LLM Fine-Tuning: A Comprehensive Guide

1. Introduction: What is Fine-Tuning and Why Should You Care?

Let's start with a simple question: You have a large language model (LLM) like GPT-3 or LLaMA. It's pretty good at general language tasks, but you want it to be AMAZING at your specific use case. Maybe you want it to:

- Write code in your company's style
- Answer medical questions accurately
- Be a helpful customer support agent
- Generate legal documents

This is where fine-tuning comes in. Think of it like this:

Pre-training = Teaching a child general language skills (reading, writing, grammar)

Fine-tuning = Teaching that child to become a specialist (doctor, lawyer, programmer)

Why Fine-Tuning Matters

Here's the thing - pre-training is EXPENSIVE. GPT-3 cost millions of dollars to train from scratch. But fine-tuning? That's where the magic happens at a fraction of the cost. You're not starting from zero; you're taking a model that already understands language and teaching it your specific requirements.

Let me give you a concrete example. Imagine you have GPT-3 and you ask it: "Write a function to sort an array."

Before fine-tuning:

```
def sort_array(arr):
    # This function sorts an array
```

```
    # There are many ways to sort...
    # Let me explain bubble sort first...
```

After fine-tuning on your codebase:

```
def sort_array(arr: List[int]) → List[int]:
    """Sort array in ascending order using quicksort."""
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return sort_array(left) + middle + sort_array(right)
```

See the difference? The fine-tuned model knows YOUR style, YOUR conventions, YOUR requirements.

---

2. The Big Picture: From Pre-training to Fine-tuning

Let's zoom out and understand the entire pipeline of creating a useful LLM:

Stage 1: Pre-training (The Foundation)

```
Raw Text (Internet) → Transformer → Next Token Prediction → Base Model
```

During pre-training, the model sees TRILLIONS of tokens. It learns:

- Grammar and syntax
- World knowledge
- Reasoning patterns
- Multiple languages
- Code, math, science, history...

But here's the catch - it learns to predict the next token, not necessarily to be helpful or accurate. If the internet says "the earth is flat" enough times, the model might learn that pattern too.

Stage 2: Fine-tuning (The Specialization)

This is where we shape the model for our needs. There are several approaches:

```
Base Model → Fine-tuning → Specialized Model
          ↓
   - Supervised Fine-tuning (SFT)
   - Unsupervised Adaptation
   - Instruction Tuning
   - RLHF (Reinforcement Learning from Human Feedback)
```

Think of it like this: Pre-training gives you a smart employee who knows a lot but doesn't know what you want them to do. Fine-tuning is the job training that teaches them exactly how to help you.

A Historical Note

This paradigm shift happened around 2018-2019 with BERT and GPT-2. Before this, people trained new models from scratch for each task. Then someone realized: "Wait, why not train one big model on general text and then specialize it?"

This was revolutionary. It's like realizing you don't need to teach every medical student how to read from scratch - you can assume they know language and just teach them medicine.

---

3. Supervised vs Unsupervised Fine-Tuning

Now let's dive into the two main flavors of fine-tuning. I'll explain both, then show you when to use each.

Supervised Fine-Tuning (SFT)

The Idea: You have input-output pairs, and you teach the model to map inputs to outputs.

```
Training Data:
Input: "Translate 'Hello' to Spanish"
Output: "Hola"

Input: "What's the capital of France?"
Output: "The capital of France is Paris."

Input: "Fix this code: prin('hello')"
Output: "print('hello')"
```

How it works:

1. You show the model an input
2. The model generates an output
3. You calculate the loss (how wrong it was)
4. You update the weights to reduce this loss
5. Repeat thousands of times

The Good:

- Precise control - you get exactly what you train for
- High quality if your data is good
- Clear evaluation metrics (accuracy, F1, etc.)

The Bad:

- Need labeled data (expensive!)
- Can overfit on small datasets
- Only learns what's in your data

Code Intuition (simplified):

```
def supervised_finetune(model, labeled_data):
    for input_text, target_output in labeled_data:
        model_output = model(input_text)
```

```
        loss = calculate_loss(model_output, target_output)
        update_weights(model, loss)
```

Unsupervised Fine-Tuning (Continued Pre-training)

The Idea: Keep training the model on raw text, but now it's YOUR text.

```
Training Data:
"In quantum mechanics, the wave function ψ describes..."
"The patent filed by ACME Corp on January 15th..."
"def quicksort(arr): if len(arr) <= 1: return arr..."
```

No labels! Just raw text from your domain.

How it works:

1. Same as pre-training - predict next token
2. But now on YOUR domain's text
3. Model absorbs vocabulary, facts, styles from your domain

The Good:

- No labeling needed
- Can use massive amounts of text
- Learns broad domain knowledge

The Bad:

- No direct task learning
- Can't control specific behaviors
- Harder to evaluate

Code Intuition:

```
def unsupervised_finetune(model, domain_text):
    for text_chunk in domain_text:
        # Same as pre-training!
        for i in range(len(text_chunk)-1):
```

```
        input_tokens = text_chunk[:i]
        target_token = text_chunk[i+1]
        loss = model.predict_next_token_loss(input_tokens, target_token)
        update_weights(model, loss)
```

When to Use Which?

Here's my rule of thumb:

Use Supervised Fine-tuning when:

- You have a specific task (classification, Q&A, translation)
- You have or can create quality labeled data
- You need precise behavior

Use Unsupervised Fine-tuning when:

- You want domain adaptation (medical, legal, scientific)
- You have lots of domain text but no labels
- You want the model to "speak the language" of your field

The Power Move: Do both! First unsupervised to learn the domain, then supervised to nail the task.

```
Base Model → Unsupervised on Domain Text → Supervised on Task Data → E
xpert Model
```

Example: Building a medical Q&A system

1. Start with LLaMA-2
2. Unsupervised fine-tune on medical textbooks, papers
3. Supervised fine-tune on (medical question, doctor answer) pairs
4. Result: Model that knows medicine AND answers questions well

---

4. Domain and Task Adaptation (DAPT/TAPT)

Let me tell you about a clever insight from the "Don't Stop Pretraining" paper (2020). They asked: "If we're going to fine-tune anyway, should we do some intermediate adaptation?"

Domain-Adaptive Pre-Training (DAPT)

The Idea: Before fine-tuning on your task, continue pre-training on text from your domain.

Think of it like this:

- Base model knows general English
- Your task is biomedical NER (finding disease names in text)
- DAPT says: "First, read a bunch of medical textbooks"

```
General Model → Read Medical Texts → Now Better at Medical Tasks
```

Example:

```
# Start with BERT
model = load_pretrained_bert()

# DAPT: Continue training on PubMed abstracts
medical_texts = load_pubmed_abstracts()  # Millions of medical papers
for batch in medical_texts:
    loss = model.masked_language_modeling(batch)
    update_weights(model, loss)

# Now model knows medical jargon, drug names, symptoms...
```

Task-Adaptive Pre-Training (TAPT)

The Idea: Even more specific - continue pre-training on the exact texts from your task.

Example: Your task is sentiment analysis on movie reviews

- Don't just use general text

- Don't even use general review text
- Use the EXACT movie reviews (without labels)

```
General Model → Read Task's Texts → Even Better at Specific Task
```

The Surprising Finding: TAPT often works as well as DAPT, despite using 100x less data!

Why? Because it's SO targeted. The model learns:

- The exact vocabulary of your task
- The writing style
- Common patterns and phrases

The Optimal Strategy

The paper found this hierarchy:

```
Best:  Base Model → DAPT → TAPT → Supervised Fine-tuning
Good:  Base Model → TAPT → Supervised Fine-tuning
OK:    Base Model → Supervised Fine-tuning
```

Real Example: Building a legal contract analyzer

1. Start with GPT-3
2. DAPT on legal texts (law books, court decisions)
3. TAPT on contracts specifically
4. Supervised fine-tune on (contract clause, interpretation) pairs

Each step adds value. It's like training a lawyer - first law school (DAPT), then contract law specifically (TAPT), then your firm's specific style (supervised).

---

5. Instruction Tuning: Teaching LLMs to Follow Orders

This is where things get really interesting. Let me explain why instruction tuning changed the game.

The Problem with Base Models

A base model (just pre-trained) is like a brilliant person who doesn't know how to be helpful. Watch this:

User: "What's the capital of France?"

Base Model: "? The capital of France is Paris. Paris is known for the Eiffel Tower, which was built in..."

It continues your text rather than answering! Why? Because it was trained to predict next tokens, not to be an assistant.

Enter Instruction Tuning

The Idea: Fine-tune the model on thousands of (instruction, response) pairs so it learns the general skill of following instructions.

```
Training Data:
"Summarize this text: [long text]" → "Here's a summary: [concise summary]"
"Translate to French: Hello" → "Bonjour"
"Write a poem about cats" → "Whiskers twitch in moonlight..."
"Explain quantum physics simply" → "Imagine particles as..."
```

Why This Works

Here's the beautiful insight: By training on DIVERSE instructions, the model learns the META-SKILL of following instructions, not just memorizing specific ones.

It's like teaching someone to cook by showing them 1000 different recipes. They don't just memorize recipes; they learn the general principles of cooking.

The Data Collection Strategies

Approach 1: Human-Written (Expensive but High Quality)

```
instructions = [
    ("Extract key points from this text: [text]", "Key points:\n1. ...\n2. ..."),
    ("Write Python code to sort a list", "def sort_list(lst):\n    return sorted(lst)"),
```

```
    ("Explain like I'm 5: gravity", "Imagine Earth is a big magnet...")
]
```

Approach 2: Model-Generated (Cheaper but Needs Care)

```
# Use GPT-4 to generate training data
prompt = "Generate an instruction and a high-quality response"
synthetic_data = gpt4.generate(prompt, n=10000)
# But be careful - check quality and licenses!
```

Key Examples

Google's FLAN: Instruction-tuned on 62 NLP tasks → Zero-shot performance skyrocketed

T0 (BigScience): Similar idea, focused on making models that follow prompts without examples

Stanford Alpaca: Used GPT-3.5 to generate 52K instructions, fine-tuned LLaMA → Surprisingly good assistant

The Technical Details

```
def instruction_tuning(model, instruction_data):
    for instruction, response in instruction_data:
        # Format as conversation
        input_text = f"Human: {instruction}\nAssistant:"
        target_text = response

        # Standard supervised loss
        loss = model.compute_loss(input_text, target_text)
        model.backward(loss)
```

The Magic

After instruction tuning, the model becomes actually useful:

User: "What's the capital of France?"

Instruction-Tuned Model: "The capital of France is Paris."

User: "Write a haiku about programming"

Instruction-Tuned Model:
"Bugs hide in the code

Coffee fuels the debugging

Sunrise, it compiles"

It just... works! The model understands it should be helpful, concise, and responsive.

---

6. RLHF: Aligning Models with Human Preferences

Now we get to the secret sauce of ChatGPT. RLHF is complex, but I'll break it down step by step.

The Limitation of Supervised Fine-Tuning

Here's the problem: Some qualities are hard to define in training data:

- What makes an explanation "clear"?
- What makes a joke "funny"?
- What makes an answer "helpful but not harmful"?

You can't easily create labeled data for these subjective qualities. Enter RLHF.

The RLHF Pipeline

RLHF has three stages. Let me walk through each:

Stage 1: Supervised Fine-Tuning (SFT)

First, create a decent baseline model:

```
# Start with base model
model = load_base_model()
```

```
# Fine-tune on high-quality demonstrations
demos = [
    ("Explain gravity", "Gravity is a force that attracts objects..."),
    ("Write a joke", "Why did the chicken cross the road? To get to the other sid
e!"),
]
sft_model = supervised_finetune(model, demos)
```

This gives us a model that's already pretty good.

Stage 2: Train a Reward Model

This is the clever part. Instead of asking humans to write perfect responses, ask them to COMPARE responses:

```
# Generate multiple responses
prompt = "Explain machine learning"
response_a = "ML is when computers learn patterns from data..."
response_b = "Machine learning is a subset of AI that... *technical jargon*"

# Human feedback
human_preference = "A > B"  # A is better (clearer, more helpful)

# Train reward model
reward_training_data = [
    (prompt, response_a, response_b, "A > B"),
    # ... thousands more
]

reward_model = train_reward_model(reward_training_data)
# Now reward_model(prompt, response) → scalar score
```

The reward model learns to predict human preferences!

Stage 3: Optimize Policy with RL (PPO)

Now the magic: use reinforcement learning to optimize the model to maximize reward:

```
def rlhf_training_loop(policy_model, reward_model, prompts):
    for prompt in prompts:
        # Generate response
        response = policy_model.generate(prompt)

        # Get reward
        reward = reward_model.score(prompt, response)

        # Important: Add KL penalty to prevent model from going crazy
        kl_penalty = kl_divergence(policy_model, original_model)
        final_reward = reward - lambda * kl_penalty

        # Update model using PPO
        update_with_ppo(policy_model, final_reward)
```

The KL Penalty: Why It Matters

Without the KL penalty, something weird happens. The model finds ways to hack the reward model:

```
Prompt: "Write a poem"
Without KL penalty: "POEM POEM POEM EXCELLENT BEAUTIFUL AMAZING P
OEM!!!"
With KL penalty: "Roses are red, violets are blue..."
```

The KL penalty keeps the model from departing too far from sensible language.

Why RLHF Works So Well

1. **Captures Nuance**: Humans are better at comparing than creating from scratch
2. **Scalable**: Can improve continuously with more feedback
3. **Handles Subjectivity**: What's "helpful" or "harmless" emerges from preferences

The Challenges

1. **Reward Hacking**: Models find unexpected ways to maximize reward
2. **Expensive**: Need lots of human feedback
3. **Complex**: Three models (policy, reward, reference), RL is tricky

Recent Developments

DPO (Direct Preference Optimization): Skip the reward model! Directly optimize from preferences:

```
# Instead of: Preferences → Reward Model → RL
# Just do: Preferences → Direct Policy Update
loss = -log(P(preferred_response)) + log(P(dispreferred_response))
```

Simpler, often works just as well!

---

7. Parameter-Efficient Fine-Tuning (PEFT)

This is where we get clever about resources. Full fine-tuning of a 70B parameter model requires multiple GPUs and lots of memory. But what if I told you we could get 95% of the performance by training less than 1% of the parameters?

The Core Insight

Large models are over-parameterized. The changes needed for a new task often lie in a much lower-dimensional space than the full model. We can exploit this!

Method 1: Prompt Tuning / Prefix Tuning

The Idea: Don't change the model at all. Instead, learn a "soft prompt" - a sequence of continuous vectors that guide the model.

```
# Traditional prompting (discrete tokens)
prompt = "Translate to French:"
output = model(prompt + input_text)

# Soft prompting (continuous vectors)
```

```
soft_prompt = nn.Parameter(torch.randn(20, embedding_dim))  # 20 learnable vectors
output = model(concat(soft_prompt, embed(input_text)))
```

Intuition: It's like learning the perfect way to ask the model to do your task.

Prefix Tuning: Even more powerful - add learnable vectors at EVERY layer:

```
class PrefixTuning:
    def __init__(self, num_prefix_tokens, num_layers):
        # Learn prefixes for keys and values at each layer
        self.prefixes = nn.ParameterList([
            nn.Parameter(torch.randn(num_prefix_tokens, hidden_dim))
            for _ in range(num_layers * 2)  # keys and values
        ])

    def forward(self, hidden_states, layer_idx):
        # Prepend learned prefix to keys/values
        prefix_key = self.prefixes[layer_idx * 2]
        prefix_value = self.prefixes[layer_idx * 2 + 1]
        # Model attends to these learned vectors
```

Results: Can match full fine-tuning performance while training <0.1% of parameters!

Method 2: Adapter Layers

The Idea: Insert small neural networks into the model that learn task-specific adjustments.

```
class Adapter(nn.Module):
    def __init__(self, hidden_dim, bottleneck_dim):
        super().__init__()
        # Down-project to bottleneck
        self.down = nn.Linear(hidden_dim, bottleneck_dim)  # e.g., 4096 → 64
        self.activation = nn.ReLU()
        # Up-project back
```

```
        self.up = nn.Linear(bottleneck_dim, hidden_dim)    # e.g., 64 → 4096

    def forward(self, x):
        return x + self.up(self.activation(self.down(x)))  # Residual connection

# Insert into transformer
class AdaptedTransformerBlock(nn.Module):
    def __init__(self, original_block):
        super().__init__()
        self.original_block = original_block
        self.adapter = Adapter(hidden_dim=4096, bottleneck_dim=64)

    def forward(self, x):
        x = self.original_block.attention(x)
        x = self.adapter(x)  # Add adapter after attention
        x = self.original_block.ffn(x)
        x = self.adapter(x)  # Add adapter after FFN
        return x
```

Why It Works:

- Original model preserves general knowledge
- Adapters learn task-specific tweaks
- Bottleneck forces efficient representation

The Numbers:

- BERT: 110M params → 3.6% trainable with adapters → same performance!
- Can have different adapters for different tasks, swap them out

Method 3: LoRA (Low-Rank Adaptation)

This is my favorite - it's elegant and powerful.

The Math Insight: Fine-tuning updates to weight matrices are often low-rank. Instead of updating $W \rightarrow W + \Delta W$, we say $\Delta W = BA$ where B and A are small.

```
class LoRA(nn.Module):
    def __init__(self, original_linear, rank=16):
        super().__init__()
        # Original weight frozen
        self.original_linear = original_linear
        in_dim, out_dim = original_linear.weight.shape

        # Low-rank decomposition
        self.A = nn.Parameter(torch.randn(in_dim, rank) / rank)
        self.B = nn.Parameter(torch.zeros(rank, out_dim))
        self.scaling = 0.01

    def forward(self, x):
        # Original forward
        orig_out = self.original_linear(x)
        # Low-rank update
        lora_out = x @ self.A @ self.B * self.scaling
        return orig_out + lora_out
```

The Beautiful Properties:

1. **Tiny Training**: GPT-3 (175B params) → LoRA trains 18M params (0.01%)!
2. **No Inference Overhead**: Can merge weights: W_new = W + scaling * B @ A
3. **Composable**: Different LoRAs for different tasks, can even combine them

Real Example:

```
# Fine-tuning 65B LLaMA on a single 24GB GPU!
model = load_llama_65b()  # Would need 260GB normally!

# Add LoRA to attention layers only
for layer in model.transformer.layers:
    layer.q_proj = LoRA(layer.q_proj, rank=8)
    layer.v_proj = LoRA(layer.v_proj, rank=8)
```

```
# Now only training ~0.1% of parameters
trainable_params = sum(p.numel() for p in model.parameters() if p.requires_gr
ad)
print(f"Training {trainable_params/1e6:.1f}M params instead of 65,000M!")
```

Method 4: QLoRA (Quantized LoRA)

The ultimate efficiency hack. Combines LoRA with 4-bit quantization:

```
# 1. Quantize base model to 4-bit (NF4 format)
model_int4 = quantize_to_4bit(model)  # 65B model now fits in 13GB!

# 2. Add LoRA adapters in 16-bit
add_lora_layers(model_int4, compute_dtype=torch.float16)

# 3. Train only LoRA parameters
# Result: Fine-tune 65B model on single consumer GPU!
```

The Magic:

- Base model in 4-bit (frozen)
- LoRA computations in 16-bit
- Achieves same results as full 16-bit fine-tuning!

Choosing a PEFT Method

Here's my decision tree:

```
Do you need to merge the fine-tuned model?
├─ Yes → Use LoRA (can merge BA into W)
└─ No → Consider all options
        │
        ├─ Minimal parameters? → Prompt/Prefix tuning
        ├─ Multiple tasks? → Adapters (easy to swap)
        └─ Best performance? → LoRA usually wins
```

Combining Methods: You can mix these!

- LoRA + Prefix tuning
- Adapters + LoRA at different layers
- The field is still exploring

---

8. Low-Memory Training Techniques

Now let's get into the engineering tricks that make fine-tuning possible on realistic hardware. These techniques are what let you fine-tune a 30B model on a single GPU instead of needing a cluster.

Understanding Memory Usage

First, let's understand where memory goes during training:

```
# Memory breakdown for a 7B parameter model
Model weights:          7B * 4 bytes = 28GB (FP32)
Optimizer states (Adam): 7B * 8 bytes = 56GB (2 momentum terms)
Gradients:              7B * 4 bytes = 28GB
Activations:            Variable, can be 50GB+
-----------------------------------------
Total:                  160GB+ for training!
```

Yikes! That's why we need these techniques.

Technique 1: Mixed Precision Training

The Idea: Use 16-bit (or even 8-bit) instead of 32-bit floats.

```
# Traditional training
model = model.float()  # 32-bit
loss = compute_loss(model(input))

# Mixed precision
model = model.half()  # 16-bit main weights
# But keep master weights in FP32 for stability
```

```
optimizer = torch.optim.AdamW(model.parameters())

with autocast():  # Automatic mixed precision
    output = model(input)  # Compute in FP16
    loss = loss_fn(output, target)  # Some ops stay in FP32

scaler.scale(loss).backward()  # Gradient scaling for numerical stability
scaler.step(optimizer)
```

Memory Savings: 50% on model and activations!

Going Further - 8-bit and 4-bit:

```
# 8-bit Adam (from bitsandbytes)
import bitsandbytes as bnb
optimizer = bnb.optim.Adam8bit(model.parameters())  # 8-bit optimizer state
s!

# QLoRA: 4-bit base model
model = load_model_in_4bit(
    "meta-llama/Llama-2-70b",
    bnb_4bit_compute_dtype=torch.float16,
    bnb_4bit_use_double_quant=True,  # Quantize the quantization!
)
```

Technique 2: Gradient Checkpointing

The Problem: Activations (intermediate values) eat memory:

```
Layer1 → [activation1] → Layer2 → [activation2] → ... → LayerN
        ^-- Stored for backward pass
```

The Solution: Don't store them! Recompute during backward:

```
# Without checkpointing
def forward(x):
```

```
    x1 = layer1(x)    # Store x1
    x2 = layer2(x1)   # Store x2
    x3 = layer3(x2)   # Store x3
    return x3

# With checkpointing
from torch.utils.checkpoint import checkpoint

def forward(x):
    # Only store x and x3, recompute x1, x2 during backward
    x3 = checkpoint(lambda x: layer3(layer2(layer1(x))), x)
    return x3
```

Trade-off:

- Memory: ~50% reduction
- Speed: ~20-30% slower (recomputation)
- Worth it? YES, if it's the difference between OOM and training!

Technique 3: Gradient Accumulation

The Idea: Can't fit batch size 32? Use batch size 4 and accumulate for 8 steps!

```
accumulation_steps = 8
optimizer.zero_grad()

for i, batch in enumerate(dataloader):
    outputs = model(batch)
    loss = criterion(outputs, targets)
    loss = loss / accumulation_steps  # Normalize
    loss.backward()

    if (i + 1) % accumulation_steps == 0:
        optimizer.step()
        optimizer.zero_grad()
```

Effect: Simulates larger batch size without memory cost.

Technique 4: Smart Optimizer Choices

Problem: Adam stores 2 extra values per parameter!

Solution 1 - Adafactor:

```
# Instead of Adam storing full second moment matrix
# Adafactor factorizes it into row and column vectors
from transformers import Adafactor

optimizer = Adafactor(
    model.parameters(),
    scale_parameter=True,
    relative_step=True,
    warmup_init=True,
)
# Memory: ~2x parameters instead of 3x
```

Solution 2 - Distributed Optimizer States (ZeRO):

```
# Each GPU stores only part of optimizer states
from deepspeed import zero

model_engine = deepspeed.init_distributed(
    model=model,
    optimizer=optimizer,
    config={
        "zero_optimization": {
            "stage": 2,  # Partition optimizer states and gradients
            # stage 3 would also partition parameters
        }
    }
)
```

Technique 5: The Full Memory-Saving Stack

Here's how to combine everything for maximum efficiency:

```
def setup_memory_efficient_training(model_name, task_data):
    # 1. Load model in 4-bit
    model = AutoModelForCausalLM.from_pretrained(
        model_name,
        load_in_4bit=True,
        bnb_4bit_compute_dtype=torch.float16,
        bnb_4bit_use_double_quant=True,
    )

    # 2. Add LoRA adapters
    peft_config = LoRAConfig(
        r=8,  # rank
        lora_alpha=32,
        target_modules=["q_proj", "v_proj"],
        lora_dropout=0.05,
    )
    model = get_peft_model(model, peft_config)

    # 3. Enable gradient checkpointing
    model.gradient_checkpointing_enable()

    # 4. Use 8-bit Adam
    optimizer = bnb.optim.Adam8bit(model.parameters())

    # 5. Mixed precision training
    scaler = torch.cuda.amp.GradScaler()

    return model, optimizer, scaler

# Now you can fine-tune a 65B model on a single 24GB GPU!
```

Real-World Example: QLoRA on LLaMA-65B

The QLoRA paper showed this incredible result:

```
Hardware: 1× 48GB GPU
Model: LLaMA-65B (260GB in FP32!)
Technique: 4-bit base + LoRA + gradient checkpointing
Result: Guanaco-65B matching ChatGPT performance!
Training time: 24 hours
Memory used: ~40GB
```

This was impossible before these techniques!

---

9. Putting It All Together: A Practical Framework

Now let's synthesize everything into a practical decision framework. I'll walk you through how to approach a real fine-tuning project.

Step 1: Define Your Goal Precisely

Ask yourself:

- What specific behavior do I want?
- What does success look like?
- What resources do I have?

Example Goals:

- "Make model answer medical questions accurately" → Need medical Q&A pairs
- "Make model write in our company's style" → Need company documents
- "Make model refuse harmful requests" → Need preference data

Step 2: Choose Your Base Model

Consider:

- **Size vs Quality**: Bigger isn't always better if you can't train it
- **License**: Can you use it commercially?
- **Architecture**: Some models are more efficient for fine-tuning

```
# Decision matrix
if gpu_memory < 16GB:
    use_model("meta-llama/Llama-2-7b")  # Or smaller
elif gpu_memory < 24GB:
    use_model("meta-llama/Llama-2-13b")
elif gpu_memory < 48GB:
    use_model("meta-llama/Llama-2-30b") + QLoRA
else:
    use_model("meta-llama/Llama-2-70b") + QLoRA
```

Step 3: Prepare Your Data

This is CRUCIAL. Good data > fancy techniques.

For Supervised Fine-tuning:

```
# Quality checklist
def validate_training_example(input_text, output_text):
    checks = [
        len(output_text) > 0,  # Not empty
        is_relevant(input_text, output_text),  # Actually answers
        is_high_quality(output_text),  # Good grammar, accurate
        not is_harmful(output_text),  # Safe
    ]
    return all(checks)

# Format for chat models
def format_for_chat(instruction, response):
    return f"""<|user|>
{instruction}
<|assistant|>
{response}
<|endoftext|>"""
```

For Domain Adaptation:

```
# Clean your corpus
def prepare_domain_corpus(raw_texts):
    cleaned = []
    for text in raw_texts:
        # Remove boilerplate, headers, footers
        text = clean_text(text)
        # Ensure quality
        if len(text) > 100 and detect_language(text) == "en":
            cleaned.append(text)
    return cleaned
```

Step 4: Design Your Training Pipeline

Here's my template that combines everything we've learned:

```
def create_finetuning_pipeline(
    model_name,
    training_data,
    output_dir,
    training_approach="qlora"  # or "full", "lora", "prefix"
):
    # 1. Load model with memory optimizations
    if training_approach == "qlora":
        model = AutoModelForCausalLM.from_pretrained(
            model_name,
            load_in_4bit=True,
            bnb_4bit_compute_dtype=torch.float16,
            bnb_4bit_quant_type="nf4",
            bnb_4bit_use_double_quant=True,
        )
    else:
        model = AutoModelForCausalLM.from_pretrained(
            model_name,
            torch_dtype=torch.float16
        )
```

```
# 2. Add parameter-efficient layers if needed
if training_approach in ["lora", "qlora"]:
    peft_config = LoraConfig(
        r=16,  # Higher rank for complex tasks
        lora_alpha=32,
        target_modules=["q_proj", "k_proj", "v_proj", "o_proj"],
        lora_dropout=0.05,
        bias="none",
        task_type="CAUSAL_LM",
    )
    model = get_peft_model(model, peft_config)

# 3. Setup training arguments
training_args = TrainingArguments(
    output_dir=output_dir,
    num_train_epochs=3,
    per_device_train_batch_size=4,
    gradient_accumulation_steps=4,  # Effective batch = 16
    gradient_checkpointing=True,
    optim="adamw_8bit",  # or "adafactor"
    logging_steps=10,
    save_strategy="epoch",
    learning_rate=2e-4,
    warmup_steps=100,
    fp16=True,
    push_to_hub=False,
)

# 4. Create trainer
trainer = SFTTrainer(  # For supervised fine-tuning
    model=model,
    train_dataset=training_data,
    tokenizer=tokenizer,
    args=training_args,
    max_seq_length=2048,
```

```
        dataset_text_field="text",  # Field containing formatted examples
    )

    return trainer
```

Step 5: Multi-Stage Training Strategy

For best results, combine techniques:

```
def advanced_finetuning_pipeline(base_model, domain_corpus, task_data):
    """
    Three-stage training for maximum performance
    """

    # Stage 1: Domain adaptation (unsupervised)
    if domain_corpus:
        print("Stage 1: Domain-Adaptive Pre-training")
        model = continue_pretraining(
            base_model,
            domain_corpus,
            learning_rate=1e-4,
            steps=10000
        )
    else:
        model = base_model

    # Stage 2: Instruction tuning (if not already done)
    if not model.is_instruction_tuned:
        print("Stage 2: Instruction Tuning")
        instruction_data = load_general_instructions()
        model = instruction_tune(
            model,
            instruction_data,
            epochs=1
        )
```

```
    # Stage 3: Task-specific fine-tuning with LoRA
    print("Stage 3: Task-Specific Fine-tuning")
    model = add_lora_adapters(model)
    model = supervised_finetune(
        model,
        task_data,
        epochs=3
    )

    # Optional Stage 4: RLHF or DPO for alignment
    if has_preference_data:
        print("Stage 4: Preference Alignment")
        model = dpo_training(
            model,
            preference_pairs,
            beta=0.1  # KL penalty weight
        )

    return model
```

Step 6: Evaluation and Iteration

Don't just train blindly. Evaluate carefully:

```
def evaluate_finetuned_model(model, test_set):
    metrics = {
        'task_performance': evaluate_on_task(model, test_set),
        'general_capability': evaluate_on_benchmarks(model),  # MMLU, etc.
        'safety': evaluate_safety(model),
        'efficiency': measure_inference_speed(model),
    }

    # Check for common issues
    issues = []
    if metrics['general_capability'] < baseline * 0.9:
        issues.append("Catastrophic forgetting detected")
```

```
    if metrics['safety'] < threshold:
        issues.append("Safety degradation")

    return metrics, issues
```

Common Pitfalls and Solutions

1. Overfitting on Small Datasets

```
# Solution: Regularization techniques
config = LoraConfig(
    r=8,  # Lower rank
    lora_dropout=0.1,  # Higher dropout
)
# Also: Mix in some general data during training
```

2. Catastrophic Forgetting

```
# Solution: Elastic Weight Consolidation or simple mixing
def create_mixed_dataset(task_data, general_data, mix_ratio=0.9):
    # 90% task data, 10% general data
    return combine_datasets(task_data, general_data, mix_ratio)
```

3. Slow Training

```
# Solution: Profile and optimize
with torch.profiler.profile() as prof:
    trainer.train()
print(prof.key_averages())
# Often the issue is data loading, not model training!
```

A Complete Example: Building a Code Assistant

Let me walk through a real example:

```
# Goal: Fine-tune model to write better Python code

# 1. Start with a code-trained base
base_model = "codellama/CodeLlama-7b-Python"

# 2. Collect data
code_corpus = collect_github_python_files(stars=">1000")
code_instructions = create_coding_tasks()  # "Write function to...", etc.

# 3. Three-stage training
# Stage 1: Adapt to your coding style
model = continue_pretraining_on_code(base_model, code_corpus)

# Stage 2: Instruction tuning for code tasks
model = instruction_tune(model, code_instructions)

# Stage 3: LoRA fine-tuning on your specific codebase
company_code_data = prepare_company_code_examples()
model = lora_finetune(model, company_code_data, r=16)

# 4. Evaluate
test_tasks = ["Write a REST API endpoint", "Refactor this function", ...]
results = evaluate_code_generation(model, test_tasks)
```

The Decision Tree

Here's how I decide on an approach:

```
Start: What's your goal?
│
├─ Domain Adaptation (medical, legal, etc.)
│   └─ Unsupervised fine-tuning on domain text
│       └─ Then supervised on specific tasks
│
├─ Task-Specific (Q&A, summarization, etc.)
```

```
│   ├── Have lots of labeled data?
│   │   └── Supervised fine-tuning
│   └── Limited labeled data?
│       └── Few-shot prompting or instruction tuning
│
├── Behavioral Alignment (helpful, harmless)
│   └── RLHF or DPO with human preferences
│
└── Resource Constrained?
    ├── Single GPU? → QLoRA
    ├── Multiple small GPUs? → DeepSpeed + LoRA
    └── CPU only? → Quantization + tiny models
```

Final Thoughts

Fine-tuning is both an art and a science. The techniques I've shown you are powerful, but remember:

1. **Data quality beats everything**: 1K excellent examples > 100K mediocre ones
2. **Start simple**: Try LoRA before full fine-tuning
3. **Measure everything**: Track loss, but also real task performance
4. **Iterate quickly**: Small experiments to find what works
5. **Combine techniques**: DAPT + TAPT + SFT + RLHF = SOTA

The field is moving fast. New techniques appear monthly. But the fundamentals remain:

- Understand your task
- Prepare good data
- Choose efficient methods
- Evaluate thoroughly
- Iterate based on results

With these tools, you can take any pre-trained model and shape it into exactly what you need. The barrier to entry has never been lower - you can fine-tune

powerful models on consumer hardware that would have required millions in compute just years ago.

Appendix: Quick Reference

Memory Requirements Cheat Sheet

```
Model Size | FP32  | FP16  | INT8  | INT4 (QLoRA)
-----------|-------|-------|-------|-------------
7B         | 28GB  | 14GB  | 7GB   | 3.5GB
13B        | 52GB  | 26GB  | 13GB  | 6.5GB
30B        | 120GB | 60GB  | 30GB  | 15GB
65B        | 260GB | 130GB | 65GB  | 32.5GB
```

Training Time Estimates (Single A100 80GB)

```
Method          | 7B Model | 13B Model | 30B Model
----------------|----------|-----------|----------
Full Fine-tune  | 2 days   | 4 days    | OOM
LoRA            | 3 hours  | 6 hours   | 12 hours
QLoRA           | 4 hours  | 8 hours   | 16 hours
Prefix Tuning   | 2 hours  | 4 hours   | 8 hours
```

Key Libraries

```
# Hugging Face ecosystem
from transformers import AutoModelForCausalLM, Trainer
from peft import LoraConfig, get_peft_model
from trl import SFTTrainer, DPOTrainer

# Quantization
import bitsandbytes as bnb

# Distributed training
```

```
import deepspeed

# Experiment tracking
import wandb
```