**Aditya's Modern LLM + Memory Research Course**

**PhD-Level Content with Consolidated Lectures**

**🎯 PART 1: MODERN TRANSFORMER ARCHITECTURE (Weeks 1-3)**

**Week 1: Complete Attention Mechanisms (2 hours)**

**Single Deep Lecture: "Everything About Attention in LLMs 2025"**

**What You'll Learn:**

1. **~~Self-Attention Fundamentals (30 min)~~**
   * **~~Q, K, V mathematics from first principles~~**
   * **~~Attention score computation: softmax(QK^T/√d)~~**
   * **~~Why scaled dot-product (√d) prevents vanishing gradients~~**
   * **~~Computational complexity: O(n²) space, O(n²d) time~~**
   * **~~Memory bottleneck analysis for long sequences~~**
2. **Multi-Head Attention (25 min)**
   * **~~Why multiple heads (different representation subspaces)~~**
   * **~~Head specialization: what different heads learn~~**
   * **~~Concatenation and linear projection~~**
   * **~~Trade-off: d\_model vs num\_heads~~**
   * **~~Implementation details & optimization~~**
3. **Modern Attention Variants (40 min)**
   * **GQA (Grouped-Query Attention): Llama 2/3 approach** 
     + **Reduces KV heads from h to g groups**
     + **Memory savings: KV cache ≈ 2x smaller**
     + **Quality trade-offs with full MQA**
   * **MQA (Multi-Query Attention): Single KV head** 
     + **Extreme compression but quality loss**
     + **When to use (inference-only scenarios)**
   * **MLA (Multi-Head Latent Attention): DeepSeek innovation** 
     + **Latent dimension compression**
     + **KV cache reduction to O(d\_latent) per token**
     + **Projection matrices for compression**
   * **Sliding Window Attention: Mistral/Llama approach** 
     + **Local attention windows (4k tokens)**
     + **Memory: O(w\*d) per position (w = window size)**
     + **Combines local + global through stacking**
   * **Sparse Attention Patterns** 
     + **Strided attention**
     + **Block-sparse patterns**
     + **Limitations and when to use**
4. **Flash Attention Evolution (25 min)**
   * **Flash Attention 1: IO-aware algorithm** 
     + **GPU memory hierarchy: HBM (slow) vs SRAM (fast)**
     + **Tiling strategy for sequential I/O**
     + **Online softmax recomputation**
     + **4x speedup with lower VRAM**
   * **Flash Attention 2: Work partitioning improvements** 
     + **Inner-loop reductions**
     + **Warp-level parallelization**
     + **10x+ speedup over vanilla**
   * **Flash Attention 3: Latest (2024)** 
     + **Producer-consumer overlap**
     + **H100-specific optimizations**
     + **8x speedup potential**
   * **Implementation considerations and integration**

**Math Deep Dive:**

* **Softmax stability: exp(x - max(x))**
* **Gradient flow through attention**
* **Proving attention is O(n²) unavoidable for full attention**

**Deliverables:**

* **modern\_attention.py: All variants implemented**
* **attention\_complexity\_analysis.py: Memory/computation profiling**
* **Research papers: Vaswani et al. (2017), GQA, MLA papers, Flash Attention 1-3**

**Key Research Questions:**

* **How much quality loss with GQA vs MQA?**
* **Optimal window size for sliding window?**
* **Can we do better than O(n²)?**

**Week 2: Positional Encodings Deep Dive (2 hours)**

**Single Deep Lecture: "Position Information in Modern LLMs"**

**What You'll Learn:**

1. **Why Positions Matter (20 min)**
   * **Attention is permutation-invariant (fundamental problem)**
   * **Relative vs absolute positions**
   * **Extrapolation beyond training context**
   * **Position bias in long sequences**
2. **RoPE: Rotary Position Embeddings (35 min)**
   * **Geometric intuition: rotating query/key vectors**
   * **Mathematics: rotation matrices in 2D subspaces**
   * **Relative position encoding property: exp(im(θ\_j - θ\_i))**
   * **Why RoPE enables extrapolation**
   * **Frequency choices and scaling**
   * **Implementation: complex number representation**
   * **Llama 2/3 configuration (base=10000, dim=128)**
3. **ALiBi: Attention with Linear Biases (25 min)**
   * **No position embeddings needed**
   * **Linear bias: attention\_scores += bias \* (-1)^(i-j)**
   * **Train short (2k tokens), test long (8k+ tokens)**
   * **Extrapolation properties vs RoPE**
   * **Llama 1 approach, trade-offs**
4. **YaRN: Context Extension (30 min)**
   * **RoPE frequency scaling for long contexts**
   * **NTK-aware interpolation**
   * **Extending from 4k → 128k tokens**
   * **Formula: θ\_i' = θ\_i \* scaling\_factor**
   * **Perplexity preservation on extended context**
   * **Parameter tuning: α, β values**
5. **NoPE: No Positional Encodings (15 min)**
   * **SmolLM3 approach**
   * **Implicit position learning**
   * **When this works (and doesn't)**
   * **Context length limitations**
6. **Comparison & Trade-offs (15 min)**
   * **RoPE: Strong extrapolation, standard choice**
   * **ALiBi: Simple, works but extrapolation limited**
   * **YaRN: Best for extending context**
   * **Empirical benchmarks**

**Advanced Topics:**

* **Position interpolation vs frequency scaling**
* **Attention patterns at different positions**
* **How models use position information**

**Deliverables:**

* **position\_encodings.py: All implementations**
* **position\_extrapolation\_test.py: Benchmark script**
* **rope\_math\_derivation.py: From-scratch explanation**
* **Papers: RoPE, ALiBi, YaRN original works**

**Research Gaps:**

* **Optimal position scaling formula?**
* **Why does RoPE extrapolate better theoretically?**
* **Can we design positions for specific tasks?**

**Week 3: Modern Architecture Components (2 hours)**

**Single Deep Lecture: "Building Stable, Efficient Transformers"**

**What You'll Learn:**

1. **Normalization Strategies (30 min)**
   * **LayerNorm: Original normalization** 
     + **Statistics per sample, per feature**
     + **Learnable scale and shift (γ, β)**
     + **Numerical stability tricks**
   * **RMSNorm: Simpler, faster (Llama, Mistral)** 
     + **Root mean square normalization**
     + **2x faster than LayerNorm**
     + **Empirical equivalence with proper tuning**
   * **Pre-Norm vs Post-Norm** 
     + **Pre-norm: norm before sublayer (training stability)**
     + **Post-norm: norm after sublayer (better for large models)**
     + **Modern trend: pre-norm for easier training**
   * **Placement in residual connections**
2. **Advanced Feed-Forward Networks (25 min)**
   * **Standard FFN: Linear → ReLU → Linear**
   * **SwiGLU: Gated linear unit variant** 
     + **(xW + b) ⊗ (xV + c), where ⊗ is element-wise product**
     + **Llama 2+ standard**
     + **Why gating helps (feature selection)**
   * **GeGLU: Gaussian error linear unit variant**
   * **MoE-FFN Preview: Sparse routing to expert FFNs**
   * **Dimension choices: hidden\_dim = 4 \* d\_model typically**
3. **Residual Connections & Gradient Flow (20 min)**
   * **Why residuals matter (skip connections)**
   * **Gradient flow through deep networks**
   * **f(x) = x + sublayer(x) properties**
   * **Scaling residuals: importance of initialization**
   * **muP (maximal update parameterization)** 
     + **Hyperparameter transfer across model sizes**
     + **Residual scaling by √N (number of residual branches)**
4. **Complete Transformer Block (15 min)**
   * **Layer ordering: LN → Attention → Residual → LN → FFN → Residual**
   * **Gradient flow analysis through blocks**
   * **Depth scaling considerations**
5. **Decoder-Only Philosophy (15 min)**
   * **Why decoder-only won (vs encoder-decoder, encoder-only)**
   * **Causal masking for autoregressive generation**
   * **GPT vs BERT architecture differences**
   * **Bidirectional attention (encoder) vs causal (decoder)**
6. **Model Initialization & Stability (15 min)**
   * **Xavier/Glorot initialization for linear layers**
   * **Special initialization for embedding layers**
   * **Residual scaling: weight\_init \*= 1/√N**
   * **muP initialization strategies**
   * **Temperature scaling for stability**

**Deep Dives:**

* **Gradient flow through 100+ layer networks**
* **Why scaling matters: variance of activations**
* **Numerical stability in softmax**

**Deliverables:**

* **modern\_transformer.py: Complete, optimized implementation**
* **gradient\_flow\_analysis.py: Visualization and profiling**
* **stability\_test.py: Training with different initializations**
* **Papers: Post-LN, Pre-LN analysis, muP**

**Key Experiments:**

* **Train models with different norm types**
* **Measure gradient variance across layers**
* **Compare with/without residual scaling**

**🔥 PART 2: TRAINING & OPTIMIZATION (Weeks 4-6)**

**Week 4: Training Fundamentals (2.5 hours)**

**Single Deep Lecture: "Modern Training: Optimizers, Schedules, and Stability"**

**What You'll Learn:**

1. **Modern Optimizers (45 min)**
   * **SGD with Momentum: Baseline**
   * **Adam & AdamW: Standard (2014)** 
     + **First moment (mean): m\_t = β₁\*m\_{t-1} + (1-β₁)\*g\_t**
     + **Second moment (variance): v\_t = β₂\*v\_{t-1} + (1-β₂)\*g\_t²**
     + **Update: θ\_t = θ\_{t-1} - α\*m̂\_t/(√v̂\_t + ε)**
     + **Why AdamW (decoupled weight decay) is better**
   * **Lion Optimizer: Modern alternative** 
     + **Sign-based update: θ ← θ - α\*sign(m\_t)**
     + **Lower memory than Adam**
     + **10% faster training observed**
   * **Sophia: Hessian-based (recent)** 
     + **Second-order information for better convergence**
     + **When to use (large models benefit most)**
   * **Empirical comparisons and trade-offs**
2. **Learning Rate Schedules (40 min)**
   * **Warmup: Why it matters** 
     + **Linear warmup from 0 to max\_lr over N steps**
     + **Critical for stability, especially large batch**
     + **Typical: 1-10% of training**
   * **Cosine Annealing: Standard approach** 
     + **LR = min\_lr + 0.5\*(max\_lr - min\_lr)\*(1 + cos(πt/T))**
     + **Smooth decay to minimum**
   * **Constant LR with Warmup: Llama approach** 
     + **Simpler, often works as well**
   * **Linear Decay: Simple baseline**
   * **WSD (Warmup-Stable-Decay): Modern** 
     + **Warmup → stable phase → decay**
     + **Better for very long training**
   * **Cyclical schedules and restarts**
   * **Practical schedule choice guidelines**
3. **Gradient Clipping & Stability (20 min)**
   * **Global Gradient Clipping: By norm** 
     + **||g|| > clip\_value → g = g \* (clip\_value / ||g||)**
     + **Prevents explosion in large learning rates**
   * **Per-Parameter Clipping: Element-wise**
   * **Gradient Norm Monitoring: Early warning signs**
   * **Loss instability detection**
   * **When to increase/decrease clip value**
4. **Mixed Precision Training (25 min)**
   * **FP32 (full precision): Baseline**
   * **FP16 (half precision): GPU native, 2x memory savings** 
     + **But: underflow/overflow problems**
     + **Solution: Loss scaling**
   * **BF16 (brain float): Wider exponent** 
     + **More stable than FP16**
     + **Nvidia, AMD support**
     + **Google TPU standard**
   * **Loss Scaling: Multiply loss by scale factor before backward** 
     + **Prevents gradient underflow**
     + **Dynamic scaling (adjust based on overflow)**
   * **Memory vs stability trade-offs**
   * **When to use each (model size, hardware)**
5. **Batch Size & Gradient Accumulation (20 min)**
   * **Effective Batch Size: accumulation\_steps \* batch\_per\_gpu \* num\_gpus**
   * **Gradient Accumulation: Simulate large batch on small memory** 
     + **No backward until N steps → one sync**
     + **Learning rate scaling considerations**
   * **Critical Batch Size: When further increase hurts** 
     + **Often 10M+ tokens for large LLMs**
   * **Batch Size Scaling: Effect on learning dynamics** 
     + **Larger batch → smoother gradients but less noise**
     + **Optimal: trade-off between gradient noise and compute**
6. **Complete Training Loop (15 min)**
   * **Checkpointing strategy**
   * **Validation during training**
   * **Early stopping criteria**
   * **Logging (WandB integration)**
   * **Monitoring key metrics**

**Advanced Topics:**

* **Gradient accumulation correctness**
* **Loss scale scheduling**
* **When to use which optimizer**

**Deliverables:**

* **training\_modern.py: Complete training loop**
* **optimizer\_comparison.py: Benchmark all optimizers**
* **schedule\_visualizer.py: Visualize LR schedules**
* **Papers: Adam, AdamW, Lion, Sophia**

**Key Experiments:**

* **Train same model with different optimizers**
* **Impact of warmup on stability**
* **FP16 vs BF16 convergence**

**Week 5: Scaling Laws & Compute (2 hours)**

**Single Deep Lecture: "Chinchilla Laws: How to Train Optimally"**

**What You'll Learn:**

1. **Compute Optimal Training (40 min)**
   * **Chinchilla Scaling Laws (DeepMind 2022)** 
     + **Historically: scale parameters more than tokens (Kaplan et al.)**
     + **Chinchilla: equal scaling is optimal**
     + **Formula: tokens ≈ 20 × parameters**
     + **Example: 70B parameters → 1.4T tokens**
   * **Optimal Allocation: C\_compute = 6PD where C = compute budget** 
     + **Solve to minimize loss: C = 6PD, optimal when d(loss)/dP = d(loss)/dD**
   * **Undertraining vs Overtraining** 
     + **Undertrain: too many parameters, too few tokens (high loss)**
     + **Overtrain: too few parameters, too many tokens (redundant)**
   * **FLOPs Estimation:** 
     + **Training FLOPs ≈ 6 \* num\_params \* num\_tokens**
     + **Inference FLOPs ≈ 2 \* num\_params \* seq\_length**
2. **Data Scaling (30 min)**
   * **Data Quantity vs Quality** 
     + **More data > better data (usually)**
     + **But quality matters enormously**
     + **Deduplication can help quality**
   * **Data Mixing Ratios** 
     + **Not uniform across domains**
     + **Optimal: code > math > reasoning > general**
     + **Domain-specific ratios for specialized models**
   * **Deduplication Impact** 
     + **Near-duplicate removal improves quality**
     + **Typical savings: 5-15% of tokens**
   * **Data Curriculum** 
     + **Start with easier/higher-quality data**
     + **Gradually mix harder data**
     + **Can improve final performance**
3. **Model Scaling Predictions (25 min)**
   * **Loss Prediction Formulas** 
     + **L(N,D) = α/N^β + γ/D^δ (power law form)**
     + **α, β, γ, δ are fitted constants**
     + **β ≈ 0.07, δ ≈ 0.1 empirically**
   * **Extrapolation: Predict loss at unseen scale**
   * **When to Stop Training: Loss plateau detection**
   * **Scaling Experiment Design** 
     + **Train small models to convergence**
     + **Fit curves, extrapolate**
     + **Saves massive compute**
4. **Compute-Optimal Training (20 min)**
   * **Budget Allocation** 
     + **50% to compute, 50% to data (rough)**
     + **Or: maximize loss reduction per FLOPs**
   * **Parameter Selection: Given compute budget** 
     + **Solve C = 6PD for optimal P, D**
   * **Token Count Planning: From parameters**
5. **Post-Training (Inference-Time) Compute (25 min)**
   * **Test-Time Scaling: o1-style thinking** 
     + **Allocate compute at inference**
     + **More tokens for harder problems**
   * **Scaling Beyond Training: Can improve accuracy**
   * **When to use: Higher-stakes applications**
   * **Trade-off: latency vs accuracy**
6. **Building Calculator Tools (10 min)**
   * **Compute budget → optimal P, D**
   * **FLOPs estimation**
   * **Timeline predictions**

**Math Deep Dives:**

* **Deriving optimal allocation from Lagrange multipliers**
* **Fitting power laws to experimental data**
* **Sensitivity analysis**

**Deliverables:**

* **scaling\_laws.py: Calculator and predictor**
* **scaling\_experiments.py: Design and run scaling studies**
* **compute\_budget\_optimizer.py: Given budget, find P and D**
* **Papers: Chinchilla, Kaplan et al., compute optimal training**

**Key Experiments:**

* **Fit power laws on your own data**
* **Verify 20:1 token-to-param ratio**
* **Test allocation formulas**

**Week 6: Data Processing at Scale (2.5 hours)**

**Single Deep Lecture: "Production Data Pipeline: Collection to Tokenization"**

**What You'll Learn:**

1. **Data Collection (40 min)**
   * **Web Sources** 
     + **CommonCrawl: Massive but noisy**
     + **FineWeb-Edu: Higher quality subset**
     + **Specific domains: GitHub (code), ArXiv (papers)**
   * **Scraping Infrastructure** 
     + **Scrapy framework**
     + **Rate limiting and politeness**
     + **Error handling and retries**
     + **Distributed scraping architecture**
   * **Licensed Data** 
     + **Book corpora (careful licensing)**
     + **News archives**
     + **Academic papers (arXiv)**
   * **Data Rights: Respect licensing, copyright**
   * **Scale: Typical LLM uses 5-20TB raw data**
2. **Quality Filtering (35 min)**
   * **Heuristic Filters** 
     + **Line length distribution**
     + **Punctuation ratio**
     + **Language detection**
     + **Remove near-duplicates before processing**
   * **Perplexity Filtering** 
     + **Train reference LM on clean data**
     + **Filter high-perplexity documents**
     + **Threshold selection crucial**
   * **Classifier-Based Filtering** 
     + **Train binary classifier (good/bad)**
     + **Apply to entire corpus**
     + **Can be expensive at scale**
   * **URL-Based Filtering** 
     + **Blacklist known low-quality domains**
     + **Whitelist trusted sources**
   * **Content Filtering** 
     + **Remove gibberish (character n-grams)**
     + **Remove ads, boilerplate**
     + **Remove PII (privacy)**
3. **Deduplication (30 min)**
   * **Exact Deduplication: Hash-based** 
     + **SHA256(document) for identity**
     + **O(1) lookup with hash table**
     + **Simple but misses near-duplicates**
   * **Fuzzy/Near-Duplicate Detection** 
     + **MinHash + LSH (Locality Sensitive Hashing)**
     + **Jaccard similarity on shingles**
     + **Fast approximate matching**
   * **Document-Level vs Line-Level** 
     + **Document: entire pages**
     + **Line: within documents (less common)**
   * **Dedup Across Folds** 
     + **Train/val/test contamination**
     + **Must deduplicate across all splits**
   * **Efficiency at Scale** 
     + **Map-reduce deduplication**
     + **Multiple passes acceptable**
4. **Tokenization at Scale (25 min)**
   * **BPE Training** 
     + **Byte pair encoding algorithm**
     + **Merges most frequent token pairs iteratively**
     + **Vocab size typically 30k-100k**
   * **Vocabulary Selection** 
     + **Size vs coverage trade-off**
     + **Larger vocab: fewer tokens, more parameters**
     + **Smaller vocab: more tokens, less coverage**
   * **Multilingual Tokenization** 
     + **Single vocab for multiple languages (Llama)**
     + **Separate vocabs per language (alternative)**
     + **Code-mixing complexity**
   * **Special Tokens** 
     + **[BOS], [EOS], [PAD], [UNK]**
     + **Chat templates use custom tokens**
   * **Tokenizer Training Infrastructure** 
     + **SentencePiece, Hugging Face tokenizers**
     + **Streaming large text files**
5. **Data Mixing & Curriculum (20 min)**
   * **Domain Mixing** 
     + **Web: 90%**
     + **Code: 5%**
     + **Books: 3%**
     + **Academic: 2%**
     + **(Example ratios, tunable)**
   * **Upsampling/Downsampling** 
     + **Boost important domains (code, math)**
     + **Reduce noise**
   * **Curriculum Learning** 
     + **Easy → Hard progression**
     + **Wikipedia (easy) → Research papers (hard)**
     + **Better convergence on diverse tasks**
   * **Annealing Strategies** 
     + **Shift ratios during training**
     + **Emphasize different domains at different phases**
6. **Data Pipeline Architecture (20 min)**
   * **Distributed Processing** 
     + **Map-reduce framework**
     + **Multiple workers processing partitions**
   * **Sharding Strategies** 
     + **Hash-based: deterministic**
     + **Random: load balancing**
   * **Storage Optimization** 
     + **Compressed format (bz2, zstd)**
     + **Indexed for random access**
   * **Streaming Data** 
     + **On-the-fly processing**
     + **No need to store intermediate results**
   * **Monitoring Pipeline** 
     + **Progress tracking**
     + **Error rates**
     + **Quality metrics**

**Advanced Topics:**

* **Scaling deduplication to petabytes**
* **Streaming tokenization**
* **Dynamic mixing ratios**

**Deliverables:**

* **data\_pipeline/: Complete production pipeline** 
  + **scraper.py: Web scraping**
  + **filter.py: Quality filtering**
  + **dedup.py: Deduplication with MinHash**
  + **tokenizer.py: BPE training and application**
  + **mixer.py: Data mixing and sampling**
* **data\_stats.py: Analyze corpus statistics**
* **Papers: Data quality studies, MinHash, curriculum learning**

**Key Experiments:**

* **Impact of deduplication on model quality**
* **Optimal domain mixing ratios**
* **Perplexity-based filtering effectiveness**

**🧠 PART 3: MEMORY SYSTEMS (Weeks 7-12)**

**Week 7: Long Context Crisis & Solutions (2 hours)**

**Single Deep Lecture: "Why Long Context is Hard and How to Fix It"**

**What You'll Learn:**

1. **The Context Length Problem (30 min)**
   * **O(n²) Complexity** 
     + **Attention: QK^T is n×n matrix**
     + **Memory: O(n\*d) for KV cache per token**
     + **Total for seq\_len=1M: ~1TB+ VRAM**
   * **Attention Cost Analysis** 
     + **seq\_len=4k: 16M attention scores (~64MB FP32)**
     + **seq\_len=128k: 16B attention scores (~64GB)**
     + **seq\_len=1M: 1T scores → infeasible**
   * **KV Cache Bottleneck** 
     + **Inference memory dominated by KV cache**
     + **Scales linearly with sequence length**
     + **Batch size × seq\_len × 2 × heads × head\_dim × bytes**
   * **Why 1M Context is Hard** 
     + **Computational cost: 10x-100x vs 4k context**
     + **Memory requirements exceed available VRAM**
     + **Optimization essential for feasibility**
2. **KV Cache Deep Dive (25 min)**
   * **What is KV Cache?** 
     + **Stores all previous K, V for auto-regressive generation**
     + **Avoids recomputing attention for old tokens**
     + **Trade-off: memory for speed**
   * **Memory Calculation** 
     + **Per-token cost: 2 × num\_layers × heads × head\_dim × bytes**
     + **Example: 70B model with GQA** 
       - **80 layers × 8 heads × 128 dim × 2 × 2 bytes = 40KB/token**
       - **1M tokens = 40GB just for KV cache**
   * **Cache Management** 
     + **Reuse across generation steps**
     + **Efficient update operations**
   * **Optimization Strategies** 
     + **Quantization (8-bit, 4-bit)**
     + **Pruning (remove unimportant tokens)**
     + **Compression (keep essential tokens)**
3. **Lost in the Middle Problem (20 min)**
   * **Phenomenon: Models attend middle of long context less**
   * **Causes** 
     + **Position bias (prefer recent, first tokens)**
     + **Information bottleneck in attention**
   * **Mitigation** 
     + **Attention sinking: preserve early tokens**
     + **Dense retrieval: retrieve important tokens**
     + **Instruction placement: put key info first/last**
4. **Long Context Benchmarks (15 min)**
   * **RULER: Targeted long-context evaluation**
   * **Needle in Haystack: Find key info in long context**
   * **LongBench: Chinese long-context benchmark**
   * **ZeroSCROLLS: Zero-shot long-document understanding**
   * **Metrics: Recall@k, accuracy, F1**
5. **Memory Profiling & Analysis (15 min)**
   * **VRAM Tracking** 
     + **torch.cuda.memory\_stats()**
     + **Identify bottlenecks**
   * **Attention Cost Breakdown** 
     + **Compute vs memory bandwidth**
     + **Roofline model analysis**
   * **Throughput Measurement** 
     + **Tokens/second at different seq\_lengths**
     + **Quality degradation curve**
   * **Bottleneck Identification** 
     + **Memory-bound vs compute-bound**
     + **Where to optimize**
6. **Context Length Experiments (15 min)**
   * **Scaling Experiments** 
     + **Train/eval at 4k, 8k, 16k, 32k**
     + **Measure quality degradation**
     + **Fit scaling curves**
   * **Cost Analysis** 
     + **Latency vs context length**
     + **Memory vs context length**
     + **Practical limits with hardware**
   * **Quality Measurement** 
     + **Loss on held-out data**
     + **Task-specific metrics**
   * **Optimal Length: Best balance of quality, speed, memory**

**Deep Dives:**

* **O(n²) complexity proof and alternatives**
* **Exact KV cache memory formula derivation**
* **How position encoding affects "lost in middle"**

**Deliverables:**

* **long\_context\_profiler.py: Comprehensive profiling tools**
* **context\_scaling\_experiments.py: Evaluate at multiple lengths**
* **memory\_calculator.py: Estimate VRAM for any config**
* **Papers: Long-context studies, lost in middle, benchmarks**

**Key Experiments:**

* **Profile your model at 4k, 8k, 16k context**
* **Measure quality on retrieval task**
* **Identify bottleneck (compute vs memory)**

**Week 8: Segment-Based Memory (Transformer-XL) (2 hours)**

**Single Deep Lecture: "Transformer-XL & Compressive Transformers"**

**What You'll Learn:**

1. **Transformer-XL Architecture (35 min)**
   * **Segment-Level Recurrence** 
     + **Process document as segments (e.g., 512 tokens)**
     + **Maintain hidden states between segments**
     + **Each segment can attend to previous segment**
   * **Memory Mechanism** 
     + **Store hidden states h\_seg from previous segment**
     + **Use as extended context: concat([h\_previous, h\_current])**
     + **Effective context grows multiplicatively**
   * **Relative Position Encoding** 
     + **Why relative (not absolute) positions matter**
     + **Enables segment recurrence (no position reset)**
     + **Decomposes: attention uses relative distances**
   * **How It Works** 
     + **Segment 1: process normally**
     + **Segment 2: attend to Seg 1's outputs + Seg 2's inputs**
     + **Gradient: can backprop through segment boundaries**
   * **Memory Efficiency** 
     + **Effective context = segment\_len × depth**
     + **Example: 512 token segments, 12 layers → 6k effective context**
     + **Linear scaling with depth, not quadratic**
2. **Relative Positional Encodings (25 min)**
   * **Why Relative Positions?** 
     + **Absolute: position resets at segment boundary (breaks continuity)**
     + **Relative: distance-based (continuous across segments)**
   * **Mathematics** 
     + **Decompose attention: content + position terms**
     + **Position bias: relative\_pos\_bias(i, j)**
     + **Applied after QK^T computation**
   * **Implementation** 
     + **Precompute relative position biases**
     + **Indexing: distance ranges from -max\_rel to +max\_rel**
     + **Memory: O(seq\_len²) but cacheable**
   * **Comparison with Absolute** 
     + **Absolute: trains on fixed length, struggles to generalize**
     + **Relative: works across segment boundaries**
3. **Segment Caching Strategy (20 min)**
   * **Cache Management** 
     + **Store activations from previous segment**
     + **Decide: keep all or subsample?**
   * **Rolling Memory** 
     + **Forget very old segments (prevent memory explosion)**
     + **Slide window of recent segments**
     + **Trade-off: more recent context vs older memory**
   * **Segment Size Choice** 
     + **Larger segments: fewer overhead, but less flexibility**
     + **Smaller segments: more segments, more merges**
     + **Typical: 256-1024 tokens**
   * **Efficiency Gains** 
     + **Training: similar cost to standard transformer**
     + **Inference: reuse KV cache from previous segment**
4. **Training Transformer-XL (15 min)**
   * **Training Modifications** 
     + **Forward: use memory from previous batch element**
     + **Backward: gradient flows through memory (carefully)**
   * **Backpropagation Through Time (BPTT)** 
     + **Can truncate: don't backprop too far**
     + **Or full backprop: slower but better gradients**
   * **Gradient Flow & Stability** 
     + **Careful scaling of gradients through memory**
     + **Risk: exploding gradients over long sequences**
     + **Mitigation: gradient clipping more aggressive**
5. **Compressive Transformers (20 min)**
   * **Key Idea: Old memory is less useful** 
     + **Compress old segments before storing**
     + **Keep recent segments in full detail**
   * **Compression Mechanism** 
     + **Average pooling, learned compression, attention-based**
     + **Compress after N segments**
   * **Update Rules** 
     + **H\_short: recent full segments**
     + **H\_long: compressed old segments**
     + **Attend to both**
   * **Quality Trade-offs** 
     + **More compression: less memory, but more info loss**
     + **Optimal compression ratio: empirical tuning**
6. **Implementation & Benchmarking (15 min)**
   * **Complete Implementation** 
     + **Cache management code**
     + **Forward/backward passes**
     + **Relative position bias computation**
   * **Training Loop** 
     + **Initialize memory**
     + **Process segments sequentially**
     + **Update memory between segments**
   * **Evaluation** 
     + **Long-document understanding tasks**
     + **Compare vs vanilla attention**

**Advanced Topics:**

* **Gradient flow analysis through recurrence**
* **Optimal compression ratios**
* **Trade-offs: context vs memory**

**Deliverables:**

* **transformer\_xl.py: Complete Transformer-XL**
* **compressive\_transformer.py: With compression**
* **relative\_position\_encoding.py: Detailed implementation**
* **memory\_management.py: Cache operations**
* **Papers: Transformer-XL, Compressive Transformers**

**Key Experiments:**

* **Train on long documents (16k+ tokens)**
* **Compare convergence: vanilla vs Transformer-XL**
* **Measure effective context vs memory usage**

**Week 9: Retrieval-Augmented Memory (2.5 hours)**

**Single Deep Lecture: "RAG Systems: From kNN-LM to RETRO"**

**What You'll Learn:**

1. **RAG Fundamentals (30 min)**
   * **Core Concept** 
     + **Generate from retrieved context, not just parametric memory**
     + **Combines: retrieval + generation**
   * **Architecture Overview** 
     + **Retriever: encode query, find similar documents**
     + **Generator: condition LM on retrieved docs**
   * **When to Use RAG** 
     + **Knowledge-intensive tasks (QA, fact-checking)**
     + **When you have external knowledge base**
     + **Long-tail facts not in training data**
   * **Benefits** 
     + **Lower parameters (don't memorize everything)**
     + **Interpretable (can show retrieved docs)**
     + **Updatable (change retrieval without retraining)**
   * **Limitations** 
     + **Retrieval quality critical (garbage in, garbage out)**
     + **Added latency (retrieval + generation)**
     + **May hallucinate if retrieval fails**
2. **kNN-LM: Non-Parametric Memory (25 min)**
   * **Concept** 
     + **Augment LM with nearest-neighbor memory**
     + **At each step: retrieve k nearest neighbors**
     + **Interpolate: α\*P\_LM(token) + (1-α)\*P\_kNN(token)**
   * **Datastore Creation** 
     + **Encode all contexts and their continuations**
     + **Context = (token\_1, ..., token\_n)**
     + **Store: (embedding, continuation\_token)**
   * **Nearest Neighbor Search** 
     + **Efficient: FAISS, LSH**
     + **Find k closest context embeddings**
     + **Average their continuations**
   * **Interpolation** 
     + **Temperature-based weighting**
     + **Learnable or fixed α**
   * **Why It Works** 
     + **Non-parametric: memorize exact continuations**
     + **Doesn't require retraining**
   * **Limitations** 
     + **Slow inference (search per token)**
     + **Large datastore memory**
     + **Limited to retrieval, no reasoning**
3. **RETRO: Retrieval-Enhanced Transformers (30 min)**
   * **Architecture Design** 
     + **Transformer + retrieval blocks**
     + **Retrieval happens at specific layers**
   * **Chunked Cross-Attention** 
     + **Split sequence into chunks**
     + **For each chunk: retrieve similar docs**
     + **Attend: self-attention + cross-attention to retrieved**
   * **Retrieval Mechanism** 
     + **Chunk encoder: embed chunk independently**
     + **Retrieve: k documents similar to chunk**
     + **Each document: stored embeddings**
   * **Training Process** 
     + **End-to-end training of whole model**
     + **Gradient flow: through retrieval attention**
     + **Tricky: non-differentiable top-k search** 
       - **Solution: straight-through estimators or differentiable approx**
   * **Efficiency** 
     + **Retrieval latency: offset by generation quality**
     + **Memory: external database, not model parameters**
   * **Results** 
     + **25B parameter model matches 175B GPT-3**
     + **Efficient scaling**
4. **FAISS for Billion-Scale Search (20 min)**
   * **Library Overview** 
     + **Facebook AI Similarity Search**
     + **Optimized for large-scale retrieval**
   * **Index Types** 
     + **Flat: Brute force (exact, slow)**
     + **IVF (Inverted File): Approximate, faster** 
       - **Divide space into cells, search cell + neighbors**
     + **HNSW (Hierarchical NSW): Graph-based** 
       - **Small-world properties, very fast**
     + **Product Quantization: Compress vectors**
   * **Scaling to Billions** 
     + **Use IVF + PQ for compression**
     + **Multi-GPU search**
     + **Distributed index across machines**
   * **Optimization** 
     + **Batch queries for throughput**
     + **GPU acceleration (FAISS GPU)**
5. **Embedding Models for Retrieval (20 min)**
   * **Dense Retrievers** 
     + **Encode query and documents into vectors**
     + **Similarity: dot-product or cosine**
   * **Contrastive Learning** 
     + **Training: similar pairs → close embedding, dissimilar → far**
     + **Loss: InfoNCE, triplet loss**
     + **Hard negatives: challenge the model**
   * **Embedding Quality** 
     + **Affects retrieval quality downstream**
     + **Must align with generator's needs**
   * **Model Selection** 
     + **Pretrained: e5-base, bge-base**
     + **Custom: train on your data**
6. **Retrieval Strategies (20 min)**
   * **Top-k Retrieval: Return k most similar** 
     + **Simple, standard**
   * **Reranking: First-stage retrieval, then rerank top-k** 
     + **Expensive ranker on fewer candidates**
     + **Improves quality at small compute cost**
   * **Hybrid Search: Combine dense + sparse** 
     + **BM25 (sparse) + dense**
     + **Better coverage (keywords + semantics)**
   * **Query Augmentation: Expand query** 
     + **Generate hypothetical document**
     + **Retrieve based on generated content**
7. **Complete RAG System (25 min)**
   * **Components** 
     + **Document ingestion and chunking**
     + **Embedding and indexing**
     + **Query processing**
     + **Retrieval**
     + **Generation conditioning**
     + **Serving**
   * **Pipeline** 
     + **Offline: build index**
     + **Online: retrieve + generate**
   * **Evaluation** 
     + **Retrieval: recall, precision, MRR**
     + **Generation: ROUGE, F1**
     + **End-to-end: task accuracy**
   * **Deployment** 
     + **Latency: retrieval + generation**
     + **Throughput: batch size**
     + **Scaling: sharded index, distributed generation**

**Advanced Topics:**

* **Learning to retrieve (jointly train retriever + generator)**
* **Dense passage retrieval training**
* **Trade-offs: quality vs speed**

**Deliverables:**

* **rag\_system/: Complete implementation** 
  + **retriever.py: Embedding, FAISS integration**
  + **generator.py: LM conditioning**
  + **pipeline.py: End-to-end pipeline**
* **knn\_lm.py: kNN-LM implementation**
* **retro.py: RETRO architecture blocks**
* **Papers: kNN-LM, RETRO, DPR, RAG**

**Key Experiments:**

* **Build RAG on Wikipedia corpus**
* **Measure retrieval quality vs generation quality**
* **Compare with fine-tuning alone**

**Week 10: Product-Key Memory (2 hours)**

**Single Deep Lecture: "Trainable External Memory: Product-Key Memory"**

**What You'll Learn:**

1. **PKM Architecture Overview (30 min)**
   * **Core Idea** 
     + **Transformer queries external memory (A × B codebook)**
     + **Better than retrieval-based (differentiable training)**
   * **Product Quantization Concept** 
     + **Split high-dimensional space into products**
     + **A × B: two separate codebooks**
     + **Key = A\_vector ⊕ B\_vector (concatenate)**
   * **Why It Works** 
     + **Memory capacity: exponential in dimension ratio**
     + **Training: gradients flow to memory**
     + **Queries: learned to retrieve useful info**
   * **Integration with Transformer** 
     + **Each position: query external memory**
     + **Retrieve top-k memory slots**
     + **Aggregate retrieved values**
     + **Add to hidden state**
2. **Product Quantization Mathematics (25 min)**
   * **Quantization Theory** 
     + **Original: d-dimensional vector**
     + **Quantized: index into codebook (smaller)**
     + **Trade-off: compression vs information loss**
   * **Product Space** 
     + **Split into d/m subspaces (m=2 for binary)**
     + **Each subspace: own codebook**
     + **Key index: product of subspace indices**
   * **Key Selection** 
     + **Query q, Codebooks A, B**
     + **Distances: ||q - a\_i ⊕ b\_j||² for all i, j**
     + **Top-k selection (top-k product combinations)**
   * **Value Retrieval** 
     + **K nearest keys retrieved**
     + **Aggregate values via weighted sum**
     + **Weights: inverse of distance (or softmax)**
   * **Gradients** 
     + **Straight-through estimator for top-k**
     + **Codebooks and keys updated via SGD**
3. **Training PKM (20 min)**
   * **Integration Strategy** 
     + **Add memory module to specific layers**
     + **Queries generated from hidden states**
   * **Loss Functions** 
     + **Reconstruction loss: predict next token**
     + **Plus memory-specific losses (if any)**
   * **Gradient Flow** 
     + **Through top-k: use straight-through estimator**
     + **Smooth approximation or gumbel-softmax alternative**
   * **Stability** 
     + **Codebook collapse: all codes become similar**
     + **Mitigation: codebook reset, diversity loss**
     + **Learning rate scheduling for stability**
4. **Collision Analysis (15 min)**
   * **Key Collisions** 
     + **Multiple inputs map to same key**
     + **Reduces effective memory capacity**
   * **Utilization Metrics** 
     + **% of slots actually used**
     + **Histogram of collision counts**
   * **Addressing Strategies** 
     + **Increase codebook size (A, B)**
     + **Entropy regularization (encourage diversity)**
     + **Temperature annealing**
   * **Impact on Quality** 
     + **More collisions → worse performance**
     + **Trade-off: memory capacity vs quality**
5. **PKM Ablation Studies (15 min)**
   * **Memory Size: Affect of |A| × |B|** 
     + **Larger: better capacity, more parameters**
     + **Smaller: faster, less params**
   * **Number of Heads: Slot retrieval per head** 
     + **More heads: redundancy**
     + **Fewer: efficiency**
   * **Top-k Selection: How many slots to aggregate** 
     + **k=1: Fast, limited**
     + **k=8: Balance**
     + **k=64: Very rich but slow**
   * **Temperature: Softmax temperature for weighting** 
     + **High: uniform, smooth**
     + **Low: sharp, peaky**
6. **Implementation from Scratch (15 min)**
   * **Data Structures** 
     + **Codebook A: learnable parameter**
     + **Codebook B: learnable parameter**
     + **Values: associated with each product code**
   * **Forward Pass** 
     + **Query projection: q = W\_q \* hidden\_state**
     + **Distance computation: d\_ij = ||q - (a\_i ⊕ b\_j)||²**
     + **Top-k retrieval: efficient selection**
     + **Value aggregation: weighted sum**
   * **Training Loop** 
     + **Compute loss on reconstructed output**
     + **Backprop through aggregation**
     + **Codebook updates**
   * **Evaluation** 
     + **Memory utilization**
     + **Collision rates**
     + **Task performance**

**Advanced Topics:**

* **Preventing codebook collapse**
* **Learned value aggregation**
* **Combining with other memory types**

**Deliverables:**

* **product\_key\_memory.py: Complete implementation**
* **pkm\_ablation.py: Systematic study**
* **memory\_analyzer.py: Utilization and collisions**
* **Papers: Product-Key Memory, PKM variants**

**Key Experiments:**

* **Train model with PKM on language modeling task**
* **Measure memory utilization and collisions**
* **Ablate: codebook size, k, temperature**

**Week 11: KV Cache Optimization (2.5 hours)**

**Single Deep Lecture: "KV Cache Optimization: Complete Survey"**

**What You'll Learn:**

1. **KV Cache as Critical Bottleneck (25 min)**
   * **Memory Analysis** 
     + **Inference dominated by KV cache for long sequences**
     + **Scales linearly: O(batch\_size \* seq\_len \* d)**
     + **Example: 70B model, batch=32, seq\_len=2k** 
       - **KV cache: ~250GB (most VRAM!)**
   * **Scaling Issues** 
     + **Doubling sequence length → doubles KV memory**
     + **Limits batch size for long contexts**
     + **Prohibitive for 100k+ length sequences**
   * **Inference Costs** 
     + **Compute: bounded by model FLOPs**
     + **Memory: KV cache bandwidth**
     + **Often memory-bound (not compute-bound)**
   * **Optimization Motivation** 
     + **Reduce KV cache size**
     + **Enable longer contexts**
     + **Increase batch size**
     + **Preserve quality**
2. **SnapKV: Attention-Based Eviction (30 min)**
   * **Core Idea** 
     + **Identify important tokens based on attention patterns**
     + **Keep: high attention tokens**
     + **Evict: low attention tokens**
   * **Importance Scoring** 
     + **Compute attention weights: softmax(QK^T)**
     + **Score: max attention across all heads**
     + **Or: aggregated over heads/layers**
   * **Head-Wise Selection** 
     + **Each head: independent selection**
     + **Different heads attend to different tokens**
     + **Improves flexibility vs global selection**
   * **Top-p Retention** 
     + **Keep top-p% of tokens by importance**
     + **Discard rest (don't store in cache)**
     + **p typically 0.9 (keep 90%)**
   * **Quality Metrics** 
     + **Measure: prediction quality on evaluation set**
     + **Find optimal p for your task**
   * **Efficiency Gains** 
     + **KV cache: ≈ p × original size**
     + **p=0.9 → 10% savings**
     + **Multiple methods combined: larger savings**
   * **Limitations** 
     + **Early tokens may have low attention (but important)**
     + **Attn sinks: special initial tokens very attended**
3. **StreamingLLM: Sink Tokens & Infinite Context (30 min)**
   * **Attention Sink Phenomenon** 
     + **First few tokens get very high attention**
     + **Independent of content (positional bias)**
     + **Stabilizes model output**
   * **Architecture** 
     + **Keep: sink tokens (e.g., first 4) + recent window**
     + **Discard: middle-aged tokens**
     + **Result: bounded cache size, infinite sequence**
   * **Sink + Window** 
     + **Sink: first N tokens (4-8 typical)**
     + **Window: last M tokens (2k typical)**
     + **Total cache: (N + M) × d = constant**
   * **Rolling KV Cache** 
     + **As new token arrives: add to cache**
     + **Oldest non-sink token removed**
     + **Sliding window property**
   * **Infinite Length** 
     + **Sequence can grow arbitrarily long**
     + **Cache stays bounded**
     + **Trade-off: loss attention to middle history**
   * **Quality** 
     + **Works surprisingly well empirically**
     + **Some loss for long-range dependencies**
     + **Good for streaming/online scenarios**
   * **Implementation** 
     + **Track: oldest valid index**
     + **Circular buffer management**
     + **Attention masking for validity**
4. **KV Cache Quantization (20 min)**
   * **Motivation** 
     + **KV cache often stored in FP32 (4 bytes)**
     + **Quantize to INT8 (1 byte) or INT4 (0.5 bytes)**
     + **4-8x reduction**
   * **Quantization Methods** 
     + **Per-channel: same scale per vector**
     + **Per-token: different scale per token**
     + **Per-token: better quality, more complex**
   * **Bit Widths** 
     + **INT8: 8 bits, minimal loss**
     + **INT4: 4 bits, more loss, 8x savings**
     + **Empirical: INT8 ≈ imperceptible, INT4 ≈ small loss**
   * **Dequantization** 
     + **At usage: KV → FP16/FP32 for attention**
     + **Cost: dequant + attention compute**
     + **Usually still faster than full FP32 cache**
   * **Scale Factors** 
     + **Min-max scaling: scale = (max - min) / (2^bits - 1)**
     + **Per-token: scale varies**
   * **Quality Trade-off** 
     + **Mostly empirical (depends on model, task)**
     + **Generally: INT8 safe, INT4 acceptable**
5. **H2O: Heavy Hitters Only (20 min)**
   * **Concept** 
     + **Local tokens (recent) + heavy hitter tokens (high attention)**
     + **Combine locality with importance**
   * **Selection Algorithm** 
     + **Keep: most recent M tokens (local window)**
     + **Keep: top-k historical tokens by attention**
     + **Total: M + k tokens cached**
   * **Dynamic Selection** 
     + **As new token arrives: recompute top-k**
     + **May evict old heavy hitters for new ones**
     + **Adaptive to changing attention patterns**
   * **Memory Savings** 
     + **Instead of full seq\_len: keep ~2k tokens**
     + **Achieves 10-100x reduction (depends on seq\_len)**
   * **Quality** 
     + **Heavy hitters: usually content tokens**
     + **Nearby: local context**
     + **Combination works well empirically**
   * **Efficiency** 
     + **Tracking top-k: minor overhead**
     + **Selection: can be amortized**
6. **Unified KV Policy Framework (15 min)**
   * **Comparison Framework** 
     + **API for pluggable policies**
     + **Common interface**
     + **Easy swapping**
   * **Benchmarking** 
     + **Same evaluation for all policies**
     + **Measure: quality, speed, memory**
     + **Trade-off curves**
   * **Best Practices** 
     + **No single best policy**
     + **Task-dependent**
     + **Considerations: seq\_len, model, hardware**
7. **Implementing All KV Policies (20 min)**
   * **SnapKV Implementation** 
     + **Attention tracking**
     + **Per-head importance**
     + **Top-p eviction**
   * **StreamingLLM** 
     + **Sink token tracking**
     + **Circular buffer management**
     + **Masking logic**
   * **Quantization** 
     + **Scale calculation**
     + **Quantize/dequantize ops**
   * **H2O** 
     + **Local window tracking**
     + **Top-k heap or sort**
     + **Token labeling**
   * **Evaluation Framework** 
     + **Standard tasks**
     + **Metric computation**

**Advanced Topics:**

* **Combining policies (SnapKV + quantization)**
* **Learned eviction policies**
* **Attention pattern modeling**

**Deliverables:**

* **kv\_policies/: All implementations** 
  + **snapkv.py: Attention-based**
  + **streaming\_llm.py: Sink tokens**
  + **quantization.py: KV quantization**
  + **h2o.py: Heavy hitters**
  + **unified\_api.py: Common interface**
* **kv\_benchmark.py: Compare all policies**
* **Papers: SnapKV, StreamingLLM, H2O, KV quantization**

**Key Experiments:**

* **Implement each policy**
* **Benchmark on long-context tasks**
* **Measure: quality, speed, memory**
* **Find optimal policy for your model**

**Week 12: Modern Memory Systems (2 hours)**

**Single Deep Lecture: "Infini-Attention & Hybrid Memory Approaches"**

**What You'll Learn:**

1. **Infini-Attention Architecture (35 min)**
   * **Core Design** 
     + **Compressive memory + local attention**
     + **Local: recent context (e.g., last 2k tokens)**
     + **Compressive: compressed history**
   * **Local Attention** 
     + **Standard attention on recent window**
     + **Maintains full information**
   * **Global (Compressive) Attention** 
     + **Compress old KV into memory**
     + **Query: attend to compressed memory**
     + **Low-rank matrix (key-value pairs)**
   * **Memory Updates** 
     + **As new tokens arrive: compress and store**
     + **Running average: mu = (mu \* t + new\_key) / (t+1)**
   * **Key Insight** 
     + **Local: detail, recent**
     + **Global: summary, historical**
     + **Combined: efficient long-context**
   * **Memory Format** 
     + **Beta: m × d matrix (m compressed vectors)**
     + **Key-value associations**
     + **Low-rank compression**
2. **Memory as Associative Array (20 min)**
   * **Concept** 
     + **Memory: key-value pairs**
     + **Query: retrieve by key similarity**
     + **No explicit storage bounds**
   * **Update Rules** 
     + **Additive: M\_new = M\_old + key\_value\_pair**
     + **Normalization: keep magnitude bounded**
   * **Retrieval Mechanism** 
     + **Query q: compute similarity with all stored keys**
     + **Aggregate values by similarity**
     + **Weights: softmax of similarities**
   * **Delta Rule (Hopfield-like)** 
     + **M += q ⊗ v (outer product update)**
     + **Normalizes by trace**
   * **Properties** 
     + **Content-addressable (retrieve by similarity)**
     + **Distributed storage (no single address)**
     + **Robust to noise (distributed redundancy)**
3. **Implementing Infini-Attention (20 min)**
   * **Architecture Details** 
     + **Dual attention: local + global**
     + **Local: standard MultiHeadAttention**
     + **Global: low-rank key-value matrix**
   * **Memory Module** 
     + **Initialize: random or zero**
     + **Update: accumulate key-value pairs**
     + **Retrieval: normalized dot-product**
   * **Training** 
     + **Gradients: through both local and global**
     + **Learning rates: may need tuning**
     + **Numerical stability: normalization**
   * **Inference** 
     + **Initialize empty memory**
     + **Process tokens, updating memory**
     + **Generate with both local + global context**
4. **RMT: Recurrent Memory Transformer (20 min)**
   * **Segment-Based Processing** 
     + **Process text in segments**
     + **Output: hidden states + memory summary**
   * **Memory Tokens** 
     + **Special tokens representing summary**
     + **Passed to next segment**
     + **Compresses information**
   * **Gradient Flow** 
     + **Recurrent: memory from previous segment**
     + **Backprop: through memory connections**
     + **Careful scaling to prevent explosion**
   * **Long Context** 
     + **Effective context: segment\_len × num\_layers × memory\_compression**
     + **Efficient scaling**
5. **Memory Systems Comparison (15 min)**
   * **Transformer-XL** 
     + **Pros: Simple, effective, well-studied**
     + **Cons: Fixed segment size, recurrence overhead**
   * **Compressive Transformers** 
     + **Pros: Handles arbitrary-length sequences**
     + **Cons: Compression loss, tuning complexity**
   * **PKM** 
     + **Pros: Trainable, flexible**
     + **Cons: Implementation complex, collision issues**
   * **Infini-Attention** 
     + **Pros: Local + global balance, simple math**
     + **Cons: Recent (less battle-tested)**
   * **Trade-off Analysis** 
     + **Memory vs compute**
     + **Quality vs speed**
     + **Simplicity vs sophistication**
6. **Building Hybrid Memory System (15 min)**
   * **Combining Approaches** 
     + **Local attention: recent tokens**
     + **Compressive: historical summary**
     + **Retrieval: important facts**
   * **Architecture Design** 
     + **Layer selection: where to add memory**
     + **Integration: minimize overhead**
   * **Implementation** 
     + **Modular design**
     + **Easy to swap components**
   * **Evaluation** 
     + **Long-document tasks**
     + **Compare against baselines**
     + **Ablation studies**

**Advanced Topics:**

* **Optimal compression ratios**
* **Learned memory update rules**
* **Multi-level hierarchy (short + long memory)**

**Deliverables:**

* **infini\_attention.py: Complete implementation**
* **rmt.py: Recurrent Memory Transformer**
* **memory\_systems.py: Comparison framework**
* **hybrid\_memory.py: Combined system**
* **Papers: Infini-Attention, RMT, memory comparisons**

**Key Experiments:**

* **Implement Infini-Attention**
* **Compare with Transformer-XL**
* **Long-document evaluation**

**⚡ PART 4: EFFICIENCY & SYSTEMS (Weeks 13-15)**

**Week 13: Flash Attention & GPU Kernels (2 hours)**

**Single Deep Lecture: "GPU Optimization: From Memory Hierarchy to Custom Kernels"**

**What You'll Learn:**

1. **GPU Memory Hierarchy (30 min)**
   * **HBM vs SRAM** 
     + **HBM (High Bandwidth Memory): Large (40GB), slow (~900GB/s)**
     + **SRAM: Small (96KB), fast (~20TB/s)**
     + **20x bandwidth difference!**
   * **Memory Access Patterns** 
     + **Sequential: fast (use all bandwidth)**
     + **Random: slow (cache misses)**
   * **IO Bottleneck** 
     + **Matrix multiply: compute-intensive (good)**
     + **Softmax/attention: IO-intensive (bad)**
     + **Goal: minimize data movement**
   * **Roofline Model** 
     + **Peak compute (FLOPs/s)**
     + **Peak bandwidth (GB/s)**
     + **Arithmetic intensity: FLOPs / bytes**
     + **Model is compute-bound if: AI > peak\_compute / peak\_bandwidth**
   * **Attention Analysis** 
     + **QK^T: n² operations, but only n² elements moved (!)**
     + **Softmax: O(n) compute, O(n) IO**
     + **Very low arithmetic intensity → IO-bound**
2. **Flash Attention Algorithm (25 min)**
   * **Tiling Strategy** 
     + **Split Q into tiles (e.g., 16x16)**
     + **For each Q tile:** 
       - **Load into SRAM**
       - **Iterate through K tiles**
       - **Compute partial attention, update max/sum online**
       - **Write to HBM only once**
   * **Online Softmax** 
     + **Standard: compute QK^T fully, then softmax**
     + **Online: update max and sum incrementally**
     + **Numerically stable: subtract running max**
     + **Recompute attention on backward (no storage)**
   * **Block-Sparse Attention** 
     + **Compute only relevant blocks**
     + **Skip blocks with low probability**
     + **Trade-off: accuracy vs speed**
   * **Memory Savings** 
     + **Vanilla: O(n²) storage for intermediate attention**
     + **Flash: O(1) (no intermediate storage)**
     + **Example: 4k → 4k² = 16M attention scores ≈ 64MB**
     + **Flash: constant memory regardless of n**
   * **Mathematics** 
     + **Proof: online computation equivalent to standard**
     + **Numerically stable due to careful normalization**
   * **Complexity** 
     + **Same O(n²) in time**
     + **But: reduced constant factors, better memory**
     + **Speedup: typically 3-5x on practical hardware**
3. **Flash Attention 2 (20 min)**
   * **Improvements Over v1** 
     + **Inner-loop reductions: fewer synchronizations**
     + **Work partitioning: better thread utilization**
     + **Warp-level optimizations**
   * **Performance Gains** 
     + **2x improvement over Flash 1**
     + **10x vs vanilla attention**
   * **Key Optimizations** 
     + **Reduce global synchronization**
     + **Better register usage**
     + **More thread parallelism**
4. **Flash Attention 3 (15 min)**
   * **Latest Optimizations** 
     + **Producer-consumer overlap: prefetch while computing**
     + **Warp specialization: some warps fetch, others compute**
     + **H100-specific: asynchronous execution**
   * **Performance** 
     + **8-10x faster than vanilla**
     + **Close to theoretical peak on H100**
   * **When It Helps Most** 
     + **Long sequences (amortize IO cost)**
     + **Smaller batch sizes (less compute parallelism)**
5. **Integration & Practical Usage (15 min)**
   * **PyTorch Integration** 
     + **torch.nn.functional.scaled\_dot\_product\_attention**
     + **Automatic Flash Attention if available**
   * **xFormers Library** 
     + **Pluggable attention implementations**
     + **Easy to use different backends**
   * **Memory Savings** 
     + **50% reduction reported**
     + **Depends on sequence length, batch size**
   * **Speed Improvements** 
     + **3-10x depending on hardware**
     + **More dramatic for longer sequences**
6. **Custom CUDA Kernels (15 min)**
   * **Triton Introduction** 
     + **High-level CUDA kernel programming**
     + **Abstracts: thread management, synchronization**
     + **Easier than raw CUDA**
   * **Kernel Optimization** 
     + **Minimize memory bandwidth**
     + **Maximize compute throughput**
     + **Profile: identify bottlenecks**
   * **Profiling Tools** 
     + **NVIDIA Nsight**
     + **PyTorch profiler**
     + **Measure: memory, compute, bandwidth**
   * **Best Practices** 
     + **Prefer bandwidth-efficient algorithms**
     + **Coalesce memory access**
     + **Minimize synchronization**

**Advanced Topics:**

* **Sparse attention patterns**
* **Handling attention across GPU boundaries**
* **Fused operations (attention + residual + norm)**

**Deliverables:**

* **flash\_attention/: Implementation notes**
* **triton\_kernels/: Example Triton kernels**
* **profiler.py: Profiling tools**
* **benchmark.py: Compare implementations**
* **Papers: Flash Attention 1-3, GPU optimization**

**Key Experiments:**

* **Profile vanilla vs Flash Attention**
* **Measure bandwidth utilization**
* **Benchmark on your hardware**

**Week 14: Parallelism & Distributed Training (2.5 hours)**

**Single Deep Lecture: "Scaling Training Across GPUs/TPUs: Complete Guide"**

**What You'll Learn:**

1. **Data Parallelism (30 min)**
   * **Basic Concept** 
     + **Replicate model on each GPU**
     + **Each GPU: different batch of data**
     + **Gradients: averaged across GPUs (AllReduce)**
   * **AllReduce Operation** 
     + **Each GPU sends gradients to all others**
     + **Sum and distribute result**
     + **Network: ring allreduce (efficient)**
   * **Gradient Synchronization** 
     + **Compute on local batch**
     + **Sync at end of batch**
     + **Update: use global gradient average**
   * **Scaling Efficiency** 
     + **N GPUs: ideally N× speedup**
     + **Practice: communication overhead**
     + **Efficiency ≈ 90-95% typically (loss: communication)**
   * **Implementation** 
     + **DistributedDataParallel (PyTorch)**
     + **Standard approach for many GPUs (<100)**
2. **Tensor Parallelism (30 min)**
   * **Why Needed** 
     + **Single GPU: not enough memory for huge models**
     + **Shard model across GPUs**
   * **Model Sharding** 
     + **Split weights across devices**
     + **Computation: each device partial**
     + **Communication: sync at layer boundaries**
   * **Megatron-LM Approach** 
     + **Column-wise parallelism (for matmul)**
     + **Row-wise parallelism (for another matmul)**
     + **Careful synchronization**
   * **Column/Row Parallelism** 
     + **W1 (d → 4d): shard across columns**
     + **W2 (4d → d): shard across rows**
     + **All-gather, computation, reduce-scatter**
   * **Communication** 
     + **Required at each layer**
     + **Bandwidth-intensive**
     + **Requires high-bandwidth interconnect (NVLink)**
   * **Pros/Cons** 
     + **Pros: No model redundancy, scales to huge models**
     + **Cons: Communication overhead, complexity**
   * **Best For** 
     + **Models too large for single GPU**
     + **High communication bandwidth available**
3. **Pipeline Parallelism (25 min)**
   * **Motivation** 
     + **Split model vertically (layers)**
     + **Each GPU: subset of layers**
     + **Reduce per-GPU memory (vs replicate whole model)**
   * **Micro-Batching** 
     + **Divide batch into micro-batches**
     + **Pipeline: send small batch to first GPU while others compute**
     + **Increases hardware utilization**
   * **Pipeline Stages** 
     + **GPU0: layers 0-3**
     + **GPU1: layers 4-7**
     + **GPU2: layers 8-11**
     + **Data flows through pipeline**
   * **Communication** 
     + **Activations: forward pass (GPU0 → GPU1)**
     + **Gradients: backward pass (GPU2 ← GPU1)**
     + **Relatively low volume (activations, not weights)**
   * **Bubble Mitigation** 
     + **Unavoidable: last GPU idle during forward pass**
     + **Minimize via better scheduling**
     + **GPipe, PipeDream algorithms**
   * **Trade-offs** 
     + **Pros: Linear memory reduction, low communication**
     + **Cons: Some idle time, complexity**
4. **Fully Sharded Data Parallel (FSDP) (30 min)**
   * **ZeRO Optimizer (Microsoft)** 
     + **Stage 1: Shard gradients across GPUs** 
       - **Each GPU: only its portion of gradient**
       - **Reduce communication in AllReduce**
       - **Memory savings: linear in number of GPUs**
     + **Stage 2: Shard optimizer state (momentum, variance)** 
       - **Adam state: 2× model size (m, v)**
       - **Only store state for own parameters**
       - **2N memory savings (N = number of GPUs)**
     + **Stage 3: Shard model weights** 
       - **Each GPU: only subset of weights**
       - **All-gather weights for forward pass**
       - **Offload to CPU if needed**
       - **Total savings: 4N (grads + optimizer state + weights)**
   * **Memory Reduction Formula** 
     + **Vanilla: M (full model replica)**
     + **ZeRO-1: M/N + M (gradients shared, weights replicated)**
     + **ZeRO-2: M/N + M/N + M**
     + **ZeRO-3: M/N + M/N + M/N (full sharding!)**
     + **For 8 GPUs: ZeRO-3 ≈ 8× memory savings**
   * **Communication Overhead** 
     + **All-gather: collect weights before forward**
     + **Reduce-scatter: distribute gradients**
     + **Trade-off: memory for communication**
   * **When to Use** 
     + **Models that don't fit on single GPU**
     + **Moderate number of GPUs (8-64)**
     + **High-bandwidth interconnect helpful**
5. **3D Parallelism (25 min)**
   * **Combining All Approaches** 
     + **Data parallelism: replicate & average**
     + **Tensor parallelism: shard weights**
     + **Pipeline parallelism: split layers**
   * **Configuration** 
     + **Dimension 1 (DP): number of DP groups**
     + **Dimension 2 (TP): tensor parallel size**
     + **Dimension 3 (PP): pipeline stages**
     + **Total: DP × TP × PP = total GPUs**
   * **Real-World Example** 
     + **256 GPUs, 70B model**
     + **DP=8, TP=4, PP=8**
     + **8 DP groups × 4 TP × 8 PP = 256**
   * **Communication Patterns** 
     + **DP: global AllReduce (high latency)**
     + **TP: local ring AllReduce (low latency)**
     + **PP: point-to-point (lowest latency)**
   * **Optimal Configuration** 
     + **Depends on: model size, number of GPUs, interconnect**
     + **Rule of thumb: TP ≤ nodes per group, PP ≤ nodes**
     + **Empirical tuning usually necessary**
   * **Scaling Laws** 
     + **Not perfectly linear (communication overhead)**
     + **256 GPUs: ~0.8x throughput of 1 GPU (per-GPU)**
6. **DeepSpeed & Accelerate (20 min)**
   * **DeepSpeed Features** 
     + **ZeRO implementations (stages 1-3)**
     + **Data parallelism, tensor parallelism**
     + **CPU offloading (swap weights/activations to CPU)**
     + **Mixed precision training**
     + **Gradient checkpointing**
   * **Accelerate Library (Hugging Face)** 
     + **Simpler abstraction**
     + **FSDP, DDP support**
     + **Multi-GPU/TPU/CPU automatic**
     + **Config-based setup**
   * **Configuration** 
     + **YAML files specify strategy**
     + **Easy to switch approaches**
     + **Good for experimentation**
   * **Best Practices** 
     + **Start simple (DDP), add complexity as needed**
     + **Profile communication vs compute**
     + **Balance: GPU utilization vs memory**
7. **Multi-Node Training Setup (25 min)**
   * **Cluster Architecture** 
     + **Master node coordinates**
     + **Worker nodes compute**
     + **High-speed interconnect (InfiniBand, etc.)**
   * **NCCL Configuration** 
     + **NVIDIA Collective Communications Library**
     + **Manages AllReduce, broadcast, etc.**
     + **Environment variables: NCCL\_DEBUG, NCCL\_SOCKET\_IFNAME**
   * **Debugging Distributed Training** 
     + **Common issues: hanging, gradient NaN, loss explosion**
     + **Tools: NCCL debug logs, rank-wise monitoring**
     + **Strategies: gradient checks, loss tracking per rank**
   * **Monitoring** 
     + **GPU utilization (nvidia-smi)**
     + **Network bandwidth (netstat, iperf)**
     + **Computation vs communication ratio**
   * **Deployment** 
     + **Container (Docker) for reproducibility**
     + **Job scheduler (Kubernetes, SLURM)**
     + **Checkpointing across nodes**

**Advanced Topics:**

* **Overlapping computation and communication**
* **Adaptive precision for different stages**
* **Resource-constrained scheduling**

**Deliverables:**

* **distributed\_training/: Complete examples** 
  + **ddp\_example.py: Data parallel**
  + **fsdp\_example.py: Fully sharded**
  + **3d\_parallel.py: Combined approach**
  + **config\_accelerate.yaml: Accelerate config**
* **launch\_multinode.sh: Multi-node setup script**
* **monitor\_distributed.py: Monitoring tools**
* **Papers: ZeRO, Megatron, pipeline parallelism**

**Key Experiments:**

* **Scale training from 1 GPU to 8, 16, 32**
* **Measure speedup and efficiency**
* **Profile communication breakdown**

**Week 15: Inference Optimization (2 hours)**

**Single Deep Lecture: "Production Inference: Speed, Quality, Scale"**

**What You'll Learn:**

1. **vLLM & PagedAttention (35 min)**
   * **Motivation** 
     + **Inference often bottleneck in production**
     + **Batch requests with varying lengths**
     + **KV cache management critical**
   * **PagedAttention Algorithm** 
     + **KV cache as virtual memory (like OS paging)**
     + **Physical "pages" store KV values**
     + **Pages can be shared between requests**
     + **Sequence → sequence of page tokens (logical)**
   * **Memory Efficiency** 
     + **Fragmentation problem: each sequence → separate cache**
     + **Solution: reuse pages across sequences**
     + **Example: shared context (many queries asking about same doc)**
     + **Savings: 50-100x in typical scenarios**
   * **Continuous Batching** 
     + **Requests arrive dynamically**
     + **Add to batch, remove completed**
     + **No artificial batching (improves latency)**
     + **Constant GPU utilization**
   * **Request Scheduling** 
     + **FCFS (first come first served)**
     + **Priority queues (urgent first)**
     + **SJF (short job first, minimize latency)**
   * **Implementation** 
     + **vLLM: open-source, production-ready**
     + **SGLang: newer, more features**
     + **Can integrate into deployment**
2. **Speculative Decoding (25 min)**
   * **Problem** 
     + **Autoregressive generation: 1 token/forward pass**
     + **Latency: proportional to output length**
     + **100 tokens = 100 forward passes**
   * **Approach** 
     + **Draft model: small, fast (e.g., 7B)**
     + **Main model: large, accurate (e.g., 70B)**
     + **Draft: generate K tokens quickly**
     + **Main: verify all K tokens in batch**
     + **Accept/reject tokens based on likelihood**
   * **Verification** 
     + **Batch forward pass: all K tokens**
     + **Compare: main vs draft logits**
     + **Accept: if within threshold (e.g., rejection sampling)**
     + **Reject: keep previous token, revert draft**
   * **Speedup Analysis** 
     + **Accept all K: K× speedup**
     + **Partial acceptance: 2-3× typical**
     + **Depends: draft quality, threshold**
   * **Quality** 
     + **Speculative decoding doesn't hurt quality**
     + **Same distribution as full model**
     + **Mathematically proven**
   * **When to Use** 
     + **Long generations (100+ tokens)**
     + **When draft model available**
     + **Latency-sensitive applications**
3. **Quantization for Inference (25 min)**
   * **Motivation** 
     + **Inference: only forward pass (no gradient)**
     + **Can use lower precision**
     + **4x memory savings (FP32 → INT8)**
   * **INT8 Quantization** 
     + **Per-channel: one scale per feature**
     + **Symmetric: [−127, 127] range**
     + **Minimal quality loss (<1% typically)**
   * **GPTQ Method** 
     + **Post-training quantization**
     + **Layer-wise: one layer at a time**
     + **Compensate for quantization with learned scaling**
     + **Works: 4-bit quantization with minimal loss**
   * **AWQ (Activation-Aware Quantization)** 
     + **Observe: activations not uniform**
     + **Quantize less important weights more**
     + **Preserve quality better than uniform**
   * **Quality Preservation** 
     + **Calibration: measure on representative data**
     + **Benchmark: compare quantized vs full**
     + **Trade-off: accuracy vs memory/speed**
   * **Hardware Support** 
     + **INT8: widely available**
     + **INT4: specialized (Tensor Cores, etc.)**
     + **GPU manufacturers optimizing for low-precision**
4. **Model Compilation (20 min)**
   * **TorchScript** 
     + **JIT compile PyTorch models**
     + **Optimize: reduce Python overhead, fuse ops**
     + **Export: inference without PyTorch dependency**
   * **ONNX Export** 
     + **Open Neural Network Exchange format**
     + **Standard: many frameworks support**
     + **Interoperability**
   * **TensorRT (NVIDIA)** 
     + **GPU-specific optimization**
     + **Layer fusion, kernel selection**
     + **10x+ speedup vs PyTorch**
     + **Requires model conversion**
   * **Optimization Techniques** 
     + **Operator fusion: combine layers**
     + **Kernel selection: choose best implementation**
     + **Memory optimization: in-place operations**
   * **Trade-offs** 
     + **Compilation time**
     + **Model specificity (TensorRT tied to GPU)**
     + **Difficulty debugging**
5. **Batching Strategies (20 min)**
   * **Static Batching** 
     + **Fixed batch size**
     + **Simple, predictable**
     + **Inefficient for varying lengths**
   * **Dynamic Batching** 
     + **Batch size adapts to available memory/compute**
     + **Requests grouped by length**
     + **Improves GPU utilization**
   * **Continuous Batching** 
     + **Requests added/removed dynamically**
     + **No waiting for batch assembly**
     + **Best latency**
   * **Request Scheduling** 
     + **Optimal: balance throughput vs latency**
     + **Trade-off: batching adds latency**
     + **Queue management: priority, fairness**
   * **Padding Strategy** 
     + **Pad to same length (easier, faster)**
     + **Or: variable length (memory efficient)**
     + **Mask out padding in attention**
6. **Production Inference System (25 min)**
   * **End-to-End Pipeline** 
     + **Load balancing: route to inference server**
     + **Request queuing: manage incoming requests**
     + **Generation: text generation with vLLM**
     + **Post-processing: format output**
   * **vLLM Deployment** 
     + **Single GPU: basic setup**
     + **Multiple GPUs: tensor parallelism**
     + **Distributed: across multiple machines**
   * **Monitoring** 
     + **Throughput: tokens/second**
     + **Latency: time to first token, total time**
     + **GPU utilization: should be >80%**
     + **Queue length: indicates system load**
   * **Scaling** 
     + **Horizontal: add more servers**
     + **Vertical: more GPUs per server**
     + **Load balancer: distribute requests**
   * **Optimization Tips** 
     + **Use vLLM for 2-5x speedup**
     + **Quantization: 4x memory savings**
     + **Batch requests when possible**
     + **Monitor and profile**

**Advanced Topics:**

* **Custom scheduling policies**
* **Batching with prefix caching**
* **Adaptive precision per layer**

**Deliverables:**

* **inference\_systems/: Production setup** 
  + **vllm\_server.py: vLLM-based server**
  + **speculative\_decoding.py: With draft model**
  + **quantize.py: GPTQ/AWQ quantization**
  + **batch\_scheduler.py: Dynamic batching**
* **benchmark\_inference.py: Compare setups**
* **monitoring.py: Prometheus metrics**
* **Papers: vLLM, PagedAttention, speculative decoding**

**Key Experiments:**

* **Deploy model with vLLM**
* **Compare: vanilla vs vLLM throughput**
* **Measure: quantization quality loss**
* **Profile: bottleneck identification**

**🎯 PART 5: MIXTURE OF EXPERTS (Weeks 16-17)**

**Week 16: MoE Fundamentals (2 hours)**

**Single Deep Lecture: "Mixture of Experts: Sparse Scaling Done Right"**

**What You'll Learn:**

1. **Mixture of Experts Overview (30 min)**
   * **Core Concept** 
     + **FFN → multiple expert FFNs**
     + **Router: decides which experts to use**
     + **Sparse: not all experts used for each token**
   * **Why Sparse?** 
     + **Dense: all experts always active (huge FLOPs)**
     + **Sparse: route to k experts (smaller FLOPs)**
     + **Can scale: add experts without proportional cost**
   * **Router Mechanism** 
     + **Input: hidden state**
     + **Logits: score for each expert**
     + **Top-k: select k experts**
     + **Output: weighted sum of expert outputs**
   * **Expert Specialization** 
     + **Different experts learn different domains**
     + **Some → code, some → reasoning, etc.**
     + **Improves efficiency**
   * **Benefits** 
     + **Efficient scaling: more parameters without more compute**
     + **Better than dense at same FLOPs**
     + **Theoretical: sparse better than dense (recent findings)**
2. **MoE Architecture Details (30 min)**
   * **Standard FFN** 
     + **W1: (d\_model → d\_ff)**
     + **Activation**
     + **W2: (d\_ff → d\_model)**
     + **Cost: 2 × d\_model × d\_ff FLOPs per token**
   * **MoE FFN** 
     + **Replace W1, W2 with expert copies**
     + **Router: linear layer → softmax**
     + **Top-k selection**
     + **Total cost: 2 × d\_model × (d\_ff/num\_experts) × k FLOPs**
   * **Top-k Routing** 
     + **Select top k experts by router score**
     + **k typically: 1 or 2**
     + **All tokens use same k**
   * **Expert Layers** 
     + **How many layers with MoE?**
     + **Typical: every other layer (alternating with dense)**
     + **Or: all attention layers + MoE layers**
   * **Load Balancing** 
     + **Ideal: each expert gets equal tokens**
     + **Reality: some experts more popular**
     + **Causes: wasted capacity, load imbalance**
   * **Capacity Factor** 
     + **Capacity: tokens per expert**
     + **If more tokens needed: drop or expert overflow**
     + **Typical: capacity\_factor = 1.25 (25% slack)**
3. **Router Design (25 min)**
   * **Router Types** 
     + **Linear: single dense layer**
     + **Learned: trained jointly with model**
     + **Fixed: hand-designed rules**
   * **Gating Function** 
     + **logits = MatMul(hidden, W\_router)**
     + **scores = softmax(logits) or sigmoid**
     + **Top-k: select indices**
   * **Expert Selection** 
     + **Hard top-k: select exactly k**
     + **Soft: weighted mixture (all experts)**
     + **Typical: hard for efficiency**
   * **Training Dynamics** 
     + **Initially: random expert selection**
     + **Gradually: experts specialize**
     + **Risk: collapse to single expert**
4. **Load Balancing in MoE (25 min)**
   * **Load Balancing Loss** 
     + **Auxiliary loss: encourage uniform load**
     + **L\_balance = α × sum(P\_e × C\_e)** 
       - **P\_e: fraction of tokens → expert e**
       - **C\_e: capacity used by expert e**
     + **Maximized when uniform**
   * **Auxiliary Losses** 
     + **Router z-loss: prevent large logits**
     + **Importance loss: utilize all experts**
     + **Coefficient tuning crucial**
   * **Router z-loss** 
     + **Regularize: log(sum(exp(logits)))**
     + **Prevent: router from getting overconfident**
   * **Expert Capacity** 
     + **Fixed vs dynamic**
     + **Fixed: simpler, may drop tokens**
     + **Dynamic: complex, can overflow**
   * **Dropout Strategies** 
     + **When expert capacity exceeded: drop tokens or replicate**
     + **Quality impact: small typically**
5. **MoE Training Challenges (20 min)**
   * **Expert Collapse** 
     + **All tokens routed to same expert(s)**
     + **Causes: random initialization, sparse signals**
     + **Prevention: load balancing loss, careful initialization**
   * **Load Imbalance** 
     + **Some experts unused**
     + **Wasted capacity**
     + **Mitigation: auxiliary losses**
   * **Gradient Issues** 
     + **Sparse updates: expert may see few gradients**
     + **Variance: high noise**
     + **Solution: careful learning rate scheduling**
   * **Stabilization Techniques** 
     + **Initialization: scale router carefully**
     + **Learning rate: often needs tuning**
     + **Gradient clipping: more aggressive for sparse**
6. **Fine-Grained MoE (20 min)**
   * **Mixtral Architecture** 
     + **Every position: top-2 experts**
     + **Very efficient**
     + **Simple, proven to work**
   * **Granularity Options** 
     + **Coarse: per-layer routing**
     + **Medium: per-position routing (typical)**
     + **Fine: per-head routing (experimental)**
   * **Modern Designs** 
     + **DeepSeek: MoE with multi-head latent attention**
     + **Llama: exploring MoE variants**
   * **Performance** 
     + **Mixtral 46B > Llama 70B at similar compute**
     + **Shows: MoE efficiency advantage**

**Advanced Topics:**

* **Expert specialization analysis**
* **Auxiliary loss design**
* **Routing learned vs fixed**

**Deliverables:**

* **mixture\_of\_experts/: Complete implementation** 
  + **router.py: Routing logic**
  + **moe\_layer.py: Expert layer**
  + **load\_balancing.py: Losses and monitoring**
  + **training.py: Training loop with stabilization**
* **moe\_analysis.py: Expert specialization**
* **Papers: MoE scaling laws, Mixtral, DeepSeek MoE**

**Key Experiments:**

* **Train small MoE model**
* **Monitor expert load balance**
* **Compare: dense vs sparse FLOPs**

**Week 17: Advanced MoE & Production (2 hours)**

**Single Deep Lecture: "DeepSeek-V3 MoE, Inference, and Production Deployment"**

**What You'll Learn:**

1. **DeepSeek-V3 MoE (30 min)**
   * **Architecture Innovation** 
     + **Multi-head latent attention (from Week 3)**
     + **Combined with MoE for efficiency**
   * **MoE Design** 
     + **8 experts per position**
     + **Top-2 routing**
     + **Shared expert pool + specific experts**
   * **Auxiliary-Loss-Free Approach** 
     + **Novel: doesn't need auxiliary loss**
     + **Uses: expert deallocation and reallocation**
     + **Simpler training**
   * **Performance** 
     + **671B parameters total**
     + **37B active per token**
     + **Outperforms much larger dense models**
     + **Shows: MoE + modern techniques = scaling**
2. **Llama 4 MoE Approach (20 min)**
   * **Architecture Details** 
     + **Exploring MoE variants**
     + **Different expert configurations**
     + **Integration with modern techniques**
   * **Training Approach** 
     + **Careful initialization**
     + **Load balancing tuning**
   * **Results** 
     + **Performance on benchmarks**
     + **Efficiency comparison**
3. **Expert Specialization Analysis (20 min)**
   * **What Experts Learn** 
     + **Visualization techniques**
     + **Attention patterns per expert**
     + **Task-specific experts emerging**
   * **Domain Specialization** 
     + **Some experts: code**
     + **Some: reasoning**
     + **Some: factual knowledge**
   * **Interpretability** 
     + **Can we understand expert roles?**
     + **Probing tasks**
     + **Feature visualization**
   * **Emerging Properties** 
     + **Implicit structure without explicit supervision**
     + **Shows: experts organically specialize**
4. **MoE Inference Optimization (20 min)**
   * **Expert Caching** 
     + **Different experts used by different tokens**
     + **Can cache frequently used experts**
     + **Reduces memory bandwidth**
   * **Batching Strategies** 
     + **Token-level batching: group by expert**
     + **Batch → reorder tokens by expert**
     + **Compute experts in sequence**
     + **Reorder back**
     + **Improves memory locality**
   * **Memory Management** 
     + **All experts in GPU memory?**
     + **Or CPU offloading?**
     + **Trade-off: latency vs memory**
   * **Throughput** 
     + **Sparse routing: better compute efficiency**
     + **Careful implementation: needed for speed**
     + **vLLM, TensorRT support**
5. **Soft MoE & Alternatives (20 min)**
   * **Soft Merging** 
     + **Instead of hard selection: continuous mixture**
     + **All experts contribute (softly)**
     + **Better gradients, harder implementation**
   * **Expert Pruning** 
     + **Remove low-contribution experts**
     + **Reduce inference cost**
     + **Minimal quality loss**
   * **Distillation** 
     + **Compress MoE → dense**
     + **Transfer knowledge**
     + **Simpler inference**
   * **Trade-offs** 
     + **Softness: quality vs compute**
     + **Pruning: speed vs performance**
6. **Production MoE System (25 min)**
   * **Training Pipeline** 
     + **Data preparation (same as dense)**
     + **Training: careful load balancing tuning**
     + **Checkpointing: crucial for large models**
     + **Evaluation: standard benchmarks**
   * **Inference Serving** 
     + **Model loading: all experts or selective?**
     + **Batching: token-level preferred**
     + **Latency: depends on implementation**
   * **Monitoring** 
     + **Expert load: track utilization**
     + **Route stability: changes over time?**
     + **Quality metrics: standard perplexity, tasks**
   * **Optimization** 
     + **Profile: where is time spent?**
     + **Optimize bottleneck**
     + **Typical: expert computation, not routing**

**Advanced Topics:**

* **Learned expert allocation**
* **Hierarchical MoE**
* **Conditional computation**

**Deliverables:**

* **advanced\_moe/: Production implementations** 
  + **deepseek\_moe.py: DeepSeek-style MoE**
  + **expert\_specialization.py: Analysis tools**
  + **inference\_batching.py: Efficient inference**
  + **monitoring.py: Performance tracking**
* **moe\_distillation.py: Compress to dense**
* **Papers: DeepSeek-V3, expert analysis, MoE inference**

**Key Experiments:**

* **Compare: dense vs sparse scaling**
* **Analyze expert specialization**
* **Profile inference bottlenecks**

**🔄 PART 6: ALIGNMENT & RLHF (Weeks 18-19)**

**Week 18: Supervised Fine-Tuning (2 hours)**

**Single Deep Lecture: "SFT: From Base Model to Instruction-Following"**

**What You'll Learn:**

1. **SFT Fundamentals (30 min)**
   * **What is SFT?** 
     + **Fine-tune on instruction-response pairs**
     + **Transform base model → instruction follower**
     + **Simple: supervised loss on responses**
   * **Why Needed?** 
     + **Base models: next-token prediction only**
     + **Not aligned with human preferences**
     + **Not instruction-following**
     + **SFT: intermediate step before RLHF**
   * **Instruction Tuning** 
     + **Curate high-quality instruction-response pairs**
     + **Fine-tune: minimize response loss**
     + **Result: model learns to follow instructions**
   * **Loss Functions** 
     + **Standard: cross-entropy on response tokens**
     + **Masking: ignore prompt tokens (only loss on response)**
     + **Weighted: some responses more important**
   * **Hyperparameters** 
     + **Learning rate: usually lower than pre-training**
     + **Batch size: depends on compute**
     + **Epochs: typically 1-3 (prevent memorization)**
2. **Instruction Dataset Design (25 min)**
   * **Prompt Formats** 
     + **Templates: "Given X, do Y"**
     + **Consistency: all prompts similar structure**
     + **Examples: helps model understand task**
   * **Quality Criteria** 
     + **Correctness: response is accurate**
     + **Clarity: easy to understand**
     + **Diversity: covers different task types**
     + **Length: reasonable (~100-500 tokens per response)**
   * **Dataset Diversity** 
     + **Different domains: coding, writing, Q&A, etc.**
     + **Different difficulty: easy to hard**
     + **Different styles: formal to casual**
   * **Coverage** 
     + **Must cover capabilities you want**
     + **Imbalanced data: some skills missing**
     + **Recommendations: balanced sampling**
   * **Data Size** 
     + **Typical: 50k-500k examples**
     + **More data: better generalization**
     + **Depends: model size, task diversity**
3. **Chat Templates (20 min)**
   * **Message Formatting** 
     + **Standardized format for multi-turn conversations**
     + **User → assistant → user → ...**
     + **Special tokens: mark who is speaking**
   * **System Prompts** 
     + **Initial instruction: set context**
     + **Example: "You are a helpful assistant..."**
     + **Affects: model behavior throughout conversation**
   * **Special Tokens** 
     + **<|user|>, <|assistant|>, <|system|>**
     + **Tokenizer: recognizes as special**
     + **Important: consistency across dataset**
   * **Best Practices** 
     + **Template: simple, unambiguous**
     + **Use in both training and inference**
     + **Document: share with users**
4. **Multi-Turn Conversations (20 min)**
   * **Context Management** 
     + **Keep conversation history**
     + **Model sees all previous turns**
     + **Attention: considers all context**
   * **Turn-Level Training** 
     + **Option 1: loss on all turns**
     + **Option 2: loss only on assistant turns**
     + **Recommendation: loss only on assistant (cleaner)**
   * **Masking Strategies** 
     + **Mask user prompts: don't learn to predict them**
     + **Only predict: assistant responses**
     + **Cleaner training signal**
   * **Quality in Conversations** 
     + **Consistency: assistant remembers context**
     + **Relevance: responds to current turn**
     + **Coherence: maintains thread across turns**
5. **Domain-Specific SFT (20 min)**
   * **Code Generation** 
     + **Instruction: "Write Python function to do X"**
     + **Response: full code, with comments**
     + **Quality: correctness, efficiency**
   * **Math Reasoning** 
     + **Step-by-step: show work**
     + **Format: clear equations, explanations**
     + **Dataset: competitive programming, textbooks**
   * **Tool Use** 
     + **Instructions for tool format**
     + **Response: calls with appropriate args**
     + **Examples: API calls, function invocations**
   * **Specialization** 
     + **Fine-tune on domain → expert behavior**
     + **Trade-off: general capability vs specialization**
     + **Recommendations: SFT on domain, then RLHF**
6. **Complete SFT Pipeline (25 min)**
   * **Data Preparation** 
     + **Collect/curate dataset**
     + **Format: templates, chat structure**
     + **Split: train/val/test**
   * **Training Loop** 
     + **Load base model**
     + **Setup: learning rate schedule, optimizer**
     + **Train: minimize loss on responses**
     + **Validate: eval loss, task accuracy**
   * **Evaluation** 
     + **Loss curves: training vs validation**
     + **Benchmark: standard tasks (MMLU, etc.)**
     + **Manual: sample and review outputs**
   * **Deployment** 
     + **Save checkpoint**
     + **Test: try on new queries**
     + **A/B test: vs base model**

**Advanced Topics:**

* **Mixture of LoRA adapters for different domains**
* **Continued pre-training mixed with SFT**
* **Instruction diversity optimization**

**Deliverables:**

* **sft\_pipeline/: Complete implementation** 
  + **dataset.py: Loading and formatting**
  + **trainer.py: SFT training loop**
  + **templates.py: Chat templates**
  + **evaluate.py: Evaluation metrics**
* **domain\_sft.py: Domain-specific fine-tuning**
* **Papers: Instruction tuning, SFT best practices**

**Key Experiments:**

* **Build SFT dataset (1000 examples)**
* **Train SFT model**
* **Evaluate: instruction following improvement**
* **Compare: base vs SFT on same prompts**

**Week 19: RLHF & Alignment (2.5 hours)**

**Single Deep Lecture: "RLHF & Modern Alignment: PPO, DPO, RLAIF"**

**What You'll Learn:**

1. **RLHF Overview (30 min)**
   * **Core Concept** 
     + **Reinforce: model for human-preferred behaviors**
     + **Use: human feedback to guide learning**
     + **Reward: signal for good responses**
   * **Pipeline Steps** 
     + **Step 1: SFT model (base for RL)**
     + **Step 2: Collect preference data (human evaluation)**
     + **Step 3: Train reward model (predict preference)**
     + **Step 4: RL training (optimize policy using reward)**
   * **Why RLHF?** 
     + **SFT: mimics examples, but not aligned**
     + **RLHF: optimizes for preference, alignment**
     + **Result: more helpful, harmless, honest**
   * **Preference Data** 
     + **Generate two responses (SFT model)**
     + **Human judges: choose better one**
     + **Collect: 10k-100k preference pairs**
   * **Human Feedback Loop** 
     + **Costly: human annotation**
     + **Quality: annotator agreement matters**
     + **Scalability: limited by annotators**
2. **Reward Model Training (25 min)**
   * **Preference Data** 
     + **Input: prompt**
     + **Response A: generated response**
     + **Response B: generated response**
     + **Label: A > B or B > A (sometimes tied)**
   * **Bradley-Terry Model** 
     + **Probability: A > B ∝ exp(r(A) - r(B))**
     + **r(·): reward function (learned)**
     + **Loss: log-likelihood of preferences**
   * **Model Architecture** 
     + **Start: SFT model (has language understanding)**
     + **Output: single scalar (reward)**
     + **Training: classification (better or worse)**
   * **Training Process** 
     + **Dataset: preference pairs**
     + **Loss: cross-entropy on preference**
     + **Optimization: same as SFT (AdamW, etc.)**
   * **Evaluation** 
     + **Accuracy: predicts human preference correctly**
     + **Calibration: predicted rewards match true rank**
     + **Generalization: works on new responses**
3. **PPO for LLMs (30 min)**
   * **Proximal Policy Optimization** 
     + **Policy: LM (generates tokens)**
     + **Reward: from reward model**
     + **Objective: maximize expected reward**
   * **KL Divergence Constraint** 
     + **Naive: optimize reward (can go arbitrarily wrong)**
     + **PPO: stay close to SFT model**
     + **Loss: reward - β × KL(policy || SFT\_model)**
     + **β: controls trade-off**
   * **Value Function** 
     + **V(s): expected future reward from state s**
     + **Used: in advantage estimation**
     + **A(s,a) = Q(s,a) - V(s)**
     + **Importance: reduces variance**
   * **PPO Implementation** 
     + **Collect trajectories: model generates responses**
     + **Compute rewards: use reward model**
     + **Compute advantages: A = reward - V**
     + **Update: policy gradient with clipping**
   * **Training Stability** 
     + **PPO clip: prevent too-large updates**
     + **Value loss: minimize difference from actual return**
     + **Total loss: policy + value + entropy terms**
   * **Challenges** 
     + **Unstable: can diverge from SFT**
     + **Expensive: need model + reward model inference**
     + **Complex: many hyperparameters**
4. **DPO: Direct Preference Optimization (30 min)**
   * **Motivation** 
     + **PPO: complex, unstable, expensive**
     + **Can we optimize preferences directly?**
     + **DPO: yes! without reward model**
   * **Idea** 
     + **Reparameterize reward: r(x,y) = β \* log(π(y|x) / π\_ref(y|x))**
     + **π: policy, π\_ref: reference (SFT)**
     + **Directly optimize preference loss**
   * **Mathematics** 
     + **Preference: P(y\_w > y\_l) = sigmoid(r(y\_w) - r(y\_l))**
     + **Substitute: r in terms of policy**
     + **Loss: directly on policy (no separate reward model)**
   * **Advantages** 
     + **Simpler: single model training**
     + **More stable: no separate reward model**
     + **Faster: fewer forward passes**
   * **DPO Loss** 
     + **log(sigmoid(β \* (log(π/π\_ref(y\_w)) - log(π/π\_ref(y\_l)))))**
     + **Optimizes: preferred response over dispreferred**
   * **Comparison with PPO** 
     + **DPO: 10x faster, simpler, similar quality**
     + **PPO: more proven, sometimes better quality**
     + **Modern: DPO often preferred for speed**
5. **RLAIF: AI Feedback (20 min)**
   * **Problem with Human Feedback** 
     + **Expensive: annotation costs**
     + **Slow: bottleneck**
     + **Biased: human preferences vary**
   * **AI Feedback Approach** 
     + **Use powerful LM to judge preferences**
     + **Constitution: rules for judging**
     + **Critique: AI explains preferences**
   * **Constitutional AI** 
     + **Set of principles: helpfulness, honesty, etc.**
     + **Model: evaluate against constitution**
     + **Generate: synthetic preference data**
   * **Self-Improvement** 
     + **Train reward model on AI feedback**
     + **RLHF: optimize using AI reward**
     + **Iterate: improve model iteratively**
   * **Scalable Oversight** 
     + **AI feedback: scales (no human bottleneck)**
     + **Quality: depends on criteria**
     + **Trade-off: less alignment with human values**
   * **Results** 
     + **Works surprisingly well**
     + **Claude models: use RLAIF + human feedback blend**
6. **Online vs Offline RL (20 min)**
   * **Online RL** 
     + **Generate data: during training**
     + **Policy: improves continuously**
     + **Pros: can explore, improve fast**
     + **Cons: expensive, unstable**
   * **Offline RL** 
     + **Fixed dataset: before training**
     + **Policy: learns from static preferences**
     + **Pros: stable, cheaper, safer**
     + **Cons: no exploration, limited by data**
   * **Hybrid Approach** 
     + **Start: offline (initial improvement)**
     + **Later: online (fine-tune, explore)**
     + **Recommended: safer and efficient**
   * **Sample Efficiency** 
     + **Offline: efficient (reuse data)**
     + **Online: inefficient (one-time use)**
     + **Preference: offline when possible**
   * **Best Practices** 
     + **Start: collect large offline dataset**
     + **Train: reward model, initial policy**
     + **Optional: online refinement if needed**
7. **Complete RLHF Pipeline (25 min)**
   * **Preference Collection** 
     + **Generate responses: SFT model**
     + **Annotate: human preference**
     + **Quality: ensure high annotator agreement**
   * **Reward Training** 
     + **Split: train/val reward model**
     + **Train: minimize preference loss**
     + **Validate: accuracy on held-out**
   * **PPO or DPO Implementation** 
     + **If PPO: complex but proven**
     + **If DPO: simpler, modern**
     + **Initialize: from SFT checkpoint**
   * **Evaluation** 
     + **Benchmarks: standard tasks**
     + **Manual: review samples**
     + **Compare: vs baseline (SFT)**
   * **Iteration** 
     + **Monitor: alignment metrics**
     + **Collect: more preferences if needed**
     + **Refine: improve reward model**

**Advanced Topics:**

* **Multi-objective reward optimization**
* **Instability detection and mitigation**
* **Combining multiple reward signals**

**Deliverables:**

* **rlhf\_pipeline/: Complete system** 
  + **preference\_collection.py: Generate, format**
  + **reward\_model.py: Train reward model**
  + **ppo\_trainer.py: PPO implementation**
  + **dpo\_trainer.py: DPO implementation**
  + **evaluate.py: Benchmark evaluation**
* **constitutional\_ai.py: RLAIF framework**
* **Papers: RLHF, DPO, Constitutional AI**

**Key Experiments:**

* **Collect 1000 preference pairs**
* **Train reward model**
* **Fine-tune with DPO**
* **Evaluate: improvement on benchmarks**

**🧪 PART 7: INDIC RESEARCH (Weeks 20-24)**

**Week 20: LLCE-Indic Benchmark Design (2 hours)**

**Single Deep Lecture: "Building a Long-Context Benchmark for Indic Languages"**

**What You'll Learn:**

1. **Motivation for LLCE-Indic (30 min)**
   * **Indic Languages** 
     + **22 official Indian languages**
     + **~1 billion speakers**
     + **Under-resourced in NLP**
     + **No good long-context benchmarks**
   * **Research Gap** 
     + **English: many long-context benchmarks**
     + **Indic: none specifically designed**
     + **Opportunity: study memorization vs retrieval**
   * **Unique Challenges** 
     + **Code-mixing: Hindi + English common**
     + **Script variety: Devanagari, Tamil, etc.**
     + **Less training data: memorization limited**
   * **Our Contribution** 
     + **LLCE-Indic: comprehensive benchmark**
     + **Multiple languages: Hindi, Tamil, Telugu, etc.**
     + **Diverse tasks: retrieval, reasoning, temporal**
   * **Why This Matters** 
     + **Inclusion: underrepresented communities**
     + **Research: new insights from different languages**
     + **Practical: need tools for Indic language LLMs**
2. **Task Taxonomy (30 min)**
   * **Needle in Haystack** 
     + **Find specific fact in long document**
     + **Long context, important info at different positions**
     + **Measures: retrieval capability**
   * **Temporal Reasoning** 
     + **Track changes over time**
     + **Updates in documents (version history)**
     + **Measures: temporal understanding**
   * **Cross-Lingual QA** 
     + **Question in one language, answer in another**
     + **Tests: multilingual understanding**
     + **Realistic: code-mixing scenarios**
   * **Aggregation Tasks** 
     + **Combine information from multiple documents**
     + **Summarize: key points**
     + **Measures: holistic understanding**
   * **Contradiction Resolution** 
     + **Conflicting information in documents**
     + **Identify: which is correct**
     + **Reasoning: temporal ordering, source credibility**
3. **Data Sources (30 min)**
   * **Government Documents** 
     + **RBI (Reserve Bank India) reports**
     + **SEBI (Securities Exchange Board) filings**
     + **High quality, official**
   * **Legal Corpus** 
     + **Court judgments, legislation**
     + **Complex language, important precedents**
     + **Multi-language documents**
   * **Educational Materials** 
     + **Textbooks, course materials**
     + **Structured, explanatory**
     + **Language: formal, clear**
   * **Licensing** 
     + **Check permissions for use**
     + **Some: public domain**
     + **Some: restricted use**
     + **Document: all attributions**
4. **Schema Design (20 min)**
   * **Metadata Structure** 
     + **Document ID, language, length**
     + **Source, date, category**
     + **Structured, queryable**
   * **Annotation Format** 
     + **Task type, question, answer**
     + **Difficulty level, position of answer**
     + **Language pair (if multilingual)**
   * **Quality Checks** 
     + **Validation: schema conformance**
     + **Completeness: all required fields**
     + **Consistency: similar tasks format similarly**
5. **Multilingual Challenges (20 min)**
   * **Code-Mixing** 
     + **Mixed Hindi + English text**
     + **Common in real usage**
     + **Challenge: tokenization, understanding**
   * **Script Variations** 
     + **Devanagari (Hindi), Tamil script, etc.**
     + **Some documents: transliterated (Roman chars)**
     + **Normalization: needed for consistency**
   * **Translation Quality** 
     + **Translate questions across languages**
     + **Must be natural, not literal**
     + **Verification: native speakers check**
   * **Evaluation Metrics** 
     + **String match: exact answers**
     + **Semantic: fuzzy match**
     + **BLEU/ROUGE: summarization tasks**
     + **Multilingual: language-specific metrics**
6. **Benchmark Infrastructure (20 min)**
   * **Scraping Pipeline** 
     + **Automated download of documents**
     + **Error handling, retries**
     + **Rate limiting (respect servers)**
   * **Storage** 
     + **Centralized database**
     + **Versioning: track changes**
     + **Backup: ensure availability**
   * **API Design** 
     + **Easy access: leaderboard, downloads**
     + **Submission system: track entries**
     + **Documentation: usage examples**
   * **Access Control** 
     + **Public vs restricted**
     + **License attribution**
     + **Usage terms**

**Deliverables:**

* **llce\_indic/schema/: Schema definitions**
* **data\_collection/: Scraping infrastructure**
* **benchmark\_api.py: Access interface**
* **Papers: Benchmark design, motivation**

**Week 21: Data Collection (2 hours)**

**Single Deep Lecture: "Large-Scale Indic Text Collection and Cleaning"**

**Covers: Web scraping, PDF extraction, text cleaning, deduplication, corpus statistics**

**Week 22: Annotation (2 hours)**

**Single Deep Lecture: "Building High-Quality Annotated Dataset"**

**Covers: Annotation guidelines, LLM-assisted annotation, temporal task creation, cross-lingual setup, quality assurance, tools, dataset analysis**

**Week 23: Experiments (2 hours)**

**Single Deep Lecture: "Comprehensive Evaluation of Memory Systems on LLCE-Indic"**

**Covers: Baseline models, memory system evaluation, KV cache policies, positional encodings, ablations, cross-lingual analysis, results**

**Week 24: Paper & Release (1.5 hours)**

**Single Deep Lecture: "Publishing Research: Paper, Code, Dataset Release"**

**Covers: Paper structure and writing, reproducibility package, code release, documentation, journey recap, future directions**

**📊 CONSOLIDATED OUTPUT**

**Total Lectures: ~30 comprehensive 2-hour lectures Plus: Code implementations, papers, datasets Plus: Optional short clips for social media (1-2 per lecture) Duration: 24 weeks, 1-2 lectures/week, 4-8 hours work**

**This gives you:**

* **✅ PhD-level depth**
* **✅ Manageable teaching load**
* **✅ Kids understand (not death by fragmentation)**
* **✅ Complete from theory to production**
* **✅ Original research included**
* **✅ Real code, real papers**

**🎯 WEEKLY SCHEDULE (REALISTIC)**

**Monday-Tuesday: Record 1-2 lectures (2-3 hours recording) Wednesday: Edit lecture, write blog post Thursday-Friday: Code implementation + short clips Weekend: Rest + planning**

**Total Time: ~15-20 hours/week (doable alongside full-time work)**