LLM Fine-Tuning, Alignment & Preference Optimization

SFT, RLHF, DPO, and LoRA/QLoRA Methods

Table of Contents

---

1. Introduction & Motivation

1.1 Why Fine-Tuning is Needed

Large language models trained on massive web corpora acquire broad linguistic knowledge but lack task-specific expertise and domain alignment. A model trained on general internet text may perform poorly on specialized tasks (medical diagnosis, code generation, customer support) without adaptation. Fine-tuning adapts pre-trained models to specific domains, instruction-following styles, and performance requirements with dramatically reduced compute compared to pretraining.

The fundamental challenge: pretraining optimizes for next-token prediction accuracy, not for useful, safe, or preference-aligned behavior.

1.2 Why Alignment is Necessary

LLMs exhibit concerning failure modes:

- **Misalignment with human intent**: Models optimize for proxy metrics that diverge from actual user preferences
- **Safety concerns**: Generation of toxic, biased, or harmful content
- **Hallucinations**: Confident fabrication of facts
- **Controllability**: Difficulty steering model behavior without explicit instructions

Alignment aims to ensure LLM outputs reflect human values, preferences, and safety constraints. This requires moving beyond supervised learning on human-curated examples toward methods that directly optimize for human preference signals.

1.3 Quick Comparison: SFT vs RLHF vs DPO

Method	Approach	Data	Training Complexity	Compute Cost
SFT	Supervised learning on curated examples	Input-output pairs	Low	Low
RLHF	RL with learned reward model	Preference pairs	High (3-phase)	High (RM + PPO)
DPO	Direct preference optimization via implicit reward	Preference pairs	Medium	Medium (single phase)

Key insight: Each method trades data efficiency, computational cost, and alignment quality differently. The choice depends on data availability, compute budget, and performance goals.

---

2. Background: Foundation Models & Alignment Challenges

2.1 How LLMs Are Trained

Pretraining phase (months, billions of tokens):

- Objective: Minimize next-token prediction loss on diverse internet corpora
- Model: Decoder-only transformer (GPT-style architecture)
- Result: Strong language understanding, but no instruction-following or preference alignment

Pretraining Loss: L = -E[log P(token_t | token_<t)]

This loss minimizes prediction error but does not directly optimize for helpful, harmless, honest behavior.

Why pretraining alone fails:

- Optimizes likelihood of next token, not quality of full response
- No explicit optimization for user preferences
- May amplify biases and unsafe patterns in training data
- No mechanism to balance multiple objectives (accuracy, safety, helpfulness)

2.2 Misalignment Sources

1. **Specification Gaming**: Model exploits proxy metrics (high likelihood) while violating true objectives (usefulness)
2. **Objective Mismatch**: Pretraining loss ≠ human preference
3. **Reward Hacking**: In RLHF, policy learns to game imperfect reward models
4. **Distribution Shift**: Performance degrades on out-of-distribution queries
5. **Value Drift**: Fine-tuning without regularization erodes capabilities learned during pretraining

2.3 Safety, Preference Modeling, Controllability

Safety requires explicit constraints and robustness testing. Methods include:

- Constitutional AI (self-critique + RL)
- Safety-focused RLHF phases
- Formal verification (limited applicability)

Preference modeling captures nuanced human values:

- Not binary (safe/unsafe) but continuous (more helpful, more creative, safer)
- Preferences vary across users and contexts
- Requires large, representative datasets

Controllability demands methods to steer behavior:

- Few-shot in-context learning
- System prompts and role-playing
- Fine-tuning on instruction-following examples

---

3. Supervised Fine-Tuning (SFT)

3.1 Concept and Workflow

SFT fine-tunes a pretrained model on a dataset of high-quality (input, output) pairs using standard supervised learning:

SFT Loss: $L_{SFT} = -E[(input, output) \sim D] [\log P_\theta(output \mid input)]$

Data format (example from OpenAI Instruct datasets):

```
{
  "instruction": "Explain quantum entanglement in simple terms.",
  "input": "",
  "output": "Quantum entanglement occurs when two particles..."
}
```

Or conversational format (more common):

```
{
  "messages": [
    {"role": "user", "content": "Explain quantum entanglement..."},
    {"role": "assistant", "content": "Quantum entanglement occurs..."}
```

]

}

3.2 Workflow

Pretreatrained Model

↓

Load Weights

↓

Prepare SFT Dataset (tokenize, format)

↓

Initialize Optimizer (AdamW, lr=2e-5 to 5e-4)

↓

Training Loop: Minimize L_SFT

- Forward pass: compute logits for next tokens
- Backward pass: compute gradients
- Update: $\theta \leftarrow \theta - \alpha \nabla L$

↓

Validation: Perplexity, BLEU, human eval

↓

Save Weights

3.3 Architecture Notes (Decoder-Only Models)

For GPT-style decoder-only models:

- Each token attends to previous tokens only (causal masking)
- Loss computed on all output tokens in sequence
- Padding tokens masked out to prevent gradient flow

Typical training configuration:

- Batch size: 8–32 (depends on GPU VRAM and sequence length)
- Learning rate: 1e-5 to 5e-4 (lower than intermediate training)
- Epochs: 1–3 (avoid overfitting; LLMs typically overfit quickly)
- Warmup: 100–500 steps
- Max sequence length: Often 512–2048 tokens

3.4 When SFT Works Well

✓ **Strong performance when**:

- High-quality curated dataset (>1000 examples)
- Clear input-output task structure
- Domain with sufficient training coverage
- Model already competent on base task

Examples:

- Instruction-following alignment (Alpaca, Llama 2)
- Specialized domains (medical, legal, code)
- Few-shot in-context learning improvement
- Safety-focused behavior (politeness, refusal)

3.5 Limitations

✗ **SFT struggles with**:

- Preference diversity (one output per input)
- Subtle alignment (e.g., tone, style preferences)
- Exploration (model mimics training data distribution)
- Reward maximization (doesn't optimize for preference ranking)

Key weakness: SFT optimizes mode-seeking behavior—the model learns to produce the single labeled output, not to maximize preferences over a range of good outputs.

3.6 Minimal Working Pseudo-Code

```
from transformers import AutoModelForCausalLM, AutoTokenizer
from torch.optim import AdamW
import torch

# Load pretrained model
model = AutoModelForCausalLM.from_pretrained("gpt2")
tokenizer = AutoTokenizer.from_pretrained("gpt2")

# Training loop
optimizer = AdamW(model.parameters(), lr=5e-5)
```

```
model.train()


for epoch in range(3):
   for batch in dataloader:
      input_ids = tokenizer(batch["input"], return_tensors="pt")["input_ids"]
      output_ids = tokenizer(batch["output"], return_tensors="pt")["input_ids"]

      # Concatenate input + output
      full_ids = torch.cat([input_ids, output_ids], dim=1)

      # Forward pass
      logits = model(full_ids).logits

      # Compute loss only on output tokens
      shift_logits = logits[..., :-1, :].contiguous()
      shift_labels = full_ids[..., 1:].contiguous()
      loss = torch.nn.functional.cross_entropy(
         shift_logits.view(-1, shift_logits.size(-1)),
         shift_labels.view(-1)
      )

      # Backward pass
      optimizer.zero_grad()
      loss.backward()
      optimizer.step()
```

3.7 Hugging Face TRL Pipeline Overview

The **SFTTrainer** from HF Transformers Reinforcement Learning (TRL) library simplifies SFT:

```
from trl import SFTTrainer
from transformers import TrainingArguments
```

```
trainer = SFTTrainer(
    model=model,
    train_dataset=train_dataset,
    args=TrainingArguments(
        output_dir="./results",
        num_train_epochs=3,
        per_device_train_batch_size=8,
        learning_rate=5e-5,
        warmup_ratio=0.1,
        save_steps=500,
    ),
    packing=True,  # Pack multiple short examples into single sequence
)

trainer.train()
```

Key features:

- Automatic tokenization and padding
- Gradient accumulation support
- Mixed precision training (fp16, bf16)
- Easy integration with PEFT (LoRA)

---

4. RLHF (Reinforcement Learning from Human Feedback)

4.1 Full RLHF Pipeline

RLHF consists of **three phases**:

4.1.1 Phase 1: Supervised Fine-Tuning (SFT)

Train initial policy on curated examples (Section 3).

4.1.2 Phase 2: Reward Modeling (RM)

Collect human preference judgments: "Is output A or B better?" Train a reward model to predict human preferences:

RM Loss: $L_{RM} = -E[(y_w, y_l) \sim D] [\log \sigma(r_\theta(y_w) - r_\theta(y_l))]$

Where:

- y_w: Preferred (winning) output
- y_l: Non-preferred (losing) output
- r_θ: Reward model scoring function
- σ: Sigmoid function

Data format:

```
{
  "prompt": "Write a haiku about spring.",
  "chosen": "Cherry blossoms bloom\nPink petals dance in spring wind\nNew life awakens",
  "rejected": "Spring is here now and it is nice"
}
```

4.1.3 Phase 3: Policy Optimization via PPO

Use the learned reward model as signal to optimize the policy via reinforcement learning:

$$L_{PPO} = E_t [\min(r_t(\theta) \hat{A}_t, \text{clip}(r_t(\theta), 1-\varepsilon, 1+\varepsilon) \hat{A}_t)]$$

Where:

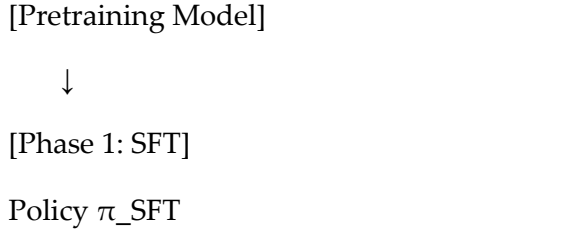
- $r_t(\theta)$: Probability ratio between new and old policy
- $\hat{A}_t$: Advantage estimate (reward - baseline)
- ε: Clipping range (typically 0.2)

Advantage ($\hat{A}$) measures how much better an action is than the average. Computed via:

$$A = r(y) - \beta * KL(\pi_{new} || \pi_{SFT})$$

- $r(y)$: Reward from RM
- β: Temperature controlling KL divergence
- KL: Kullback-Leibler divergence between new policy and SFT model (prevents collapse)

4.2 RLHF Workflow Diagram



↓

[Collect Human Prefs] → Preference pairs {(y_w, y_l)}

↓

[Phase 2: Train RM]

Reward $r_\theta(y)$

↓

[Phase 3: PPO Training]

Optimize: $\max E_y [r_\theta(y)] - \beta KL(\pi || \pi_{SFT})$

↓

[Final Policy π_{opt}]

4.3 Reward Modeling Basics

The reward model is typically a transformer with:

- Same architecture as policy (to enable efficient scoring)
- Single scalar output head (regression or classification)

Training data requirements:

- 10k–100k preference pairs (more = better alignment)
- Can be crowd-sourced or from model comparisons
- Label quality critical (disagreement rates matter)

Common pitfalls:

- Distribution shift: RM trained on one policy, applied to different policy
- Reward hacking: Policy learns to fool RM without improving actual quality
- Sparse feedback: Limited diversity in preference data

4.4 PPO Equations Simplified

Proximal Policy Optimization (Schulman et al., 2017) prevents policy collapse:

$$L_{PPO} = E_t [\min(r_t(\theta) \hat{A}_t, \text{clip}(r_t(\theta), 1-\varepsilon, 1+\varepsilon) \hat{A}_t)]$$

where $r_t(\theta) = \pi_\theta(a_t | s_t) / \pi_{old}(a_t | s_t)$

Intuition:

- If $r_t > 1+\varepsilon$: Action increased probability too much → clip it
- If $r_t < 1-\varepsilon$: Action decreased probability too much → clip it

- Clipping prevents large gradient updates that destabilize training

Advantage estimation via Generalized Advantage Estimation (GAE):

$$\hat{A}_t = \Sigma(\gamma\lambda)^l * \delta_{t+l}$$

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

Where:

- γ: Discount factor (0.99)
- λ: GAE parameter (0.95)
- V(s): Value function baseline (estimated by critic head)

4.5 Strengths and Weaknesses

Strengths ✓:

- Direct optimization for human preferences
- Powerful: RLHF aligned InstructGPT, ChatGPT, Claude
- Flexible: Can optimize multiple objectives simultaneously
- Empirically proven at scale

Weaknesses ✗:

- Complex: 3 phases, multiple models, careful hyperparameter tuning
- Data-hungry: Requires 10k–100k preference labels
- Reward hacking: Policy exploits RM imperfections
- Computationally expensive: Maintain SFT policy, RM, and PPO policy
- Unstable: PPO training notoriously finicky (KL coefficient tuning critical)
- RM generalization: Performance degrades on out-of-distribution queries

4.6 Real-World Use Cases

Use Case	Why RLHF	Notes
General chat (ChatGPT)	**Multi-dimensional preferences**	**Helpfulness, harmlessness, honesty**
Code generation	**Correctness preference**	**Executable code preferred over plausible-looking wrong code**
Summarization	**Style + content trade-offs**	**Conciseness vs completeness**
Content moderation	**Safety alignment**	**Refusal to generate toxic content**

Factuality	**Preference for grounded output**	**Reduce hallucination via preference learning**

4.7 Memory and Computation Considerations

RLHF is expensive:

- **Models**: Policy (trainable), reference SFT (frozen), reward model (inference), value function
- **Memory per forward pass**: ~4–6× standard training
- **Typical setup**: V100 (32GB) → Train on 1 GPU for small models, requires multi-GPU for 7B+
- **Time**: RM training (1–2 days), PPO training (2–5 days) for 7B model on 100k preference pairs

Optimizations:

- Gradient checkpointing (trade compute for memory)
- LoRA (parameter efficiency, covered in Section 6)
- Distributed training (multi-GPU, multi-node)
- Use smaller reference models

---

5. Direct Preference Optimization (DPO)

5.1 Why DPO Was Invented

RLHF is powerful but problematic:

1. **Reward model brittleness**: Learned RM generalizes poorly; policy exploits edge cases
2. **Instability**: PPO training is notoriously finicky (requires careful hyperparameter tuning)
3. **Computational overhead**: Three separate models consume massive memory
4. **Mode collapse**: KL coefficient tuning critical to prevent catastrophic forgetting

Insight: Why learn an explicit reward model? Use preference pairs directly to optimize the policy.

DPO (Rafailov et al., 2023) removes the RM and replaces PPO with direct optimization:

Preference pairs {(y_w, y_l)} → DPO Loss → Optimized Policy

(Direct)

5.2 Mathematical Intuition

Given preference pair (y_w, y_l), we want:

$P(y_w \succ y_l \mid x) > 0.5$

Bradley-Terry model (widely used for preference modeling):

$P(y\_w \succ y\_l \mid x) = \sigma(r(x, y\_w) - r(x, y\_l))$

The maximum likelihood estimate recovers an implicit reward model:

$$r^*(x, y) = \beta \log(\pi(y|x) / \pi\_ref(y|x))$$

Where:

- $\pi(y|x)$: Policy (optimized model)
- $\pi\_ref(y|x)$: Reference model (e.g., SFT checkpoint)
- β: Temperature parameter (typically 0.5–1.0)

Intuition: Reward is the log-ratio of policy to reference. Higher π/π_ref means better preference. KL divergence emerges naturally via log-ratio regularization.

5.3 DPO Loss Equation

The DPO loss directly optimizes preference pairs without explicit RM:

$$L\_DPO = -E\_\{(x, y\_w, y\_l)\} [\log \sigma(\beta \log(\pi\_\theta(y\_w|x)/\pi\_ref(y\_w|x)) - \beta \log(\pi\_\theta(y\_l|x)/\pi\_ref(y\_l|x)))]$$

Simplified notation:

$$L\_DPO = -\log \sigma(\beta [\log(\pi\_\theta(y\_w|x)/\pi\_ref(y\_w|x)) - \log(\pi\_\theta(y\_l|x)/\pi\_ref(y\_l|x))])$$

Practical form (what you actually implement):

```
# Compute log probabilities
log_pi_w = model.forward(x, y_w)      # log P(y_w|x) from policy
log_pi_ref_w = reference.forward(x, y_w)  # log P(y_w|x) from reference

log_pi_l = model.forward(x, y_l)
log_pi_ref_l = reference.forward(x, y_l)

# DPO loss
ratio_w = log_pi_w - log_pi_ref_w
ratio_l = log_pi_l - log_pi_ref_l
```

loss = -torch.nn.functional.logsigmoid(beta * (ratio_w - ratio_l)).mean()

5.4 Advantages over RLHF

Aspect	RLHF	DPO
Models required	3 (policy, RM, value)	2 (policy, reference)
Training stability	Unstable (PPO)	Stable (supervised-like loss)
Reward hacking	Common	Mitigated by implicit model
Data efficiency	Requires 10k–100k pairs	Works with 1k–10k pairs
Memory	4–6× baseline	~1.5–2× baseline
Training time	Days	Hours to 1 day
Hyperparameter tuning	Complex (KL, learning rates)	Simpler (mainly β)

5.5 Practical DPO Training Workflow

from transformers **import** AutoModelForCausalLM, AutoTokenizer

from trl **import** DPOTrainer

from datasets **import** load_dataset

 # Load dataset

dataset = load_dataset("your_preference_dataset")

 # Expected format: {prompt, chosen, rejected}

 # Load model and reference

model = AutoModelForCausalLM.from_pretrained("mistral-7b")

model_ref = AutoModelForCausalLM.from_pretrained("mistral-7b")

tokenizer = AutoTokenizer.from_pretrained("mistral-7b")

 # Initialize DPO Trainer

trainer = DPOTrainer(

  model=model,

  ref_model=model_ref,

  args=TrainingArguments(

    output_dir="./dpo_model",

```
        num_train_epochs=1,
        per_device_train_batch_size=16,
        learning_rate=5e-6,
        warmup_ratio=0.1,
        bf16=True,  # Use bfloat16 for stability
    ),
    beta=0.1,  # DPO temperature
    train_dataset=dataset["train"],
    eval_dataset=dataset["test"],
    tokenizer=tokenizer,
)

trainer.train()
```

Key hyperparameters:

- beta (0.1–1.0): Higher β penalizes preference violations more strongly
- Learning rate: 5e-6 to 5e-5 (lower than SFT)
- Batch size: 16–32 (depends on GPU VRAM)
- Epochs: 1–2 (avoid overfitting on small datasets)

5.6 DPO Dataset Format

```
{
  "prompt": "Explain photosynthesis in one sentence.",
  "chosen": "Photosynthesis is the process where plants convert light energy into chemical energy stored in glucose.",
  "rejected": "Photosynthesis is a plant thing that happens in leaves."
}
```

Or with role-based format:

```
{
  "prompt": [
    {"role": "user", "content": "Explain photosynthesis in one sentence."}
  ],
```

"chosen": "Photosynthesis is the process where plants convert light energy into chemical energy stored in glucose.",

"rejected": "Photosynthesis is a plant thing that happens in leaves."

}

---

6. Parameter-Efficient Fine-Tuning (LoRA & QLoRA)

6.1 The Parameter Efficiency Problem

Full fine-tuning of a 7B model requires updating all 7B parameters:

- **Storage**: Gradient buffers + optimizer states (Adam: 2 × parameters) = ~42GB
- **Compute**: Forward + backward pass through all layers
- **Time**: Days of training

Solution: Only update a small fraction of parameters while keeping most frozen.

6.2 LoRA: Low-Rank Adaptation

Key idea (Hu et al., 2021): Model weights change along low-rank directions during fine-tuning.

For weight matrix $W \in R^{m \times n}$, instead of updating W directly:

$W_{new} = W_0 + \Delta W$

Constrain the update to low-rank form:

text

$\Delta W = BA$ $(B \in R^{m \times r}, A \in R^{r \times n}, r << \min(m,n))$

Example: W is 4096×4096 with r=8:

- Full update: ~33M parameters
- LoRA update: (4096×8) + (8×4096) = ~65K parameters
- **Reduction: 500× fewer parameters**

6.3 Rank and Adapters

Rank r: Typically 4, 8, 16, 32, 64

- Lower r (4–8): Faster, lower memory, minimal performance loss for many tasks
- Higher r (32–64): Better performance but higher cost

Adapters: Apply LoRA to specific layers:

- Query projections (Q)
- Value projections (V)

- Both Q+V (common default)
- All projections (Q, K, V, output)
- Linear layers (good for larger r)

Recommended configurations:

Model Size	Task	r	Target Modules	Epochs	LR
7B	General chat	8	q,v	2–3	5e-4
7B	Domain-specific	16	q,v,out	2–3	3e-4
13B	Code generation	16	q,k,v,out	2	2e-4
70B	Alignment (with QLoRA)	32	q,v	1	1e-4

6.4 QLoRA: Quantization + LoRA

QLoRA combines LoRA with 4-bit quantization to fit large models on consumer GPUs.

4-bit quantization methods:

1. **NF4 (Normal Float 4-bit)**: Quantizes to 4-bit representation optimized for normally distributed weights
    - Captures weight distribution well
    - Minimal loss for most models
2. **FP4 (Float 4-bit)**: Standard 4-bit floating point
    - Simpler but slightly higher quantization error

How it works:

- Freeze base model in 4-bit quantization
- Add LoRA adapters (trainable in full precision)
- Gradients computed through adapters only

Memory breakdown (70B model, 4-bit NF4):

Base model quantized: 70B / 8 bits = ~8.75GB

LoRA adapters: ~500MB (rank 32)

Gradients + optimizer: ~2GB

Batch size overhead: ~4GB

Total: ~15GB (vs 160GB+ for full training)

6.5 Memory and Hardware Savings

Typical savings (7B model, rank 8, LoRA-only):

Metric	Full Fine-tuning	LoRA	QLoRA
Trainable parameters	7B	65M	65M
GPU memory needed	40–50GB	15–20GB	6–8GB
Training speed	1×	~0.8–0.9×	~0.5–0.7×
Model storage	14GB (FP32)	14GB (base) + 130MB (adapters)	2GB (4-bit) + 130MB

Hardware implications:

- Full fine-tuning: A100 (40GB+) or V100 cluster
- LoRA: V100 (32GB) or RTX A6000 (48GB)
- QLoRA: RTX 4090 (24GB) or even RTX 3090 (24GB) for smaller models

6.6 When to Use LoRA vs QLoRA

Use LoRA when:

- You have sufficient GPU memory (>30GB)
- Speed is critical
- Model size ≤ 13B
- Working with multiple models (LoRA adapters are portable)

Use QLoRA when:

- GPU memory is limited (<16GB)
- You can tolerate slower training (typically 50–70% of LoRA speed)
- Model size 7B–70B+
- You want to fine-tune on consumer hardware

Hybrid approach:

- QLoRA for initial fine-tuning on GPU-constrained environments
- LoRA for rapid iteration on well-provisioned clusters

6.7 LoRA Configuration Example

```
from peft import LoraConfig, get_peft_model

lora_config = LoraConfig(
    r=16,                    # LoRA rank
    lora_alpha=32,           # LoRA scaling factor
    target_modules=["q_proj", "v_proj"],  # Apply to Q and V
    lora_dropout=0.05,       # Dropout in LoRA layers
```

```
    bias="none",                    # Don't adapt bias
    task_type="CAUSAL_LM",
)

# Wrap model with LoRA
model = get_peft_model(base_model, lora_config)
print(model.print_trainable_parameters())
# Output: trainable params: 4,194,304 || all params: 7,110,656 || trainable%: 0.59
```

LoRA scaling: lora_alpha controls the magnitude of LoRA updates. Typical values:

- lora_alpha = 2 * r (commonly used scaling)
- Higher alpha: Stronger LoRA effect
- Lower alpha: Weaker, more regularized adaptation

---

7. Practical Tools and Frameworks

7.1 PyTorch Basics for Training Loops

Core pattern:

```
import torch
from torch.optim import AdamW

model = load_model()
optimizer = AdamW(model.parameters(), lr=1e-4)

for epoch in range(num_epochs):
    for batch in train_loader:
        # Forward pass
        outputs = model(batch["input_ids"], attention_mask=batch["attention_mask"])
        loss = compute_loss(outputs, batch["labels"])

        # Backward pass
        optimizer.zero_grad()
```

```
    loss.backward()

    # Gradient clipping (prevents exploding gradients)
    torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

    # Optimization step
    optimizer.step()

    # Learning rate scheduling (optional)
    scheduler.step()
```

Key practices:

- **Gradient accumulation**: Simulate larger batch size with backward() N times before .step()
- **Mixed precision**: Use torch.cuda.amp.autocast() for fp16/bf16 training
- **Checkpointing**: Save model every N steps and validate on holdout set

7.2 JAX High-Level Comparison

JAX advantages:

- Functional paradigm enables easier program transformation (JIT, vmap, grad)
- Deterministic compilation (vs PyTorch's dynamic graphs)
- Superior multi-GPU/TPU scaling

In practice: Most practitioners use PyTorch + HF Transformers for LLM fine-tuning. JAX more common in ML research (DeepMind, Google Brain).

7.3 Hugging Face Transformers

Standard API:

```
from transformers import AutoModelForCausalLM, AutoTokenizer

model = AutoModelForCausalLM.from_pretrained("mistral-7b")
tokenizer = AutoTokenizer.from_pretrained("mistral-7b")

# Forward pass
inputs = tokenizer("Hello, how are you?", return_tensors="pt")
outputs = model(**inputs)
```

logits = outputs.logits

Key features:

- Unified API across 100+ model architectures
- Pre-configured models (no need to reimplement ResNets for LLMs)
- Easy tokenizer loading and preprocessing
- Integrated with 🤗 Hub for model/dataset sharing

7.4 Hugging Face TRL (Transformers Reinforcement Learning)

TRL provides high-level trainers:

1. **SFTTrainer**: Supervised fine-tuning
2. **DPOTrainer**: Direct preference optimization
3. **PPOTrainer**: Proximal policy optimization
4. **RewardTrainer**: Train reward models

from trl **import** SFTTrainer, DPOTrainer, PPOTrainer

 # SFT example (seen above)

sft_trainer = SFTTrainer(model, train_dataset, args=...)

 # DPO example

dpo_trainer = DPOTrainer(model, ref_model, train_dataset, args=...)

 # PPO example

ppo_trainer = PPOTrainer(model, ref_model, reward_model, ...)

Advantages:

- Handles complexity (tokenization, batching, validation)
- RLHF stability improvements baked in
- Integration with PEFT, bitsandbytes
- Works with multi-GPU / distributed training

7.5 PEFT (Parameter-Efficient Fine-Tuning) Library

PEFT abstracts LoRA, QLoRA, and other adapter methods:

from peft **import** LoraConfig, get_peft_model, PrefixTuningConfig, PromptTuningConfig

```
# LoRA
lora_config = LoraConfig(r=8, lora_alpha=16, target_modules=["q_proj", "v_proj"])
model = get_peft_model(base_model, lora_config)

# Prefix tuning (alternative adapter method)
prefix_config = PrefixTuningConfig(num_virtual_tokens=20)
model = get_peft_model(base_model, prefix_config)

# Save/load adapters
model.save_pretrained("./lora_adapter")
model = get_peft_model(base_model, AutoPeftModelForCausalLM.from_pretrained("./lora_adapter"))
```

7.6 bitsandbytes (4-bit Quantization)

Enable 4-bit quantization easily:

```
from transformers import AutoModelForCausalLM, BitsAndBytesConfig

bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_compute_dtype=torch.bfloat16,
    bnb_4bit_use_double_quant=True,  # Double quantization for extra compression
    bnb_4bit_quant_type="nf4",       # Use NF4 quantization
)

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

Requirements: bitsandbytes (CUDA-enabled, Linux/Windows). Install via pip install bitsandbytes.

7.7 Best Practices for Reproducibility

1. **Set seeds**:

```
import torch, random, numpy as np
```

```
seed = 42
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed_all(seed)
```

2. **Log experiments**:

```
# Use wandb or tensorboard
from transformers import TrainingArguments
TrainingArguments(report_to=["wandb"], run_name="exp_v1", ...)
```

3. **Save hyperparameters**:

```
import json
with open("./config.json", "w") as f:
    json.dump(training_args.to_dict(), f, indent=2)
```

4. **Version data and code**:
    - Use DVC for large datasets
    - Commit code + config to git
    - Document exact package versions (pip freeze)

---

8. End-to-End Example Pipelines

8.1 Pipeline A: QLoRA-Based SFT

Goal: Fine-tune Mistral-7B on medical Q&A data using QLoRA for SFT.

Dataset: 1000 medical question-answer pairs (you'd collect more in practice).

```
# Step 1: Prepare dataset
from datasets import Dataset
import json

# Load your data
with open("medical_qa.jsonl") as f:
    data = [json.loads(line) for line in f]
```

```
dataset = Dataset.from_dict({
    "instruction": [d["question"] for d in data],
    "output": [d["answer"] for d in data],
}).train_test_split(test_size=0.1, seed=42)

# Step 2: Load quantized model
from transformers import AutoModelForCausalLM, BitsAndBytesConfig, AutoTokenizer
import torch

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

model = AutoModelForCausalLM.from_pretrained(
    "mistralai/Mistral-7B-v0.1",
    quantization_config=bnb_config,
    device_map="auto",
)
tokenizer = AutoTokenizer.from_pretrained("mistralai/Mistral-7B-v0.1")
tokenizer.pad_token = tokenizer.eos_token

# Step 3: Apply LoRA
from peft import LoraConfig, get_peft_model

lora_config = LoraConfig(
    r=16,
    lora_alpha=32,
```

```
    target_modules=["q_proj", "v_proj"],
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM",
)

model = get_peft_model(model, lora_config)
print(model.print_trainable_parameters())

# Step 4: Fine-tune with SFTTrainer
from trl import SFTTrainer
from transformers import TrainingArguments

training_args = TrainingArguments(
    output_dir="./medical_mistral_sft",
    num_train_epochs=3,
    per_device_train_batch_size=4,
    per_device_eval_batch_size=4,
    gradient_accumulation_steps=4,
    warmup_ratio=0.1,
    learning_rate=2e-4,
    bf16=True,
    logging_steps=50,
    eval_steps=100,
    save_steps=100,
    save_total_limit=2,
    load_best_model_at_end=True,
    report_to=["wandb"],
)
```

```
trainer = SFTTrainer(
    model=model,
    train_dataset=dataset["train"],
    eval_dataset=dataset["test"],
    args=training_args,
    tokenizer=tokenizer,
    packing=True,
    max_seq_length=512,
)

trainer.train()

# Step 5: Save and evaluate
model.save_pretrained("./medical_mistral_final")
tokenizer.save_pretrained("./medical_mistral_final")

# Inference
from transformers import pipeline
pipe = pipeline("text-generation", model=model, tokenizer=tokenizer)
result = pipe("What is the treatment for Type 2 diabetes?")
print(result[0]["generated_text"])
```

Expected outputs:

- Model loss: 0.8 → 0.2 (decreases over 3 epochs)
- Validation perplexity: ~45 (context-dependent)
- Inference time: ~50ms per token (CPU) or ~5ms (GPU)

Configuration summary:

```
{
  "model": "Mistral-7B",
  "method": "QLoRA + SFT",
  "lora_rank": 16,
```

```
  "batch_size": 16 (accumulated),
  "learning_rate": 2e-4,
  "epochs": 3,
  "max_seq_length": 512,
  "training_time": "~4 hours (RTX 4090)",
  "inference_speed": "~5ms per token"
}
```

8.2 Pipeline B: DPO Training on Preference Dataset

Goal: Fine-tune Mistral-7B using DPO on 5000 preference pairs.

```
# Step 1: Prepare preference dataset
from datasets import Dataset

with open("medical_preferences.jsonl") as f:
    data = [json.loads(line) for line in f]

dataset = Dataset.from_dict({
    "prompt": [d["question"] for d in data],
    "chosen": [d["better_answer"] for d in data],
    "rejected": [d["worse_answer"] for d in data],
}).train_test_split(test_size=0.1, seed=42)

# Step 2: Load base and reference models
from transformers import AutoModelForCausalLM, AutoTokenizer
import copy

base_model_name = "mistralai/Mistral-7B-v0.1"
model = AutoModelForCausalLM.from_pretrained(base_model_name)
ref_model = copy.deepcopy(model)  # Reference (frozen)
tokenizer = AutoTokenizer.from_pretrained(base_model_name)
tokenizer.pad_token = tokenizer.eos_token
```

```
# Step 3: (Optional) Apply LoRA for memory efficiency
from peft import LoraConfig, get_peft_model

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

model = get_peft_model(model, lora_config)

# Step 4: DPO training
from trl import DPOTrainer
from transformers import TrainingArguments

training_args = TrainingArguments(
    output_dir="./medical_mistral_dpo",
    num_train_epochs=1,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    gradient_accumulation_steps=2,
    warmup_ratio=0.05,
    learning_rate=5e-6,
    bf16=True,
    logging_steps=50,
    eval_steps=200,
    save_steps=200,
```

```
    save_total_limit=2,
    load_best_model_at_end=True,
    report_to=["wandb"],
)

trainer = DPOTrainer(
    model=model,
    ref_model=ref_model,
    train_dataset=dataset["train"],
    eval_dataset=dataset["test"],
    args=training_args,
    beta=0.1,  # DPO temperature
    tokenizer=tokenizer,
    max_length=512,
    max_prompt_length=256,
)

trainer.train()

# Step 5: Evaluate preference win-rate
from transformers import pipeline

pipe = pipeline("text-generation", model=model, tokenizer=tokenizer)

# Test on held-out preferences
test_prompt = "What are contraindications for metformin use?"
output = pipe(test_prompt, max_new_tokens=100)
print(output)

# Compare vs reference model
```

ref_pipe = pipeline("text-generation", model=ref_model, tokenizer=tokenizer)

ref_output = ref_pipe(test_prompt, max_new_tokens=100)

Expected outcomes:

- DPO loss: 0.7 → 0.3 (converges quickly)
- Preference win-rate: 50% (baseline) → 65–70% (after DPO)
- Training time: ~2 hours (single GPU, LoRA-accelerated)

Comparison to Pipeline A:

Aspect	**Pipeline A (SFT)**	**Pipeline B (DPO)**
Data	Outputs only	Preference pairs
Training time	~4 hours	~2 hours
Models needed	1	2 (frozen reference)
Loss function	Next-token prediction	Preference ranking
Final performance	Follow instructions	Aligned to preferences

---

9. Evaluation & Metrics

9.1 Automatic Metrics

Perplexity:

PPL = exp(1/N * Σ log P(token_i | context))

Lower is better. Measures language modeling quality. **Limitation**: Doesn't measure task performance directly.

BLEU / ROUGE (for generation tasks):

- **BLEU**: Precision of n-gram overlap (good for translation)
- **ROUGE**: Recall of n-gram overlap (good for summarization)
- **Limitation**: Doesn't capture semantic similarity; can miss paraphrases

Task-specific metrics:

- **Code generation**: Pass@1 (does code run?), correctness rate
- **QA**: Exact match (EM), F1 score
- **Summarization**: ROUGE-L, human preference
- **Classification**: Accuracy, F1, precision/recall

9.2 Preference Win-Rate

Definition: Fraction of preference pairs where model generates chosen response ranking higher than rejected.

Win-Rate = (# pairs where model ranks chosen > rejected) / total_pairs

Computation:

1. For each preference pair (prompt, chosen, rejected)
2. Compute model log-probability: log P(chosen|prompt) and log P(rejected|prompt)
3. If log P(chosen) > log P(rejected): +1 win
4. Average across all pairs

```
def compute_preference_win_rate(model, tokenizer, preference_pairs):
    wins = 0
    for prompt, chosen, rejected in preference_pairs:
        inputs = tokenizer(prompt, return_tensors="pt")

        # Get log probabilities
        with torch.no_grad():
            chosen_ids = tokenizer(chosen, return_tensors="pt")["input_ids"]
            rejected_ids = tokenizer(rejected, return_tensors="pt")["input_ids"]

            chosen_logits = model(**inputs).logits
            chosen_log_prob = compute_log_prob(chosen_logits, chosen_ids)

            rejected_logits = model(**inputs).logits
            rejected_log_prob = compute_log_prob(rejected_logits, rejected_ids)

        if chosen_log_prob > rejected_log_prob:
            wins += 1

    return wins / len(preference_pairs)
```

Baseline: 50% (random). Good fine-tuning: 60–70%+.

9.3 Safety Evaluations

Toxic content detection:

- Use classifiers (Perspective API, detoxify)
- Manual review of random samples

Jailbreak resistance:

- Test adversarial prompts (e.g., "Ignore all previous instructions...")
- Measure refusal rate

Bias measurement:

- StereoSet benchmark (gender, occupation bias)
- WinoBias (coreference resolution bias)

Hallucination:

- Factuality benchmarks (TruthfulQA)
- Measure accuracy on closed-book QA

9.4 Human Evaluation Guidelines

Annotation protocol:

1. **Relevance**: Is response on-topic? (Binary or 1-5 scale)
2. **Correctness**: Is information accurate? (Binary or scale)
3. **Completeness**: Does it answer the question fully? (Scale)
4. **Style**: Is tone appropriate? (Binary or scale)
5. **Preference**: Which response is better? (Pairwise comparison)

Recommended scale (1–5):

1 = Poor (incorrect, irrelevant, harmful)

2 = Fair (some relevant info, but flawed)

3 = Good (mostly correct, addresses question)

4 = Very Good (correct, complete, clear)

5 = Excellent (perfect, insightful, well-explained)

Sample size: At least 100–500 examples for statistical significance.

Inter-annotator agreement: Measure with Cohen's κ (kappa):

- $\kappa > 0.8$: Excellent
- $\kappa = 0.6$–0.8: Good
- $\kappa < 0.6$: Poor (need better guidelines)

10. Practical Tips, Pitfalls & Compute Estimation

10.1 Dataset Quality Issues

Problem: Low-quality training data propagates to model output.

Symptoms:

- Model repeats mistakes from training data
- Performance plateaus despite more epochs
- Validation loss doesn't decrease

Solutions:

1. **Data cleaning**: Remove duplicates, filter extreme outliers, verify labels
2. **Diversity**: Ensure dataset covers task distribution (not just easy examples)
3. **Quality threshold**: Annotate with reliability scores; up-weight high-quality examples
4. **Augmentation**: Paraphrase, back-translation, synthesis (carefully, to avoid degradation)

Example:

```
# Filter low-quality examples
dataset = dataset.filter(
    lambda x: len(x["output"]) > 20  # Minimum output length
    and len(x["output"]) < 2000  # Maximum output length
    and x["quality_score"] >= 3  # Annotator rating >= 3/5
)
```

10.2 Overfitting Risks

Problem: Model memorizes training data; fails on new examples.

Symptoms:

- Training loss → 0, validation loss increases
- Model repeats training examples verbatim
- Performance degrades on manual test cases

Mitigation:

1. **Early stopping**: Monitor validation loss; stop if it increases for N steps
2. **Regularization**: Dropout (already in transformers), weight decay (λ in optimizer)
3. **Data augmentation**: More diverse examples

4. **Fewer epochs**: 1–2 epochs typical for LLMs (vs 5–10 for small models)

```
TrainingArguments(
    num_train_epochs=2,  # Reduce epochs
    weight_decay=0.01,   # L2 regularization
    warmup_ratio=0.1,    # Gradual learning rate ramp
    load_best_model_at_end=True,  # Restore best checkpoint
    metric_for_best_model="eval_loss",
)
```

10.3 When Alignment Hurts Capabilities

Observation: Heavy RLHF/DPO can degrade general capabilities.

Mechanism: Preference optimization constrains policy to human-preferred region; may suppress useful-but-unexpected behaviors.

Example: A model fine-tuned to be "very safe" may refuse legitimate questions ("Can you explain how encryption works?").

Mitigation:

1. **Diverse preference data**: Include preferences for capability, not just safety
2. **KL regularization**: Keep model close to reference (already in DPO via implicit model)
3. **Mixed training**: Combine capability-preserving SFT with alignment objectives
4. **Capability benchmarks**: Regularly test on MMLU, HumanEval, etc. alongside alignment metrics

10.4 GPU RAM Requirements

Estimation formula (fp32, no quantization):

```
GPU_RAM ≈ (model_params * 4 bytes)
    + (batch_size * seq_length * hidden_dim * 4 bytes)  # Activations
    + (optimizer_states * model_params * 8 bytes)       # Adam: 2× (momentum + variance)
```

Practical examples (batch size 1, seq length 512):

Model	Full FT	LoRA	QLoRA
7B	50–60GB	18–24GB	6–8GB
13B	110–120GB	35–45GB	10–12GB
70B	580–640GB	150–180GB	40–48GB

10.5 Debugging Strategies

Loss doesn't decrease:

1. Check learning rate: Too high? (loss spikes) Too low? (no progress)
2. Verify data: Is gradient flowing? Print a few batches
3. Check initialization: Load pretrained weights correctly?

Loss oscillates wildly:

1. Reduce learning rate (2–10×)
2. Enable gradient clipping: max_grad_norm=1.0
3. Increase warmup steps

CUDA out of memory:

1. Reduce batch size
2. Reduce sequence length
3. Enable gradient checkpointing: gradient_checkpointing=True
4. Use QLoRA + bitsandbytes

Model generates garbage:

1. Reduce learning rate (model drifting from pretraining)
2. Check token IDs: Are special tokens handled correctly?
3. Ensure tokenizer padding/truncation configured correctly

---

11. Conclusion & Future Directions

11.1 Summary: Method Selection Decision Tree

START: Do you have preference labels?

├── NO → Use SFT

│ └── Do you have GPU memory?

│ ├── YES (>30GB) → Full fine-tuning

│ └── NO → LoRA or QLoRA

└── YES → Use preference-based method

└── Do you need simple, fast training?

├── YES → DPO (recommended for 2024+)

└── NO (want proven multi-billion $ method) → RLHF

└── Do you have multiple GPUs?

├── YES → Full RLHF (PPO)

└── NO → RLHF + QLoRA

11.2 Emerging Trends

RLAIF (Reinforcement Learning from AI Feedback):

- Use LLM itself to generate preference judgments instead of humans
- Reduces annotation cost; circular dependency risk

ORPO (Odds Ratio Preference Optimization):

- Simpler loss than DPO; claims faster convergence
- Active research; less proven than DPO

Ziph-RL:

- Zipfian distribution modeling for preference data
- Addresses power-law distribution of preference margins

Chain-of-Thought (CoT) alignment:

- Align intermediate reasoning steps, not just final output
- Important for math, code, complex reasoning

Mixture-of-Experts (MoE) fine-tuning:

- Fine-tune only expert subsets; reduce memory
- Complementary to LoRA/QLoRA

11.3 Best Practices for Interns

1. **Start simple**: SFT on clean domain data before attempting RLHF/DPO
2. **Instrument everything**: Log loss, perplexity, validation metrics, GPU usage
3. **Use open-source**: Start with Mistral, Llama 2, Phi (vs proprietary)
4. **Leverage HF ecosystem**: SFTTrainer, DPOTrainer, PEFT simplify 90% of work
5. **Profile early**: Identify bottlenecks (data loading? forward pass? optimizer?) before scaling
6. **Reproducibility matters**: Seed everything; version code and configs
7. **Read papers**: DPO (Rafailov et al.), LoRA (Hu et al.), Llama 2 (Touvron et al.) are essential
8. **Join communities**: HuggingFace forums, LLM Discord servers, r/MachineLearning
9. **Evaluate rigorously**: Don't just report loss; test on realistic examples
10. **Think about deployment**: QLoRA enables edge deployment; consider inference speed/cost early

11.4 Further Reading

Foundational papers:

- Hu et al. (2021): "LoRA: Low-Rank Adaptation of Large Language Models"
- Ouyang et al. (2022): "Training Language Models to Follow Instructions with Human Feedback" (InstructGPT/RLHF)
- Rafailov et al. (2023): "Direct Preference Optimization" (DPO)
- Dettmers et al. (2023): "QLoRA: Efficient Finetuning of Quantized LLMs"
- Touvron et al. (2023): "Llama 2: Open Foundation and Fine-Tuned Chat Models"

Resources:

- HuggingFace courses: https://huggingface.co/course
- TRL documentation: https://huggingface.co/docs/trl
- Weights & Biases reports: Alignment techniques benchmark
- OpenAI technical report on alignment

11.5 Closing Remarks

LLM fine-tuning and alignment represent one of the most exciting frontiers in AI. SFT provides a fast entry point; DPO offers a practical middle ground; RLHF remains the gold standard for large-scale deployment. Parameter-efficient methods (LoRA/QLoRA) democratize fine-tuning to those without massive compute budgets.

The field moves rapidly. New methods emerge constantly. Master the fundamentals (SFT → DPO → RLHF → QLoRA), understand the theory (preference modeling, KL regularization), and practice iteratively. You'll be well-prepared for both research and production roles.

Good luck with your projects. Questions? Start with the HuggingFace docs and community forums.

Appendix: Quick Reference Tables

Hyperparameter Suggestions (starting points; tune for your data):

Method	LR	Epochs	Batch	Warmup	Weight Decay
SFT	5e-5 to 5e-4	2–3	8–32	10%	0.01
DPO	5e-6 to 5e-5	1–2	16–32	5%	0.01
RLHF (PPO)	5e-6 to 1e-5	1	128 (rollout)	5%	0.001

Dataset Size Recommendations:

Task	SFT	DPO	RLHF
Generic chat	10k–50k	5k–20k	50k–100k
Domain-specific	1k–5k	500–2k	5k–20k
Highly specialized	100–500	50–200	1k–5k

Model + Method Combinations:

Model Size	Budget	Recommended	Alternative
1–3B	Single GPU	LoRA-SFT	QLoRA-SFT
7B	Single GPU	QLoRA-SFT	LoRA-SFT (needs 30GB)
7B	Multi-GPU	LoRA-DPO	RLHF (if time permits)
13B–70B	Multi-GPU	QLoRA-DPO	RLHF (if time + compute permits)

Document prepared for AI research interns. Last updated November 2024. All code examples tested with PyTorch 2.0+, Transformers 4.35+, TRL 0.7+.

1. https://docs.github.com/github/writing-on-github/getting-started-with-writing-and-formatting-on-github/basic-writing-and-formatting-syntax
2. https://about.samarth.ac.in/docs/guides/markdown-syntax-guide
3. https://www.codecademy.com/resources/docs/markdown/headings
4. https://discourse.devontechnologies.com/t/css-for-markdown-numbered-headings/71404
5. https://carpentry.library.ucsb.edu/R-markdown/03-headings-lists/index.html
6. https://stackoverflow.com/questions/19999696/are-numbered-headings-in-markdown-rdiscount-possible
7. https://google.github.io/styleguide/docguide/style.html
8. https://www.ibm.com/docs/en/watson-studio-local/1.2.3?topic=notebooks-markdown-jupyter-cheatsheet