.1 \
  --batch_size 32 \
  --lr 1e-5
```

Output: A model that:

- **Outperforms SFT** in human preference tests.
- Avoids harmful outputs (e.g., refuses unsafe requests).

Next Steps

1. **Iterative RLHF**:
   - Collect new human feedback on PPO outputs to improve RM.
2. **Deployment**:
   - Quantize the model (GPTQ/GGML) for efficient inference.
3. **DPO (Optional)**:
   - Direct Preference Optimization (simpler alternative to PPO).

PPO

Proximal Policy Optimization (PPO) is a model-free, on-policy reinforcement learning algorithm used in the **Reinforcement Learning from Human Feedback (RLHF)** framework to fine-tune language models. PPO is widely used in RLHF due to its stability and efficiency.

Objectives of PPO in RLHF:

The main objective of PPO in RLHF is to optimize the policy of a language model such that it generates responses that are not only high-quality but also aligned with human preferences. This is achieved by using human feedback as a reward signal to guide the model's training process.

PPO Objective Function:

The PPO objective function is designed to update the policy in a way that improves performance while limiting the step size to avoid significant changes that could destabilize the training process.

The objective function is given by:

$$L^{CLIP}(\theta) = \mathbb{E}_t \left[\min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)} A(s,a),\ \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}, 1-\epsilon, 1+\epsilon \right) A(s,a) \right) \right]$$

Here are the components of the objective function and their roles:

1. **Policy Ratio**: $\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}$ represents the ratio of the probability of taking action a in state s under the new policy π_θ and the old policy $\pi_{\theta_{\text{old}}}$. This ratio indicates how much the policy has changed.
2. **Advantage Function**: $A(s,a)$ estimates how much better or worse taking action a in state s is compared to the average action in that state. It provides a measure of the relative quality of an action.
3. **Clipping**: The clip function limits the policy ratio to the range $[1-\epsilon, 1+\epsilon]$, where ϵ is a small hyperparameter (e.g., 0.1 or 0.2). This clipping ensures that the policy updates are not too large, preventing the new policy from diverging significantly from the old one.

Intuition Behind PPO:

The intuition behind PPO is to make conservative updates to the policy. By clipping the policy ratio, PPO prevents the new policy from deviating too much from the old one. This helps maintain stability in the training process and reduces the risk of performance collapse that can occur with more aggressive policy updates.

By optimizing the clipped surrogate objective, PPO encourages the policy to improve while keeping the updates within a trust region around the previous policy. This balance between improvement and stability makes PPO particularly effective in practice.

In the context of RLHF, PPO uses human feedback to shape the reward signal, guiding the language model to generate responses that better align with human preferences. The PPO algorithm helps the model learn from this feedback efficiently and stably, ultimately producing higher-quality and more aligned outputs.