

Training LLMs from Base Models

A Comprehensive Workshop Guide

PART 1: FOUNDATIONS & HISTORY

Section 1: Historical Evolution of AI & ML

1943-1960s: The Beginning

McCulloch-Pitts Neuron (1943) - First mathematical model of neural network - Binary threshold activation

Perceptron (Rosenblatt, 1958)

Output = $\begin{cases} 1 & \text{if } \sum(w_i \times x_i) > \theta \\ 0 & \text{otherwise} \end{cases}$

- Linear classifier only
- Minsky & Papert (1969): XOR problem killed first AI wave



Perceptron: A simple algorithm that makes binary decisions by weighing inputs. Like a function that returns true/false based on weighted sum of inputs. Foundation of neural networks but can only solve linearly separable problems.


1960s-1980s: Symbolic AI Era

Expert Systems - DENDRAL (1965): Chemical analysis - MYCIN (1976): Medical diagnosis - Knowledge representation: IF-THEN rules




Expert Systems: Programs that encode human expertise as rules (if condition then action). Like a giant switch statement with domain knowledge. Worked well for narrow domains but couldn't generalize.


Fuzzy Logic (Zadeh, 1965) - Degrees of truth $[0,1]$ vs binary - Linguistic variables
- Applications: Control systems, washing machines

 **Fuzzy Logic:** Allows partial truths between 0 and 1, not just true/false. Like having a float instead of boolean for “how true” something is. Useful when categories have soft boundaries.


1980s-1990s: Connectionist Revival

Multi-Layer Perceptrons + Backpropagation (1986) - Rumelhart, Hinton, Williams - Hidden layers solve XOR - Gradient descent through networks

 **Backpropagation:** Algorithm that calculates gradients by chain rule from output back to input. Like git blame but for neural network errors - traces which weights caused the mistake. Enables training deep networks.


 **Hidden Layers:** Intermediate layers between input and output that learn representations. Like middleware that transforms data into useful features. More layers = ability to learn more complex patterns.

Universal Approximation Theorem (1989) - Single hidden layer can approximate any continuous function - But how many neurons needed?


 **Universal Approximation Theorem:** Proof that a neural network with one hidden layer can approximate any continuous function. Like saying you can draw any curve with enough straight line segments. Theoretical guarantee but may need infinite neurons.

1990s-2000s: Diversification


Support Vector Machines (Vapnik, 1995) - Maximum margin classifier - Kernel trick for non-linear separation - Dominated until ~2010

 **SVM:** Finds the best boundary between classes by maximizing margin (distance to nearest points). Like drawing a line between two groups with maximum buffer zone. Uses “kernel trick” to handle non-linear boundaries.

Ensemble Methods - Random Forests (Breiman, 2001) - Gradient Boosting (Friedman, 2001) - AdaBoost (Freund & Schapire, 1997)

 **Ensemble Methods:** Combine multiple models for better predictions than any single model. Like taking a vote from multiple experts instead of trusting one. Random Forest=parallel trees, Boosting=sequential improvement.

Evolutionary Algorithms - Genetic algorithms: Selection, crossover, mutation - Genetic programming: Evolve programs - Particle swarm optimization

 **Evolutionary Algorithms:** Optimization by simulating evolution - create population, select best, mutate, repeat. Like breeding programs that get progressively better at a task. No gradients needed.


Probabilistic Graphical Models - Hidden Markov Models - Conditional Random Fields - Bayesian Networks

2000s-2010s: Statistical Learning

Factorization Machines (Rendle, 2010)


$$\hat{y}(x) = w_0 + \sum w_i x_i + \sum \sum \langle v_i, v_j \rangle x_i x_j$$

- Feature interactions without combinatorial explosion
- Key for recommender systems

 **Factorization Machines:** Efficiently model interactions between features using dot products of latent vectors. Like collaborative filtering but for any features. Avoids creating all feature pairs explicitly.


Deep Belief Networks (Hinton, 2006) - Layer-wise pre-training - Restricted Boltzmann Machines - Breakthrough: Deep networks finally trainable


Convolutional Neural Networks Renaissance - LeNet-5 (1998) → AlexNet (2012) - ImageNet moment: 16.4% error (vs 25.8% next best) - ReLU, Dropout, Data Augmentation

 **CNN:** Neural networks with convolutional layers that scan for patterns regardless of position. Like having a sliding window feature detector. Exploits spatial structure in images.


2010-2015: Deep Learning Explosion

Key Innovations: - **ReLU Activation:** $f(x) = \max(0, x)$ - No vanishing gradient - **Dropout** (2012): Randomly zero neurons during training - **Batch Normalization** (2015): Normalize layer inputs - **Adam Optimizer** (2014): Adaptive learning rates


 **Dropout:** Randomly disable neurons during training to prevent overfitting. Like training with random team members absent each day. Forces redundancy and robustness.

 **Adam Optimizer:** Gradient descent with per-parameter adaptive learning rates and momentum. Like having cruise control that adjusts speed per road condition. Combines benefits of AdaGrad and RMSProp.

Recurrent Networks Evolution - Vanilla RNN → LSTM (1997) → GRU (2014) - Sequence-to-Sequence (2014) - Attention mechanism (Bahdanau, 2014)

 **RNN/LSTM:** Networks with loops that maintain state across sequence elements. Like a for-loop that remembers previous iterations. LSTM adds gates to control what to remember/forget.

Word Embeddings - Word2Vec (2013): Skip-gram, CBOW - GloVe (2014): Global vectors - Distributed representations

 **Word Embeddings:** Representing words as dense vectors where similar words are nearby in vector space. Like mapping words to coordinates where distance represents semantic similarity. King - Man + Woman \approx Queen.

Why This History Matters for LLMs


1. **Vanishing gradients** → Solved by ReLU, then layer norm
2. **Deep network training** → Pre-training paradigm from DBNs
3. **Sequence modeling** → RNN limitations led to transformers
4. **Feature learning** → Word embeddings → Contextual embeddings
5. **Attention mechanism** → Key building block for transformers

Reference: Negnevitsky, M. (2005). *Artificial Intelligence: A Guide to Intelligent Systems*. Pearson Education.

Section 2: The Path to Transformers

The Sequence Modeling Problem (2015-2017)


RNN/LSTM Limitations - Sequential computation: Can't parallelize training - Long-range dependencies: Information bottleneck through hidden states - Gradient flow: Even LSTMs struggle beyond ~200 tokens - Training time: Days/weeks for translation models

 **Sequence Modeling:** Processing ordered data where context matters (text, time series). Like parsing text where meaning depends on previous words. RNNs process sequentially, limiting parallelization.


Attention as Band-Aid (2014-2016)

Bahdanau Attention: $\alpha_{ij} = \exp(e_{ij}) / \sum_k \exp(e_{ik})$
where $e_{ij} = a(s_{i-1}, h_j)$

- Still required RNN backbone
- Attention helped but didn't solve parallelization

 **Attention Mechanism:** Dynamically weight which parts of input to focus on for each output. Like highlighting relevant parts of documentation when answering a question. Originally added on top of RNNs.

Google's NMT (2016) - 8-layer LSTM with attention - Training: 6 days on 96 NVIDIA K80 GPUs - BLEU score: 26.30 (EN-DE)

 **BLEU Score:** Metric for translation quality comparing to human references (0-100). Like unit test coverage but for translation accuracy. Higher is better, 30+ is decent, 40+ is very good.


“Attention Is All You Need” (Vaswani et al., 2017)

Core Innovation: Self-Attention

$\text{Attention}(Q, K, V) = \text{softmax}(QK^T / \sqrt{d_k})V$

- Q, K, V: Query, Key, Value matrices from same input


- No recurrence needed
- Full parallelization possible

 **Self-Attention:** Each position can attend to all positions in previous layer. Like each word looking at all other words to understand context. Enables parallelization unlike sequential RNNs.

Multi-Head Attention


$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$
 where $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$

- 8 parallel attention operations
- Different representation subspaces

 **Multi-Head Attention:** Running attention multiple times in parallel with different projections. Like having multiple experts look at the same data from different angles. Each head can learn different relationships.

Position Encodings (No Built-in Order)

$\text{PE}(\text{pos}, 2i) = \sin(\text{pos}/10000^{(2i/d_{\text{model}})})$
 $\text{PE}(\text{pos}, 2i+1) = \cos(\text{pos}/10000^{(2i/d_{\text{model}})})$

 **Position Encodings:** Adding position information since transformers don't inherently know word order. Like adding line numbers to code so order matters. Uses sin/cos waves of different frequencies.

Architecture Choices - Encoder-Decoder structure (for translation) - 6 layers each - Model dimension: 512 - FFN dimension: 2048 - Training: 3.5 days on 8 P100 GPUs (10× faster!) - BLEU score: 28.4 (SOTA)

 **Encoder-Decoder:** Two-part architecture where encoder processes input, decoder generates output. Like compiler frontend (parse) and backend (codegen). Decoder can't see future tokens (masked).

Key Takeaways from the Paper

1. Computational Efficiency - Complexity per layer: $O(n^2 \cdot d)$ vs RNN's $O(n \cdot d^2)$ - Maximum path length: $O(1)$ vs RNN's $O(n)$ - Parallelizable: All positions simultaneously

2. Attention Visualizations - Model learns linguistic structure unsupervised - Different heads capture different phenomena: - Head 5: Direct objects - Head 6: Coreference - Head 7: Syntactic dependencies

3. Scaling Properties - Performance improves log-linearly with model size - Bigger models = Better performance (foreshadowing GPT)

Section 3: From Transformers to GPTs


The Decoder-Only Revolution


GPT (OpenAI, 2018) - Key insight: Drop the encoder, use only decoder -
Unsupervised pre-training + Supervised fine-tuning - 117M parameters, 12 layers -
BookCorpus: 7,000 unpublished books

Architecture simplification:


Transformer: Encoder + Decoder (complex)

GPT: Decoder only (simple, scalable)

 **Decoder-Only:** Using just the decoder part of transformer with causal masking. Like using only the output generation part, trained to predict next token. Simpler than encoder-decoder but equally powerful.

 **Fine-tuning:** Taking a pre-trained model and training it further on specific task/data. Like taking a generalist programmer and training them on your codebase. Much faster than training from scratch.

GPT-2 (2019): “Zero-Shot Task Transfer” - 1.5B parameters (10× larger) -
WebText: 40GB of internet text - Emergent abilities: Translation, summarization without training - “Too dangerous to release” controversy

 **Zero-Shot Learning:** Model performs tasks it wasn’t explicitly trained for, just from instructions. Like a programmer who can write in a new language just from seeing the documentation. No task-specific training needed.


GPT-3 (2020): The Scaling Hypothesis

Parameters: 175B (100× GPT-2)

Training tokens: 300B

Cost: \$4.6M in compute

- In-context learning: Task specification via prompts
- Few-shot learning: Learn from examples in prompt
- No fine-tuning needed

 **Few-Shot Learning:** Learning from just a few examples provided in the prompt. Like showing 2-3 examples of input/output and model figures out the pattern. No gradient updates needed.


BERT and the Bidirectional Branch (2018)

Different Philosophy - Encoder-only architecture - Bidirectional context (vs GPT’s left-to-right) - Masked Language Modeling: Predict [MASK] tokens - Dominated NLU benchmarks (GLUE, SQuAD)

GPT line: Autoregressive generation

BERT line: Understanding/classification

Both important, but GPT won the scaling race

 **Masked Language Modeling:** Training by hiding random words and predicting them from context. Like fill-in-the-blank exercises. Allows bidirectional context but can't generate text naturally.


The Scaling Laws (Kaplan et al., 2020)

Power Laws for Neural LMs

$$L(N) = (N_c/N)^{\alpha N}$$

- L: Loss
- N: Parameters
- N_c : Critical size constant
- $\alpha N \approx 0.076$

Key Findings: 1. Model size matters most (not width/depth ratio) 2. Optimal batch size scales with loss 3. Sample efficiency improves with scale 4. Convergence is predictable

 **Scaling Laws:** Mathematical relationships between model size, data, compute, and performance. Like Moore's Law but for model capabilities. Predictable improvements with scale increase.

Chinchilla (2022): Compute-Optimal Training - Previous models were undertrained - Optimal: 20 tokens per parameter - 70B Chinchilla > 175B GPT-3

Why Decoder-Only Won

1. **Simplicity:** One stack vs two
2. **Generative:** Can do any text task
3. **Scaling:** Cleaner scaling properties
4. **Training:** Simpler objective (next token)
5. **Data:** Can use any text (no pairs needed)

The LLM Cambrian Explosion (2022-2024)

Open Models - LLaMA (Meta): Efficient architecture, public weights - Mistral: European, efficient - Qwen (Alibaba): Multilingual focus - Gemma (Google): Distilled knowledge


Architectural Convergence - RMSNorm instead of LayerNorm - SwiGLU/GeGLU activation instead of ReLU - Rotary Position Embeddings (RoPE) - Grouped-Query Attention (GQA) - Flash Attention for efficiency

References: - Vaswani et al. (2017). "Attention Is All You Need" - Radford et al. (2018). "Improving Language Understanding by Generative Pre-Training" - Kaplan et al. (2020). "Scaling Laws for Neural Language Models"

PART 2: ARCHITECTURE & CORE CONCEPTS

Section 4: Core Concepts - Tensors & Matrix Operations

Tensors: The Fundamental Data Structure

 **Tensors:** Multi-dimensional arrays (1D=vector, 2D=matrix, 3D+=tensor). Like nested arrays in programming but optimized for GPU operations. All neural network data flows through tensors.

Dimensions in LLMs

```
# Typical tensor shapes in transformers
embeddings: [batch_size, seq_len, d_model]           # (32, 512, 768)
attention_weights: [batch, heads, seq_len, seq_len]  # (32, 12, 512, 512)
ffn_weights: [d_model, d_ff]                         # (768, 3072)
logits: [batch_size, seq_len, vocab_size]             # (32, 512, 50257)
```

Memory Calculation

```
# FP32: 4 bytes per parameter
# FP16/BF16: 2 bytes per parameter
# INT8: 1 byte per parameter

model_size_gb = (num_parameters * bytes_per_param) / 1e9

# Example: LLaMA-7B in different precisions
FP32: 7B * 4 = 28GB
FP16: 7B * 2 = 14GB
INT8: 7B * 1 = 7GB
```

Matrix Multiplications in Transformers

Attention Computation


QK^T : [batch, heads, seq, d_k] × [batch, heads, d_k, seq]
= [batch, heads, seq, seq]

Attention × V: [batch, heads, seq, seq] × [batch, heads, seq, d_k]
= [batch, heads, seq, d_k]

FLOPs per Token (Forward Pass)

Attention: $4 * \text{seq_len} * \text{d_model}^2$
FFN: $2 * \text{d_model} * \text{d_ff}$
Total per layer: $\sim 12 * \text{d_model}^2$ (assuming $\text{d_ff} = 4 * \text{d_model}$)

For 32 layers, $\text{d_model}=4096$:
FLOPs per token $\approx 6.4\text{B}$ operations

 **FLOPs:** Floating Point Operations - measure of computational work. Like counting CPU instructions but for math operations. More FLOPs = more compute time/cost.

Section 5: Transformer Architecture Deep Dive

Layer Components


1. Layer Normalization


RMSNorm (modern choice)

```
def rmsnorm(x, weight, eps=1e-5):  
    norm = torch.sqrt(torch.mean(x**2, dim=-1, keepdim=True) +  
                           eps)  
    return x / norm * weight
```

vs Original LayerNorm

```
def layernorm(x, weight, bias, eps=1e-5):  
    mean = x.mean(dim=-1, keepdim=True)  
    var = x.var(dim=-1, keepdim=True, unbiased=False)  
    return weight * (x - mean) / torch.sqrt(var + eps) + bias
```

 **Layer Normalization:** Normalizes inputs across features for each sample independently. Like standardizing data but per-example rather than per-batch. Stabilizes training in deep networks.

 **RMSNorm:** Simplified layer norm using only root mean square, no mean centering. Like normalization but skips the mean subtraction step. Slightly faster with similar performance.

2. Attention Mechanisms Evolution

2017: Multi-Head Attention (MHA)
– k heads, each with $\text{d_model}/k$ dimension

2019: Multi-Query Attention (MQA)
– Share K,V across heads, unique Q
– Memory: $1/k$ reduction

2023: Grouped-Query Attention (GQA)

- Groups share K,V (e.g., 8 heads, 2 groups)
- Balance between MHA quality and MQA efficiency


3. Position Embeddings

Rotary Position Embeddings (RoPE) – Modern choice

```
def apply_rotary_pos_emb(q, k, cos, sin):
    q_embed = (q * cos) + (rotate_half(q) * sin)
    k_embed = (k * cos) + (rotate_half(k) * sin)
    return q_embed, k_embed
```

Advantages over sinusoidal:

- # – Relative positions*
- # – Extrapolates to longer sequences*
- # – Better performance*

 **RoPE:** Rotary Position Embeddings encode position by rotating vectors in complex space. Like encoding position as rotation angle rather than addition. Better extrapolation to longer sequences.

4. FFN Variants

Original (GELU activation)


```
ffn_output = w2(gelu(w1(x)))
```


SwiGLU (LLaMA choice)

```
ffn_output = w2(silu(w1(x)) * w3(x))
```

GeGLU (BERT variant)

```
ffn_output = w2(gelu(w1(x)) * w3(x))
```


 **FFN (Feed-Forward Network):** Two linear layers with activation in between, processes each position independently. Like a per-token MLP that adds capacity. Usually 4× model dimension for hidden size.

 **SwiGLU:** Gated linear unit using Swish activation ($x \cdot \text{sigmoid}(x)$). Like having a gate that controls information flow. Improves model quality with small compute increase.

PART 3: EFFICIENCY TECHNIQUES

Section 6: Quantization - Making Models Fit

INT8 Quantization (LLM.int8())

 **Quantization:** Reducing numerical precision from FP32/16 to INT8/4 to save memory. Like using byte instead of float for numbers, trading precision for memory. Enables larger models on smaller GPUs.

Concept: Mixed-Precision Decomposition

```
# Identify outlier features (0.1% of values)
outliers = abs(x) > threshold # e.g., 6.0

# Compute in different precisions
output = matmul_int8(x_int8, w_int8) + matmul_fp16(x_outliers,
                                                    w_outliers)
```

Impact: - 2× memory reduction - 10-20% slowdown (acceptable) - <1% accuracy loss

4-bit Quantization (QLoRA)

NormalFloat4 (NF4) Datatype

```
# Optimized for normally distributed weights
# Quantization levels chosen for N(0,1) distribution
nf4_levels = [-1.0, -0.6962, -0.5251, -0.3949, -0.2844,
              -0.1848, -0.0911, 0.0, 0.0796, 0.1609,
              0.2461, 0.3379, 0.4407, 0.5626, 0.7230, 1.0]
```

Double Quantization

Original: FP16 weights (2 bytes/param)

↓

First: NF4 weights (0.5 bytes/param)

↓

Second: Quantize the quantization constants

Result: 0.375 bytes/param (5.3× reduction)


Practical Memory (7B model)

Base FP16:	14 GB
INT8:	7-8 GB

QLoRA (NF4): 3–4 GB
→ Fits on consumer GPU!

Section 7: LoRA - Parameter Efficient Fine-tuning

The Math Behind LoRA

 **LoRA (Low-Rank Adaptation):** Fine-tune by adding small trainable matrices instead of updating all weights. Like adding a small patch to existing code rather than rewriting everything. Reduces memory by 100×.


Problem: Full fine-tuning updates $W \in \mathbb{R}^{(d \times k)}$

Solution: Decompose updates into low-rank matrices

$$W' = W_0 + \Delta W = W_0 + BA$$

where:

- W_0 : Frozen pretrained weights
- $B \in \mathbb{R}^{(d \times r)}$, $A \in \mathbb{R}^{(r \times k)}$
- $r \ll \min(d, k)$ (typically $r=8, 16, 32$)

 **Low-Rank Decomposition:** Representing a large matrix as product of two smaller matrices. Like compressing data by finding patterns. Rank r controls compression ratio vs quality.

Trainable Parameters Reduction

Original: $d \times k$ parameters

LoRA: $d \times r + r \times k = r(d + k)$ parameters

Example ($d=4096$, $k=4096$, $r=16$):

Original: 16,777,216 parameters

LoRA: 131,072 parameters (128× reduction!)

Implementation Details

```
class LoRALayer(nn.Module):
    def __init__(self, base_layer, r=16, alpha=16):
        super().__init__()
        self.base_layer = base_layer  # Frozen
        d, k = base_layer.weight.shape

        # Initialization matters!
        self.lora_A = nn.Parameter(torch.randn(r, k) * 0.01)
        self.lora_B = nn.Parameter(torch.zeros(d, r))
        self.scaling = alpha / r

    def forward(self, x):
        base_out = self.base_layer(x)  # No gradient
```

```
lora_out = x @ self.lora_A.T @ self.lora_B.T
return base_out + self.scaling * lora_out
```

Where to Apply LoRA


Typical Configuration:


```
target_modules = [
    "q_proj", "k_proj", "v_proj",  # Attention
    "o_proj",                               # Output projection
    "gate_proj", "up_proj", "down_proj"  # FFN
]

# Memory usage (7B model):
# Full fine-tuning: 28GB (optimizer states)
# LoRA r=16:          ~1GB
```

Section 8: GPU, CUDA & Parallel Training

GPU Architecture for LLMs

 **VRAM:** Video RAM on GPU, like system RAM but for GPU computations. Determines max model size you can load. More VRAM = bigger models or larger batches.

 **Tensor Cores:** Specialized GPU units for matrix multiplication, much faster than regular CUDA cores. Like SIMD instructions but for matrix ops. 10× speedup for transformer operations.

Key Specs That Matter

Memory Bandwidth: How fast we can move tensors

- A100: 1.6 TB/s
- V100: 900 GB/s
- T4: 320 GB/s


Tensor Cores: Specialized matrix multiply units

- A100: 312 TFLOPS (FP16)
- V100: 125 TFLOPS (FP16)
- T4: 65 TFLOPS (FP16)

VRAM: How big a model we can fit

- A100: 40/80 GB
- V100: 16/32 GB
- T4: 16 GB

CUDA Kernels in Transformers

 **CUDA Kernels:** GPU functions that run in parallel across thousands of threads. Like parallel map operations but at hardware level. Custom kernels can fuse operations for efficiency.


Flash Attention (Dao et al., 2022)

Standard Attention:

- Materialize $S = QK^T$ matrix: $O(n^2)$ memory
- Memory bandwidth bottleneck

Flash Attention:

- Tiled computation in SRAM
- Never materialize full attention matrix
- 2-4× faster, 10-20× less memory

 **Flash Attention:** Optimized attention that computes in GPU SRAM without materializing full attention matrix. Like streaming computation instead of storing intermediate results. Major memory and speed improvement.

Kernel Fusion Examples


Unfused (3 kernel launches)

```
x = layer_norm(x)
x = linear(x)
x = gelu(x)
```

Fused (1 kernel launch)

```
x = fused_norm_linear_gelu(x)  # Custom CUDA kernel
```

Data Parallelism Strategies

 **Data Parallel (DDP):** Each GPU has full model copy, processes different batch samples. Like multiple workers processing different parts of dataset. Gradients averaged across GPUs.

Distributed Data Parallel (DDP)

Each GPU has full model copy

Split batch across GPUs

```
world_size = 4
```

```
per_gpu_batch = total_batch // world_size
```

Gradient synchronization

```
all_reduce(gradients)  # Average gradients across GPUs
```

Gradient Accumulation

Simulate larger batch without more memory

```
accumulation_steps = 4
```

```
for i, batch in enumerate(dataloader):
```

```
    loss = model(batch) / accumulation_steps
```

```
loss.backward()
```

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



Gradient Accumulation: Accumulate gradients over multiple forward passes before updating weights. Like collecting multiple small batches into one large update. Simulates larger batch size without extra memory.

Memory Optimization Techniques

Gradient Checkpointing - Recompute activations during backward pass - 30-50% memory savings - 20-30% slower

Mixed Precision (FP16/BF16) - Half the memory - 2-3× faster on modern GPUs - Requires loss scaling

Fully Sharded Data Parallel (FSDP) - Shards model across GPUs - For models that don't fit on one GPU - Complex but necessary for large models

PART 4: MODEL SELECTION & HYPERPARAMETERS

Section 9: Choosing Your Base Model


⚠ The Specialization Trap

Specialized Models Resist New Knowledge

Real-world example: Training CodeLlama on a new programming language often fails, while base Llama succeeds. Why?

Model Flexibility Spectrum

```
flexibility_ranking = {  
    "Base Models": "██████████ Most Flexible", #  
        Llama-base, Qwen-base  
    "Continued Pretrained": "██████░░░░ Domain Rigid", #  
        CodeLlama, BioMedLM  
    "Instruction-Tuned": "██░░░░░░░░ Nearly Frozen", #  
        ChatGPT, Llama-Instruct  
    "RLHF/DPO Aligned": "██░░░░░░░░ Extremely Rigid", #  
        Claude, GPT-4  
}
```

 **Model Rigidity:** Specialized models have “crystallized” weights highly optimized for their domain. Like concrete that’s already set - hard to reshape. Base models are like clay - still moldable.

Why This Happens: 1. **Continued pretraining** reinforces existing patterns (500B tokens for CodeLlama) 2. **Regularization** during specialization prevents forgetting but also prevents learning 3. **Representation collapse** - internal features become too specialized to repurpose

Decision Framework

```
def choose_base_or_specialized(task_type, domain):  
    """Critical decision: base vs specialized model"""  
  
    if task_type == "novel_patterns":  
        # Learning new language, new format, new domain  
        return "USE_BASE_MODEL" # ← Critical for success  
  
    elif task_type == "extend_existing":  
        # Improving within known domain (Python → better Python)  
        if domain == "code":
```

```

        return "CodeLlama/DeepSeek-Coder"
    elif domain == "medical":
        return "BioMedLM"

    elif task_type == "behavioral_change":
        # Different outputs, new style, new rules
        return "USE_BASE_MODEL" # Avoid instruction-tuned!

```

Real Examples - What Works vs What Fails

Task	Wrong Choice ❌	Right Choice ✓	Why
New programming language	CodeLlama	Llama-base	CodeLlama too rigid for novel syntax
Company-specific Python style	Llama-base	CodeLlama	CodeLlama already knows Python
Medical notes → Legal docs	BioMedLM	Llama-base	Domain shift needs flexibility
Chat → Different personality	Llama-Instruct	Llama-base	Instruction tuning locks behavior
SQL → Novel query syntax	SQLCoder	Qwen-base	Specialized model resists new patterns

Selection Criteria

Use BASE models when: - Learning genuinely new patterns/languages/formats - Changing fundamental behavior - Cross-domain transfer - Your data contradicts model's training

Use SPECIALIZED models when: - Extending within their domain - Your data aligns with their training - You want to preserve specialized knowledge - Fine-tuning with very little data

AVOID instruction-tuned models for fine-tuning unless: - Using specialized techniques (DPO, ORPO) - Only doing prompt engineering - Using them for inference only

For This Workshop

```

# Recommended: Base models for maximum flexibility
RECOMMENDED_MODELS = {
    "primary": "Qwen/Qwen2.5-0.5B", # Base model ✓

```

```

"alternative": "meta-llama/Llama-3.2-1B", # Base model ✓

# AVOID for novel patterns:
# "CodeLlama-7b-Python" ✗ (too specialized)
# "Llama-3.2-1B-Instruct" ✗ (instruction-tuned)
}


# Quick test: Is it a base model?
def is_base_model(model_name):
    red_flags = ["instruct", "chat", "code", "sql", "med", "fin",
                 "rlhf"]
    return not any(flag in model_name.lower() for flag in
                   red_flags)

```

References: - HuggingFace Model Hub: <https://huggingface.co/models> - Qwen Technical Report: <https://arxiv.org/abs/2407.10671> - Llama 3.2 Release: <https://ai.meta.com/blog/llama-3-2>

Section 10: Critical Hyperparameters

Learning Rate & Schedulers


 **Learning Rate:** Step size for gradient descent, controls how much to update weights. Like throttle control - too fast and you overshoot, too slow and training takes forever. Most critical hyperparameter.

Cosine Schedule with Warmup

```

def get_lr(step, warmup_steps=1000, max_steps=10000,
           max_lr=5e-4, min_lr=5e-5):
    # Linear warmup
    if step < warmup_steps:
        return max_lr * step / warmup_steps
    # Cosine decay
    progress = (step - warmup_steps) / (max_steps - warmup_steps)
    return min_lr + 0.5 * (max_lr - min_lr) * (1 + cos(pi *
        progress))

```

 **Warmup:** Gradually increasing learning rate from 0 at start of training. Like warming up an engine before driving. Prevents large unstable updates when model is random.

Typical Values by Model Size

<1B params: lr = 5e-4 to 1e-3
 1-7B params: lr = 1e-4 to 5e-4
 >7B params: lr = 5e-5 to 2e-4

Batch Size Economics

Effective Batch Size

```
effective_batch = micro_batch * gradient_accumulation * num_gpus
```

```
# Examples:
```

```
# 1 GPU, 16GB memory:
```

```
micro_batch = 1
```

```
gradient_accumulation = 32
```

```
effective_batch = 32
```

```
# 8 GPUs with DDP:
```

```
micro_batch = 4
```


```
gradient_accumulation = 4
```


```
num_gpus = 8
```

```
effective_batch = 128
```

Key Hyperparameters Ranked by Impact

Parameter	Impact	Typical Range	Notes
Learning Rate	Critical	1e-5 to 1e-3	Too high = divergence
Batch Size	High	8-512	Larger = more stable
Warmup	High	3-10% of steps	Prevents early divergence
Weight Decay	Medium	0.01-0.1	Regularization
Gradient Clip	Medium	0.5-1.0	Stability
Dropout	Low	0.0-0.1	Often 0 for LLMs
Adam β_2	Low	0.95-0.999	0.95 for long sequences
Adam ϵ	Low	1e-8 to 1e-6	Numerical stability

 **Weight Decay:** L2 regularization that shrinks weights toward zero. Like a penalty for large weights to prevent overfitting. Acts as regularization to improve generalization.

 **Gradient Clipping:** Limiting gradient magnitude to prevent exploding gradients. Like a speed limiter for updates. Essential for stable training of transformers.

Training Stability Tips

```
# Gradient clipping - essential for stability
```

```
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
```

```
# Mixed precision - careful with loss scaling
```

```
from torch.cuda.amp import GradScaler
```

```
scaler = GradScaler(init_scale=2**16, growth_interval=2000)
```

```
# Monitor gradient norms
```

```
grad_norm = sum(p.grad.norm()**2 for p in model.parameters())**0.5
```

```
if grad_norm > 100:
    print("Warning: Large gradients detected!")
```

Training Optimization Strategies

Speed vs Quality Trade-offs

Technique	Memory Savings	Speed Impact	Quality Impact
LoRA	10-100×	Slightly faster	Slightly lower
QLoRA	5× additional	10% slower	Minimal
Flash Attention	10×	2-4× faster	None
Gradient Checkpointing	30-50%	20-30% slower	None
Mixed Precision	2×	2-3× faster	Minimal

Optimization Decision Tree

Memory Constrained?

Yes → QLoRA + Gradient Checkpointing

No → Continue

Time Constrained?

Yes → Larger batch + Mixed Precision + Flash Attention

No → Continue

Quality Critical?

Yes → Full precision + Longer training + Lower LR

No → Standard optimizations

PART 5: DATA PIPELINE

Section 11: Data Pipeline Architecture

Data Preparation Workflow

Collection → Cleaning → Formatting → Tokenization → Loading

Data Sources & Formats

Sources: - Web scraping (Common Crawl, custom) - Existing datasets (HuggingFace, Kaggle) - Proprietary data (company documents) - Synthetic generation (from larger models)

Storage Formats: - **JSONL:** Simple, human-readable, inefficient - **Parquet:** Columnar, compressed, fast - **Arrow:** Memory-mapped, zero-copy - **TFRecord:** TensorFlow native - **WebDataset:** Streaming-optimized

Trade-offs: - Size vs Speed vs Simplicity - Streaming vs In-memory - Random access vs Sequential

Data Formats & Loading

Common Formats

```
# JSONL – Most common for text
{"text": "Example document 1"}
{"text": "Example document 2"}
```

```
# Parquet – Efficient for large datasets
df.to_parquet("data.parquet")
```

```
# HuggingFace Datasets
dataset = load_dataset("json", data_files="data.jsonl")
dataset = load_dataset("parquet", data_files="data.parquet")
dataset = load_dataset("squad") # From hub
```

Tokenization Strategy



Tokenization: Converting text to numbers that models understand. Like compiling source code to bytecode. Critical for model performance and efficiency.

Tokenization Approaches: - On-the-fly: Flexible, slower - Pre-tokenized: Fast, rigid - Cached: Balance of both

```
def tokenize_function(examples, tokenizer, max_length=512):
    """Standard tokenization with padding/truncation"""
    return tokenizer(
        examples["text"],
        padding="max_length",
        truncation=True,
        max_length=max_length,
        return_tensors="pt"
    )

# For conversation data
def format_conversation(examples):
    """Format as: <user>Question</user><assistant>Answer</assistant>"""
    texts = []
    for q, a in zip(examples["question"], examples["answer"]):
        texts.append(f"<user>{q}</user><assistant>{a}</assistant>")
    return {"text": texts}
```

Batching Strategies

- **Dynamic batching:** Memory efficient
- **Fixed batching:** Predictable
- **Bucketing:** Groups similar lengths

Data Quality Checks

```
def validate_dataset(dataset, tokenizer):
    """Pre-training sanity checks"""
    issues = []

    # Check for empty samples
    empty_count = sum(1 for x in dataset if not x["text"].strip())
    if empty_count > 0:
        issues.append(f"Found {empty_count} empty samples")

    # Check token distribution
    lengths = [len(tokenizer.encode(x["text"])) for x in dataset[:100]]
    if max(lengths) > 2048:
        issues.append(f"Very long sequences detected: max={max(lengths)}")

    # Check for duplicates
    texts = [x["text"] for x in dataset[:1000]]
    if len(texts) != len(set(texts)):
        issues.append("Duplicate samples detected")
```

```
    return issues
```

```
# Cleaning pipeline
```

```
def clean_text(text):
```

```
    # Remove excessive whitespace
```

```
    text = " ".join(text.split())
```

```
    # Remove zero-width characters
```

```
    text = text.replace("\u200b", "")
```

```
    # Normalize unicode
```

```
    text = unicodedata.normalize("NFKC", text)
```

```
    return text
```

Efficient Data Loading

```
# Streaming for large datasets
```

```
dataset = load_dataset("json", data_files="huge_file.jsonl",  
                        streaming=True)
```

```
# Data collator for dynamic batching
```

```
from transformers import DataCollatorForLanguageModeling
```

```
data_collator = DataCollatorForLanguageModeling(  
    tokenizer=tokenizer,
```

```
    mlm=False, # False for causal LM
```

```
    pad_to_multiple_of=8 # Optimize for tensor cores
```

```
)
```

```
# Efficient dataloader
```

```
dataloader = DataLoader(  
    dataset,
```

```
    batch_size=batch_size,
```

```
    collate_fn=data_collator,
```

```
    num_workers=4, # Parallel data loading
```

```
    pin_memory=True, # Faster GPU transfer
```


```
)
```

PART 6: MLOPS & TRAINING INFRASTRUCTURE

Section 12: MLOps Overview & Tools Ecosystem

The Training Lifecycle

Development Phase 1. **Experimentation** - Jupyter notebooks, quick iterations 2. **Configuration** - Hyperparameters, model selection 3. **Data Pipeline** - Collection, cleaning, formatting 4. **Training** - Model optimization, checkpointing 5. **Evaluation** - Metrics, quality assessment 6. **Deployment** - Model serving, versioning

 **MLOps:** Machine Learning Operations - practices that combine ML, DevOps, and data engineering. Like DevOps but for model training pipelines. Focuses on reproducibility, scalability, and monitoring.

Key Infrastructure Components


Compute Layer - Local GPUs for prototyping - Cloud GPUs for full training (AWS, GCP, Lambda Labs) - Cluster management (Kubernetes, Slurm) - Cost optimization strategies

Storage Layer - Dataset storage (S3, GCS, local NFS) - Model checkpoints and versioning - Experiment artifacts - Cache management

Orchestration Layer - Experiment tracking - Hyperparameter management - Distributed training coordination - Pipeline automation

Model & Dataset Hubs

HuggingFace - Central repository for pre-trained models - Standardized model APIs - Dataset hosting and versioning - Community discussions and model cards - Alternative: ModelScope (Asia), Civitai (images)

 **Model Hub:** Centralized repository for sharing pre-trained models. Like DockerHub but for neural networks. Handles versioning, licensing, and distribution.

Kaggle - Dataset marketplace - Free GPU compute (30 hrs/week) - Competition platform - Notebook sharing - Alternative: Google Colab, Paperspace Gradient

Experiment Tracking

Weights & Biases (W&B) - Real-time metrics visualization - Hyperparameter tracking - Model versioning - Team collaboration - Alternatives: MLflow, TensorBoard, Neptune.ai, Aim

Trade-offs: - W&B: Best UI, costly at scale - MLflow: Open source, self-hosted - TensorBoard: Simple, limited features - Neptune: Good middle ground

Configuration Management

Hydra - Hierarchical configuration - Command-line overrides - Multi-run sweeps - Config composition - Alternatives: ConfigArgParse, ML Collections, YAML/JSON



Config Composition: Hydra allows mixing configs like LEGO blocks. You can override any parameter from CLI. Makes experimentation systematic and reproducible.

Trade-offs: - Hydra: Powerful but complex - YAML: Simple but no validation - Python configs: Type-safe but less flexible

Section 13: Training Infrastructure

Compute Options Comparison

Provider	Strengths	Weaknesses	Best For
Local	Full control, no limits	High upfront cost	Development
Cloud (AWS/GCP)	Enterprise features	Expensive	Production
GPU Clouds	Cost-effective	Less reliable	Training runs
Colab/Kaggle	Free	Limited time/resources	Prototyping

Distributed Training Strategies

Data Parallel (DP) - Split batch across GPUs - Each GPU has full model - Good for small models - Simple to implement

Distributed Data Parallel (DDP) - Improved version of DP - Better GPU utilization - Standard for multi-GPU

Fully Sharded Data Parallel (FSDP) - Shards model across GPUs - For models that don't fit on one GPU - Complex but necessary for large models

Pipeline Parallel - Split model layers across GPUs - Good for very deep models - Hard to balance



Distributed Training: Spreading training across multiple GPUs/nodes. Like parallel processing but for neural networks. Critical for large models and datasets.

Hands-On Lab: Project Structure

```

hackathon/
├── configs/                                # Hydra configuration files
│   ├── train.yaml                        # Training configuration
│   ├── eval.yaml                        # Evaluation configuration
│   └── eval_baseline.yaml                # Baseline model evaluation
├── data/                                  # Training and evaluation data
│   ├── *.jsonl                          # Training data files
│   └── eval/                             # Evaluation data
│       └── test.jsonl
├── src/                                  # Source code
│   ├── main.py                          # CLI entry point (Typer)
│   ├── train/
│   │   └── pipeline.py                  # Training orchestration
│   ├── eval/
│   │   └── evaluator.py                 # Evaluation framework
│   ├── data/
│   │   ├── loader.py                   # Dataset loading
│   │   └── sources.py                   # Data source abstraction
│   └── deploy/
│       └── push.py                      # Model publishing
├── outputs/                              # Training outputs
│   └── runs/
│       ├── 2024-01-15_10-30-45/
│       │   ├── model/                   # Saved model + tokenizer
│       │   ├── config_used.yaml
│       │   └── logs/                    # TensorBoard logs
│       └── latest -> ...                 # Symlink to latest run
├── Makefile                              # Command shortcuts
└── requirements.txt                      # Python dependencies

```

Training Configuration (configs/train.yaml)

```

# Model
model_name: meta-llama/Llama-3.1-8B

# Data
dataset_path: ./data
dataset_pattern: "*.jsonl"
data_format: prompt_completion # or "messages"

# Training Hyperparameters
epochs: 5
batch_size: 4
lr: 2e-4
gradient_accumulation_steps: 2 # Effective batch = 8
max_length: 512

```

```
# Learning Rate Schedule
warmup_ratio: 0.03
weight_decay: 0
lr_scheduler_type: cosine

# Precision (use fp16 for older GPUs, bf16 for modern)
bf16: false
fp16: true

# LoRA Configuration
lora_r: 64
lora_alpha: 128
lora_dropout: 0.05
lora_target_modules:
  - q_proj
  - k_proj
  - v_proj
  - o_proj
  - gate_proj
  - up_proj
  - down_proj

# Memory Optimization
gradient_checkpointing: false

# Logging
logging_steps: 20
save_steps: 100
```

Quick Start Commands

```
# 1. Setup environment
make venv && source .venv/bin/activate
make install

# 2. Prepare data (if needed)
make convert # Convert ChatGPT format to JSONL

# 3. Train model
make train

# 4. Evaluate fine-tuned model
make eval

# 5. Compare with baseline
make eval-baseline

# 6. Publish to HuggingFace
make publish-hf
```

Training Pipeline (src/train/pipeline.py)

```
def run_training(cfg):
    """Fine-tune a model using LoRA."""

    # 1. Create timestamped run directory
    run_dir = create_run_dir() # outputs/runs/
    2024-01-15_10-30-45/
    OmegaConf.save(config=cfg, f=str(run_dir /
    "config_used.yaml"))

    # 2. Load dataset from JSONL files
    dataset = data_source.load_dataset(data_pattern, config=cfg)

    # 3. Tokenize with proper prompt/completion handling
    def tokenize(example):
        # Build input: BOS + prompt + newline + completion + EOS
        # Labels: mask prompt tokens with -100
        prompt_ids = tokenizer(prompt + "\n",
        add_special_tokens=False).input_ids
        completion_ids = tokenizer(completion,
        add_special_tokens=False).input_ids

        labels = [-100] * len(prompt_ids) + completion_ids
        # Only learn completion!
        return {"input_ids": input_ids, "labels": labels, ...}

    # 4. Load base model with LoRA
    base_model = AutoModelForCausalLM.from_pretrained(
        cfg.model_name,
        torch_dtype=torch.float16,
        device_map="auto",
    )

    lora_cfg = LoraConfig(
        r=cfg.lora_r, # 64 = high capacity
        lora_alpha=cfg.lora_alpha, # 128 = 2x rank
        lora_dropout=cfg.lora_dropout,
        target_modules=cfg.lora_target_modules,
    )
    model = get_peft_model(base_model, lora_cfg)
    model.print_trainable_parameters() # Shows ~1-2% trainable

    # 5. Train with HuggingFace Trainer
    trainer = Trainer(
        model=model,
        args=TrainingArguments(
            output_dir=str(run_dir),
            num_train_epochs=cfg.epochs,
            per_device_train_batch_size=cfg.batch_size,
            learning_rate=cfg.lr,
            report_to=["tensorboard"],
            ...
```

```

    ),
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
)
trainer.train()

# 6. Save model + update symlink
model.save_pretrained(run_dir / "model")
tokenizer.save_pretrained(run_dir / "model")
_update_latest_symlink(run_dir)
    # outputs/runs/latest -> this run

```

Evaluation Framework (src/eval/evaluator.py)

```

def run_evaluation(cfg):
    """Evaluate model with multiple metrics."""

    # Load model (handles both LoRA adapters and merged models)
    model = AutoPeftModelForCausalLM.from_pretrained(model_path)

    # Evaluation loop
    for example in dataset:
        prompt = example["prompt"]
        expected = example["completion"]

        # Generate prediction
        input_ids = build_eval_input(prompt, tokenizer, device,
                                       max_len=4096)
        output = model.generate(input_ids, max_new_tokens=256)
        generated = tokenizer.decode(output[0],
                                      skip_special_tokens=True)

        # Collect metrics
        exact_matches += int(generated.lower() ==
                              expected.lower())
        overlaps.append(jaccard_overlap(expected, generated))

    # Compute ROUGE/BLEU scores
    rouge = evaluate.load("rouge")
    bleu = evaluate.load("bleu")

    metrics = {
        "exact_match": exact_matches / total,
        "avg_jaccard_overlap": sum(overlaps) / total,
        "rouge1": rouge_scores["rouge1"],
        "bleu": bleu_scores["bleu"],
    }

```

Data Formats

Prompt/Completion Format (Recommended)

```
{"prompt": "What is FLOWROLL?", "completion": "FLOWROLL is a  
rolling aggregation..."}
```

Messages Format (Chat/Conversational)

```
{"messages": [  
  {"role": "user", "content": "What is FLOWROLL?"},  
  {"role": "assistant", "content": "FLOWROLL is a rolling  
aggregation..."}  
]}
```

Monitoring Training

```
# Watch TensorBoard  
tensorboard --logdir outputs/runs/latest/logs  
  
# Healthy signs:  
# - Loss decreasing steadily  
# - Grad norm stable (slight increase is normal)  
# - No sudden spikes  
  
# Example output:  
# {'loss': 2.73, 'epoch': 0.25} # Starting  
# {'loss': 1.55, 'epoch': 1.0} # Decreasing  
# {'loss': 0.61, 'epoch': 5.0} # Good progress
```

Model Publishing

```
# Set credentials  
export HF_TOKEN="hf_XXXXXXXXXXXXX"  
export HF_REPO_ID="username/model-name"  
  
# Merge LoRA into base model (optional)  
make merge  
  
# Convert to GGUF for Ollama/llama.cpp  
make gguf QUANT=Q5_K_M  
  
# Push to HuggingFace  
make publish-hf
```

PART 7: EVALUATION, MONITORING & DEPLOYMENT

Section 14: Evaluation & Validation

Evaluation Strategies

Automatic Metrics - **Perplexity**: Lower is better, measures prediction confidence - **BLEU/ROUGE**: Translation/summarization quality - **Task-specific**: Accuracy, F1, Exact Match - **Loss curves**: Training dynamics visualization

Human Evaluation - Quality assessment (1-5 scale ratings) - Safety checking - Edge case testing - A/B comparisons

Benchmark Suites - MMLU (knowledge across 57 subjects) - HumanEval (coding - 164 problems) - BBH (reasoning - 23 hard tasks) - Custom benchmarks for your use case

Validation Best Practices

Data Splits - Train: 80-90% - Validation: 5-10% - Test: 5-10% - Ensure no leakage between splits

Overfitting Detection - Monitor validation loss vs training loss - Track train-val gap (growing gap = overfitting) - Use early stopping (patience = 3-5 evals) - Regular checkpointing (keep best K by val loss)

Quality Assurance Checklist - Generate diverse sample outputs - Test edge cases and adversarial inputs - Compare to baseline/previous version - Check for capability regressions - Verify no training data memorization

Section 15: Monitoring & Debugging

What to Monitor

Training Metrics - Loss curves (training/validation) - Learning rate schedules - Gradient norms - Parameter distributions

System Metrics - GPU utilization (target: 90%+) - Memory usage (near max = good) - Temperature (<85°C safe) - I/O bottlenecks

Model Quality - Perplexity - Task-specific metrics - Generation samples - Overfitting indicators

Common Issues & Diagnosis

Problem	Symptoms	Likely Causes	Solution
Loss = NaN	Training crashes	LR too high, bad data	Reduce LR by 10x
Loss not decreasing	Flat curve	LR too low, poor init	Check data, increase LR
OOM Errors	CUDA memory	Batch/model too big	Reduce batch, use gradient checkpointing
Loss spikes	Sudden jumps	Bad data samples	Add gradient clipping
Slow training	Low GPU util	I/O bottleneck	Increase dataloader workers
Eval worse than train	Growing gap	Overfitting	Add dropout, reduce epochs

Debugging Tools

GPU Monitoring

```
# Real-time GPU stats
watch -n 1 nvidia-smi
```

```
# Better TUI alternatives
nvidia-smi
gpustat
```

Gradient Debugging

```
def check_gradients(model):
    """Log gradient statistics"""
    total_norm = 0
    for p in model.parameters():
        if p.grad is not None:
            param_norm = p.grad.data.norm(2)
            total_norm += param_norm.item() ** 2

    total_norm = total_norm ** 0.5

    if total_norm > 100:
        print(f"⚠ Large gradient norm: {total_norm}")

    return total_norm
```

Sample Output Debugging

```
def generate_samples(model, tokenizer, prompts, step):
    """Generate and log sample outputs"""
    model.eval()
    with torch.no_grad():
        for i, prompt in enumerate(prompts):
            inputs = tokenizer(prompt, return_tensors="pt")
            outputs = model.generate(**inputs, max_length=100)
            text = tokenizer.decode(outputs[0],
                                   skip_special_tokens=True)
            print(f"Step {step} - Prompt {i}: {text}")
    model.train()
```

Performance Profiling

```
# PyTorch profiler for bottleneck detection
from torch.profiler import profile, ProfilerActivity

with profile(
    activities=[ProfilerActivity.CPU, ProfilerActivity.CUDA],
    record_shapes=True,
    profile_memory=True
) as prof:
    for batch in dataloader:
        output = model(batch)
        loss = output.loss
        loss.backward()

# Print slowest operations
print(prof.key_averages().table(sort_by="cuda_time_total",
                                row_limit=10))
```

Section 16: Production Deployment

Deployment Options

Inference Frameworks - **vLLM**: High throughput, PagedAttention, OpenAI-compatible API - **TGI (Text Generation Inference)**: HuggingFace solution, Flash Attention - **Ollama**: Local-first, easy setup, works on laptop - **llama.cpp**: Pure C++, CPU inference, edge/embedded

User Interfaces - **Open WebUI**: Self-hosted ChatGPT-like interface, works with Ollama - **text-generation-webui**: Feature-rich, many backends - **LM Studio**: Desktop app, easy model management - **Jan.ai**: Privacy-focused, offline-first

Managed Platforms - **Hyperstac.cloud**: Managed GPU cloud, pay-per-use - **Replicate**: Serverless, easy deployment - **Modal**: Python-native, auto-scaling - **RunPod**: GPU marketplace - **Baseten**: Enterprise features

Production Considerations

Performance Optimization - Continuous batching (2-4× throughput) - KV cache management - Tensor parallelism for large models - Speculative decoding (2-3× speed)

Monitoring Requirements - Latency tracking (p99, not average!) - Throughput metrics (tokens/sec) - Error rates - Cost per token

Safety & Compliance - Content filtering - Rate limiting - Audit logging - Privacy compliance

Scaling Strategies

Vertical Scaling - Bigger GPUs (T4 → A10G → A100 → H100) - More VRAM - Simple but expensive at top

Horizontal Scaling - Multiple replicas - Load balancing (round-robin, least-connections, latency-based) - Auto-scaling triggers (queue depth, GPU util, latency p99)

Model Optimization for Serving - Quantization (GPTQ, AWQ, GGUF) - Distillation (train smaller model to mimic large) - Pruning (remove low-importance weights) - Caching (prompt, semantic, response)

Deployment Decision Matrix

Scenario	Recommended	Why
Local dev/testing	Ollama	Easy setup, works on laptop
Team/company chatbot	Ollama + Open WebUI	Self-hosted, multi-user, private
High-traffic production	vLLM / TGI	Optimized throughput, batching
Edge/mobile/embedded	llama.cpp	Minimal deps, CPU inference
Serverless/scale-to-zero	Replicate / Modal	Pay only for usage

Appendix: Quick Reference

Key Formulas

Attention

$$\text{Attention}(Q, K, V) = \text{softmax}(QK^T/\sqrt{d_k})V$$

LoRA

$$W' = W_0 + BA \quad (\text{where } r \ll d)$$

Perplexity

$$\text{PPL} = \exp(\text{avg_loss})$$

Memory (approximate)

Model: $\text{params} \times \text{bytes_per_param}$

Optimizer: 2-3× model size (Adam states)

Activations: $\text{batch} \times \text{seq} \times \text{hidden} \times \text{layers}$

Recommended Models for Workshop

Model	Size	VRAM (FP16)	VRAM (QLoRA)
Qwen2.5-0.5B	0.5B	~1GB	~0.5GB
Llama-3.2-1B	1B	~2GB	~1GB
Qwen2.5-1.5B	1.5B	~3GB	~1.5GB
Llama-3.2-3B	3B	~6GB	~2GB

Essential Commands

Training

```
make train TRAIN_CONFIG=train
```

Evaluation

```
make eval
```

Merge LoRA weights

```
make merge
```

Convert to GGUF (for Ollama)

```
make gguf
```

Push to HuggingFace

```
make publish-hf
```

```
# Monitor GPU
```

```
watch -n 1 nvidia-smi
```

```
# View training logs
```

```
tensorboard --logdir outputs/runs
```

Learning Resources

Books

Foundational - *Deep Learning* by Ian Goodfellow, Yoshua Bengio, Aaron Courville — The “bible” of deep learning, free online at deeplearningbook.org - *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow* by Aurélien Géron — Practical ML from basics to production - *Neural Networks and Deep Learning* by Michael Nielsen — Free online, excellent intuition building

NLP & Transformers - *Natural Language Processing with Transformers* by Lewis Tunstall, Leandro von Werra, Thomas Wolf — HuggingFace team, very practical - *Speech and Language Processing* by Dan Jurafsky & James Martin — Comprehensive NLP textbook, free online drafts

LLM-Specific - *Build a Large Language Model (From Scratch)* by Sebastian Raschka — Step-by-step LLM implementation - *Designing Machine Learning Systems* by Chip Huyen — MLOps and production considerations

Online Courses

Free - *fast.ai Practical Deep Learning* — Top-down approach, highly recommended for practitioners - *Stanford CS224N: NLP with Deep Learning* — Excellent theory, lectures on YouTube - *Stanford CS229: Machine Learning* — Andrew Ng’s foundational course - *Andrej Karpathy’s Neural Networks: Zero to Hero* — YouTube series, builds GPT from scratch - *HuggingFace NLP Course* — Free, practical, covers transformers end-to-end

Paid - *DeepLearning.AI Specializations* — Andrew Ng, well-structured courses - *Full Stack LLM Bootcamp* — Practical LLM deployment focus

Blogs & Newsletters

Technical Deep Dives - *Lilian Weng’s Blog* (lilianweng.github.io) — Exceptional technical summaries of ML topics - *Jay Alammar’s Blog* (jalammar.github.io) — Visual explanations of transformers, attention - *Sebastian Raschka’s Blog* — LLM research summaries, practical tips - *The Gradient* — Research paper summaries - *Chip Huyen’s Blog* — MLOps, production ML

Newsletters - *The Batch* by DeepLearning.AI — Weekly AI news summary - *Import AI* by Jack Clark — AI research and policy - *Last Week in AI* — Comprehensive weekly roundup - *Ahead of AI* by Sebastian Raschka — LLM-focused research digest

People to Follow

Researchers & Practitioners - **Andrej Karpathy** (@karpathy) — Former Tesla AI, OpenAI, exceptional educator - **Sebastian Raschka** (@rasaborsch) — LLM researcher, author, practical tutorials - **Chip Huyen** (@chipro) — MLOps, production ML systems - **François Chollet** (@fchollet) — Keras creator, deep learning philosophy - **Yann LeCun** (@ylecun) — Meta AI Chief Scientist, foundational contributions - **Ilya Sutskever** (@ilyasut) — OpenAI co-founder, scaling insights

HuggingFace Team - **Lewis Tunstall** — Transformers book author - **Leandro von Werra** — TRL (training) library - **Thomas Wolf** (@Thom_Wolf) — HuggingFace co-founder

MLOps & Engineering - **Hamel Husain** (@HamelHusain) — Practical ML engineering - **Eugene Yan** (@eugeneyan) — ML systems, applied ML

Key Papers

Foundational Transformers - *Attention Is All You Need* (Vaswani et al., 2017) — The transformer architecture - *BERT: Pre-training of Deep Bidirectional Transformers* (Devlin et al., 2019) - *Language Models are Unsupervised Multitask Learners* (GPT-2, Radford et al., 2019)

Scaling & Training - *Scaling Laws for Neural Language Models* (Kaplan et al., 2020) — How scale affects performance - *Training Compute-Optimal Large Language Models* (Chinchilla, Hoffmann et al., 2022)

Efficient Fine-tuning - *LoRA: Low-Rank Adaptation of Large Language Models* (Hu et al., 2021) - *QLoRA: Efficient Finetuning of Quantized LLMs* (Dettmers et al., 2023) - *LLM.int8(): 8-bit Matrix Multiplication for Transformers* (Dettmers et al., 2022)

Instruction Tuning & RLHF - *Training language models to follow instructions with human feedback* (InstructGPT, Ouyang et al., 2022) - *Constitutional AI: Harmlessness from AI Feedback* (Anthropic, 2022)

Communities & Forums

- **HuggingFace Forums** — Model training, debugging help
- **r/LocalLLaMA** — Self-hosted LLM community, practical tips
- **r/MachineLearning** — Research discussions
- **EleutherAI Discord** — Open-source LLM development
- **MLOps Community Slack** — Production ML discussions

Tools Documentation

Essential Reading - HuggingFace Transformers docs — transformers.readthedocs.io - PEFT (LoRA) docs — huggingface.co/docs/peft - PyTorch tutorials — pytorch.org/tutorials - Weights & Biases guides — docs.wandb.ai

Deployment - vLLM documentation — vllm.readthedocs.io - Ollama docs — ollama.com - llama.cpp wiki — github.com/ggerganov/llama.cpp

Last updated: December 2025