Instruction Fine-tuning: Aligning Models with Human Instructions

Prof. Rongyu Lin

Lecture 8

Quinnipiac University
School of Computing and Engineering

Reading: Raschka Ch. 7 (pp. 239-289)
"Instruction Fine-tuning and Alignment"

Learning Objectives

By the end of this lecture, you will be able to:

- Understand the motivation and importance of instruction fine-tuning
- Distinguish between pre-training, fine-tuning, and instruction fine-tuning
- Prepare and format datasets for instruction-based training
- Implement complete instruction fine-tuning pipelines
- Apply parameter-efficient methods like LoRA and QLoRA
- Evaluate and deploy instruction-tuned models effectively

Transforming foundation models into helpful AI assistants

Lecture Outline

Part I

Foundations of Instruction Fine-tuning

Understanding the alignment problem and training pipeline

What is Instruction Fine-tuning?

Definition:

- Process of training LLMs to follow human instructions
- Transforms general language models into helpful assistants
- Bridges gap between pre-training and deployment

Key Characteristics:

- Supervised learning on instruction-response pairs
- Teaches desired behavior patterns
- Improves alignment with user expectations

Examples:

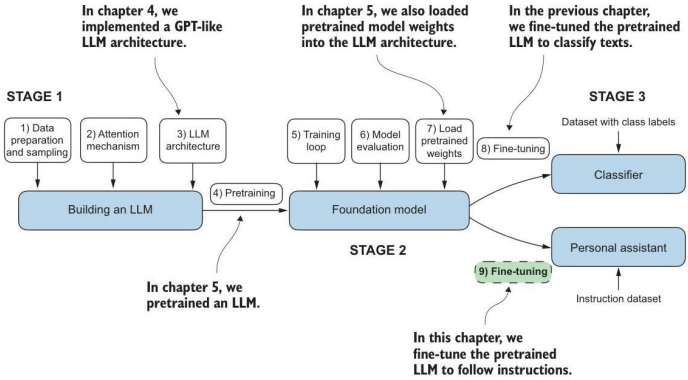
Before Fine-tuning

```
Input:  "Write a poem about AI"
Output:  "about AI is a topic..."
```

After Fine-tuning

```
Input:  "Write a poem about AI"
Output:  "Digital minds awake,
Silicon dreams take flight..."
```

The Complete LLM Training Pipeline



Three-Stage Process:

1. **Pre-training:** Learn language patterns
2. **Instruction Fine-tuning:** Align with instructions
3. **Preference Alignment:** Human feedback refinement

Pre-training vs Fine-tuning vs Instruction Fine-tuning

Pre-training

- Unsupervised learning
- Massive text corpora
- Next-token prediction
- General language understanding
- Months of training
- Thousands of GPUs

Fine-tuning

- Task-specific adaptation
- Domain-focused data
- Classification/regression
- Specialized capabilities
- Hours to days
- Single GPU possible

Instruction Fine-tuning

- Behavioral alignment
- Instruction-response pairs
- Following commands
- General helpfulness
- Days to weeks
- Multiple GPUs typical

Each stage builds upon the previous, creating increasingly capable models

The Alignment Problem

Pre-trained Models:

- Complete text probabilistically
- No concept of helpfulness
- May generate harmful content
- Unpredictable behavior

Challenges:

- Hallucination and fabrication
- Bias amplification
- Unsafe or unethical outputs
- Poor instruction following

Alignment Goals:

- Helpful: Assist users effectively
- Harmless: Avoid dangerous outputs
- Honest: Acknowledge limitations

Solution Approaches:

- Instruction fine-tuning (this lecture)
- RLHF (Reinforcement Learning)
- Constitutional AI
- Direct Preference Optimization

Industry Success Stories

OpenAI GPT Series:

- GPT-3 $\rightarrow$ InstructGPT $\rightarrow$ ChatGPT
- 100x smaller but preferred outputs
- Instruction tuning + RLHF

Google's Models:

- LaMDA $\rightarrow$ Bard $\rightarrow$ Gemini
- Multi-modal instruction following
- Safety-first alignment approach

Open Source Models:

- Alpaca (Stanford): 52K instructions
- Vicuna (LMSYS): ShareGPT conversations
- Dolly (Databricks): Employee-generated
- WizardLM: Complex instruction evolution

Impact:

- Democratized AI assistants
- Enabled custom deployments
- Accelerated research progress

Instruction-Response Format

```
Instruction:
Write a Python function to check
if a number is prime.

Response:
def is_prime(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True
```

Template Components:

- Instruction: User's request or query
- Input: Optional context or data
- Response: Model's desired output

Common Formats:

- Alpaca: Instruction-Input-Response
- ChatML: Role-based conversations
- Llama2: Special tokens [INST]
- Custom: Domain-specific templates

Consistent formatting is crucial for training

Types of Instruction Datasets

Human-Generated

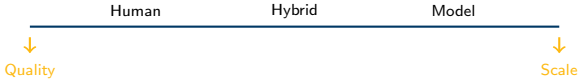
- High quality
- Expensive to create
- Limited scale
- Examples:
  - Dolly-15k
  - OpenAssistant
  - Super-NaturalInstructions

Model-Generated

- Scalable approach
- Quality varies
- Potential biases
- Examples:
  - Alpaca (GPT-3)
  - Vicuna (ShareGPT)
  - WizardLM

Hybrid Approaches

- Model generation + human review
- Instruction evolution
- Self-instruct methods
- Examples:
  - Orca (explanation tuning)
  - Constitutional AI
  - LIMA (Less Is More)



Evaluation Metrics for Instruction-Tuned Models

Automatic Metrics:

- **Perplexity:** Language modeling quality
- **BLEU/ROUGE:** N-gram overlap
- **BERTScore:** Semantic similarity
- **Task-specific:** Accuracy on benchmarks

Benchmarks:

- MMLU: Multi-task knowledge
- HumanEval: Code generation
- TruthfulQA: Factual accuracy
- MT-Bench: Multi-turn conversations

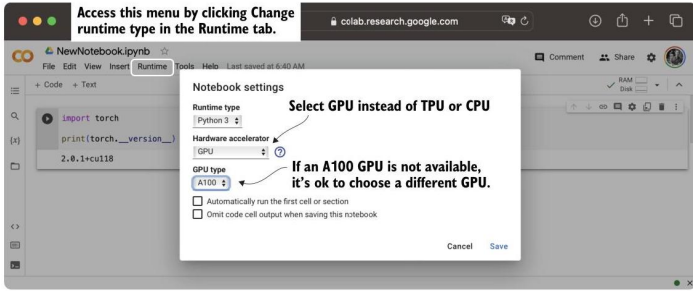
Human Evaluation:

- Helpfulness ratings
- Safety assessments
- Preference comparisons (A/B tests)
- Red-teaming for adversarial inputs

LLM-as-Judge:

- GPT-4 evaluation
- Alpaca Eval leaderboard
- Arena (pairwise comparisons)
- Self-evaluation techniques

Hardware Requirements and Setup



Minimum Requirements:

- GPU: 16GB VRAM (RTX 3090/4090)
- RAM: 32GB system memory
- Storage: 100GB for models/data

Optimization Techniques:

- Gradient checkpointing
- Mixed precision (FP16/BF16)
- Parameter-efficient methods (LoRA)

Current Challenges in Instruction Fine-tuning

Technical Challenges:

- Catastrophic forgetting
- Distribution shift
- Overfitting to instruction format
- Computational costs

Data Challenges:

- Quality vs quantity tradeoff
- Instruction diversity
- Multilingual coverage
- Domain expertise gaps

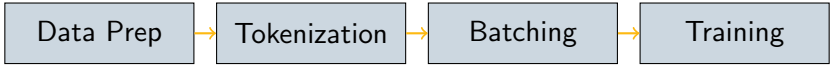
Ethical Considerations:

- Bias amplification
- Safety guarantees
- Misuse potential
- Transparency requirements

Research Directions:

- Constitutional AI methods
- Self-supervised alignment
- Continual learning approaches
- Efficient fine-tuning at scale

Looking Ahead: Implementation Pipeline



Next: Deep dive into dataset preparation and formatting

- Loading and exploring the Alpaca dataset
- Creating instruction templates
- Implementing tokenization pipelines
- Building efficient data loaders

Part II

Dataset Preparation and Formatting

Building the foundation for instruction fine-tuning

The Alpaca Dataset

Overview:

- 52,000 instruction-following examples
- Generated using GPT-3.5 (text-davinci-003)
- Based on self-instruct methodology
- Cost: $500 to generate

Dataset Structure:

- **instruction:** The task description
- **input:** Optional context (can be empty)
- **output:** Expected response

```
# Example entry
{
  "instruction": "Translate to French",
  "input": "Hello, how are you?",
  "output": "Bonjour, comment allez-vous?"
}

# Another example
{
  "instruction": "Write a haiku about AI",
  "input": "",
  "output": "Silicon minds wake\nData flows like endless
      streams\nFuture writes itself"
}
```

Simple format, powerful results

Loading the Alpaca Dataset

```python
import json
from torch.utils.data import Dataset
import pandas as pd

# Load Alpaca dataset
def load_alpaca_dataset(path):
    with open(path, 'r') as f:
        data = json.load(f)
    return data

# Analyze dataset
data = load_alpaca_dataset('alpaca_data.json')
print(f"Total examples: {len(data)}")

# Convert to DataFrame for analysis
df = pd.DataFrame(data)
print(f"Unique instructions: {df['instruction'].nunique()}")
print(f"Examples with input: {(df['input'] != '').sum()}")

# Sample distribution
inst_lengths = df['instruction'].str.len()
print(f"Avg instruction length: {inst_lengths.mean():.0f}")
```

Dataset Statistics:

- 52,002 total examples
- 39,825 unique instructions
- 17,858 with input context
- Avg length: 85 characters

Instruction Categories:

- Generation: 28%
- Classification: 18%
- Q&A: 22%
- Rewriting: 15%
- Translation: 8%
- Other: 9%

Diverse tasks ensure generalization

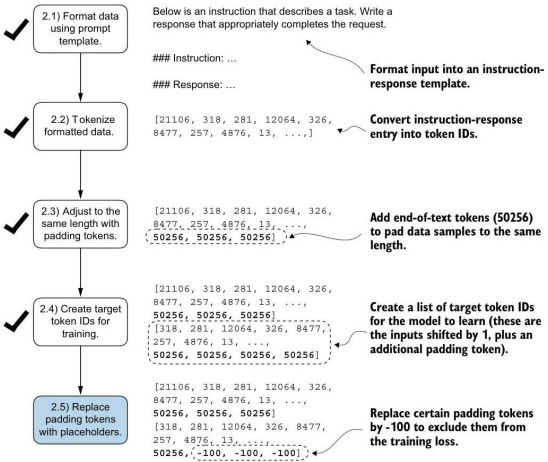
Data Preprocessing Pipeline

Five-Step Process:

1. **Data Loading:** Read JSON/CSV files into memory
2. **Format Conversion:** Apply instruction template formatting
3. **Tokenization:** Convert text to token IDs
4. **Padding/Truncation:** Ensure uniform sequence lengths
5. **Target Creation:** Generate labels for training

Key Considerations:

- Memory efficiency
- Batch processing
- Data shuffling
- Special tokens handling



Instruction Template Formatting

```python
def format_instruction(entry):
    """Format entry with Alpaca template"""
    instruction = entry['instruction']
    input_text = entry['input']

    if input_text:
        prompt = f"""Below is an instruction that describes a task,
      paired with an input that provides further context. Write a
      response that appropriately completes the request.

        Instruction:
        {instruction}

        Input:
        {input_text}

        Response:
        """
    else:
        prompt = f"""Below is an instruction that describes a task. Write
       a response that appropriately completes the request.

        Instruction:
        {instruction}

        Response:
        """
    return prompt + entry['output']
```

Template Components:

- System prompt for context
- Clear section markers
- Consistent formatting
- Flexibility for input presence

Best Practices:

- Use consistent delimiters
- Preserve whitespace structure
- Include system instructions
- Handle edge cases gracefully

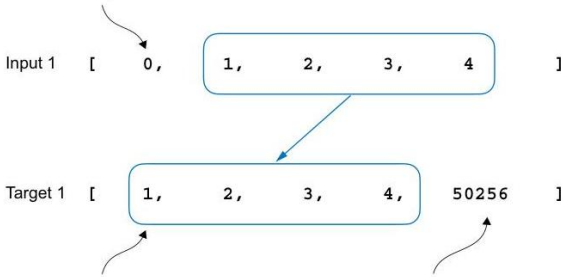
Output Example:

```
Below is an instruction...

Instruction:
Explain quantum computing

Response:
Quantum computing uses...
```

Tokenization for Instruction Fine-tuning



Input Processing:

- Convert text to token IDs
- Add special tokens (BOS/EOS)
- Maintain instruction boundaries

Target Creation:

- Shift tokens by one position
- Mask instruction tokens (optional)
- Preserve response for loss calculation

Implementing Tokenization

```python
from transformers import AutoTokenizer

# Initialize tokenizer
tokenizer = AutoTokenizer.from_pretrained("meta-llama/Llama-2-7b-hf")
tokenizer.pad_token = tokenizer.eos_token

def tokenize_example(example, max_length=512):
    """Tokenize instruction-response pair"""
    # Format the example
    text = format_instruction(example)

    # Tokenize with padding and truncation
    tokens = tokenizer(
        text,
        max_length=max_length,
        padding="max_length",
        truncation=True,
        return_tensors="pt")

     Create labels (shift by 1 for next-token prediction)
    labels = tokens["input_ids"].clone()
    labels[tokens["attention_mask"] == 0] = -100

    return {
        "input_ids": tokens["input_ids"].squeeze(),
        "attention_mask": tokens["attention_mask"].squeeze(),
        "labels": labels.squeeze()}
```

Key Parameters:

- `max_length`: Sequence limit
- `padding`: Strategy for short sequences
- `truncation`: Handle long sequences
- `return_tensors`: Output format

Label Masking:

- Use -100 for ignored tokens
- PyTorch ignores in loss calculation
- Can mask instruction portion
- Focus learning on responses

Proper tokenization is critical for training

Advanced: Instruction Masking

```python
def mask_instruction_tokens(tokens, response_start_idx):
    """Mask instruction portion, keep response"""
    labels = tokens["input_ids"].clone()

    # Find response delimiter position
    response_token = tokenizer.encode("Response:",
                                      add_special_tokens=False)

    # Find where response starts in sequence
    for i in range(len(labels) - len(response_token)):
        if labels[i:i+len(response_token)].tolist() == response_token:
            response_start_idx = i + len(response_token)
            break

    # Mask everything before response
    if response_start_idx > 0:
        labels[:response_start_idx] = -100

    # Also mask padding tokens
    labels[tokens["attention_mask"] == 0] = -100

    return labels

# Apply masking
tokens = tokenizer(formatted_text, ...)
labels = mask_instruction_tokens(tokens, response_idx)
```

Why Mask Instructions?

- Focus learning on responses
- Prevent overfitting to prompts
- Improve generation quality
- Reduce training noise

Masking Strategy:

```
[KEEP] Below is an instruction...
[MASK] Instruction:
[MASK] Write a poem
[MASK] Response:
[KEEP] Roses are red...
[KEEP] Violets are blue...
```

Optional but often improves results

Creating PyTorch Dataset

```python
import torch
from torch.utils.data import Dataset

class InstructionDataset(Dataset):
    def __init__(self, data, tokenizer, max_length=512):
        self.data = data
        self.tokenizer = tokenizer
        self.max_length = max_length

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        example = self.data[idx]
        text = format_instruction(example)
        encoding = self.tokenizer(
            text,
            max_length=self.max_length,
            padding="max_length",
            truncation=True,
            return_tensors="pt")

        labels = encoding["input_ids"].clone()
        labels[encoding["attention_mask"] == 0] = -100

        return {"input_ids": encoding["input_ids"].squeeze(),
            "attention_mask": encoding["attention_mask"].squeeze(),
            "labels": labels.squeeze()}
```

Dataset Methods:

- `__init__`: Initialize with data
- `__len__`: Return dataset size
- `__getitem__`: Get single example

Usage:

```python
# Create dataset
train_dataset = InstructionDataset(
    data=train_data,
    tokenizer=tokenizer,
    max_length=512)

# Check size
print(f"Dataset size: {len(train_dataset)}")

# Get sample
sample = train_dataset[0]
print(f"Input shape: {sample['input_ids'].shape}")
```

PyTorch Dataset enables efficient loading

Data Validation and Quality Checks

```python
def validate_dataset(dataset, tokenizer):
    """Perform quality checks on dataset"""
    issues = []

    for idx in range(min(100, len(dataset))):
        sample = dataset[idx]

        if sample['attention_mask'].sum() == 0:
            issues.append(f"Empty sequence at index {idx}")

        if sample['attention_mask'][-1] == 1:
            text = tokenizer.decode(sample['input_ids'])
            if not text.endswith(tokenizer.eos_token):
                issues.append(f"Truncated at index {idx}")

        valid_labels = sample['labels'][sample['labels'] != -100]
        if len(valid_labels) < 10:
            issues.append(f"Too few labels at index {idx}")

    # Report findings
    if issues:
        print(f"Found {len(issues)} potential issues:")
        for issue in issues[:5]:
            print(f"  - {issue}")
    else:
        print("Dataset validation passed!")

    return len(issues) == 0
```

Common Issues:

- Empty or very short responses
- Truncated sequences
- Encoding errors
- Missing special tokens
- Inconsistent formatting

Quality Metrics:

- Response length distribution
- Token vocabulary coverage
- Special token placement
- Label/input ratio

Remediation:

- Filter problematic examples
- Adjust max_length if needed
- Fix formatting issues
- Balance dataset if skewed

Data Splitting Strategy

```python
from sklearn.model_selection import train_test_split

def split_instruction_data(data, val_size=0.1, test_size=0.05):
    """Split data into train/val/test sets"""

    # First split: train+val vs test
    train_val, test = train_test_split(data, test_size=test_size,
        random_state=42,shuffle=True)

    # Second split: train vs val
    val_ratio = val_size / (1 - test_size)
    train, val = train_test_split(train_val,test_size=val_ratio,
        random_state=42,shuffle=True)

    print(f"Dataset splits:")
    print(f"  Train: {len(train):,} examples")
    print(f"  Val: {len(val):,} examples")
    print(f"  Test: {len(test):,} examples")

    return train, val, test

# Apply splitting
train_data, val_data, test_data = split_instruction_data(
    alpaca_data, val_size=0.1, test_size=0.05)
```

Split Ratios:

- Training: 85% (44,202 examples)
- Validation: 10% (5,200 examples)
- Test: 5% (2,600 examples)

Considerations:

- Maintain task distribution
- Avoid data leakage
- Stratified splitting if needed
- Keep similar examples together

Advanced Splitting:

- By instruction type
- By difficulty level
- By response length
- Cross-validation for small data

Data Augmentation Techniques

Instruction Paraphrasing:

- Rephrase instructions
- Maintain semantic meaning
- Increase diversity

Response Variations:

- Multiple valid responses
- Different writing styles
- Varying detail levels

Back-translation:

- Translate to another language
- Translate back to original
- Creates natural variations

```python
def augment_instruction(example):
    """Simple augmentation example"""
    templates = [
        "Please {task}",
        "Can you {task}",
        "I need you to {task}",
        "{task}",
        "Help me {task}"
    ]

    instruction = example['instruction']
    # Random template selection
    import random
    template = random.choice(templates)

    # Apply template
    if instruction.lower().startswith(('please', 'can you')):
        task = instruction
    else:
        task = instruction.lower()

    augmented = template.format(task=task)

    return {
        'instruction': augmented,
        'input': example['input'],
        'output': example['output']
    }
```

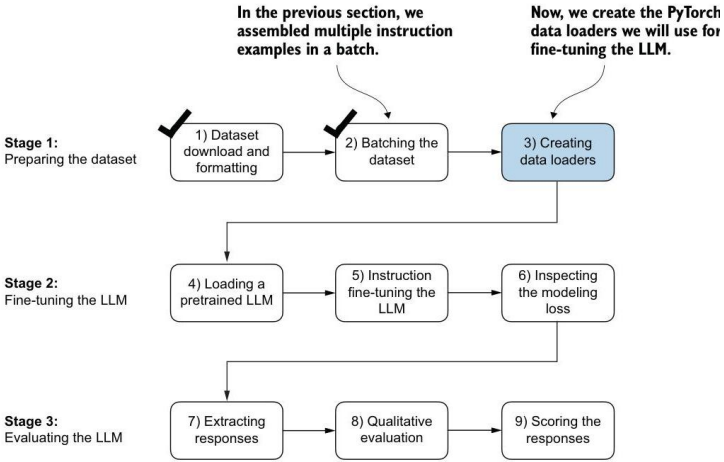
From Data to Batches

Next Steps:

- Create efficient data loaders for batching
- Handle variable-length sequences
- Optimize memory usage
- Prepare for distributed training

Key Concepts:

- Dynamic batching
- Padding strategies
- Sequence bucketing
- Memory pinning



Part III

Batching and Data Loading

Efficient data pipelines for model training

Why Batching Matters

Benefits of Batching:

- GPU utilization improvement
- Stable gradient estimates
- Faster training convergence
- Memory efficiency

Batch Size Considerations:

- GPU memory limits
- Model size constraints
- Gradient accumulation
- Learning rate scaling

Typical Batch Sizes:

- 7B model: 4-8 per GPU
- 13B model: 2-4 per GPU
- 30B model: 1-2 per GPU
- With gradient accumulation: effectively larger

Memory Formula:

$$\begin{aligned}\text{Memory} &= \text{Model} + \text{Gradients} \\ &\quad + \text{Optimizer} + \text{Activations} \\ &\approx 4 \times \text{Model Size}\end{aligned}$$

Creating Efficient DataLoaders

```python
from torch.utils.data import DataLoader
import torch

def create_dataloaders(train_dataset, val_dataset,
                       batch_size=4, num_workers=4):

    # Training dataloader with shuffling
    train_loader = DataLoader(
        train_dataset,
        batch_size=batch_size,
        shuffle=True,
        num_workers=num_workers,
        pin_memory=True,
        drop_last=True,   # Drop incomplete batches
        collate_fn=instruction_collate_fn)

    # Validation dataloader without shuffling
    val_loader = DataLoader(
        val_dataset,
        batch_size=batch_size,
        shuffle=False,
        num_workers=num_workers,
        pin_memory=True,
        drop_last=False,
        collate_fn=instruction_collate_fn)

    print(f"Train batches: {len(train_loader)}")
    print(f"Val batches: {len(val_loader)}")
```

DataLoader Parameters:

- `batch_size`: Samples per batch
- `shuffle`: Randomize order
- `num_workers`: Parallel loading
- `pin_memory`: GPU transfer optimization
- `drop_last`: Handle final batch
- `collate_fn`: Custom batching logic

Performance Tips:

- Use multiple workers for I/O
- Enable pin_memory for GPU
- Prefetch next batch
- Profile loading time

Custom Collate Function

```python
def instruction_collate_fn(batch):
    """Custom collate for instruction data"""
    input_ids = torch.stack([item['input_ids'] for item in batch])
    attention_mask = torch.stack([item['attention_mask'] for item in
      batch])
    labels = torch.stack([item['labels'] for item in batch])

    actual_max_len = attention_mask.sum(dim=1).max().item()

    if actual_max_len < input_ids.shape[1]:
        input_ids = input_ids[:, :actual_max_len]
        attention_mask = attention_mask[:, :actual_max_len]
        labels = labels[:, :actual_max_len]

    return {'input_ids': input_ids,
        'attention_mask': attention_mask,
        'labels': labels}

# Alternative: padding on-the-fly
def dynamic_collate_fn(batch):
    """Pad to max length in current batch only"""
    # Get max length in this batch
    max_len = max(len(item['input_ids']) for item in batch)
    # Pad each sequence to batch max length
    # More memory efficient than padding to global max
```

Collate Function Role:

- Combine samples into batch
- Handle variable lengths
- Apply padding strategy
- Create tensors for model

Padding Strategies:

- **Static:** Pad to fixed max_length
- **Dynamic:** Pad to batch max
- **Bucketing:** Group similar lengths

Dynamic padding saves memory **Memory Impact:**

- Static: Consistent but wasteful
- Dynamic: Efficient but variable
- Trade-off: Speed vs memory

Gradient Accumulation for Large Batch Sizes

```python
def train_with_accumulation(model, dataloader,
                            accumulation_steps=4):
    """Train with gradient accumulation"""
    optimizer.zero_grad()

    for batch_idx, batch in enumerate(dataloader):
        # Forward pass
        outputs = model(**batch)
        loss = outputs.loss / accumulation_steps

        # Backward pass
        loss.backward()

        # Update weights after accumulation
        if (batch_idx + 1) % accumulation_steps == 0:
            optimizer.step()
            optimizer.zero_grad()

        # Log effective batch size
        effective_batch = batch_size * accumulation_steps
        print(f"Effective batch size: {effective_batch}")
```

Why Gradient Accumulation?

- Simulate larger batches
- Overcome GPU memory limits
- Improve training stability
- Better convergence

Effective Batch Formula:

$$\text{Effective} = \text{Physical} \times \text{Accumulation}$$

Memory Savings:

- 4x accumulation = 4x larger batch
- Same memory as single batch
- Trade computation for memory

Memory Optimization Strategies

Memory Reduction Techniques:

- **Mixed Precision:** FP16/BF16 training
- **Gradient Checkpointing:** Trade compute for memory
- **CPU Offloading:** Move optimizer states
- **Model Sharding:** Distribute across GPUs

Memory Breakdown:

- Model weights: 2 bytes/param (FP16)
- Gradients: 2 bytes/param
- Optimizer states: 8 bytes/param (Adam)
- Activations: Variable

```python
# Mixed precision training
from torch.cuda.amp import autocast, GradScaler

scaler = GradScaler()

for batch in dataloader:
    optimizer.zero_grad()

    with autocast():
        outputs = model(**batch)
        loss = outputs.loss

    scaler.scale(loss).backward()
    scaler.step(optimizer)
    scaler.update()

# Memory: 50% reduction
# Speed: 2-3x faster
```

Distributed and Parallel Data Loading

```python
from torch.utils.data.distributed import DistributedSampler
import torch.distributed as dist

def setup_distributed_dataloader(dataset, world_size, rank):
    """Setup distributed data loading"""
    # Create distributed sampler
    sampler = DistributedSampler(
        dataset,
        num_replicas=world_size,
        rank=rank,
        shuffle=True,
        seed=42
    )

    # Create dataloader with sampler
    dataloader = DataLoader(
        dataset,
        batch_size=batch_size // world_size,
        sampler=sampler,
        num_workers=4,
        pin_memory=True,
        persistent_workers=True
    )

    return dataloader

# Each GPU gets unique subset
# No data duplication across GPUs
```

Distributed Benefits:

- Linear scaling with GPUs
- No data overlap
- Synchronized training
- Efficient memory usage

Key Parameters:

- `world_size`: Total GPUs
- `rank`: GPU identifier
- `num_replicas`: Same as world_size
- `persistent_workers`: Keep alive

Essential for multi-GPU training

Bucketing for Efficient Batching

```python
from torch.utils.data import Sampler
import numpy as np

class BucketSampler(Sampler):
    def __init__(self, dataset, batch_size, bucket_size=100):
        self.dataset = dataset
        self.batch_size = batch_size
        self.bucket_size = bucket_size

        # Sort by sequence length
        self.lengths = [len(d['input_ids']) for d in dataset]
        self.sorted_idx = np.argsort(self.lengths)

    def __iter__(self):
        # Create buckets of similar lengths
        buckets = []
        for i in range(0, len(self.sorted_idx), self.bucket_size):
            bucket = self.sorted_idx[i:i+self.bucket_size]
            np.random.shuffle(bucket)  # Shuffle within bucket
            buckets.extend(bucket)

        # Yield batches from buckets
        for i in range(0, len(buckets), self.batch_size):
            yield buckets[i:i+self.batch_size]
```

Bucketing Benefits:

- Minimize padding waste
- Similar lengths in batch
- Faster training
- Better GPU utilization

Example Buckets:

```
Bucket 1: [100-200 tokens]
Bucket 2: [200-300 tokens]
Bucket 3: [300-400 tokens]
Bucket 4: [400-512 tokens]
```

Memory Impact:

- 30-50% memory savings
- 20-30% speed improvement
- Maintains randomness

Monitoring Data Pipeline Performance

```python
import time
from torch.utils.data import DataLoader

class DataLoaderMonitor:
    def __init__(self, dataloader):
        self.dataloader = dataloader
        self.batch_times = []

    def __iter__(self):
        start_time = time.time()

        for batch_idx, batch in enumerate(self.dataloader):
            batch_time = time.time() - start_time
            self.batch_times.append(batch_time)

            # Log statistics
            if batch_idx % 100 == 0:
                avg_time = np.mean(self.batch_times[-100:])
                throughput = len(batch['input_ids']) / batch_time

                print(f"Batch {batch_idx}:")
                print(f"  Load time: {batch_time:.3f}s")
                print(f"  Throughput: {throughput:.0f} samples/s")
                print(f"  Avg time: {avg_time:.3f}s")

            yield batch
            start_time = time.time()
```

Key Metrics:

- Data loading time
- Throughput (samples/sec)
- GPU utilization
- Memory usage
- Queue depths

Bottleneck Indicators:

- GPU idle time ¿ 10%
- Data loading ¿ forward pass
- Memory spikes
- Uneven batch times

Monitor to identify optimization opportunities

Caching for Repeated Data Access

Caching Levels:

- **Dataset Cache:** Preprocessed tokens
- **Memory Cache:** Frequently accessed batches
- **Disk Cache:** Persistent storage
- **GPU Cache:** Pinned memory

When to Cache:

- Multiple epochs training
- Expensive preprocessing
- Small datasets
- Repeated validation

```python
class CachedDataset(Dataset):
    def __init__(self, data, tokenizer, cache_dir):
        self.data = data
        self.tokenizer = tokenizer
        self.cache_dir = cache_dir
        self.cache = {}

    def __getitem__(self, idx):
        # Check cache first
        if idx in self.cache:
            return self.cache[idx]

        # Process and cache
        item = self.process_item(self.data[idx])
        self.cache[idx] = item

        # Save to disk
        cache_file = f"{self.cache_dir}/{idx}.pt"
        torch.save(item, cache_file)

        return item
```

Prefetching and Asynchronous Loading

```python
from torch.utils.data import DataLoader
import torch.cuda

class PrefetchLoader:
    def __init__(self, dataloader, device='cuda'):
        self.dataloader = dataloader
        self.device = device

    def __iter__(self):
        stream = torch.cuda.Stream()
        first = True

        for next_batch in self.dataloader:
            with torch.cuda.stream(stream):
                # Async copy to GPU
                next_batch = {k: v.to(self.device, non_blocking=
    True)
                              for k, v in next_batch.items()}

            if not first:
                yield batch  # Return previous
            else:
                first = False

            torch.cuda.current_stream().wait_stream(stream)
            batch = next_batch
        yield batch  # Last batch
# GPU processes while CPU prepares next batch
```

Prefetching Benefits:

- Hide data transfer latency
- Overlap compute and I/O
- Better GPU utilization
- Faster overall training

Pipeline Timeline:

```
CPU: Load B1 | Load B2 | Load B3
GPU:         | Proc B1 | Proc B2
```

Performance Gains:

- 10-20% speed improvement
- Near 100% GPU utilization
- Minimal code changes

Troubleshooting Common Data Loading Issues

Common Problems:

- **OOM Errors:** Batch size too large
- **Slow Loading:** I/O bottleneck
- **Deadlocks:** Multi-process issues
- **Data Corruption:** Encoding errors

Debugging Tools:

- torch.utils.bottleneck
- nvidia-smi monitoring
- Memory profilers
- Data validation scripts

```python
# Debug data loading
def debug_dataloader(loader):
    print(f"Batches: {len(loader)}")

    for i, batch in enumerate(loader):
        if i >= 3: break  # Test first 3

        print(f"\nBatch {i}:")
        for k, v in batch.items():
            print(f"  {k}: {v.shape}")
            print(f"    dtype: {v.dtype}")
            print(f"    device: {v.device}")
            print(f"    mem: {v.nbytes/1e6:.1f}MB")

        # Check for issues
        if torch.isnan(v).any():
            print("WARNING: NaN values!")
```

Part IV

Model Loading and Fine-tuning

Implementing the complete training pipeline

Loading Pre-trained Models

```python
from transformers import AutoModelForCausalLM
import torch

def load_model_for_finetuning(model_name, device_map="auto"):
    """Load pre-trained model for fine-tuning"""

    # Load with automatic device mapping
    model = AutoModelForCausalLM.from_pretrained(
        model_name,
        torch_dtype=torch.bfloat16,  # Use BF16 for stability
        device_map=device_map,       # Auto GPU assignment
        trust_remote_code=True,       # For custom models
        use_cache=False              # Disable KV cache for training
    )

    # Enable gradient checkpointing
    model.gradient_checkpointing_enable()

    # Prepare for training
    model.train()

    print(f"Model loaded: {model.num_parameters()/1e9:.1f}B params")
    print(f"Memory: {torch.cuda.memory_allocated()/1e9:.1f}GB")

    return model
```

Loading Options:

- `torch_dtype`: FP16/BF16/FP32
- `device_map`: GPU placement
- `use_cache`: KV cache control
- `load_in_8bit`: Quantization

Model Sizes:

- 7B: 14GB (FP16)
- 13B: 26GB (FP16)
- 30B: 60GB (FP16)
- 70B: 140GB (FP16)

BF16 recommended for training stability

Preparing Model for Training

```python
def prepare_model_for_training(model, use_lora=False):
    """Configure model for fine-tuning"""

    # Freeze embeddings (optional)
    for param in model.get_input_embeddings().parameters():
        param.requires_grad = False

    # Enable training mode
    model.train()

    if use_lora:
        from peft import LoraConfig, get_peft_model

        lora_config = LoraConfig(
            r=16,                    # Rank
            lora_alpha=32,           # Scaling
            target_modules=["q_proj", "v_proj"],
            lora_dropout=0.1,
            bias="none",
            task_type="CAUSAL_LM"
        )

        model = get_peft_model(model, lora_config)
        print(f"Trainable params: {model.print_trainable_parameters()}")

    return model
```

Training Configurations:

- Full fine-tuning: All parameters
- LoRA: 0.1% parameters
- Frozen embeddings: Stability
- Gradient checkpointing: Memory

Memory Requirements:

- Full: 4x model size
- LoRA: 1.5x model size
- QLoRA: 0.5x model size

Training Speed:

- Full: Baseline
- LoRA: 1.5-2x faster
- QLoRA: 1.2x faster

Training Configuration and Hyperparameters

```python
from transformers import TrainingArguments

training_args = TrainingArguments(
    output_dir="./instruction-tuned-model",
    per_device_train_batch_size=4,
    per_device_eval_batch_size=4,
    gradient_accumulation_steps=4,
    num_train_epochs=3,
    learning_rate=2e-5,
    warmup_steps=100,
    logging_steps=10,
    save_strategy="steps",
    save_steps=500,
    evaluation_strategy="steps",
    eval_steps=100,
    bf16=True,                    # Use BF16
    tf32=True,                    # Tensor cores
    gradient_checkpointing=True,
    dataloader_num_workers=4,
    report_to="tensorboard",
    load_best_model_at_end=True,
    metric_for_best_model="eval_loss",
    greater_is_better=False,
)
```

Key Hyperparameters:

- Learning rate: 1e-5 to 5e-5
- Batch size: 4-32 per GPU
- Epochs: 1-3 typical
- Warmup: 3-10% of steps

Optimization Tips:

- Start with small LR
- Use gradient accumulation
- Monitor loss closely
- Early stopping if overfitting

Adjust based on model size and data

Custom Training Loop Implementation

```python
def train_epoch(model, dataloader, optimizer, scheduler):
    """Custom training loop for one epoch"""
    model.train()
    total_loss = 0
    progress_bar = tqdm(dataloader, desc="Training")
    for step, batch in enumerate(progress_bar):
        batch = {k: v.to('cuda') for k, v in batch.items()}

        # Forward pass
        outputs = model(**batch)
        loss = outputs.loss

        # Backward pass
        loss.backward()

        # Gradient clipping
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

        # Optimizer step
        optimizer.step()
        scheduler.step()
        optimizer.zero_grad()

        # Update metrics
        total_loss += loss.item()
        progress_bar.set_postfix({'loss': loss.item()})

    return total_loss / len(dataloader)
```

Training Loop Components:

- Forward pass: Compute loss
- Backward pass: Compute gradients
- Gradient clipping: Stability
- Optimizer step: Update weights
- Scheduler step: Adjust LR

Monitoring:

- Loss per batch
- Learning rate
- Gradient norm
- GPU memory

Custom loops offer maximum control

Using HuggingFace Trainer API

```python
from transformers import Trainer, DataCollatorForLanguageModeling

# Data collator for padding
data_collator = DataCollatorForLanguageModeling(
    tokenizer=tokenizer,
    mlm=False,  # Causal LM, not masked
    pad_to_multiple_of=8  # Efficient for GPU
)

# Initialize trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    data_collator=data_collator,
    tokenizer=tokenizer,
    compute_metrics=compute_metrics,
    callbacks=[early_stopping_callback],
)

# Start training
trainer.train()

# Save the fine-tuned model
trainer.save_model("./final-model")
tokenizer.save_pretrained("./final-model")
```

Trainer Advantages:

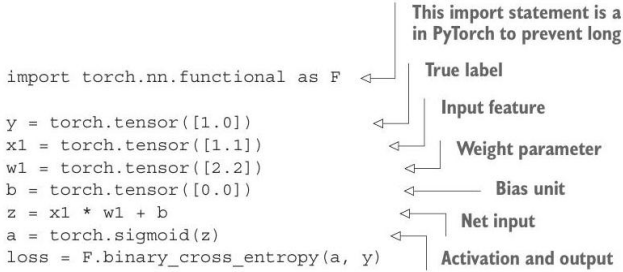
- Automatic mixed precision
- Distributed training support
- Built-in logging
- Checkpoint management
- Easy evaluation

Built-in Features:

- TensorBoard integration
- W&B logging
- Early stopping
- Learning rate scheduling
- Gradient accumulation

Recommended for most use cases

Monitoring Training Progress



Key Metrics:

- Training loss curve
- Validation loss trend
- Learning rate schedule
- Gradient norms

Warning Signs:

- Loss explosion
- Validation divergence
- Gradient vanishing
- Memory leaks

Preventing Overfitting

Regularization Techniques:

- **Dropout:** Random neuron deactivation
- **Weight Decay:** L2 regularization
- **Early Stopping:** Monitor validation
- **Data Augmentation:** Increase diversity

Detection Methods:

- Train/val loss divergence
- Performance plateau
- Memorization patterns
- Generalization failure

```python
from transformers import EarlyStoppingCallback

# Early stopping configuration
early_stopping = EarlyStoppingCallback(
    early_stopping_patience=3,
    early_stopping_threshold=0.001
)

# Dropout in model config
model.config.attention_dropout = 0.1
model.config.hidden_dropout = 0.1

# Weight decay in optimizer
optimizer = AdamW(
    model.parameters(),
    lr=2e-5,
    weight_decay=0.01  # L2 regularization
)
```

Checkpoint Management and Recovery

```python
import os
import torch
class CheckpointManager:
    def __init__(self, save_dir, max_checkpoints=3):
        self.save_dir = save_dir
        self.max_checkpoints = max_checkpoints
        self.checkpoints = []

    def save_checkpoint(self, model, optimizer, epoch, loss):
        checkpoint = {'epoch': epoch,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'loss': loss,}

        path = f"{self.save_dir}/checkpoint_epoch_{epoch}.pt"
        torch.save(checkpoint, path)
        self.checkpoints.append(path)

        if len(self.checkpoints) > self.max_checkpoints:
            old = self.checkpoints.pop(0)
            os.remove(old)

    def load_checkpoint(self, path, model, optimizer):
        checkpoint = torch.load(path)
        model.load_state_dict(checkpoint['model_state_dict'])
        optimizer.load_state_dict(checkpoint['optimizer_state_dict
    '])
        return checkpoint['epoch'], checkpoint['loss']
```

Checkpoint Contents:

- Model weights
- Optimizer state
- Learning rate scheduler
- Training metrics
- Random states

Best Practices:

- Save regularly (every N steps)
- Keep best checkpoints
- Version control configs
- Document experiments

Essential for long training runs

Multi-GPU and Distributed Training

```python
import torch.distributed as dist
from torch.nn.parallel import DistributedDataParallel as DDP

def setup_distributed(rank, world_size):
    """Initialize distributed training"""
    os.environ['MASTER_ADDR'] = 'localhost'
    os.environ['MASTER_PORT'] = '12355'

    dist.init_process_group("nccl", rank=rank, world_size=world_size)

def train_distributed(rank, world_size):
    setup_distributed(rank, world_size)

    # Create model on specific GPU
    model = model.to(rank)
    model = DDP(model, device_ids=[rank])

    # Distributed sampler
    sampler = DistributedSampler(dataset, num_replicas=world_size, rank=
      rank)

    dataloader = DataLoader(dataset, sampler=sampler, batch_size=
      batch_size)

    # Train as normal
    for epoch in range(num_epochs):
        sampler.set_epoch(epoch)  # Shuffle differently each epoch
        train_epoch(model, dataloader, optimizer)
```

Parallelism Types:

- **Data Parallel:** Split batches
- **Model Parallel:** Split model
- **Pipeline Parallel:** Split layers
- **Tensor Parallel:** Split tensors

Scaling Efficiency:

- 2 GPUs: 1.8x speedup
- 4 GPUs: 3.5x speedup
- 8 GPUs: 6.5x speedup

Essential for large models

Evaluation During Training

```python
def evaluate(model, eval_dataloader, compute_metrics=None):
    """Evaluate model on validation set"""
    model.eval()
    total_loss = 0
    predictions = []
    references = []

    with torch.no_grad():
        for batch in tqdm(eval_dataloader, desc="Evaluating"):
            batch = {k: v.to('cuda') for k, v in batch.items()}

            outputs = model(**batch)
            total_loss += outputs.loss.item()

            # Generate predictions
            generated = model.generate(batch['input_ids'],max_length=512,
                temperature=0.7,do_sample=True)

            predictions.extend(tokenizer.batch_decode(generated))
            references.extend(tokenizer.batch_decode(batch['labels']))

    # Compute metrics
    metrics = {'eval_loss': total_loss / len(eval_dataloader)}
    if compute_metrics:
        metrics.update(compute_metrics(predictions, references))

    return metrics
```

Evaluation Metrics:

- Perplexity
- BLEU score
- ROUGE scores
- Custom task metrics

Evaluation Frequency:

- Every N steps
- End of epoch
- On demand
- Best model tracking

Generation Settings:

- Temperature: 0.7-1.0
- Top-p: 0.9-0.95
- Top-k: 40-50

Generation and Inference

```python
def generate_response(model, tokenizer, instruction, input_text=""):
    """Generate response for instruction"""
    model.eval()

    # Format prompt
    if input_text:
        prompt = f"""Below is an instruction that describes a task,
      paired with an input. Write a response that appropriately completes
       the request.
            \### Instruction:
            {instruction}

            \### Input:
            {input_text}

            \### Response:
            """
    else:
        prompt = f"""Below is an instruction that describes a task. Write
      a response that appropriately completes the request.
            \### Instruction:
            {instruction}

            \### Response:
            """

    # Tokenize
    inputs = tokenizer(prompt, return_tensors="pt").to('cuda')
```

Generation Parameters:

- `max_new_tokens`: Output length
- `temperature`: Randomness
- `top_p`: Nucleus sampling
- `top_k`: Top-k sampling
- `do_sample`: Enable sampling

Decoding Strategies:

- Greedy: Deterministic
- Sampling: Random
- Beam Search: Multiple paths
- Nucleus: Dynamic vocabulary

Balance quality vs diversity

Common Training Issues and Solutions

Loss Explosion:

- Lower learning rate
- Gradient clipping
- Check data quality
- Use FP32 or BF16

Out of Memory:

- Reduce batch size
- Enable gradient checkpointing
- Use mixed precision
- Apply LoRA/QLoRA

Slow Convergence:

- Increase learning rate
- Adjust warmup steps
- Check data distribution
- Use larger batch size

Poor Quality:

- More training data
- Longer training
- Adjust temperature
- Check prompt format

Systematic debugging leads to solutions

Training Best Practices

Before Training:

- Validate data quality
- Test on small subset
- Profile memory usage
- Set up monitoring

During Training:

- Monitor loss curves
- Track GPU utilization
- Save regular checkpoints
- Evaluate periodically

After Training:

- Thorough evaluation
- Ablation studies
- Document results
- Version control models

General Tips:

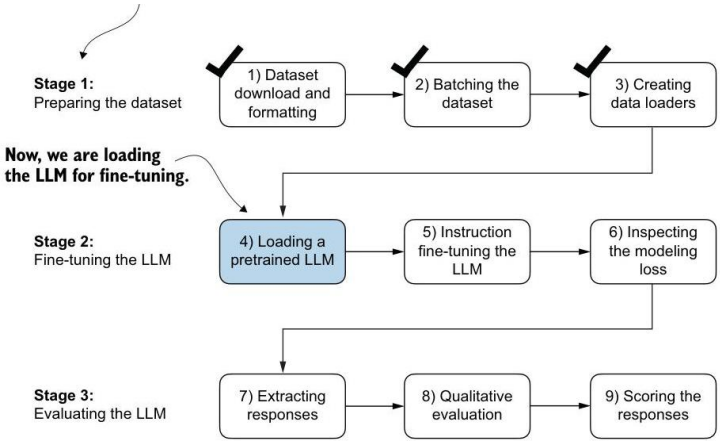
- Start simple, iterate
- Keep detailed logs
- Reproduce experiments
- Share learnings

Complete Training Pipeline Example

End-to-End Pipeline:

1. Load and preprocess data → Format instructions
2. Create datasets and dataloaders → Efficient batching
3. Load pre-trained model → Configure for training
4. Train with monitoring → Track progress
5. Evaluate and save → Deploy best model

Complete in 5 steps!



Part V

Advanced Techniques

Parameter-efficient methods and optimization

LoRA: Low-Rank Adaptation

LoRA Concept:

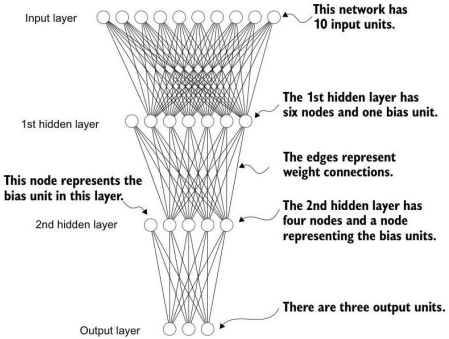
- Freeze pre-trained weights
- Add trainable rank decomposition matrices
- Merge at inference time
- Dramatically reduce parameters

Mathematical Formulation:

$$W' = W + BA$$

where:

- $W \in \mathbb{R}^{d \times k}$: Frozen weights
- $B \in \mathbb{R}^{d \times r}$: Trainable
- $A \in \mathbb{R}^{r \times k}$: Trainable



Benefits:

- 10,000x fewer parameters
- Same quality as full fine-tuning
- Multiple LoRA adapters possible

Implementing LoRA

```python
from peft import LoraConfig, get_peft_model, TaskType

# Configure LoRA
lora_config = LoraConfig(
    r=16,                           # Rank
    lora_alpha=32,                  # Scaling factor
    target_modules=[                # Which modules to adapt
        "q_proj",                   # Query projection
        "k_proj",                   # Key projection
        "v_proj",                   # Value projection
        "o_proj",                   # Output projection
        "gate_proj",                # MLP gate
        "up_proj",                  # MLP up
        "down_proj"                 # MLP down
    ],
    lora_dropout=0.1,               # Dropout for regularization
    bias="none",                    # Don't train biases
    task_type=TaskType.CAUSAL_LM    # Task type
)

# Apply LoRA to model
model = get_peft_model(model, lora_config)

# Check trainable parameters
model.print_trainable_parameters()
# Output: trainable params: 4,194,304 || all params: 6,738,415,616 ||
      trainable%: 0.06
```

LoRA Hyperparameters:

- **r (rank):** 4-64 typical
- **alpha:** r to 2r typical
- **dropout:** 0.05-0.1
- **target_modules:** Attention layers

Memory Savings:

- 7B model: 28GB $\rightarrow$ 4GB
- 13B model: 52GB $\rightarrow$ 8GB
- 30B model: 120GB $\rightarrow$ 16GB

0.1% parameters, 90% performance

QLoRA: Quantized LoRA

```python
from transformers import BitsAndBytesConfig
import torch

# 4-bit quantization config
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,                    # 4-bit quantization
    bnb_4bit_quant_type="nf4",            # NormalFloat4
    bnb_4bit_use_double_quant=True,       # Double quantization
    bnb_4bit_compute_dtype=torch.bfloat16 # Compute in BF16
)

# Load quantized model
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    quantization_config=bnb_config,
    device_map="auto",
    trust_remote_code=True
)

# Apply LoRA on top of quantized model
model = prepare_model_for_kbit_training(model)
model = get_peft_model(model, lora_config)

# Training uses ~25% of full precision memory
print(f"Memory usage: {torch.cuda.memory_allocated()/1e9:.1f}GB")
```

QLoRA Components:

- 4-bit base model (frozen)
- 16-bit LoRA adapters
- 32-bit optimizer states
- Paged optimizers

Memory Formula:

$$\text{Memory} = \frac{\text{Model}}{4} + \text{LoRA} + \text{Gradients}$$

Performance:

- 65B model on single 48GB GPU
- 95% of full fine-tuning quality
- 4x slower than LoRA

Adapter-Based Fine-tuning

Adapter Architecture:

- Insert small networks between layers
- Keep original model frozen
- Train only adapter parameters
- Multiple tasks, multiple adapters

Adapter Design:

- Down-projection: $d \to r$
- Non-linearity: ReLU/GELU
- Up-projection: $r \to d$
- Residual connection

```python
class Adapter(nn.Module):
    def __init__(self, dim, r=64):
        super().__init__()
        self.down = nn.Linear(dim, r)
        self.up = nn.Linear(r, dim)
        self.act = nn.GELU()

    def forward(self, x):
        residual = x
        x = self.down(x)
        x = self.act(x)
        x = self.up(x)
        return x + residual  # Skip connection

# Insert adapters after attention/FFN
for layer in model.transformer.layers:
    layer.attn_adapter = Adapter(dim)
    layer.ffn_adapter = Adapter(dim)
```

Prompt Tuning and P-Tuning

```python
from peft import PromptTuningConfig, get_peft_model

# Soft prompt tuning
prompt_config = PromptTuningConfig(
    task_type="CAUSAL_LM",
    prompt_tuning_init="TEXT",  # Initialize from text
    num_virtual_tokens=20,       # Soft prompt length
    prompt_tuning_init_text="Classify the following text:",
    tokenizer_name_or_path=model_name,)

# Apply prompt tuning
model = get_peft_model(model, prompt_config)

# Only soft prompts are trainable
model.print_trainable_parameters()
# trainable params: 40,960 || all params: 6,738,456,576 || 0.0006%

# P-Tuning v2: Deep prompt tuning
class PrefixEncoder(nn.Module):
    def __init__(self, prefix_len, hidden_size):
        super().__init__()
        self.embedding = nn.Embedding(prefix_len, hidden_size)
        self.trans = nn.Sequential(
            nn.Linear(hidden_size, hidden_size),
            nn.Tanh(),
            nn.Linear(hidden_size, hidden_size * 2)
        )
```

Prompt Tuning Types:

- **Hard Prompts:** Fixed text
- **Soft Prompts:** Learnable embeddings
- **Prefix Tuning:** Prepend to keys/values
- **P-Tuning v2:** Deep prompts

Advantages:

- Extremely parameter efficient
- Task-specific prompts
- No model modification
- Fast switching between tasks

Best for multi-task scenarios

Gradient Checkpointing

Memory vs Compute Trade-off:

- Don't store all activations
- Recompute during backward pass
- 60% memory reduction
- 20% slower training

How it Works:

- Checkpoint certain layers
- Forward: Store checkpoints only
- Backward: Recompute from checkpoints
- Memory: $O(\sqrt{n})$ instead of $O(n)$

```python
# Enable gradient checkpointing
model.gradient_checkpointing_enable()

# Custom checkpointing
from torch.utils.checkpoint import checkpoint

class CheckpointedLayer(nn.Module):
    def forward(self, x):
        # Checkpoint expensive operations
        def custom_forward(x):
            x = self.attention(x)
            x = self.mlp(x)
            return x

        return checkpoint(custom_forward, x)

# Memory comparison
# Without: 24GB for 7B model
# With: 10GB for 7B model
```

Mixed Precision Training

```python
from torch.cuda.amp import autocast, GradScaler

# Initialize gradient scaler
scaler = GradScaler()

for epoch in range(num_epochs):
    for batch in dataloader:
        optimizer.zero_grad()

        # Mixed precision forward pass
        with autocast(dtype=torch.bfloat16):
            outputs = model(**batch)
            loss = outputs.loss

        # Scale loss and backward
        scaler.scale(loss).backward()

        # Unscale before clipping
        scaler.unscale_(optimizer)
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

        # Optimizer step with scaling
        scaler.step(optimizer)
        scaler.update()

# BF16 vs FP16
# BF16: Better numerical stability, same range as FP32
# FP16: Better hardware support, needs loss scaling
```

Precision Formats:

- **FP32:** Full precision (baseline)
- **FP16:** Half precision, needs scaling
- **BF16:** Brain float, more stable
- **TF32:** Tensor cores on A100

Benefits:

- 2x memory reduction
- 2-3x speed improvement
- Larger batch sizes
- Minimal accuracy loss

Essential for large model training

Knowledge Distillation for Compression

```python
import torch.nn.functional as F

def distillation_loss(student_logits, teacher_logits,
                      labels, temperature=3.0, alpha=0.5):
    """Knowledge distillation loss"""
    teacher_probs = F.softmax(teacher_logits / temperature, dim=-1)
    student_log_probs = F.log_softmax(student_logits / temperature, dim
      =-1)

    kl_loss = F.kl_div(student_log_probs, teacher_probs,
                       reduction='batchmean') * (temperature ** 2)

    ce_loss = F.cross_entropy(student_logits, labels)

    total_loss = alpha * kl_loss + (1 - alpha) * ce_loss
    return total_loss

teacher_model.eval()
for batch in dataloader:
    with torch.no_grad():
        teacher_outputs = teacher_model(**batch)

    student_outputs = student_model(**batch)
    loss = distillation_loss(student_outputs.logits,
                             teacher_outputs.logits,
                             batch['labels'])
```

Distillation Process:

- Large teacher model (e.g., 70B)
- Small student model (e.g., 7B)
- Transfer knowledge via soft labels
- Maintain performance

Key Parameters:

- **Temperature:** Controls softness of probability distribution
- **Alpha:** Balance between distillation and hard labels

Knowledge Distillation for Compression - Part 2

```python
# Training loop with distillation
teacher_model.eval()  # Freeze teacher
student_optimizer = torch.optim.AdamW(
    student_model.parameters(), lr=1e-4)

for epoch in range(num_epochs):
    for batch in dataloader:
        # Get teacher predictions (no gradient)
        with torch.no_grad():
            teacher_outputs = teacher_model(**batch)

        # Get student predictions
        student_outputs = student_model(**batch)

        # Calculate distillation loss
        loss = distillation_loss(
            student_outputs.logits,
            teacher_outputs.logits,
            batch['labels'],
            temperature=3.0,
            alpha=0.7  # 70% distillation, 30% hard
        )

        # Backprop and optimize
        loss.backward()
        student_optimizer.step()
        student_optimizer.zero_grad()
```

Compression Results:

Model	Size	Perf
Teacher	70B	100%
Student	7B	95%
Tiny	1.5B	88%

Benefits:

- 10x size reduction
- 5x faster inference
- Edge deployment possible

Deploy efficiently!

Comparing Parameter-Efficient Methods

Method	Params	Memory	Speed	Quality	Use Case
Full Fine-tuning	100%	High	Baseline	Best	Small models
LoRA	0.1%	Medium	Fast	95-98%	General purpose
QLoRA	0.1%	Low	Slow	93-96%	Large models
Adapters	1-5%	Medium	Medium	92-95%	Multi-task
Prompt Tuning	0.01%	Very Low	Very Fast	85-90%	Few tasks

Selection Criteria:

- Hardware constraints
- Quality requirements
- Number of tasks
- Deployment scenario
- Training time budget
- Model size

Summary

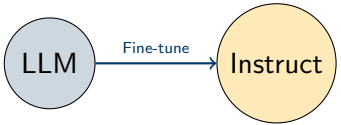
Key Takeaways and Next Steps

Putting it all together

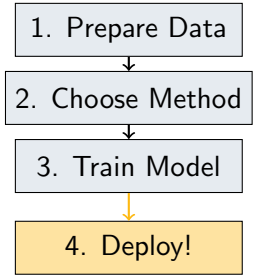
Key Takeaways

Core Concepts:

- Instruction fine-tuning transforms LLMs into assistants
- Quality data is more important than quantity
- Proper formatting ensures training success



Implementation Roadmap:



Success Formula:

Data Quality $\times$ Method Efficiency
$=$ Training Success

Practical Recommendations

Getting Started Example:

```python
# Minimal setup for beginners
from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
    TrainingArguments,
    Trainer
)
from peft import LoraConfig

# Start small
model_name = "meta-llama/Llama-2-7b"
data = load_dataset("tatsu-lab/alpaca")

# Use LoRA
lora_config = LoraConfig(
    r=8,  # Low rank
    lora_alpha=16,
    target_modules=["q_proj", "v_proj"]
)
```

Common Pitfalls:

- Overfitting to format
- Catastrophic forgetting
- Poor data quality
- Insufficient evaluation
- Deployment challenges

Resources:

- HuggingFace Transformers
- PEFT library
- Alpaca dataset
- LLM training guides
- Community forums

Future Directions in Instruction Fine-tuning

Research Frontiers:

- Constitutional AI approaches
- Self-supervised alignment
- Instruction evolution techniques
- Cross-lingual transfer
- Multi-modal instructions

Technical Advances:

- More efficient architectures
- Better quantization methods
- Improved alignment algorithms
- Faster training techniques
- Enhanced evaluation metrics

Industry Trends:

- Specialized domain models
- Edge deployment solutions
- Privacy-preserving training
- Continuous learning systems
- Automated fine-tuning pipelines

Opportunities:

- Custom AI assistants
- Domain expertise capture
- Personalized models
- Real-time adaptation
- Collaborative training

Next Lecture Preview

Advanced Topics:

- Reinforcement Learning from Human Feedback (RLHF)
- Constitutional AI and safety
- Direct Preference Optimization (DPO)
- Multi-modal instruction tuning

Practical Applications:

- Building domain-specific assistants
- Deployment strategies
- Performance optimization
- Evaluation and monitoring

Thank you! Questions?