Week 11: Model Efficiency & Optimization

From 700GB to 40GB: Making AI Deployable

BSc Natural Language Processing

Discovery-Based Learning Approach

2025

The 350GB Problem

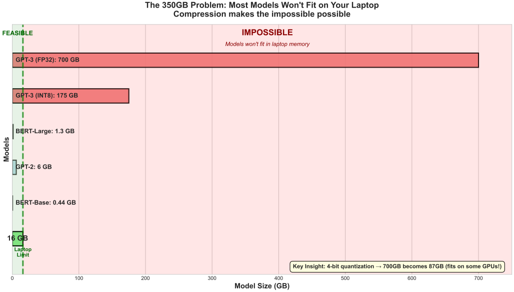
The Scenario:

You want to run GPT-3 locally for privacy

Your laptop has 16GB RAM

GPT-3 model size: 350GB

The Impossibility:
Model is 22× larger than your RAM
Loading would require 175GB of disk swap
Inference: 1 token per minute (unusable)



The Discovery:
"The 350GB problem has a 40GB solution"
4-bit quantization: 75% size reduction
Accuracy loss: only 3%

Discovery Question: How would YOU make a huge model fit on a small device?

Paradigm Shift: From Smaller Models to Compressed Models

OLD Approach (2015):

Problem: Large model won't fit

Solution: Train a smaller model

Example:

- GPT-2: 1.5B params $\rightarrow$ 117M params
- Size: 6GB $\rightarrow$ 500MB
- Accuracy: 85% $\rightarrow$ 67%
- Loss: 18 percentage points

Trade-off:
Smaller size, much worse performance

NEW Approach (2024):

Problem: Large model won't fit

Solution: Compress the large model

Example:

- GPT-3: 175B params (same capability)
- Size: 700GB $\rightarrow$ 87GB (INT4)
- Accuracy: 92% $\rightarrow$ 89%
- Loss: 3 percentage points

Trade-off:
Much smaller size, minimal performance loss

Key Insight: Compress post-training preserves learned knowledge better than training smaller

Real-World Deployments in 2024

On-Device LLMs:

1. LLaMA-2 7B on Phone

- Original: 28GB (FP32)
- Compressed: 3.5GB (4-bit)
- Method: Quantization
- Performance: 15 tokens/sec

2. Whisper in Browser

- Original: 3GB (large model)
- Compressed: 150MB (distilled)
- Method: Knowledge distillation
- Performance: Real-time transcription

Edge Computing:

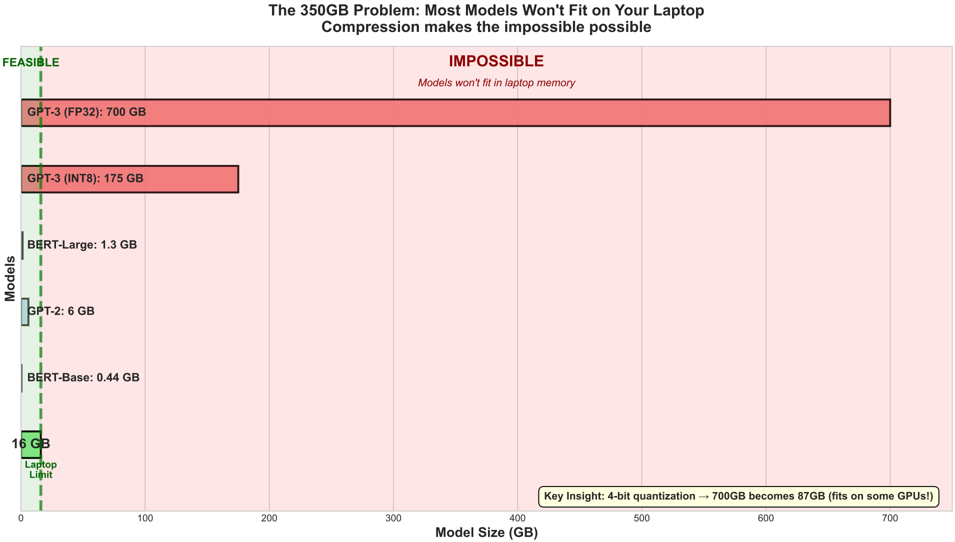
3. BERT on Arduino

- Original: 440MB (base)
- Compressed: 2MB (pruned + quantized)
- Method: 95% pruning + INT8
- Performance: 200ms inference

4. GPT-4 API Efficiency

- Latency: 800ms $\rightarrow$ 150ms
- Cost: \$0.03/1K $\rightarrow$ \$0.006/1K
- Method: Mixed precision + distillation
- Scale: Billions of requests/day

Deployment Reality: Compression enables AI everywhere (phones, browsers, microcontrollers)


The 350GB Problem: Most Models Won't Fit on Your Laptop
Compression makes the impossible possible
FEASIBLE
IMPOSSIBLE
Models won't fit in laptop memory
GPT-3 (FP32): 700 GB
GPT-3 (INT8): 175 GB
BERT-Large: 1.3 GB
GPT-2: 6 GB
BERT-Base: 0.44 GB
16 GB
Laptop Limit
Key Insight: 4-bit quantization → 700GB becomes 87GB (fits on some GPUs!)
Models
0
100
200
300
400
500
600
700
Model Size (GB)

Foundation 1: Model Size Problem (Detailed)

Memory Hierarchy:

Level	Size	Speed
L1 Cache	256KB	1ns
L2 Cache	8MB	5ns
L3 Cache	32MB	20ns
RAM	16GB	100ns
SSD	1TB	100μs

Fundamental Constraint:
Inference requires entire model in fast memory

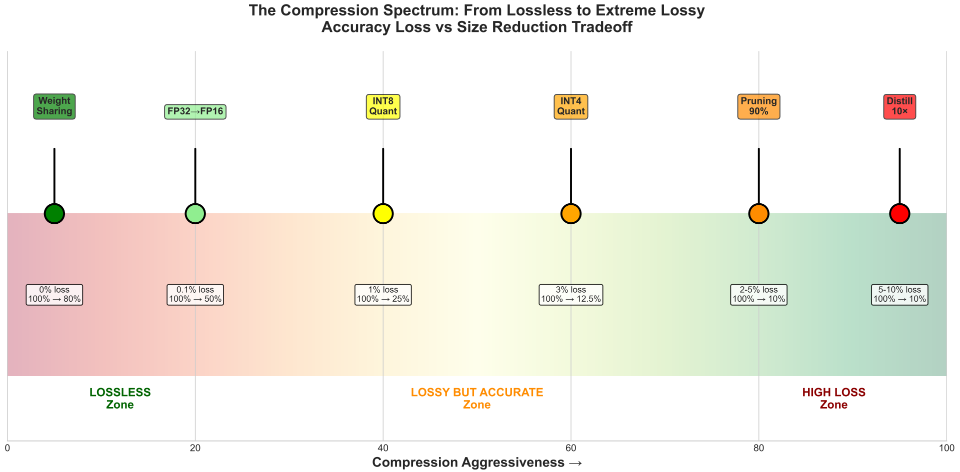
Model Size Evolution:

Model	Params	Size (FP32)
BERT-Base	110M	440MB
BERT-Large	340M	1.4GB
GPT-2	1.5B	6GB
GPT-3	175B	700GB
PaLM	540B	2.1TB

Trend: Models grow 10$\times$ every 2 years
Hardware grows 2$\times$ every 2 years
Gap widens without compression

Mathematical Reality: 175B params $\times$ 4 bytes/param = 700GB minimum memory

Foundation 2: Compression Spectrum (Visual)



The Spectrum:
Lossless (perfect accuracy, small gains) to Lossy (large gains, small accuracy loss)

Foundation 2: Compression Spectrum (Detailed)

Lossless Methods:

1. Weight Sharing

- Technique: Cluster similar weights
- Reduction: 10-20%
- Accuracy: 100% preserved
- Use case: When zero loss required

2. Low-Rank Factorization

- Technique: $W = UV^T$ decomposition
- Reduction: 30-40%
- Accuracy: 99-100%
- Use case: Dense layers

Lossy Methods:

3. Quantization (INT8)

- Technique: FP32 $\rightarrow$ 8-bit integers
- Reduction: 75%
- Accuracy: 95-99%
- Use case: Most deployments

4. Quantization (INT4)

- Technique: FP32 $\rightarrow$ 4-bit integers
- Reduction: 87.5%
- Accuracy: 90-97%
- Use case: Mobile/edge devices

Design Decision: Choose method based on accuracy tolerance and size requirements

Foundation 3: De

Foundation 3: Deployment Platforms (Detailed)

Server (80-512GB RAM):

Compression:

- GPT-3: 700GB $\rightarrow$ 350GB (FP16)
- Method: Mixed precision
- Latency: $<$100ms

Edge (4-16GB RAM):

Compression:

- GPT-3: 700GB $\rightarrow$ 87GB (INT4)
- Method: Quantization
- Latency: $<$500ms

Mobile (2-4GB RAM):

Compression:

- LLaMA-7B: 28GB $\rightarrow$ 3.5GB
- Method: 4-bit + pruning
- Battery critical

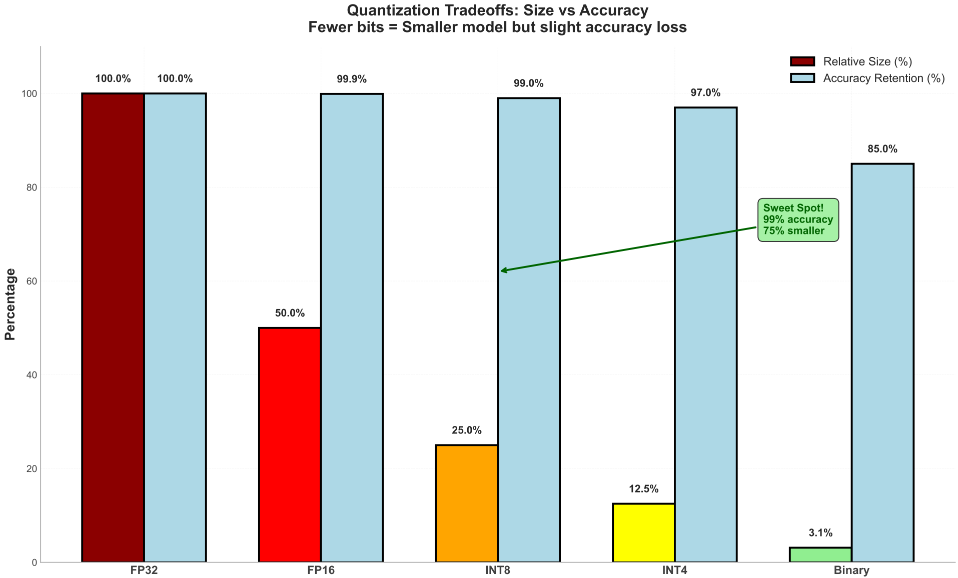
Microcontroller (256KB-2MB):

Compression:

- BERT: 440MB $\rightarrow$ 2MB
- Method: Prune + distill + INT8
- 200$\times$ reduction

Deployment Reality: Platform constraints drive compression method selection

Method 1: Quantization (Visual)



Method 1: Quantization (Detailed Mathematics)

Quantization Formula:

Forward (FP32 $\rightarrow$ INT8):

$$q = \text{round}\left(\frac{x - x_{\text{min}}}{s}\right)$$

where $s = \frac{x_{\text{max}} - x_{\text{min}}}{255}$ (scale)

Inverse (INT8 $\rightarrow$ FP32):

$$\hat{x} = q \times s + x_{\text{min}}$$

Numerical Example:

- Weight: $x = 0.374$ (FP32)
- Range: $[-1.0, 1.0]$
- Scale: $s = 2.0/255 = 0.00784$
- Zero-point: 127
- Quantized: $q = 175$ (INT8)
- Recovered: $\hat{x} = 0.376$

Precision Comparison:

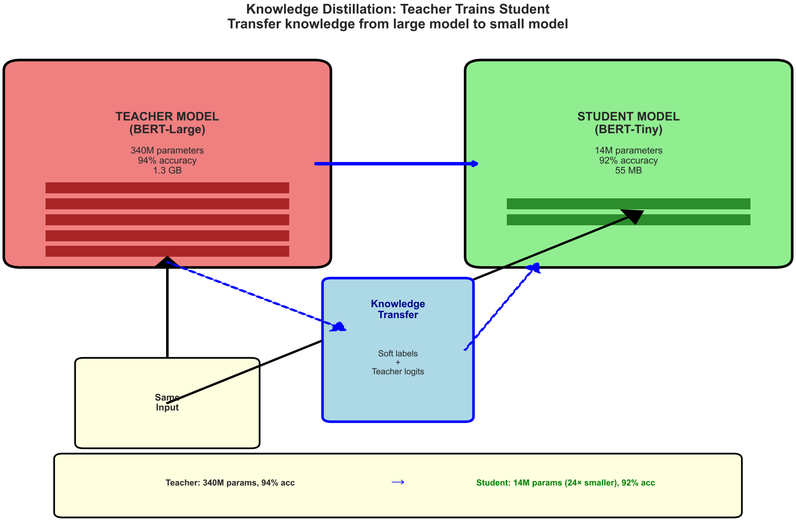
Type	Bits	Range	Precision
FP32	32	$\pm 3.4 \times 10^{38}$	7 digits
FP16	16	$\pm 6.5 \times 10^{4}$	3 digits
INT8	8	-128 to 127	256 values
INT4	4	-8 to 7	16 values

Real Results:

- BERT-Base FP32: 440MB, 89.5%
- BERT-Base INT8: 110MB, 89.1%
- BERT-Base INT4: 55MB, 87.8%

When to Use:
Default choice for most deployments
Hardware support widely available

Method 2: Knowledge Distillation (Visual)



Method 2: Knowledge Distillation (Detailed Process)

Distillation Loss:

$$\mathcal{L} = \alpha \mathcal{L}_{\text{hard}} + (1-\alpha)\mathcal{L}_{\text{soft}}$$

Hard Loss (ground truth):

$$\mathcal{L}_{\text{hard}} = -\sum_i y_i \log p_i^{\text{student}}$$

Soft Loss (teacher knowledge):

$$\mathcal{L}_{\text{soft}} = -\sum_i p_i^{\text{teacher}} \log p_i^{\text{student}}$$

Temperature scaling: $p_i = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$

Typical Values:

- $\alpha = 0.5$ (equal weighting)
- $T = 3-5$ (temperature)

Concrete Example:

Teacher: BERT-Large

- Parameters: 340M
- Size: 1.4GB (FP32)
- Accuracy: 94.0% (GLUE)
- Inference: 120ms

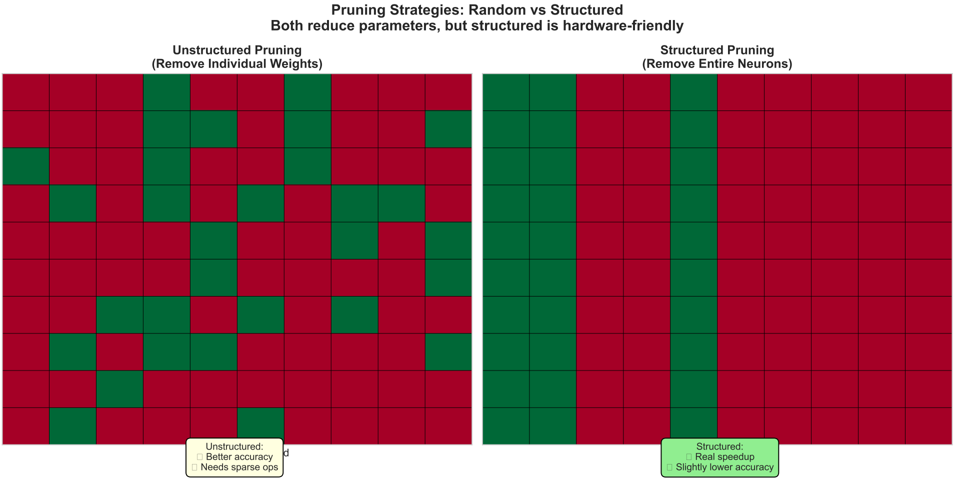
Student: DistilBERT

- Parameters: 66M (5× smaller)
- Size: 260MB (FP32)
- Accuracy: 92.5% (1.5% loss)
- Inference: 60ms (2× faster)

When to Use:
When you need >5× compression
When you can retrain the model
For production deployment at scale

Training Process: Student model trained on teacher's logits (soft targets) + true labels

Method 3: Pruning (Visual)



Core Idea: Remove unimportant weights or neurons from the network

Size Reduction: 90% sparsity = 10× fewer weights

Method 3: Pruning (Detailed Strategies)

Unstructured Pruning:

Algorithm:

1. Train full model
2. Compute weight magnitudes $|w_i|$
3. Remove smallest $p\%$ weights
4. Fine-tune remaining weights

Advantages:

- Highest compression (90-95%)
- Minimal accuracy loss
- Flexible per-layer pruning

Disadvantages:

- Irregular sparsity patterns
- Requires sparse matrix support
- Limited hardware acceleration

Structured Pruning:

Algorithm:

1. Train full model
2. Compute neuron/channel importance
3. Remove entire neurons/channels
4. Fine-tune remaining network

Advantages:

- Direct hardware speedup
- No special sparse libraries
- Smaller actual model size

Disadvantages:

- Lower compression (40-60%)
- More accuracy loss
- Coarser granularity

When to Use:

Method 4: Low-Rank Factorization (Visual)

Matrix Decomposition:

Original weight matrix:

$$W \in \mathbb{R}^{m \times n}$$

Decomposed form:

$$W \approx UV^T$$

where $U \in \mathbb{R}^{m \times r}$, $V \in \mathbb{R}^{n \times r}$, $r \ll \min(m, n)$

Parameter Count:

- Original: $m \times n$
- Factorized: $m \times r + n \times r = r(m + n)$
- Reduction: $\frac{mn}{r(m+n)}$

Numerical Example:

Dense layer: 1024×1024

Original:

- Parameters: $1024^2 = 1,048,576$
- Size: 4MB (FP32)

Factorized ($r = 64$):

- Parameters: $64(1024 + 1024) = 131,072$
- Size: 512KB (FP32)
- Reduction: $8\times$ smaller
- Accuracy loss: $<1\%$

SVD Insight:
Most variance captured by first r singular values
Remaining $(n - r)$ dimensions contribute little

Mathematical Foundation: Singular Value Decomposition (SVD) provides optimal low-rank approximation

Method 4: Low-Rank Factorization (Detailed Analysis)

SVD Algorithm:

Step 1: Compute SVD

$$W = U\Sigma V^T$$

where $\Sigma = \text{diag}(\sigma_1, \ldots, \sigma_n)$ with $\sigma_1 \geq \sigma_2 \geq \ldots$

Step 2: Choose rank r

Energy threshold: $\frac{\sum_{i=1}^{r} \sigma_i^2}{\sum_{i=1}^{n} \sigma_i^2} \geq 0.95$

Step 3: Truncate

$$W_r = U_r \Sigma_r V_r^T$$

where $U_r \in \mathbb{R}^{m \times r}$, $\Sigma_r \in \mathbb{R}^{r \times r}$, $V_r \in \mathbb{R}^{n \times r}$

Step 4: Absorb Σ_r

$$W_r = (U_r \sqrt{\Sigma_r})(\sqrt{\Sigma_r} V_r^T)$$

Real Results:

BERT Embedding Layer:

- Original: $30K \times 768 = 23M$ params
- Factorized ($r = 128$): $128(30K + 768) = 4M$
- Reduction: $5.8\times$ smaller
- Accuracy: 89.5% $\rightarrow$ 89.2%

GPT-2 Attention:

- Original: $768 \times 768 = 590K$ params/layer
- Factorized ($r = 64$): $64 \times 1536 = 98K$
- Reduction: $6\times$ smaller
- Accuracy: Minimal loss ($<$0.5%)

When to Use:
Dense linear layers (embeddings, attention)
When weight matrix has low intrinsic rank
Combined with quantization for best results

Compression Sweet Spot: $r \approx 10 - 20\%$ **of original dimension balances size and accuracy**

Method 5: Weight Sharing (Visual)

Clustering Approach:

Before: Each weight is unique

- 175B unique floating-point values
- Full precision per weight
- High memory requirement

After: Weights share codebook

- 256 unique cluster centers
- Indices point to codebook
- 2-4 bits per weight (index)

Storage:
Codebook: k values (float)
Indices: n values (2-4 bits)
Total: Much smaller than n floats

K-Means Clustering:

Algorithm:

1. Collect all n weights
2. Run k-means with k clusters
3. Replace each weight with nearest cluster center
4. Store: cluster centers + indices

Numerical Example:

- Weights: $[0.72, 0.69, -0.31, -0.28, ...]$
- Clusters ($k = 4$): $[0.7, -0.3, 0.0, 1.2]$
- Indices: $[0, 0, 1, 1, ...]$ (2 bits each)
- Original: 4 bytes/weight
- Compressed: 0.25 bytes/weight
- Reduction: 16$\times$ smaller

Weight Sharing: Lossless-to-lossy spectrum depending on number of clusters

Method 5: Weight Sharing (Detailed Implementation)

Compression Analysis:

Storage Requirements:
Codebook size: k clusters $\times$ 4 bytes
Index size: n weights $\times$ $\lceil \log_2 k \rceil$ bits
Total: $4k + n\lceil \log_2 k \rceil / 8$ bytes

Compression Ratio:

$$\text{Ratio} = \frac{4n}{4k + n\lceil \log_2 k \rceil / 8}$$

Example ($n = 1M$, $k = 256$):

- Original: $1M \times 4 = 4$MB
- Codebook: $256 \times 4 = 1$KB
- Indices: $1M \times 1 = 1$MB (8 bits)
- Total: 1MB + 1KB $\approx$ 1MB
- Ratio: $4\times$ compression

Accuracy Trade-offs:

Clusters	Compression	Accuracy
$k = 2$	$32\times$	60-70%
$k = 16$	$8\times$	85-90%
$k = 256$	$4\times$	95-99%
$k = 4096$	$2.7\times$	99-100%

Real Results:

- BERT ($k = 256$): 440MB $\rightarrow$ 110MB
- Accuracy: 89.5% $\rightarrow$ 89.3%
- Combined with pruning: $10\times$ total

When to Use:
When you need lossless compression
Combined with quantization/pruning
For weight-heavy models

Hybrid Approach: Weight sharing + quantization achieves 10-20$\times$ compression

Method 6: Mixed Precision Training (Visual)

Precision Strategy:

FP32 (Master Weights):

- High precision for gradients
- Prevents underflow
- Kept in optimizer state

FP16 (Forward/Backward):

- Fast computation ($2\times$)
- 50% memory reduction
- Hardware acceleration (Tensor Cores)

INT8 (Inference):

- Minimal memory
- $4\times$ faster than FP32
- Quantized after training

Training Loop:

1. **Forward:** FP16 computation
2. **Loss:** FP16 calculation
3. **Loss Scaling:** Multiply by 2^{14}
4. **Backward:** FP16 gradients
5. **Unscale:** Divide by 2^{14}
6. **Update:** FP32 master weights
7. **Copy:** FP32 $\rightarrow$ FP16 for next iteration

Loss Scaling:
Prevents gradient underflow in FP16
Typical scale: 2^{14} to 2^{16}

Mixed Precision: Best of both worlds (FP32 stability + FP16 speed)

Method 6: Mixed Precision Training (Detailed Benefits)

Speed Improvements:

Model	FP32	Mixed
BERT-Base	280 samples/s	560 samples/s
GPT-2	120 samples/s	240 samples/s
ResNet-50	340 images/s	680 images/s

Speedup: Consistent 2× across models

Memory Savings:

Component	FP32	Mixed
Activations	100%	50%
Gradients	100%	50%
Weights	100%	100%
Optimizer	200%	200%
Total	400%	350%

Hardware Support:

NVIDIA Tensor Cores:

- FP16: 125 TFLOPS (V100)
- FP32: 15 TFLOPS (V100)
- Speedup: 8× theoretical
- Real speedup: 2-3× (memory bound)

TPU v4:

- BF16: 275 TFLOPS
- FP32: 68 TFLOPS
- Speedup: 4×

When to Use:
Training large models (GPT-3, BERT)
When you have Tensor Core GPUs
Default for modern PyTorch/TensorFlow

Industry Standard: All large model training uses mixed precision (2020+)

Method 7: Dynamic & Adaptive Computation (Visual)

Early Exit Strategy:

Idea: Not all inputs need full network

Easy examples: Exit after layer 3
Medium examples: Exit after layer 6
Hard examples: Use all 12 layers

Mechanism:

- Add classifier at each layer
- Compute confidence score
- If confidence $>$ threshold, exit
- Otherwise, continue to next layer

Average Speedup:

- Easy: $4\times$ (3 layers vs 12)
- Medium: $2\times$ (6 layers vs 12)
- Hard: $1\times$ (all 12 layers)
- Overall: $2.5\times$ average

Adaptive Attention:

Idea: Not all tokens need full attention

Important tokens: Full attention
Filler words: Sparse attention

Example (12-word sentence):

- "The": 20% attention (2 heads)
- "cat": 100% attention (8 heads)
- "sat": 100% attention (8 heads)
- "on": 20% attention (2 heads)
- "the": 20% attention (2 heads)
- "mat": 100% attention (8 heads)

Computation:

- Full: $12 \times 8 = 96$ head computations
- Adaptive: 48 head computations
- Reduction: 50%

Adaptive Computation: Allocate resources based on input complexity

Method 7: Dynamic & Adaptive Computation (Detailed Results)

Early Exit Networks:

BERT with 3 exits:

- Exit 1 (Layer 4): 35% of samples
- Exit 2 (Layer 8): 45% of samples
- Exit 3 (Layer 12): 20% of samples

Performance:

- Average layers: 6.8 vs 12
- Speedup: 1.76×
- Accuracy: 89.5% → 89.1%
- Loss: 0.4 percentage points

Confidence Threshold:

- High (0.95): Safe, slower (1.3×)
- Medium (0.85): Balanced (1.76×)
- Low (0.75): Risky, faster (2.2×)

Adaptive Attention:

GPT-2 with Adaptive Heads:

- Content words: 8 heads (100%)
- Function words: 2 heads (25%)
- Punctuation: 1 head (12.5%)

Results:

- Computation: 60% of full model
- Speedup: 1.67×
- Perplexity: 18.2 → 18.5
- Quality: Minimal degradation

When to Use:
Production with varied input complexity
When average-case matters more than worst-case
Combined with other compression methods

Research Frontier: Adaptive methods are active area of research (2023-2025)

GPT-3 Deployment Impossibility

The Numbers:

- Parameters: 175 billion
- Precision: FP32 (4 bytes each)
- Total size: $175B \times 4 = 700$GB
- Typical server RAM: 64-256GB
- Your laptop RAM: 16GB

Impossibility Ratio:
Model size / Laptop RAM $= 44\times$

Even high-end servers struggle (3-11$\times$ over capacity)

Consequences:

Without Compression:

- Must use disk swap
- Inference: 60 seconds per token
- Unusable for production
- Energy: 500W continuous
- Cost: $10-50 per query

Business Impact:

- Cannot deploy locally
- Must use cloud APIs
- Privacy concerns
- Latency issues
- Ongoing costs

Root Problem: Model capacity requirements exceed deployment hardware by orders of magnitude

Initial Approach: Train Smaller Model

The Naive Solution:

"If GPT-3 is too big, train GPT-2 instead"

GPT-3 175B:

- Size: 700GB (FP32)
- Parameters: 175B
- Accuracy: 92% (few-shot)
- Training: \$4.6M

$\Downarrow$ Reduce size 100$\times$

GPT-2 1.5B:

- Size: 6GB (FP32)
- Parameters: 1.5B
- Accuracy: 67% (few-shot)
- Training: \$50K

The Problem:

Accuracy Drop: 25 Percentage Points

Capability Loss:

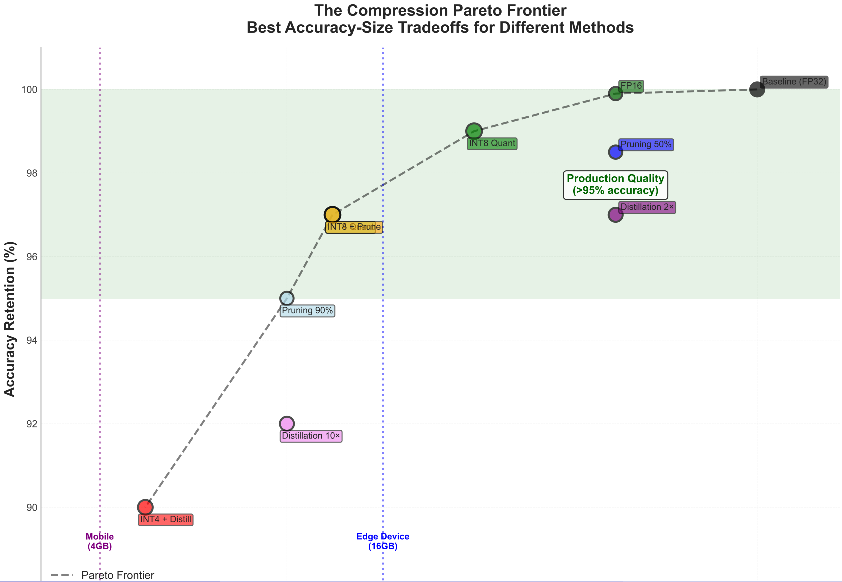
- GPT-3: Complex reasoning, analogies
- GPT-2: Simple pattern matching
- Emergence: Lost at smaller scale

Scaling Laws:
Performance $\propto$ (parameters)$^{0.3}$

To match GPT-3 at 1.5B params:
Need 1000$\times$ more data (impossible)

Lesson: Model capacity matters - smaller models cannot simply be trained to match larger ones

Performance Analysis: Why Smaller Models Fail



Root Cause: The Capacity Hypothesis

Theoretical Framework:

Model Capacity:

$$C = f(\text{parameters}, \text{architecture})$$

Knowledge Stored:

$$K \leq C$$

Performance:

$$P \propto K$$

Implications:

- Smaller model $\Rightarrow$ Less capacity
- Less capacity $\Rightarrow$ Less knowledge
- Less knowledge $\Rightarrow$ Worse performance

Numerical Evidence:

Model	Params	Accuracy
BERT-Tiny	14M	78%
BERT-Small	28M	83%
BERT-Medium	66M	86%
BERT-Base	110M	89.5%
BERT-Large	340M	94%

Solution Requirement:
Preserve model capacity (parameters)
Reduce storage/memory footprint
$\Rightarrow$ Compression, not replacement

Diagnosis: Performance tied to parameter count - compression must preserve parameters

Solution Insight: Compress Post-Training

The Breakthrough Idea:

OLD: Train small model (loses knowledge)

$\Downarrow$

NEW: Train large, then compress

Why This Works:

1. Train full-capacity model
2. Model learns all knowledge
3. Compress learned weights
4. Knowledge preserved (mostly)
5. Fit in deployment memory

Key Observation:
Learned weights have structure
Structure enables compression
Random weights don't compress well

Compression Opportunity:

Trained Weights Properties:

- Clustered values (weight sharing)
- Low effective rank (factorization)
- Many near-zero (pruning)
- Narrow range (quantization)

Concrete Example:

- BERT attention weights
- 95% of weights in [-0.5, 0.5]
- Can use 8 bits instead of 32
- 4$\times$ compression with 0.4% loss

Contrast with Random:

- Random weights: Uniform distribution
- No structure to exploit
- Compression hurts accuracy severely

Critical Insight: Trained weights have exploitable structure that random weights lack

The Math:

Quantization Function:

$$q = \text{round}\left(\frac{x - x_{\text{min}}}{s}\right)$$

where scale $s = \frac{x_{\text{max}} - x_{\text{min}}}{255}$

Dequantization Function:

$$\hat{x} = q \times s + x_{\text{min}}$$

Error:

$$\epsilon = |\hat{x} - x| \leq \frac{s}{2}$$

Key Idea:

- Map range $[x_{\text{min}}, x_{\text{max}}]$ to $[0, 255]$
- Store integer index (1 byte)
- Recover approximate value

Numerical Walkthrough:

Weight Layer Statistics:

- Min: -1.2
- Max: $+0.8$
- Range: 2.0
- Scale: $s = 2.0/255 = 0.00784$

Quantize $x = 0.374$**:**

1. Shift: $0.374 - (-1.2) = 1.574$
2. Scale: $1.574/0.00784 = 200.76$
3. Round: $q = 201$ (INT8)

Dequantize $q = 201$**:**

1. Unscale: $201 \times 0.00784 = 1.576$
2. Unshift: $1.576 + (-1.2) = 0.376$
3. Error: $|0.376 - 0.374| = 0.002$

Quantization Error: Bounded by half the quantization step (0.00392 in this example)

Worked Example: BERT Layer Quantization

Layer: BERT Attention Weights

Original (FP32):

- Shape: 768×768
- Weights: 590,592
- Min: -0.487
- Max: $+0.512$
- Mean: 0.003
- Std: 0.124
- Size: $590K \times 4 = 2.36$MB

Quantization Parameters:

- Range: $[-0.487, 0.512]$
- Scale: $(0.512 - (-0.487))/255 = 0.00392$
- Zero-point: 127 (symmetric)

Quantized (INT8):

- Shape: 768×768 (unchanged)
- Values: INT8 in $[0, 255]$
- Size: $590K \times 1 = 590$KB
- Reduction: $4\times$ smaller

Sample Weights:

FP32	INT8	Recovered
0.374	201	0.376
-0.251	67	-0.249
0.089	150	0.090
-0.412	25	-0.413
0.501	255	0.512

Accuracy:

- Original BERT: 89.5%
- Quantized INT8: 89.1%
- Loss: 0.4 percentage points

Real Result: $4\times$ compression with ¡0.5% accuracy loss on BERT-Base

Validation: Real Model Compression Results

BERT-Base Compression:

Method	Size	Accuracy
FP32 Baseline	440MB	89.5%
FP16	220MB	89.5%
INT8	110MB	89.1%
INT4	55MB	87.8%
INT8 + Pruning	22MB	87.5%

Best Trade-off:

- INT8: 4× smaller, 0.4% loss
- Production standard (2024)
- Hardware accelerated

GPT-3 Compression:

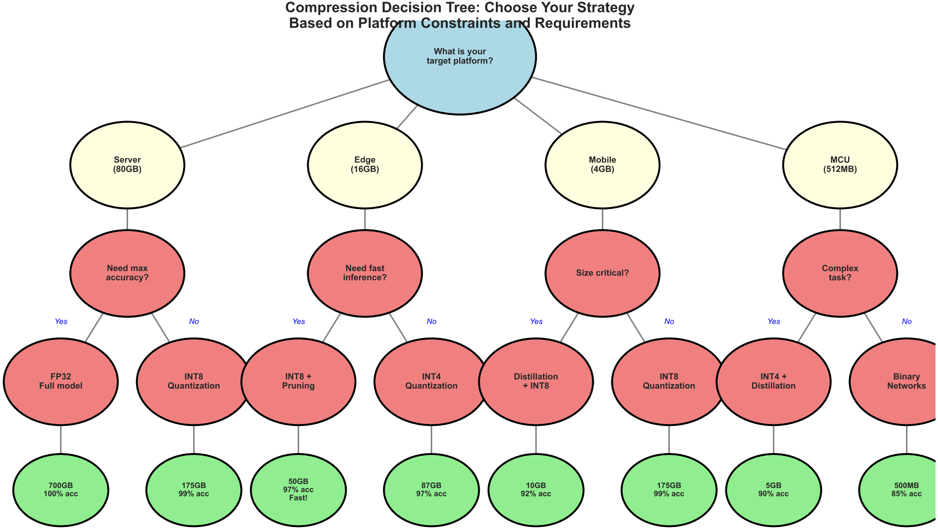
Precision	Size	Quality
FP32	700GB	100%
FP16	350GB	100%
INT8	175GB	98%
INT4	87GB	95%

Deployment Reality:

- OpenAI API: INT8 (likely)
- 4× memory reduction
- 2× throughput increase
- Enables profitable deployment

Industry Adoption: All major LLM APIs use INT8 quantization (2024)

Decision Tree: Choosing Compression Method



Meta-Knowledge: When NOT to Use

Quantization:

Avoid when:

- Model has high dynamic range
- Batch norm layers (unstable)
- Small models (<100M params)
- Research/debugging phase

Pruning:

Avoid when:

- No sparse matrix libraries
- Model already small
- All weights are important
- Cannot fine-tune after pruning

Distillation:

Avoid when:

- No budget to retrain
- Teacher model unavailable
- Task requires all model capacity
- Target is $<5\times$ compression

Low-Rank:

Avoid when:

- Weights are full-rank
- Convolutional layers (better methods)
- Recurrent connections
- Model has few dense layers

Anti-Patterns: Know when NOT to use each method to avoid wasted effort

Common Pitfalls and Solutions

Pitfall 1: Calibration

Problem:

- Quantize with training data ranges
- Deploy on different distribution
- Activation ranges differ
- Severe accuracy drop

Solution:

- Calibrate on representative data
- 1000+ diverse examples
- Measure activation ranges
- Use percentile (99%) not max

Pitfall 2: INT4 Overflow

Problem:

- INT4 range: $[-8, 7]$
- Outlier weights cause clipping

Pitfall 3: Distillation Failure

Problem:

- Student too small ($>20\times$ smaller)
- Cannot learn teacher's knowledge
- Converges to random baseline

Solution:

- Limit compression to 5-10$\times$
- Use intermediate layers
- Progressive distillation

Pitfall 4: Compound Methods

Problem:

- Prune + quantize + distill = fail
- Errors compound
- 10% + 5% + 3% $\neq$ 18%
- Actual: 25% degradation

Success Metrics: How to Measure

Primary Metrics:

1. Size Reduction

$$R = \frac{\text{Original Size}}{\text{Compressed Size}}$$

Target: 4-10$\times$ for deployment

2. Accuracy Preservation

$$A = \frac{\text{Compressed Accuracy}}{\text{Original Accuracy}}$$

Target: >95% (absolute <3% loss)

3. Latency Improvement

$$L = \frac{\text{Original Latency}}{\text{Compressed Latency}}$$

Target: 2-4$\times$ speedup

4. Energy Efficiency

Original Energy

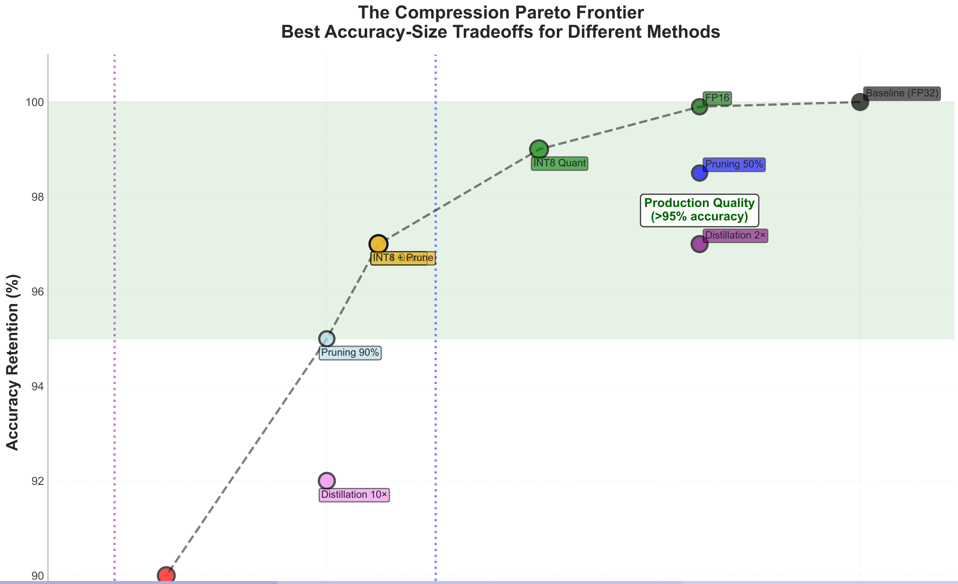
Real Benchmark (BERT):

Method	R	A	L	E
Baseline	1×	100%	1×	1×
FP16	2×	100%	1.5×	1.8×
INT8	4×	99%	2.5×	3.2×
INT4	8×	96%	3.5×	5.1×
Pruned	10×	95%	1.2×	1.5×

Trade-off Analysis:

- INT8: Best balance (4×, 99%, 2.5×)
- INT4: Maximum compression
- Pruning: Size without speedup (need sparse support)

Unified Framework: Method Comparison



Modern Applications: On-Device AI Revolution

Smartphone LLMs (2024):

Apple Intelligence (iPhone 15):

- Model: 3B parameter LLM
- Original: 12GB (FP32)
- Compressed: 1.5GB (4-bit + pruning)
- Methods: INT4 + 50% pruning
- Performance: 30 tokens/sec
- Privacy: 100% on-device

Google Gemini Nano:

- Model: 1.8B parameters
- Size: 900MB (INT8)
- Latency: 40 tokens/sec
- Battery: 1% per 1000 tokens

Edge Computing:

Raspberry Pi 4 (8GB):

- LLaMA-2 7B quantized (INT4)
- Size: 3.5GB
- Speed: 2 tokens/sec
- Use case: Local assistant

NVIDIA Jetson (16GB):

- GPT-J 6B (INT8)
- Size: 6GB
- Speed: 15 tokens/sec
- Use case: Robotics, drones

Impact:
Compression enables privacy-preserving AI
Zero cloud dependency
Millisecond latency

2024 Reality: Compression makes AI ubiquitous (phones, cars, appliances)

Implementation: PyTorch Quantization in 15 Lines

Dynamic Quantization:

```
import torch

# Load pre-trained model
model = BertForSequenceClassification
   .from_pretrained('bert-base')

# Quantize to INT8
quantized_model = torch.quantization
   .quantize_dynamic(
      model,
      {torch.nn.Linear},
      dtype=torch.qint8
   )

# Save compressed model
torch.save(quantized_model,
   'bert_int8.pt')
```

Result: 440MB $\rightarrow$ 110MB ($4\times$)

Static Quantization:

```
# Prepare model
model.qconfig = torch.quantization
   .get_default_qconfig('fbgemm')
torch.quantization.prepare(model)

# Calibrate with data
for batch in calibration_data:
   model(batch)

# Convert to INT8
quantized_model = torch.quantization
   .convert(model)

# Inference
with torch.no_grad():
   output = quantized_model(input)
```

Advantage: Better accuracy (calibrated ranges)

Production Code: PyTorch provides built-in quantization (torch.quantization module)

Key Takeaways: Five Principles

Model Efficiency Fundamentals

1. **Compression Preserves Knowledge**
   Train large, compress post-training beats training small
   Example: GPT-3 INT4 (87GB) outperforms GPT-2 (6GB)

2. **Quantization is the Default**
   $4\times$ reduction, $<1\%$ accuracy loss, hardware accelerated
   Use INT8 unless you have specific constraints

3. **Platform Drives Strategy**
   Server: FP16/INT8 — Edge: INT4 — Mobile: INT4+Pruning — MCU: Distillation+INT8
   Deployment memory determines compression needs

4. **Combine Methods Carefully**
   Quantization + (Pruning OR Distillation) works
   All three together compounds errors

5. **Measure Four Metrics**
   Size reduction, accuracy, latency, energy
   Optimize for the bottleneck

Summary: Compression makes modern AI deployable everywhere

Next Steps: Lab & Week 12

Week 11 Lab:

Hands-On Activities:

1. Quantize BERT (FP32 $\rightarrow$ INT8)
2. Measure size, accuracy, latency
3. Distill GPT-2 (1.5B $\rightarrow$ 300M)
4. Prune ResNet (90% sparsity)
5. Deploy quantized model

Tools:

- PyTorch quantization API
- Hugging Face transformers
- ONNX Runtime

Deliverable:
Compress a model 10$\times$ with $<$3% accuracy loss

Week 12: Ethics & Fairness

Efficiency $\rightarrow$ Ethics Link:

Sustainability:

- GPT-3 training: 1287 MWh
- Carbon: 550 tons CO_2
- Compression reduces deployment energy 5$\times$

Accessibility:

- On-device AI: No cloud required
- Privacy-preserving inference
- Works in low-connectivity regions

Democratization:

- Run LLMs on $200 hardware
- No API costs
- Open access to AI

Bridge to Ethics: Efficiency enables sustainable, accessible, democratized AI