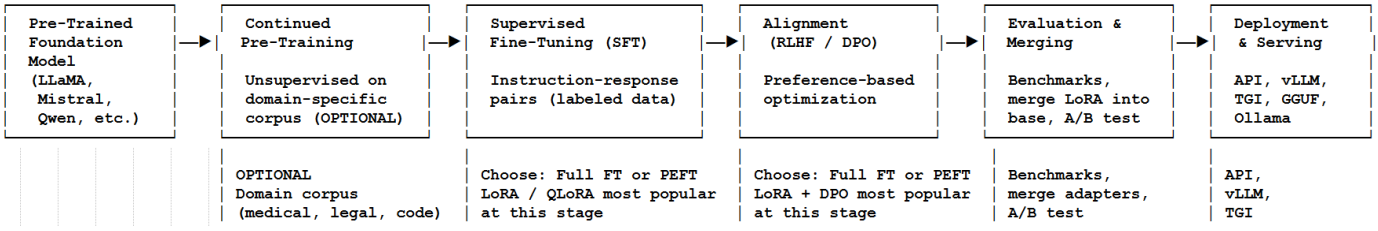
Fine Tuning

Fine Tuning:

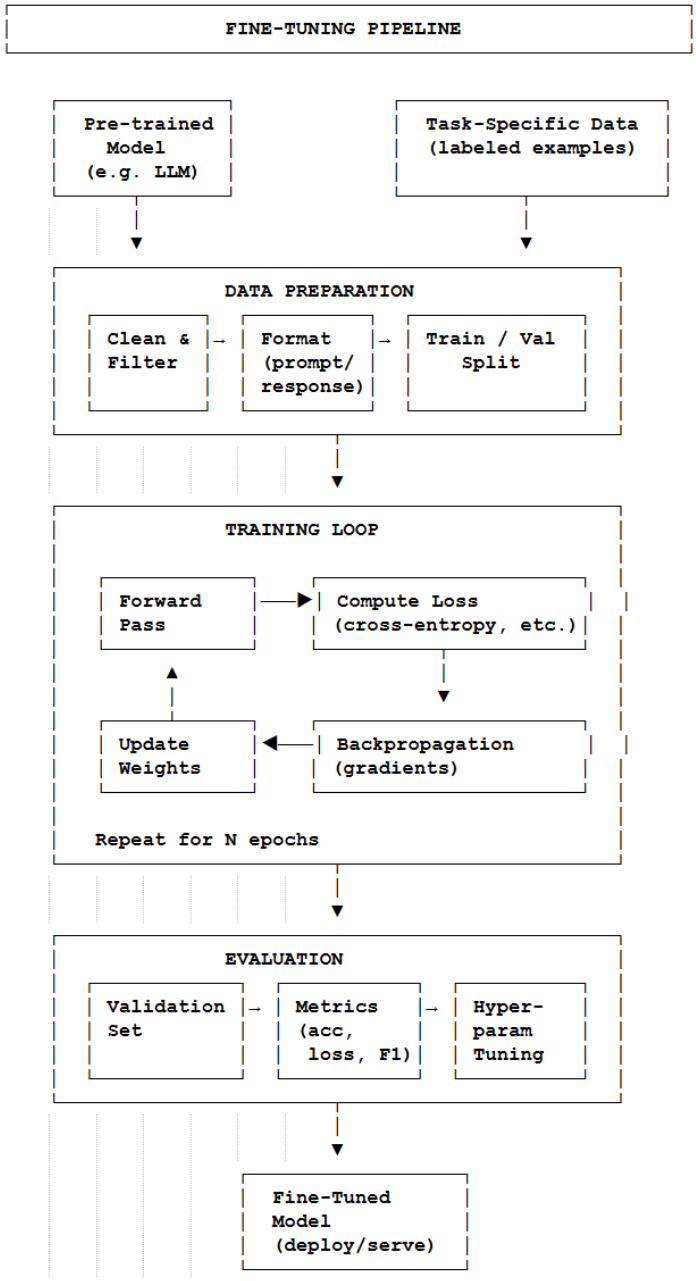
- Full Fine Tuning
  - Standard Full Fine Tuning
  - Feature Extraction
  - Gradual Unfreezing

- PEFT – Parameter Efficient Fine Tuning
  - Additive Methods
    - Bottleneck Adapters
    - Soft Prompts
    - IA^3
  - Reparameterization
    - LORA
    - QLORA
    - DORA
    - LORA +
  - Selective Methods
    - rsLORA
    - BitFit - (bias only)
    - Fish Mask
  - Hybrid Methods
    - QLORA
    - LongLORA
    - LORA-FA
  - Prompt Methods
    - Prefix Tuning
    - P-Tuning V2
    - Prompt Tuning
- Alignment Tuning
  - RLHF
  - DPO
  - ORPO / KTO

Typical End to End Pipeline



Fine Tuning Detailed Breakdown: How its done



The Big Picture

Imagine you hired a brilliant generalist — someone who has read millions of books, articles, and websites. They know a little about everything. That's your pre-trained foundation model.

Now imagine you want this generalist to become a specialist — say, a medical diagnostician.
You sit them down and train them intensively on thousands of medical cases, patient records, and diagnostic reports.

Full fine-tuning is that intensive retraining. You take the entire model — every single connection, every weight, every parameter — and allow ALL of them to be updated based on your new, task-specific data.

Fine Tuning

Why It Exists

Pre-trained models learn general language patterns during pre-training (which costs millions of dollars and takes weeks/months on huge GPU clusters). They learn grammar, facts, reasoning patterns, and general knowledge from massive internet-scale datasets.

But this general knowledge isn't enough for specific tasks. A model that can write poetry might be terrible at classifying legal contracts. A model that can summarize news might fail at generating SQL queries.

Full fine-tuning bridges this gap: take the general knowledge and reshape ALL of it toward your specific goal.

How It Works Conceptually

- **Step 1: Start With a Pre-Trained Model**

Think of the model as a massive web of interconnected numbers (called weights or parameters).
A model like LLaMA-7B has 7 billion of these numbers. GPT-3 has 175 billion.
Each number contributes to how the model processes and generates language.

During pre-training, these billions of numbers were carefully tuned so the model could predict the next word in a sentence across trillions of words of text. The result is a model that "understands" language at a deep statistical level.

- **Step 2: Prepare Your Task-Specific Dataset:**

You gather data specific to your goal. Examples:
- Sentiment analysis: thousands of movie reviews labeled "positive" or "negative"
- Medical Q&A: thousands of (medical question → accurate answer) pairs
- Code generation: thousands of (natural language description → working code) pairs
- Summarization: thousands of (long article → concise summary) pairs

The key is that this dataset is much smaller than the pre-training data.
Pre-training might use trillions of tokens. Fine-tuning might use thousands to millions of examples.

Data Formatting Matters: For modern LLM fine-tuning, the format of your data is just as important as the content.

Common formats include:
- **Instruction-tuning format**: structured as (system prompt → user instruction → assistant response) triples. This is the standard for chat/instruction models.

- **Completion-only format:** simple (input → output) pairs without role structure.

- **Chat templates:** model-specific formatting (e.g., ChatML, LLaMA chat format) that the model was pre-trained with. Using the wrong template can significantly degrade performance.

Loss Masking:
During training, you typically only compute the loss on the output/response tokens, not on the input/instruction tokens. This tells the model "learn to generate better responses" rather than "learn to memorize the questions." This is sometimes called "completion-only loss" and is controlled by flags like ****train_on_input=False**** in most fine-tuning frameworks.

- **Step 3: The Training Loop (What Actually Happens)**

Here's where the magic happens. For each example in your dataset, the model goes through a cycle:

A) Forward Pass — The Model Makes a Prediction:
You feed an input into the model. The input flows through all the layers — embedding layer, attention layers, feed-forward layers — and the model produces an output (a prediction).

Think of it like water flowing through a complex network of pipes.
Each pipe has a valve (a weight) that controls how much water flows through.
The water enters at one end (input) and comes out the other end (output/prediction).

B) Loss Calculation — How Wrong Was It?
You compare the model's prediction to the correct answer from your dataset.
The difference is called the loss. A high loss means the model was very wrong. A low loss means it was close.

For example, if the model predicted "negative" for a movie review that was actually "positive," the loss would be high.

C) Backpropagation — Tracing the Blame
This is the critical step. The algorithm works backward through the entire network, asking: "Which weights contributed to this mistake, and by how much?"
Imagine the water came out the wrong pipe at the end.
Backpropagation traces backward through every pipe junction, calculating exactly which valves need to be adjusted and by how much to redirect the water correctly.
This produces a gradient for every single weight in the model — a mathematical direction that says "adjust this weight by this amount to reduce the error."

D) Weight Update — Adjusting Everything
Using an optimizer (like Adam or SGD), every weight in the model is nudged in the direction that reduces the error.
The size of each nudge is controlled by the learning rate — a hyperparameter you set.
- Learning rate too high → model changes too aggressively, overshoots, becomes unstable
- Learning rate too low → model changes too slowly, takes forever, might get stuck
- Just right → model gradually improves

In full fine-tuning, ALL billions of weights are updated in step D. This is the defining characteristic.

E) Repeat
This cycle (forward → loss → backward → update) repeats for every batch of examples,
across multiple passes through the entire dataset (called epochs).

Fine Tuning

Batch Size and Gradient Accumulation:

In practice, you can rarely fit your ideal batch size into GPU memory all at once.
The solution is gradient accumulation: process several smaller micro-batches, accumulate the gradients from each, and only perform the weight update after N micro-batches.

For example, if your target batch size is 32 but you can only fit 4 examples in GPU memory at once, you set

gradient_accumulation_steps = 8 (4 × 8 = 32 effective batch size).

The model sees the same total gradient as if you processed all 32 at once — it just takes 8 micro-steps to get there. This is critical for full fine-tuning where memory is already tight.

The Learning Rate Dilemma:

This is one of the most important concepts in full fine-tuning. The learning rate for fine-tuning is typically much smaller (often 10x to 100x smaller) than what was used during pre-training.

Why? Because the pre-trained weights already encode valuable knowledge. You don't want to destroy that knowledge — you want to gently reshape it.

Think of it like this: the pre-trained model is a beautifully sculpted statue. Fine-tuning is like carefully chiseling new details onto it. If you chisel too aggressively (high learning rate), you destroy the original sculpture. If you're too gentle (low learning rate), you never finish the new details.

Typical learning rate ranges for full fine-tuning: 1e-6 to 5e-5 (compare this to LoRA which uses 5e-5 to 5e-4 — PEFT methods can afford higher learning rates because they're only updating a tiny fraction of parameters).

Catastrophic Forgetting :

This is the biggest danger of full fine-tuning. When you update all the weights for your specific task, the model can "forget" the general knowledge it learned during pre-training.

Imagine teaching your medical specialist so intensively that they forget how to speak grammatically, or forget basic common sense, or forget everything about every other topic.

This happens because the new task-specific gradients can overwrite the patterns stored in the weights from pre-training. The model becomes narrow — excellent at your specific task but degraded at everything else.

Mitigation strategies:

- Small learning rate: gentle updates preserve more original knowledge
- Early stopping: stop training before the model overfits and forgets
- Regularization: mathematically penalize weights that drift too far from their pre-trained values
- Data mixing: include some general-purpose data alongside your task-specific data
- Gradual unfreezing (a variant — see below)

Evaluation and Knowing When to Stop: Training without proper evaluation is flying blind. Here's what to monitor:

Fine Tuning

Validation Loss vs. Training Loss:
Split your dataset into training (~90%) and validation (~10%) sets.
Track both losses during training. The critical signal is divergence:

- Training loss keeps dropping, validation loss also drops → model is learning, keep going
- Training loss keeps dropping, validation loss plateaus → model is starting to memorize, be cautious
- Training loss keeps dropping, validation loss rises → model is overfitting, stop training

Early Stopping:
In practice, you don't train for a fixed number of epochs and hope for the best.
You monitor validation loss and stop when it hasn't improved for N evaluation steps (called "patience"). Save checkpoints at each evaluation step so you can roll back to the best one.

Task-Specific Metrics:
Loss alone doesn't tell the whole story. Track metrics relevant to your task:

- **Classification** **:** F1 score, precision, recall, accuracy
- **Generation/Summarization** **:** BLEU, ROUGE, BERTScore
- **Q&A** **:** Exact match, F1 on answer spans
- **Code generation** **:** pass@k (does the generated code actually run and pass tests?)

These metrics sometimes diverge from loss — a model might have slightly higher loss
but produce more practically useful outputs.

<u>Variants of Full Fine-Tuning:</u>

A) Standard Full Fine-Tuning
All layers, all weights, updated from the start. Maximum flexibility, maximum risk of forgetting, maximum compute cost.

B) Feature Extraction
You freeze the entire pre-trained model and only train a new classification head (a small layer added on top). The pre-trained model acts as a fixed feature extractor.

Think of it as: you don't retrain the generalist at all. You just put a specialist translator at the end who interprets what the generalist says and converts it into task-specific answers.

This is technically not "full" fine-tuning, but it's on the spectrum. It's the safest (no forgetting) but least flexible (the model can't adapt its internal representations).

C) Gradual Unfreezing
You start by freezing most layers and only training the top layers. Then, epoch by epoch, you unfreeze deeper layers, allowing more of the model to adapt.
Think of it as: first train the specialist translator (top layers), then gradually allow the generalist to adapt their thinking (deeper layers) once the translator is stable.This balances adaptation and preservation beautifully.

D) Layer-Selective Fine-Tuning
Rather than all-or-nothing, you choose specific layers to train while freezing the rest.
Common strategies include:

- Last N layers — most common, works well for classification tasks
- First + Last layers — captures both low-level and task-specific features
- Every Nth layer — distributes adaptation throughout the model

Fine Tuning

This sits between Feature Extraction and Full Fine-Tuning on the spectrum, and connects directly to PEFT selective methods like BitFit (which only trains bias terms — ~0.1% of parameters — and is surprisingly effective).

The full spectrum looks like:

Feature Extraction →	**Layer-Selective →**	**Gradual Unfreezing →**	**Full Fine-Tuning**
(least flexible, safest)	(moderate)	(balanced)	(most flexible, most risk)

The Cost Problem

Full fine-tuning is expensive. Here's why:
Memory: During training, you need to store in GPU memory:

- The model weights themselves (e.g., 7B parameters × 4 bytes = 28 GB for a 7B model in float32)
- The gradients for every weight (same size as the weights — another 28 GB)
- The optimizer states (Adam stores 2 extra values per weight — another 56 GB)
- The activations from the forward pass (needed for backpropagation — variable, can be huge)

Realistic Memory Math (Mixed Precision — How It's Actually Done):
In practice, almost nobody fine-tunes in full float32 anymore.
Mixed precision training (BF16/FP16) is standard and roughly halves the memory for weights and gradients:

For a 7B parameter model with BF16 mixed precision + Adam optimizer:

Model weights (BF16)	: 7B × 2 bytes = ~14 GB	
Gradients (BF16)	: 7B × 2 bytes = ~14 GB	
Optimizer states (FP32)	: 7B × 8 bytes = ~56 GB	← Adam keeps momentum + variance in FP32 for stability
Activations (variable):	= ~10-30 GB	(depends on batch size and sequence length)
Total:	~94-114 GB	

Note: Even with mixed precision, the optimizer states dominate memory.
This is why the Adam optimizer's FP32 states are often the bottleneck,
and why techniques like 8-bit Adam (from bitsandbytes) exist to compress them.

For a 70B parameter model: scale everything by 10× → ~1 TB+, requiring multiple GPUs.

So for a 7B parameter model, you might need 120+ GB of GPU memory just for training.
For a 70B model, you're looking at over 1 TB — requiring multiple expensive GPUs.

Fine Tuning

Gradient Checkpointing (Trading Compute for Memory):
The activations from the forward pass can consume enormous memory (they're needed for backpropagation). Gradient checkpointing is a technique that discards most intermediate activations during the forward pass and recomputes them on-the-fly during the backward pass.

- Without checkpointing: store all activations → high memory, normal speed
- With checkpointing: store only some activations, recompute the rest → ~30-40% less memory, ~20-30% slower

This is almost always enabled for full fine-tuning of large models.
Think of it as choosing to re-derive a formula during the exam rather than memorizing it saves brain space (memory) but costs time (compute).

Compute: Every training step requires a forward pass AND a backward pass through the ENTIRE model, updating ALL parameters. This is slow and costly.

Storage: You save a complete copy of all parameters. Every fine-tuned version is as large as the original model. If you fine-tune for 10 different tasks, you store 10 full copies. This cost problem is exactly why PEFT methods like LoRA were invented — but that's for a later step.

Distributed Training — How You Actually Scale
When a model doesn't fit on a single GPU (which is most full fine-tuning scenarios), you distribute the work across multiple GPUs.

<u>Two dominant approaches:</u>

DeepSpeed ZeRO (Zero Redundancy Optimizer):
By default, each GPU holds a full copy of weights, gradients, and optimizer states — massively redundant. ZeRO eliminates this redundancy in stages:
- ZeRO Stage 1: Shard optimizer states across GPUs (biggest memory win, ~4× reduction)
- ZeRO Stage 2: Also shard gradients across GPUs
- ZeRO Stage 3: Also shard model weights across GPUs (most memory efficient, enables training models that don't fit on any single GPU)

PyTorch FSDP (Fully Sharded Data Parallel):
PyTorch's native equivalent of ZeRO Stage 3. Shards weights, gradients, and optimizer states across GPUs and only gathers the full parameters when needed for computation.

- Practical example:
A 70B model in BF16 needs ~140 GB just for weights. With 8× A100 80GB GPUs and DeepSpeed ZeRO-3 or FSDP, each GPU only holds ~17.5 GB of sharded weights, leaving room for gradients, optimizer states, and activations.

These tools are what make full fine-tuning of 70B+ models physically possible. Without them, you'd need GPUs with terabytes of memory that don't exist.

This Cost Problem is Exactly Why PEFT Methods Like LoRA Were Invented

Every pain point listed above has a corresponding PEFT solution:

Full Fine-Tuning Problem	→	PEFT Solution
120+ GB memory for 7B model	→	LoRA: ~16 GB (trains only ~0.3% of params)
1 TB+ for 70B model	→	QLoRA: ~36-48 GB (4-bit base + LoRA adapters)
Full copy per task (14-140 GB each)	→	LoRA adapters: ~20-200 MB per task, swappable
Catastrophic forgetting	→	Base model frozen, only adapters updated
Hours/days of training	→	Much faster convergence with fewer parameters
Multi-GPU clusters required	→	Single GPU feasible for most model sizes

---

When Full Fine-Tuning Makes Sense

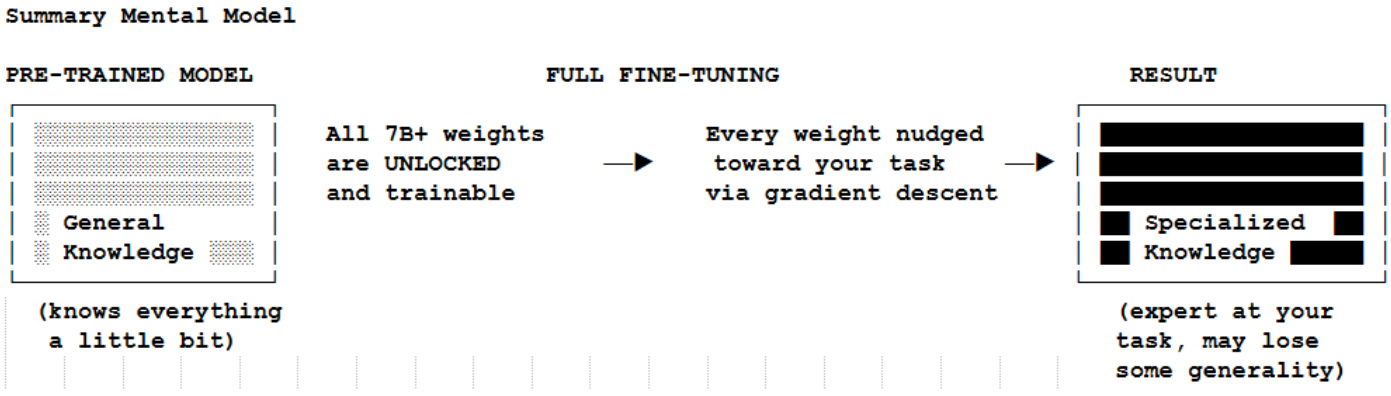
Despite its costs, full fine-tuning is the right choice when:

- You have enough data (tens of thousands to millions of examples)
- You have enough compute (multiple high-end GPUs)
- You need maximum performance on your specific task
- Your task is significantly different from what the model learned in pre-training
- You're building a production system where even small accuracy gains matter
- You're a large organization with the infrastructure (Google, Meta, OpenAI, etc.)

When It Doesn't Make Sense

- Limited compute budget → use PEFT
- Small dataset (hundreds of examples) → high risk of overfitting with full fine-tuning
- Need to serve multiple tasks from one model → PEFT adapters are modular
- Rapid experimentation → full fine-tuning is too slow for quick iterations

---



Fine Tuning

Solution to the cost Problem

PEFT and LoRA, which were invented specifically to solve the cost and forgetting problems described above.

How is Full Fine Tuning Carried Out - Under the hood

Let me walk through the full data pipeline from raw data to what actually hits the model.

Input Data:

The input data is NOT pre-computed embedding vectors. This is a common misconception. You feed the model raw text, and the model's own embedding layer converts it to vectors on the fly during training. The embedding layer itself is part of the model and gets updated during full fine-tuning.

What the Raw Data Looks Like:

Your fine-tuning dataset is just text — stored in simple, standard file formats:
JSONL (JSON Lines) — the most common format:

JSON:

```
{"instruction": "Classify the sentiment", "input": "This movie was breathtaking", "output": "positive"}
{"instruction": "Classify the sentiment", "input": "Terrible waste of time", "output": "negative"}
{"instruction": "Summarize this article", "input": "The Federal Reserve announced...", "output": "The Fed raised rates by..."}
```

Each line is one training example. A fine-tuning dataset might be a single .jsonl file that's a few hundred MB to a few GB.

Other common formats:

- CSV/TSV — simple tabular format
- Parquet — columnar format, compressed, used by Hugging Face datasets
- JSON — nested structures for multi-turn conversations
- Plain text — for continued pre-training (just raw text, no instruction structure)

For chat/instruction models, the data often looks like:

```
{
  "conversations":
  [
    {"role": "system", "content": "You are a medical assistant."},
    {"role": "user", "content": "What causes migraines?"},
    {"role": "assistant", "content": "Migraines are caused by..."}
  ]
}
```

That's it. No vectors. No embeddings. Just text in files.

Fine Tuning

How Text Becomes Numbers (Tokenization → Embedding)
Before the model can process text, two things happen — and this is where the vectors come in:

Step 1: Tokenization (text → integer IDs)
A tokenizer breaks text into subword tokens and maps each to an integer ID. This happens *before* the model, as a preprocessing step.

"This movie was breathtaking"
↓ tokenizer
[1552, 5765, 471, 4800, 28107]

These are just integer IDs — a lookup index. The tokenizer is fixed and never changes during fine-tuning.
It's stored as a separate vocabulary file (usually `tokenizer.json` or `tokenizer.model`, a few MB).

Step 2: Embedding Lookup (integer IDs → vectors)
The model's first layer is an embedding table — essentially a giant matrix where each row corresponds to one token ID. When token ID 1552 enters the model, it looks up row 1552 and pulls out a dense vector (e.g., 4096 dimensions for a 7B model).

Token ID 1552 → [0.023, -0.891, 0.445, ..., 0.112] (4096 floats)
Token ID 5765 → [0.771, 0.034, -0.562, ..., -0.338] (4096 floats)

This happens *inside the model* during the forward pass, on the GPU, in real time.
You never pre-compute or store these embedding vectors. The embedding table is part of the model's weights and gets updated during full fine-tuning like everything else.

How the Data Is Actually Stored and Loaded

Storage — it's simple files on disk:

```
my_dataset/
├── train.jsonl          # 500K examples, ~2 GB
├── validation.jsonl     # 50K examples, ~200 MB
└── metadata.json        # dataset info
```

Or if using Hugging Face datasets (the most common approach):

```
dataset/
├── data/
│   ├── train-00000-of-00004.parquet
│   ├── train-00001-of-00004.parquet
│   ├── train-00002-of-00004.parquet
│   └── train-00003-of-00004.parquet
└── dataset_info.json
```

Parquet files are split into shards for parallel loading.
Hugging Face datasets use Apache Arrow format under the hood — memory-mapped files that let you work with datasets larger than RAM without loading everything at once.

****No database involved**.** You're not querying a database during training.

The data sits on disk (local SSD or cloud storage like S3) and gets streamed into memory in batches.
Training data pipelines are optimized for sequential throughput, not random access — the opposite of what databases are designed for.

How Data Flows During Training — Not One Line, Not All at Once
This is the key part. It's neither one example at a time nor everything at once. It's **batches**

The DataLoader pipeline:

Disk (JSONL/Parquet files)
↓
DataLoader reads a batch of raw examples (e.g., 8 examples)
↓
Tokenizer converts text → token IDs for each example
↓
Padding/Truncation — all sequences in the batch must be the same length
(shorter ones get padded with a special [PAD] token, longer ones get truncated to max_seq_length, e.g., 2048 or 4096 tokens)
↓
Collation — stack into a single tensor of shape [batch_size, seq_length]
e.g., [8, 2048] — 8 examples, each 2048 tokens
↓
Move to GPU memory
↓
Model's embedding layer converts token IDs → vectors
Now shape is [8, 2048, 4096] — 8 examples, 2048 positions, 4096-dim vectors
↓
Forward pass through all transformer layers
↓
Loss, backprop, weight update
↓
Next batch

Crucially, the DataLoader:

- Shuffles the dataset at the start of each epoch (so the model sees examples in different order)
- Loads batches in parallel using multiple CPU workers while the GPU is processing the current batch
- Pre-fetches the next batch so the GPU is never waiting on data

---

What "**batch**" means in practice with gradient accumulation:
Say you want an effective batch size of 32 but can only fit 4 examples on the GPU:

```
                    ------------------------------------------------------------
Micro-batch 1:  4 examples → forward → loss → backward → accumulate gradients
Micro-batch 2:  4 examples → forward → loss → backward → accumulate gradients
Micro-batch 3:  4 examples → forward → loss → backward → accumulate gradients
…
Micro-batch 8:  4 examples → forward → loss → backward → accumulate gradients
                                          ↓
                                  Weight update (using accumulated gradients from all 32 examples)
                    ------------------------------------------------------------
```

The DataLoader pipeline Breakdown :

What's sitting in your JSONL file:

{"instruction": "Classify the sentiment", "input": "This movie was breathtaking", "output": "positive"}
{"instruction": "Classify the sentiment", "input": "Terrible waste of time", "output": "negative"}
{"instruction": "Translate to French", "input": "The cat sat on the mat", "output": "Le chat était assis sur le tapis"}

Just text. Nothing has happened yet.

Step 1: Template Formatting (Text → Structured Text)

The DataLoader first applies a **chat template** to combine the fields into a single string.
This is model-specific — different models expect different formats.
For a LLaMA-style model:

Example 1: "[INST] Classify the sentiment: This movie was breathtaking [/INST] positive"
Example 2: "[INST] Classify the sentiment: Terrible waste of time [/INST] negative"
Example 3: "[INST] Translate to French: The cat sat on the mat [/INST] Le chat était assis sur le tapis"

For a ChatML-style model (like Mistral/OpenChat):

Example 1 : "<|im_start|>user\nClassify the sentiment: This movie was breathtaking<|im_end|>\n<|im_start|>assistant\npositive<|im_end|>"

This is where **special tokens** enter the picture:

Special Token	Purpose	When Added
<s> / BOS	Beginning of sequence	Start of the whole sequence
</s> / EOS	End of sequence	End of the whole sequence
[INST] [/INST]	Instruction boundaries	Around the instruction/input
[CLS]	Classification token (BERT-era)	NOT used in modern LLM fine-tuning
[SEP]	Separator (BERT-era)	NOT used in modern LLM fine-tuning
[PAD]	Padding (comes later)	During batching — Step 4

Important distinction:
`[CLS]` and `[SEP]` are BERT-family tokens. Modern decoder-only LLMs (LLaMA, Mistral, GPT) don't use them. They use BOS, EOS, and model-specific instruction markers instead.
If you see `[CLS]` and `[SEP]`, you're looking at older encoder-model fine-tuning (BERT, RoBERTa). Still just text at this point.

No numbers yet.

Step 2: Tokenization (Text → Integer IDs)

The tokenizer breaks each formatted string into subword tokens and maps each to an integer ID from the vocabulary.

Example 1 (short):
"<s>[INST] Classify the sentiment: This movie was breathtaking [/INST] positive</s>"

↓ tokenizer

[1, 518, 25580, 29962, 4134, 1598, 278, 19688, 29901, 910, 14064, 471, 4800, 28107, 518, 29914, 25580, 29962, 6374, 2]

That's 20 tokens.

Example 3 (longer):

"<s>[INST] Translate to French: The cat sat on the mat [/INST] Le chat était assis sur le tapis</s>"

↓ tokenizer

[1, 518, 25580, 29962, 4103, 9632, 304, 5765, 29901, 450, 6635, 3290, 373, 278, 1775, 518, 29914, 25580, 29962, 997, 13563, 4496, 465, 275, 1190, 454, 260, 11690, 2]

That's 29 tokens.

Notice the problem: **different examples have different lengths** (20 vs 29 tokens).
GPUs need rectangular tensors — every row must be the same length. This is where padding comes in.

What we have now: a list of integer arrays of varying lengths. No vectors yet — just integer IDs.

Step 3: Create the Label/Target Array and Loss Mask

Before batching, the DataLoader creates the **labels** — what the model should predict — and the **loss mask** that tells training which tokens to grade.

Example 1:
Input IDs: **[1, 518, 25580, 29962, 4134, 1598, 278, 19688, 29901, 910, 14064, 471, 4800, 28107, 518, 29914, 25580, 29962, 6374, 2]**

Labels: [-100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, 6374, 2]
^^
instruction + input tokens: IGNORED by loss **output tokens: GRADED**

`-100` is a magic number in PyTorch that means "ignore this token when computing loss." The loss function skips any position with -100.

This is the **loss masking** we discussed earlier. The model sees the full sequence during the forward pass, but only gets graded on the output portion.

Step 4: Padding and Attention Masking (This Is Where Padding Happens)

Now the DataLoader collects a batch of examples (say batch_size=4) and must make them all the same length.

Before padding (variable lengths):

Example 1: [1, 518, 25580, ..., 6374, 2] → 20 tokens
Example 2: [1, 518, 25580, ..., 8178, 2] → 18 tokens
Example 3: [1, 518, 25580, ..., 11690, 2] → 29 tokens ← longest
Example 4: [1, 518, 25580, ..., 1781, 2] → 24 tokens

After RIGHT-PADDING to length 29 (length of longest example):

Example 1: [1, 518, 25580, ..., 6374, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0] → 29 tokens
Example 2: [1, 518, 25580, ..., 8178, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] → 29 tokens
Example 3: [1, 518, 25580, ..., 11690, 2] → 29 tokens (no padding needed)
Example 4: [1, 518, 25580, ..., 1781, 2, 0, 0, 0, 0, 0] → 29 tokens

The `0` here is the PAD token ID. **This is what padding is** — adding filler tokens so all sequences are the same length. It has nothing to do with CLS or SEP or EOS — those are content tokens with meaning. Padding is meaningless filler.

But there's a problem: you don't want the model to pay attention to padding tokens. So the DataLoader also creates an **attention mask**:

Attention Mask (1 = real token, 0 = padding, ignore me):

Example 1: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Example 2: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Example 3: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
Example 4: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0]

Labels are also padded with -100:

Example 1: [-100, ..., -100, 6374, 2, -100, -100, -100, -100, -100, -100, -100, -100, -100]

Padding tokens get -100 in labels (ignored by loss) AND 0 in attention mask (ignored by attention).
They're invisible to the model in every way.

Step 5: Truncation (If Needed)

If any example exceeds `max_seq_length` (e.g., 2048 or 4096), it gets chopped:

Example with 5000 tokens, max_seq_length=2048:

```
[1, 518, 25580, ..., token_2046, token_2047, 2]
 ↑                                           ↑
 kept: first 2048 tokens                     EOS forced at the end

 Tokens 2049-5000: DISCARDED
```

This is a hard cutoff.

Information beyond the max length is simply lost.
This is why choosing the right max_seq_length matters — too short and you lose data, too long and you waste GPU memory (memory scales quadratically with sequence length in standard attention).

Step 6: Collation into Tensors (What Actually Goes to GPU)

The DataLoader's collator stacks everything into rectangular tensors:

```
batch =
{
    "input_ids"      :   tensor of shape [4, 29],   # 4 examples × 29 tokens each
    "attention_mask":   tensor of shape [4, 29],   # which tokens are real
    "labels"         :   tensor of shape [4, 29],   # what model should predict (-100 = ignore)
}
```

These three tensors get moved from CPU to GPU. That's everything the model needs.

Visually as matrices:

input_ids [4 × 29]:

```
1  518  25580  29962  4134  ...  6374  2   0   0   0      Example 1
1  518  25580  29962  7301  ...  8178  2   0   0   0      Example 2
1  518  25580  29962  4103  ... 11690  2                  Example 3
1  518  25580  29962  2431  ...  1781  2   0   0   0      Example 4
```

attention_mask [4 × 29]:

```
1  1  1  1  1  ...  1  1  0  0  0
1  1  1  1  1  ...  1  1  0  0  0
1  1  1  1  1  ...  1  1
1  1  1  1  1  ...  1  1  0  0  0
```

labels [4 × 29]:

```
-100 -100 -100 -100 -100 ... 6374   2  -100  -100  -100
-100 -100 -100 -100 -100 ... 8178   2  -100  -100  -100
-100 -100 -100 -100 -100 ... 11690  2
-100 -100 -100 -100 -100 ... 1781   2  -100  -100  -100
```

Step 7: Into the Model (Token IDs → Embeddings → Forward Pass)

NOW the model's embedding layer converts integer IDs to vectors:

input_ids [4, 29]	→	**Embedding Layer**	→	**hidden_states [4, 29, 4096]**
(integers)		**(lookup table)**		(dense vectors)

Each integer ID gets replaced by a 4096-dimensional vector:
Token ID 518 → [0.023, -0.891, 0.445, ..., 0.112] (4096 floats)
This 3D tensor `[batch_size, seq_length, hidden_dim]` is what flows through all 32 transformer layers, getting transformed at each step.

Fine Tuning

The Full Picture:

DISK	STEP 1	STEP 2	STEP 3	STEP 4	STEP 5	STEP 6	STEP 7
Raw JSONL →	Template Formatting →	Tokenize to IDs →	Create Labels + Loss Mask →	Pad + Attention Mask →	Truncate (if needed) →	Stack into Tensors →	Embed + Forward Pass
"Classify sentiment: This movie was great" → positive	"<s>[INST] Classify..."	[1, 518, 25580, ...]	labels: [-100,..., 6374, 2]	input_ids padded to same length + attn_mask	(chop to max_seq_len)	[4, 29] tensor moved to GPU	[4, 29, 4096] 3D tensor flows through 32 layers

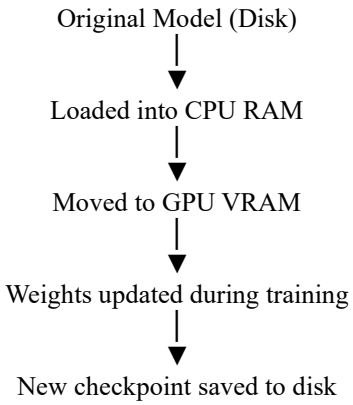
Note: padding (Step 4) is adding meaningless filler zeros so all examples in a batch have equal length.

Special tokens like BOS/EOS (Step 1) are meaningful markers that tell the model where sequences begin and end.

What Lives on GPU vs. CPU vs. Disk:

STORAGE LAYER	DISK (SSD / Cloud Storage)	CPU RAM	GPU VRAM
Dataset Storage	Full dataset files (JSONL, Parquet)	Loaded batches being tokenized / preprocessed	Current micro-batch (as tensors)
Model / Training State	Model checkpoints (saved periodically)	DataLoader workers prefetching data Optimizer states	Model weights Gradients
Dataset Index / Runtime	—	Memory-mapped dataset index (Arrow / Parquet)	Activations (for backpropagation)

Visual Flow



Fine Tuning

PEFT - Parameter-Efficient Fine-Tuning:

- Additive Methods
  - Bottleneck Adapters
  - Soft Prompts
  - IA^3
- Reparameterization
  - LORA
  - QLORA
  - DORA
  - LORA +
- Selective Methods
  - rsLORA
  - BitFit - (bias only)
  - Fish Mask
- Hybrid Methods
  - QLORA
  - LongLORA
  - LORA-FA
- Prompt Methods
  - Prefix Tuning
  - P-Tuning V2
  - Prompt Tuning

Additive : Build new small modules and insert them into the model. The original architecture gets new components bolted on. Bottleneck Adapters are the classic example.
Downside: those new modules stay in the model forever, adding inference latency.

Reparameterization: don't change the architecture at all. Instead, decompose the weight updates into smaller matrices (LoRA's A × B). Trains alongside the frozen weights as a parallel bypass, then merges back in and vanishes. Zero inference overhead. This is why LoRA dominates.

Selective: add nothing new, change nothing structurally. Just pick a tiny subset of the model's existing parameters (like bias terms in BitFit, or Fisher-information-selected weights in Fish Mask) and unfreeze only those. Everything else stays frozen. Extremely lightweight but limited expressiveness.

Hybrid: combine strategies from the categories above.

QLoRA is the poster child: it takes reparameterization (LoRA adapters in 16-bit) and combines it with quantization (base model compressed to 4-bit).
Each technique solves a different bottleneck **LoRA reduces trainable parameters, quantization reduces the memory footprint of the frozen base.**

Prompt-based: the most radical approach. Don't touch any weights at all. Instead, learn continuous "soft prompt" vectors that get prepended to the input and steer the model's behavior from the outside. The model itself is completely untouched — you're just learning a better way to talk to it. Fewest parameters (~0.001%), but also the least expressive for smaller models.

Fine Tuning

The Big Picture — Why PEFT Exists

Full fine-tuning updates ALL parameters. For a 7B model that means:

- ~28 GB just for weights (FP32)
- ~28 GB for gradients
- ~56 GB for optimizer states (Adam)
- ~10-30 GB for activations

Total: ~120+ GB of GPU VRAM

And every fine-tuned variant is a full copy of the model. Ten tasks = ten 14-28 GB checkpoints.

PEFT's core insight: you don't need to update all the parameters.

Research showed that the "intrinsic dimensionality" of fine-tuning is low —meaning the weight changes needed to adapt a model to a new task live in a much smaller subspace than the full parameter space. You can capture most of the adaptation with a tiny fraction of trainable parameters.

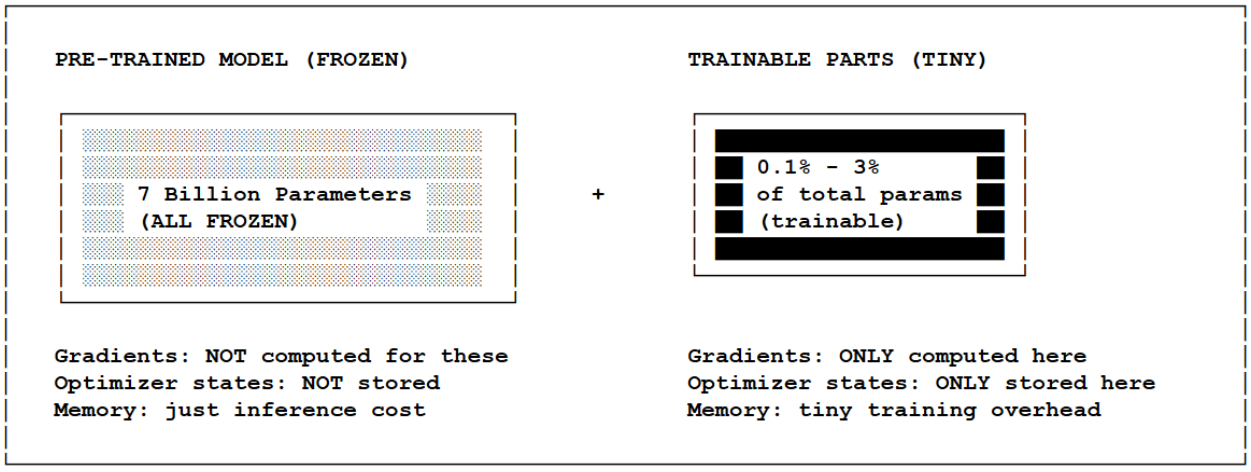
Think of it this way: a pre-trained model is a massive building. Full fine-tuning demolishes and rebuilds every room. PEFT just redecorates specific rooms — same structural integrity, fraction of the cost.

The Fundamental PEFT Principle — Freeze, Then Add or Select

Every PEFT method follows the same two-step pattern:

Step 1: FREEZE the pre-trained model weights (they become read-only)

Step 2: Either ADD new small trainable components, or SELECT a tiny subset of existing params to train



Because the frozen base model only does forward passes (no gradients, no optimizer states), the memory footprint drops dramatically:

Full Fine-Tuning (7B model, BF16 + Adam):

Weights	:	14 GB
Gradients	:	14 GB ← eliminated in PEFT
Optimizer states	:	56 GB ← eliminated in PEFT
Activations	:	10-30 GB ← reduced (fewer backward-pass paths)
Total:		~94-114 GB

PEFT / LoRA (7B model, BF16 base + LoRA adapters):

Frozen weights	:	14 GB	(forward pass only, no gradients/optimizer)
LoRA adapter weights	:	~20 MB	(trainable)
LoRA gradients	:	~20 MB	
LoRA optimizer states	:	~80 MB	
Activations	:	4-10 GB	(reduced)
Total:		~16-24 GB	

ADDITIVE – PEFT - Concept

Build new small modules and insert them into the model. The original architecture gets new components bolted on.

Bottleneck Adapters are the classic example.
Downside: those new modules stay in the model forever, adding inference latency.

You have a pre-trained transformer. Every layer in it follows this flow:

Input → Self-Attention → Add & Norm → Feed-Forward (MLP) → Add & Norm → Output

Additive PEFT physically inserts new small modules into this pipeline that didn't exist before.
The original layers are all frozen — you're literally adding new trainable components into the architecture.
The Main Additive Method: Bottleneck Adapters - step-by-step breakdown of what happens.

Step 1: Freeze the entire pre-trained model

Every single parameter in the original model gets requires_grad = False.
No gradients will be computed for them, no optimizer states stored. They become read-only.

```
for param in model.parameters():
    param.requires_grad = False    # 7 billion parameters → all frozen
```

Step 2: Insert adapter modules into every transformer layer

Two small adapter modules are inserted into each layer — one after self-attention, one after the feed-forward network:

BEFORE (standard transformer layer):

Input
↓
Self-Attention
↓
Add & LayerNorm
↓
Feed-Forward (MLP)
↓
Add & LayerNorm
↓
Output

AFTER (with adapters inserted):

Input
↓
Self-Attention ← FROZEN
↓
[ADAPTER MODULE 1] ← NEW, trainable
↓
Add & LayerNorm
↓
Feed-Forward (MLP) ← FROZEN
↓
[ADAPTER MODULE 2] ← NEW, trainable
↓
Add & LayerNorm
↓
Output

The adapter is now literally in the data path. Every token's representation must flow through it.

Step 3: Understand what's inside each adapter module

Each adapter is a tiny feed-forward network with a bottleneck — it squeezes the data down to a small dimension and expands it back.

Here's exactly what happens to a single token's hidden state vector as it passes through:

Input: h (shape: [4096]) ← the token's hidden state coming from self-attention

▼

Down-projection: W_down × h (W_down shape: [64 × 4096])

h is now compressed: [4096] → [64]
This forces the adapter to learn a COMPRESSED representation of whatever adjustment is needed.
It can't just memorize — it must generalize.

▼

Non-linearity: ReLU(compressed_h)

The non-linearity is important — without it, down-project then up-project is just a single linear transformation (matrix multiplication collapses). ReLU gives the adapter the ability to learn non-linear transformations.

▼

Up-projection: W_up × activated_h (W_up shape: [4096 × 64])

Expanded back: [64] → [4096]
Same dimensionality as the original hidden state.

▼

Residual add: output = adapter_output + h (original input added back)

THIS IS CRITICAL. The residual connection means:
- If the adapter learns nothing useful → output ≈ h
  (model behaves like the pre-trained version)
- If the adapter learns something → output = h + adjustment
  (model gets a task-specific nudge)

▼

Output: h_adapted (shape: [4096]) ← continues to the next part of the layer

---

Step 4: Count the trainable parameters: For a single adapter module with bottleneck dimension 64 and hidden dimension 4096:

W_down	: [64 × 4096]	=	262,144 parameters
b_down	: [64]	=	64 parameters
W_up	: [4096 × 64]	=	262,144 parameters
b_up	: [4096]	=	4,096 parameters
Total per adapter:			~524,448 parameters

Per transformer layer : 2 adapters × 524,448 = ~1,048,896

For 32 layers : 32 × 1,048,896 = ~33.6 million

vs. 7 billion total model parameters → ~0.48% trainable

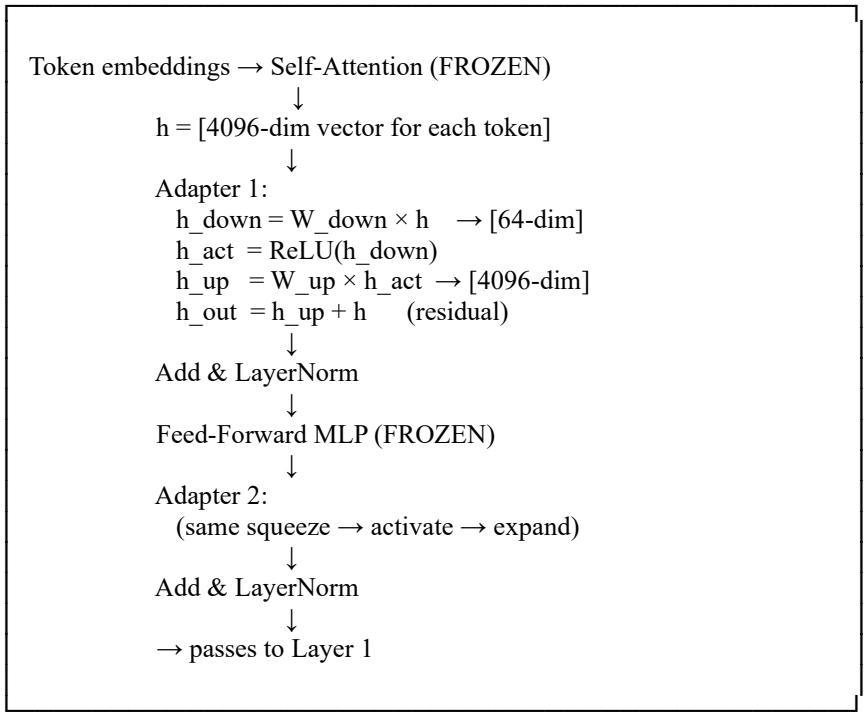
The bottleneck dimension (64 in this example) is the key hyperparameter.
It's analogous to LoRA's rank — smaller means fewer parameters but less capacity.

Step 5: Training — what actually happens during a forward pass

Let's trace a single training example through a model with adapters:

"Classify sentiment : This movie was great" → "positive"
Tokenize : [1, 518, 25580, ..., 6374, 2]

Forward pass through Layer 0:



... repeat through all 32 layers ...

Final output → compute loss on "positive" tokens

Step 6: Backward pass — where it gets efficient:

Loss computed on output
↓
Backpropagation begins (working backward through the network)
↓
At each frozen layer:
- Gradients flow THROUGH the frozen weights (needed for chain rule)
- But NO gradient is STORED for frozen weights
- NO optimizer state maintained for frozen weights
↓
At each adapter:
- Gradients ARE computed and STORED for W_down, W_up, b_down, b_up
- Optimizer (Adam) maintains momentum + variance for these params only
↓
Weight update:
- ONLY adapter parameters get updated
- Frozen model: untouched

The memory savings come from the optimizer states.
Adam stores 2 extra values per trainable parameter (momentum and variance).
For 7B frozen parameters, that's 56 GB you never allocate. For 33M adapter parameters, it's ~260 MB.

Step 7: Saving the result
After training, you save ONLY the adapter weights:

Full fine-tuning checkpoint:	Adapter checkpoint:
model.safetensors 14 GB	adapter_weights.pt ~130 MB adapter_config.json (references base model by name)

COMPONENT	FULL FINE-TUNING CHECKPOINT	ADAPTER CHECKPOINT (LoRA / PEFT)
Main Weights File	model.safetensors	adapter_weights.pt
File Size	~14 GB	~130 MB
Configuration File	(embedded in full model)	adapter_config.json
Base Model Reference	Contains full base model weights	References base model by name
What It Stores	Entire model (all parameters)	Only adapter (delta) weights
Dependency on Base Model	Independent	Requires base model to load

Why the Bottleneck Shape Matters

The squeeze-and-expand isn't arbitrary. It enforces an information bottleneck:

4096 dimensions of information
↓
FORCED through 64 dimensions ← **can't pass everything through**
↓
Back to 4096 dimensions

The adapter MUST learn which 64 dimensions of variation are most important for your task. It's forced to prioritize.

Bottleneck too small (e.g., 8)	:	Not enough capacity. Adapter can't capture the task's complexity.
Bottleneck too large (e.g., 2048)	:	Too much capacity. Approaches full fine-tuning cost. Defeats the purpose.
Sweet spot (32-128)	:	Enough to capture task-specific adjustments without excess.

The Three Additive Sub-Methods

Bottleneck Adapters are the most important, but the additive category also includes:

$(IA)^3$ — the most extreme version. Instead of inserting full modules, it just learns three scaling vectors (one each for keys, values, and feed-forward outputs). Each vector element-wise multiplies the activations — amplifying some dimensions and suppressing others. Only ~0.01% of parameters. Extremely lightweight, less expressive.

Soft Prompts — sometimes classified as additive because you're adding new trainable embedding vectors to the input sequence. These are covered more thoroughly under the "Prompt-Based" category, but conceptually they're additive — new parameters that didn't exist before.

Why Adapters Lost to LoRA:

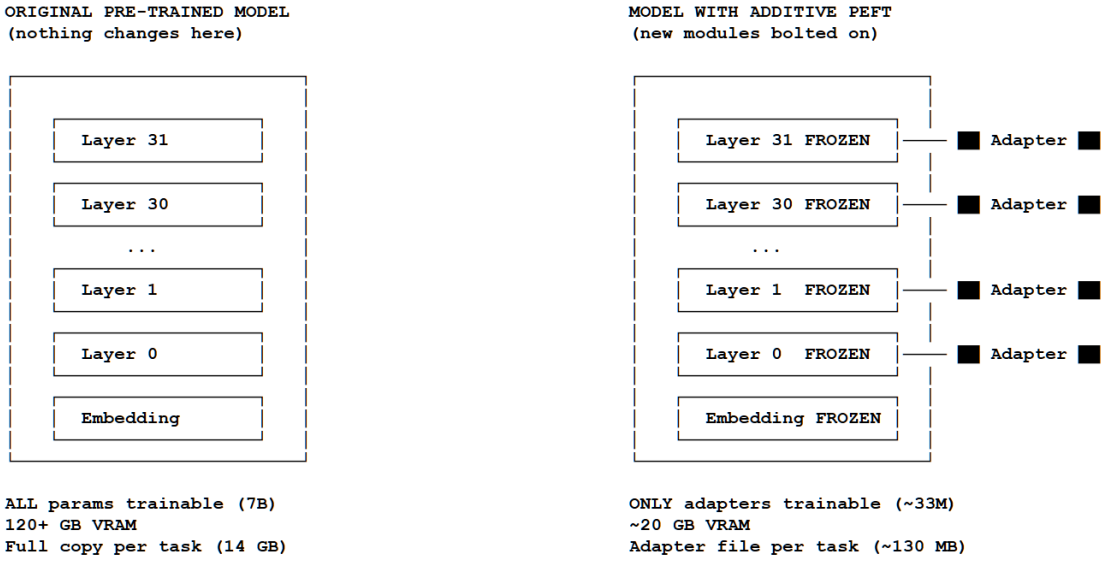
The fundamental problem: adapters can't be removed after training. They sit in the forward pass permanently, adding latency to every inference call. LoRA's parallel bypass **($h = W_0x + BAx$)** merges back into the weight matrix after training **($W\_merged = W_0 + BA$),** leaving zero trace. That single property mergeability is why LoRA became the dominant PEFT method and adapters faded into historical importance.

Additive PEFT — Complete Visual Diagram Breakdown:

Textual diagrams covering every aspect of Additive PEFT: architecture, data flow, bottleneck mechanics, training loop, memory layout, and comparison with other PEFT approaches.

THE CORE IDEA — WHAT "ADDITIVE" MEANS VISUALLY

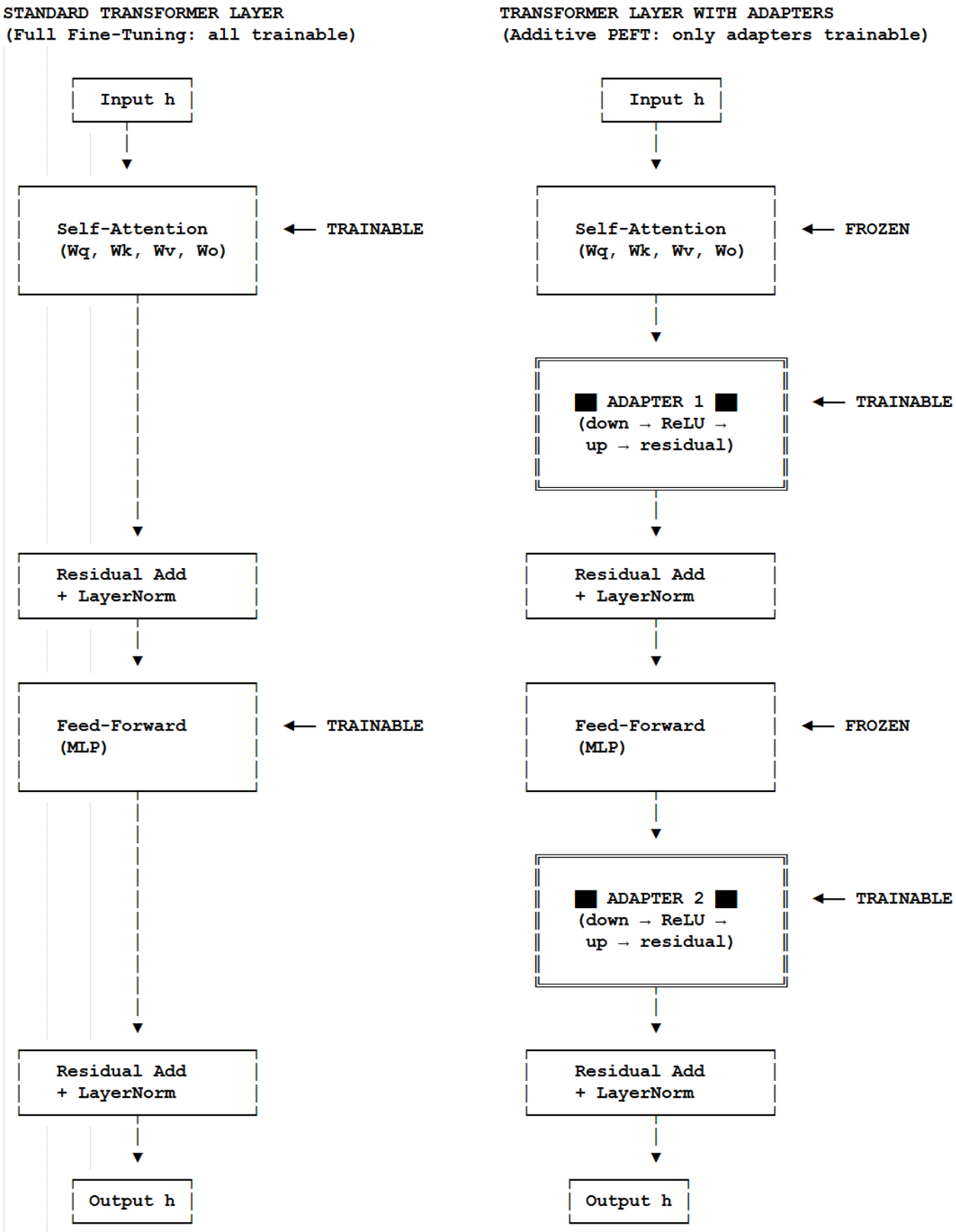
The core idea: frozen model + new modules bolted on



ALL params trainable (7B)
120+ GB VRAM
Full copy per task (14 GB)

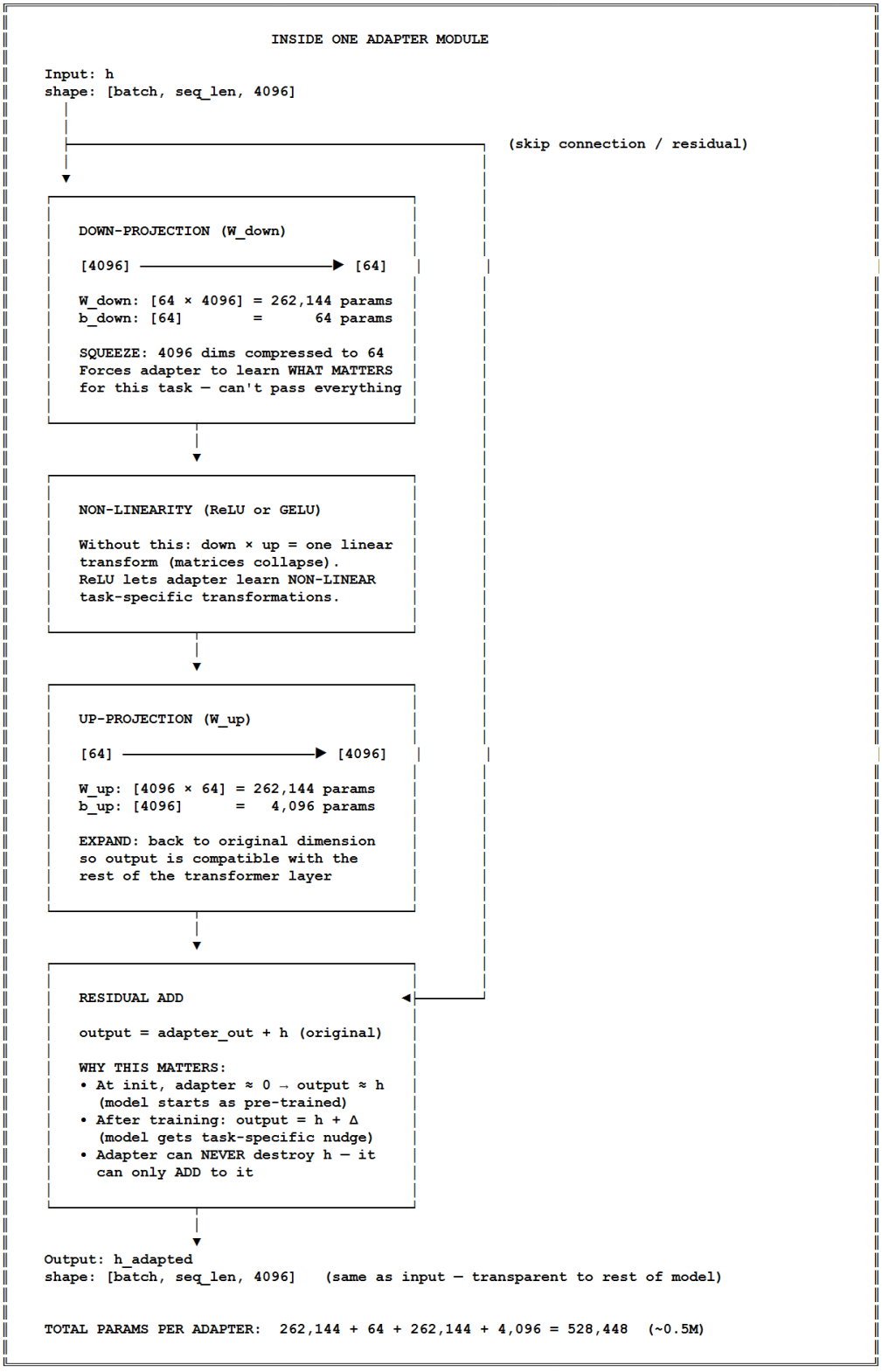
ONLY adapters trainable (~33M)
~20 GB VRAM
Adapter file per task (~130 MB)

STANDARD TRANSFORMER LAYER vs. LAYER WITH ADAPTERS



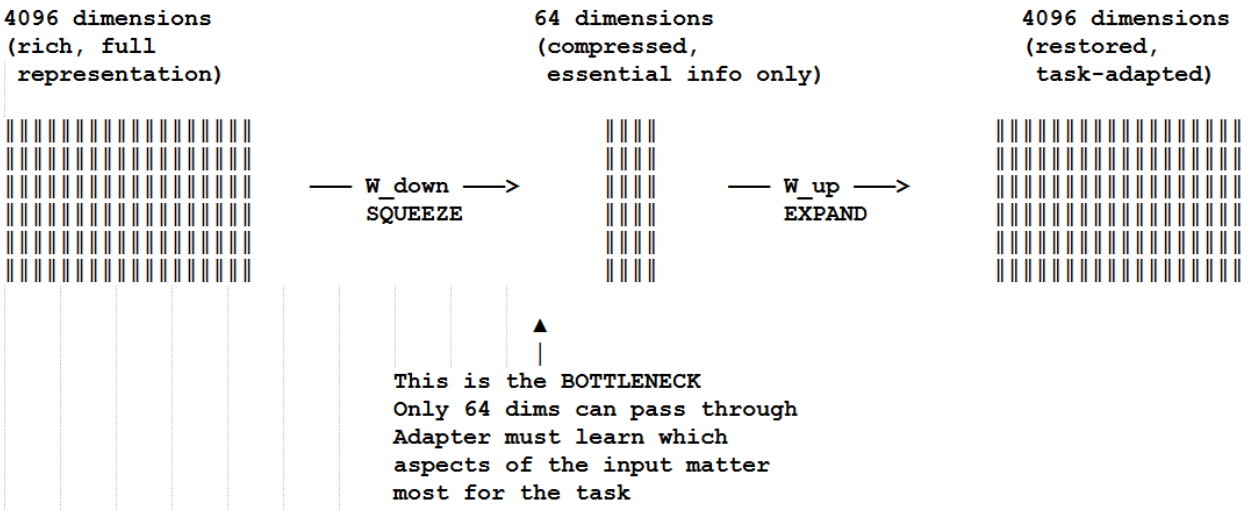
INSIDE A SINGLE ADAPTER MODULE — THE BOTTLENECK

Inside a single adapter module: the full bottleneck breakdown **(down-project → ReLU → up-project → residual),** with parameter counts and annotations on why each component exists



THE BOTTLENECK SHAPE — WHY IT WORKS

INFORMATION FLOW THROUGH THE BOTTLENECK
The bottleneck shape visualized as an information funnel, plus the size trade-off table



BOTTLENECK SIZE TRADE-OFF

Bottleneck Dim	Params/Adapter	Total (32 layers) (2 adapters/layer)	Capacity	Risk
8	~65K	~4.2M	Very low	Underfitting
32	~262K	~16.8M	Low	Good balance
64	~525K	~33.6M	Medium	Sweet spot ★
128	~1.05M	~67.1M	High	Good balance
256	~2.1M	~134M	Very high	Diminishing
2048	~16.8M	~1.07B	Excessive	Defeats purpose

★ = common default

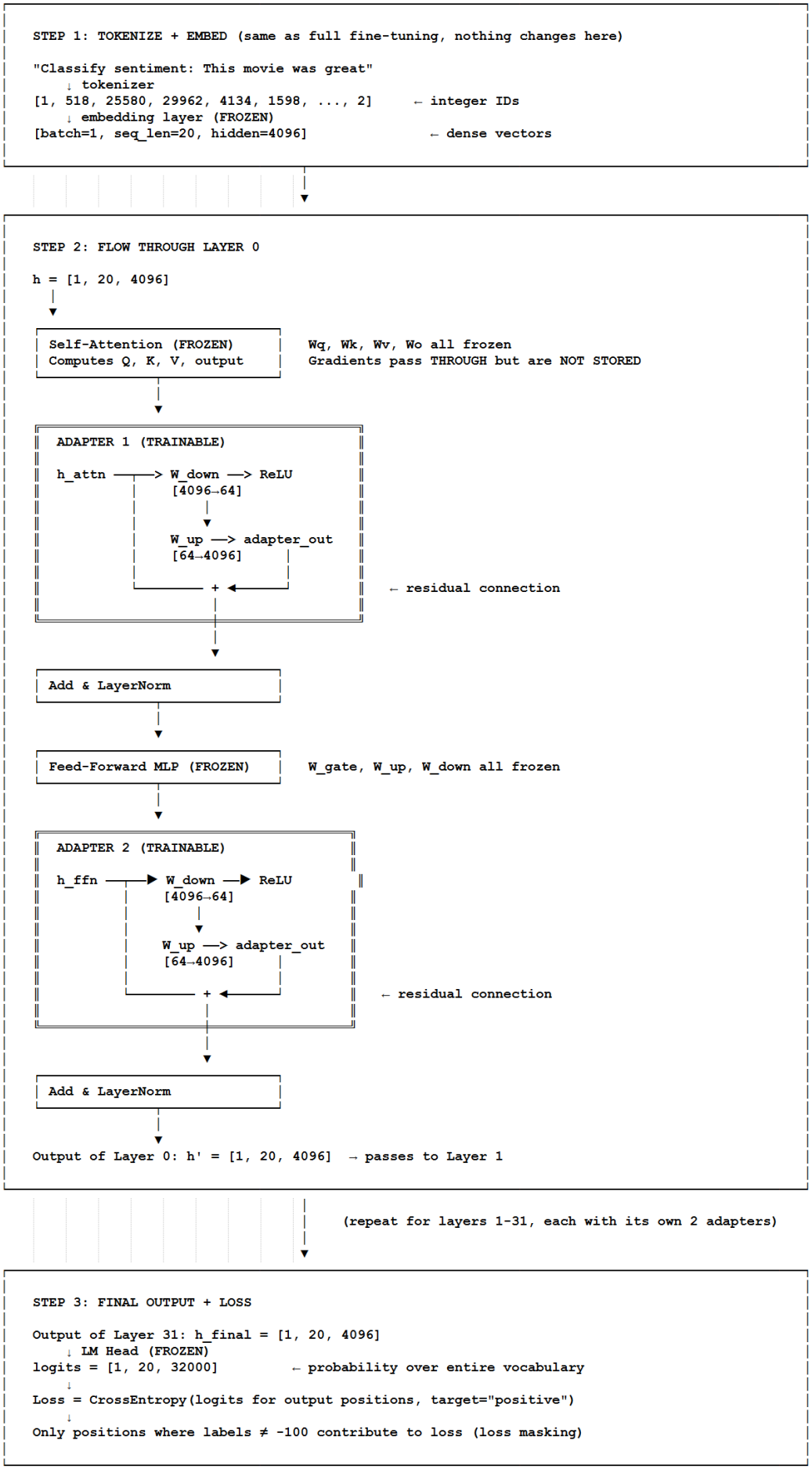
DATA FLOW THROUGH ENTIRE MODEL — FORWARD PASS WITH ADAPTERS

Input: "Classify sentiment: This movie was great" → "positive"

Complete forward pass data flow:

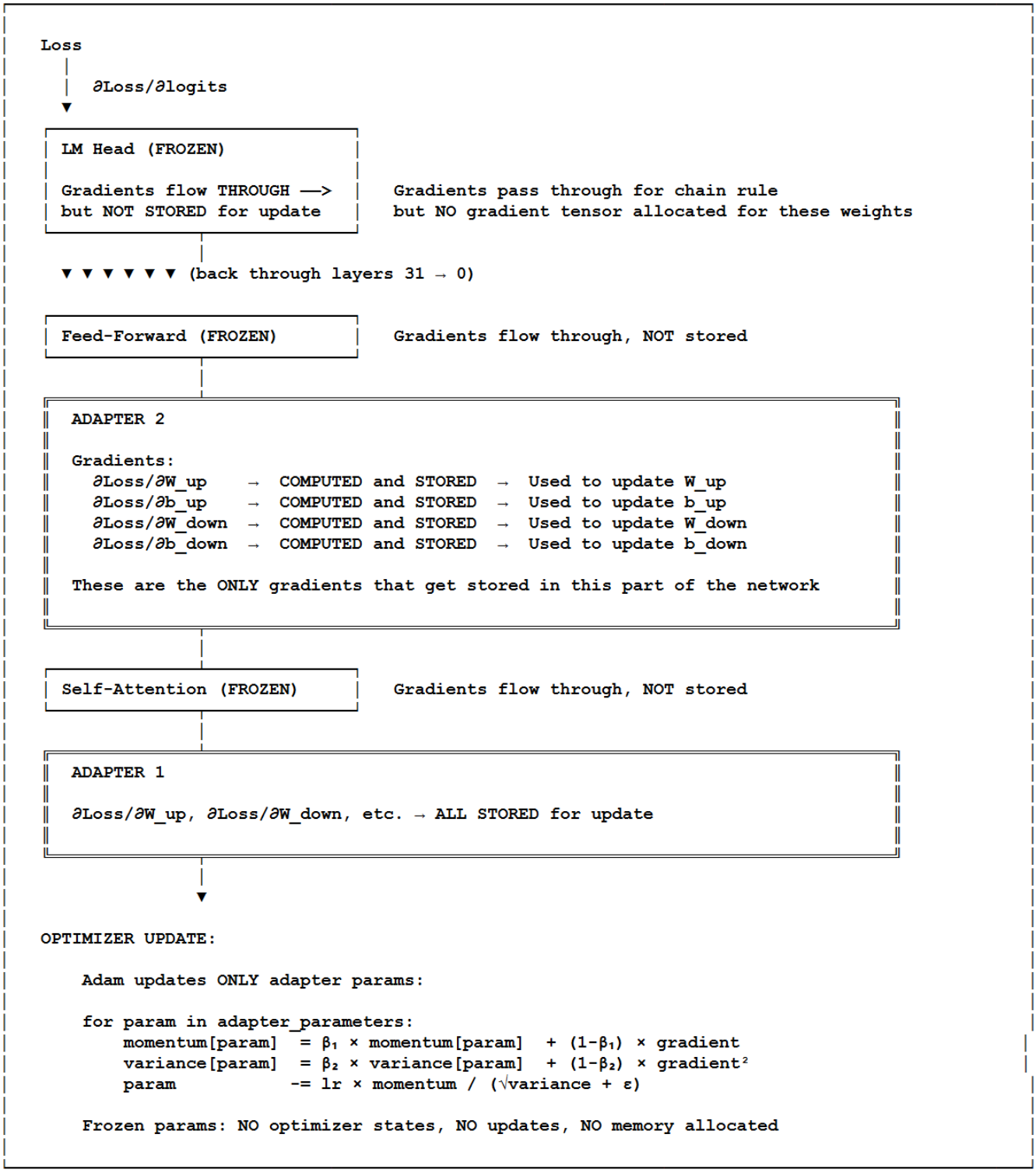
tokenize → embed → through all 32 layers with both adapters → logits → loss

Fine Tuning


STEP 1: TOKENIZE + EMBED (same as full fine-tuning, nothing changes here)
"Classify sentiment: This movie was great"
↓ tokenizer
[1, 518, 25580, 29962, 4134, 1598, ..., 2] ← integer IDs
↓ embedding layer (FROZEN)
[batch=1, seq_len=20, hidden=4096] ← dense vectors
STEP 2: FLOW THROUGH LAYER 0
h = [1, 20, 4096]
Self-Attention (FROZEN)
Computes Q, K, V, output
Wq, Wk, Wv, Wo all frozen
Gradients pass THROUGH but are NOT STORED
ADAPTER 1 (TRAINABLE)
h_attn
W_down
[4096→64]
ReLU
W_up
[64→4096]
adapter_out
+
← residual connection
Add & LayerNorm
Feed-Forward MLP (FROZEN)
W_gate, W_up, W_down all frozen
ADAPTER 2 (TRAINABLE)
h_ffn
W_down
[4096→64]
ReLU
W_up
[64→4096]
adapter_out
+
← residual connection
Add & LayerNorm
Output of Layer 0: h' = [1, 20, 4096] → passes to Layer 1
(repeat for layers 1-31, each with its own 2 adapters)
STEP 3: FINAL OUTPUT + LOSS
Output of Layer 31: h_final = [1, 20, 4096]
↓ LM Head (FROZEN)
logits = [1, 20, 32000] ← probability over entire vocabulary
↓
Loss = CrossEntropy(logits for output positions, target="positive")
↓
Only positions where labels ≠ -100 contribute to loss (loss masking)

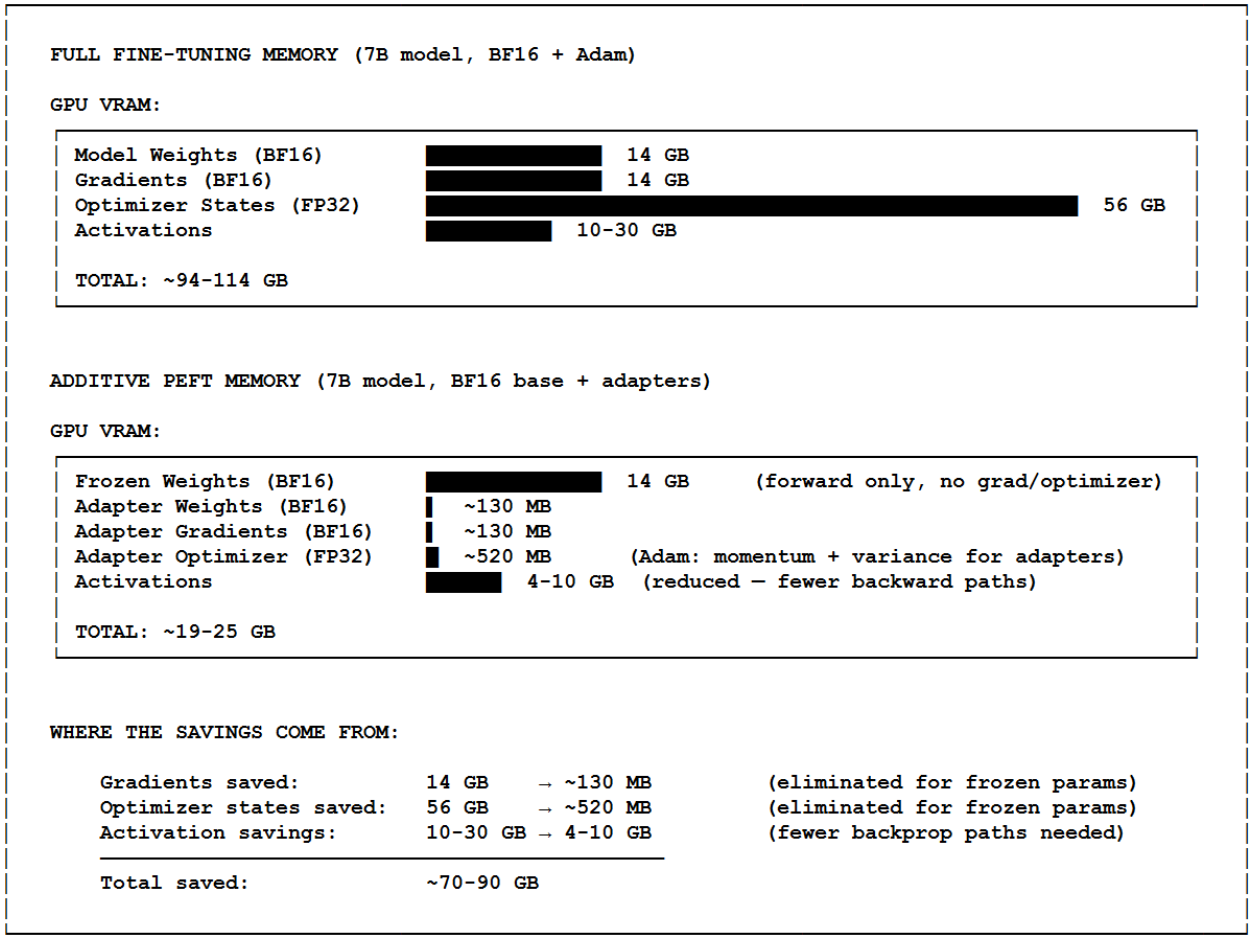
BACKWARD PASS — WHERE GRADIENTS FLOW (AND DON'T)

Backward pass showing exactly where gradients flow through (frozen layers) vs. where they're stored (adapters only), plus the optimizer update



MEMORY LAYOUT — WHAT LIVES WHERE

Memory layout comparison: Full FT (~94-114 GB) vs. Additive PEFT (~19-25 GB), with bar-style visualization showing where the savings come from



PARAMETER COUNT BREAKDOWN ACROSS THE MODEL:

Full parameter count breakdown: every component in a 7B model, which are frozen, which are trainable

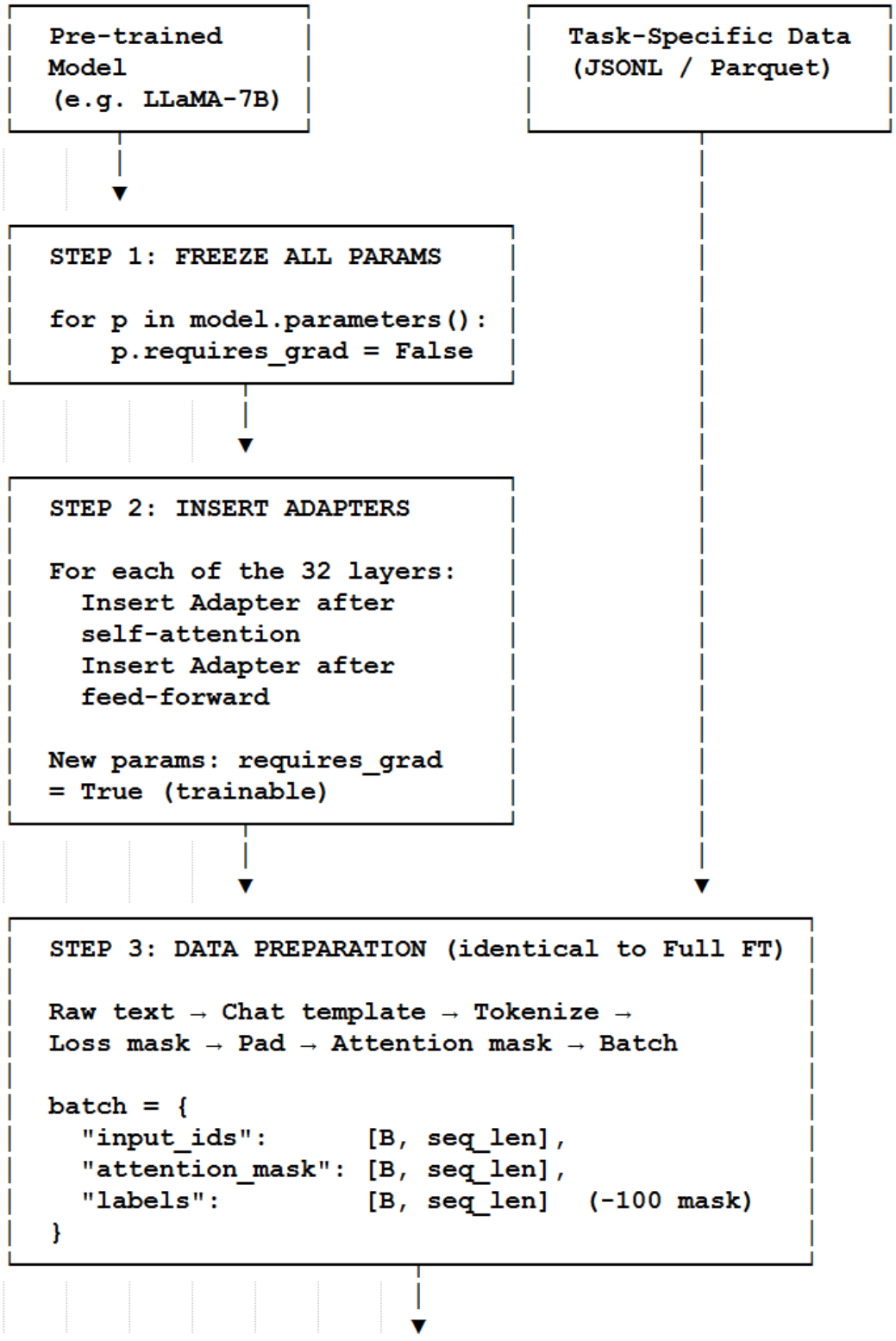
7B MODEL WITH BOTTLENECK ADAPTERS (bottleneck_dim=64, 32 layers, 2 adapters/layer)

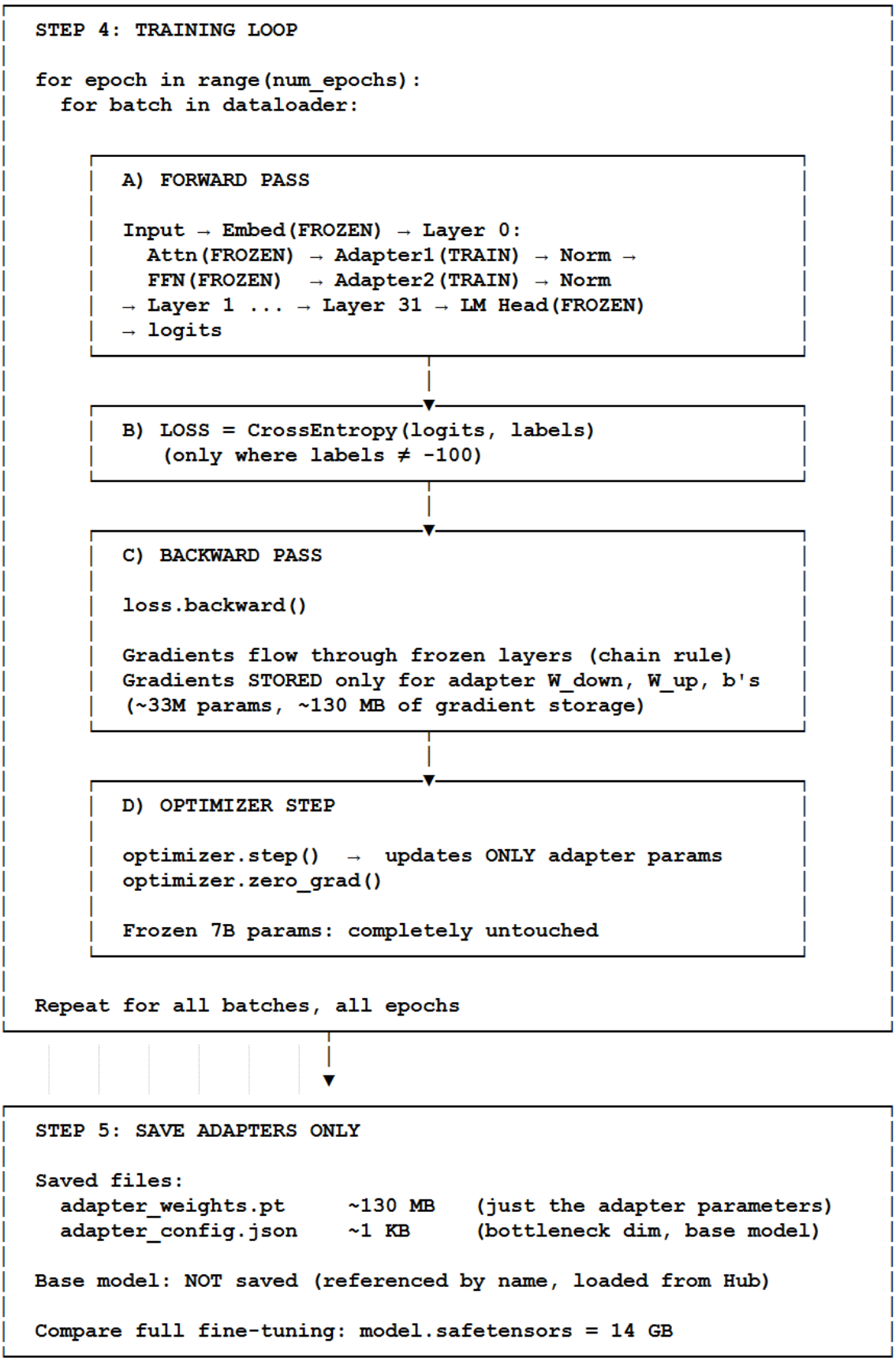
Component	Params	Trainable?	Notes
Embedding layer	131M	FROZEN	[32000 vocab × 4096]
Per transformer layer:			
Self-Attention			
W_q	16.8M	FROZEN	[4096 × 4096]
W_k	16.8M	FROZEN	[4096 × 4096]
W_v	16.8M	FROZEN	[4096 × 4096]
W_o	16.8M	FROZEN	[4096 × 4096]
▓▓ Adapter 1 ▓▓	~0.53M	★ TRAINABLE	[4096→64→4096] + biases
LayerNorm	8K	FROZEN	[4096] × 2
Feed-Forward MLP			
W_gate	45.1M	FROZEN	[4096 × 11008]
W_up	45.1M	FROZEN	[4096 × 11008]
W_down	45.1M	FROZEN	[11008 × 4096]
▓▓ Adapter 2 ▓▓	~0.53M	★ TRAINABLE	[4096→64→4096] + biases
LayerNorm	8K	FROZEN	[4096] × 2
LM Head	131M	FROZEN	[4096 × 32000]

FROZEN: ~6,738M (99.5%)
TRAINABLE: ~33.6M (0.5%) ← just the adapters

TRAINING LOOP — ADDITIVE PEFT END-TO-END

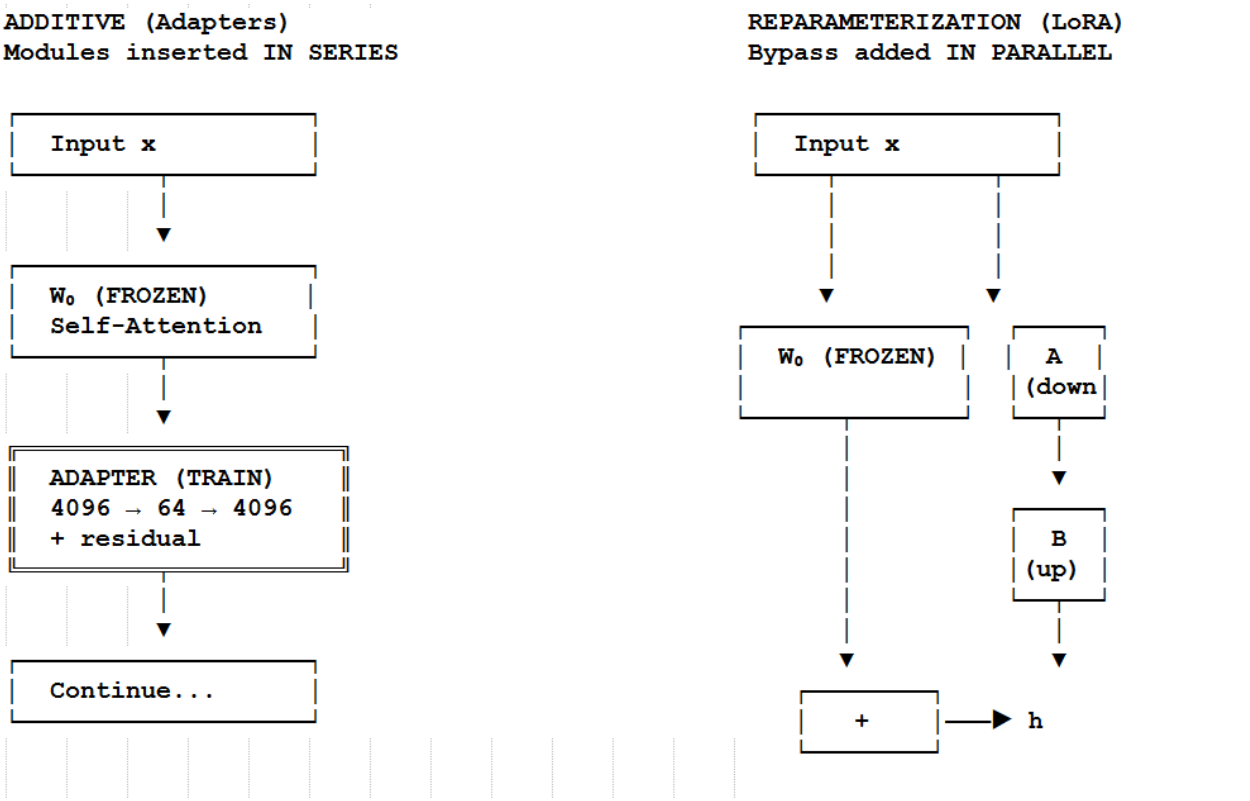
End-to-end training loop:
freeze → insert adapters → data prep → forward → loss → backward → update → save




STEP 4: TRAINING LOOP
for epoch in range(num_epochs):
for batch in dataloader:
A) FORWARD PASS
Input → Embed(FROZEN) → Layer 0:
Attn(FROZEN) → Adapter1(TRAIN) → Norm →
FFN(FROZEN) → Adapter2(TRAIN) → Norm
→ Layer 1 ... → Layer 31 → LM Head(FROZEN)
→ logits
B) LOSS = CrossEntropy(logits, labels)
(only where labels ≠ -100)
C) BACKWARD PASS
loss.backward()
Gradients flow through frozen layers (chain rule)
Gradients STORED only for adapter W_down, W_up, b's
(~33M params, ~130 MB of gradient storage)
D) OPTIMIZER STEP
optimizer.step() → updates ONLY adapter params
optimizer.zero_grad()
Frozen 7B params: completely untouched
Repeat for all batches, all epochs
STEP 5: SAVE ADAPTERS ONLY
Saved files:
adapter_weights.pt ~130 MB (just the adapter parameters)
adapter_config.json ~1 KB (bottleneck dim, base model)
Base model: NOT saved (referenced by name, loaded from Hub)
Compare full fine-tuning: model.safetensors = 14 GB

ADDITIVE vs REPARAMETERIZATION (LoRA) — ARCHITECTURAL DIFFERENCE

Adapters (in-series) vs. LoRA (in-parallel) architectural comparison, highlighting why LoRA won (mergeability)

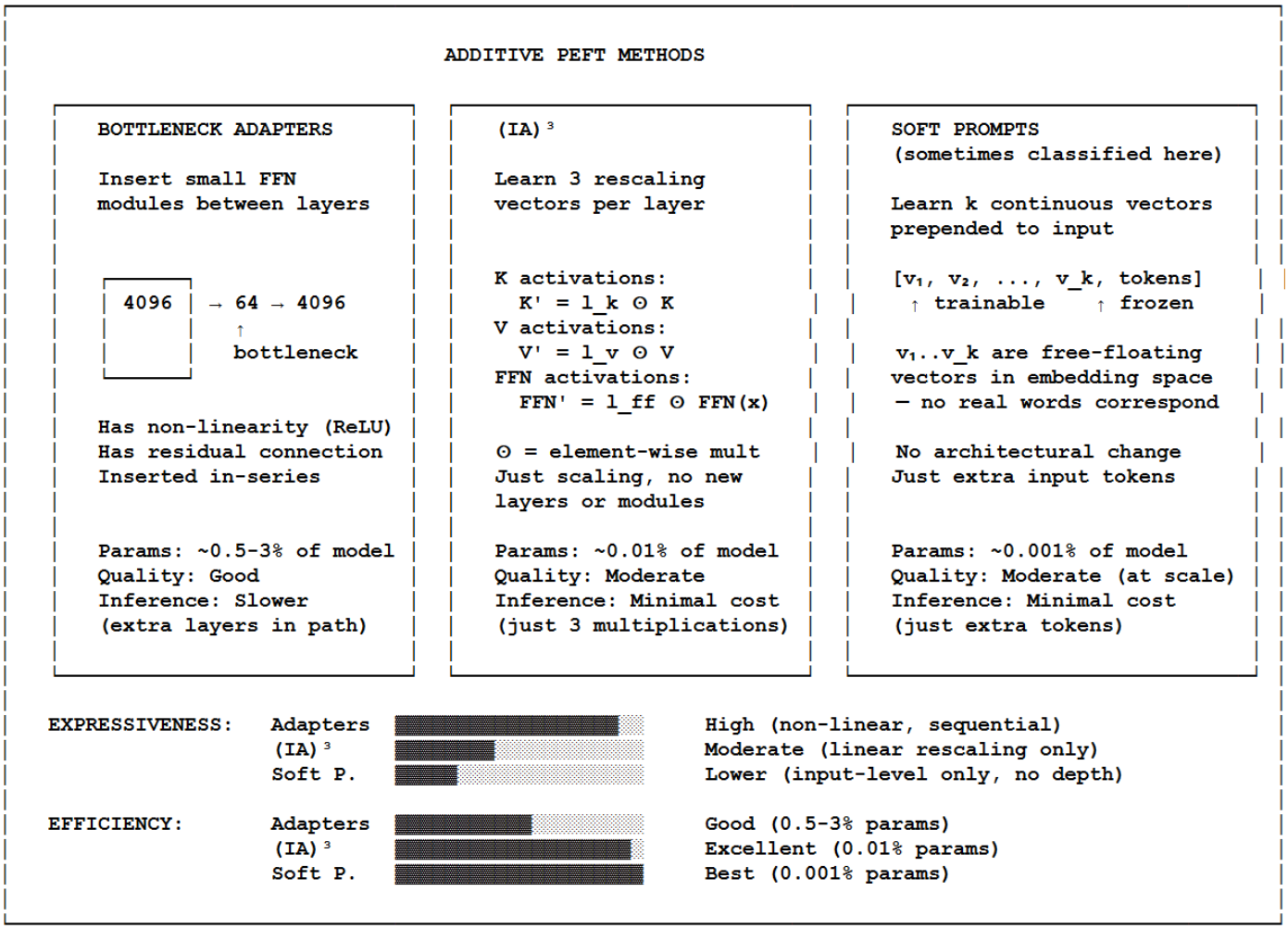


KEY DIFFERENCE:

Adapters:	CANNOT be merged.	Extra latency at inference.	PERMANENT.
LoRA:	CAN be merged.	Zero latency at inference.	REMOVABLE.

This single difference is why LoRA replaced Adapters as the standard.

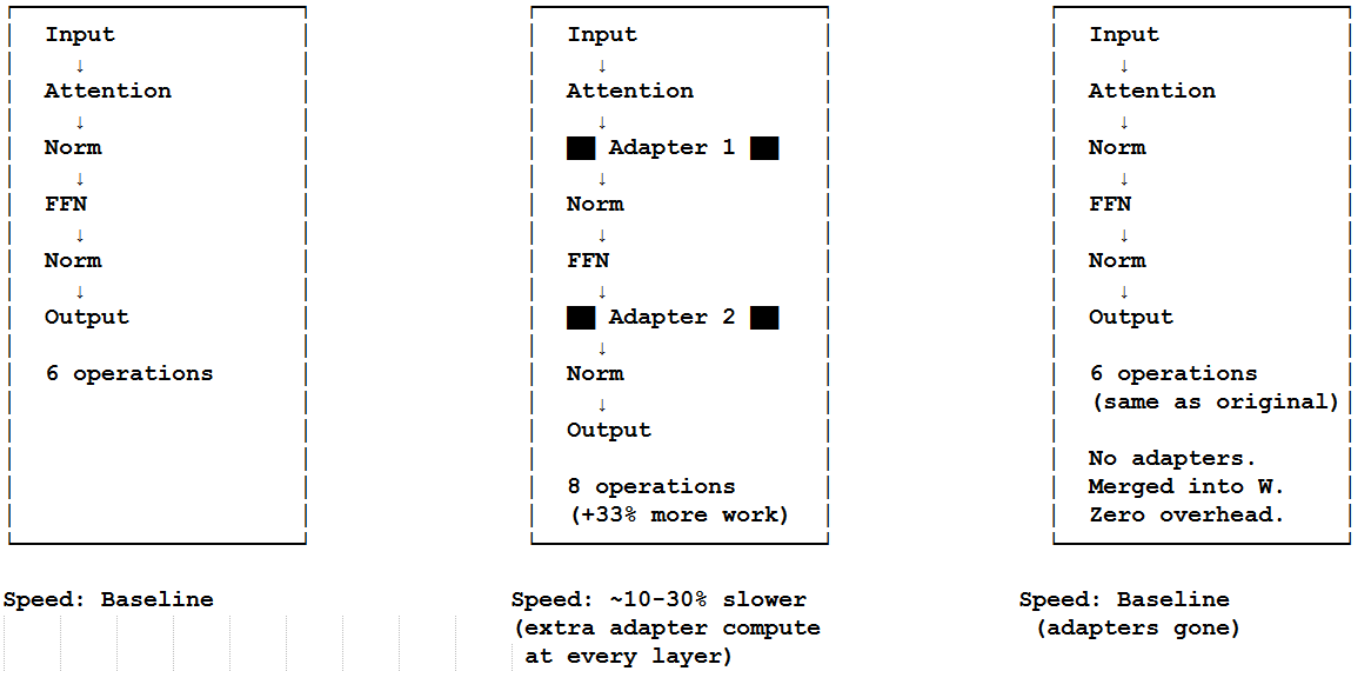
ALL THREE ADDITIVE SUB-METHODS — SIDE BY SIDE:



INFERENCE — THE PERMANENT COST OF ADAPTERS

Inference cost: showing the permanent latency penalty of adapters vs. zero overhead after LoRA merge

INFERENCE (serving predictions to users)

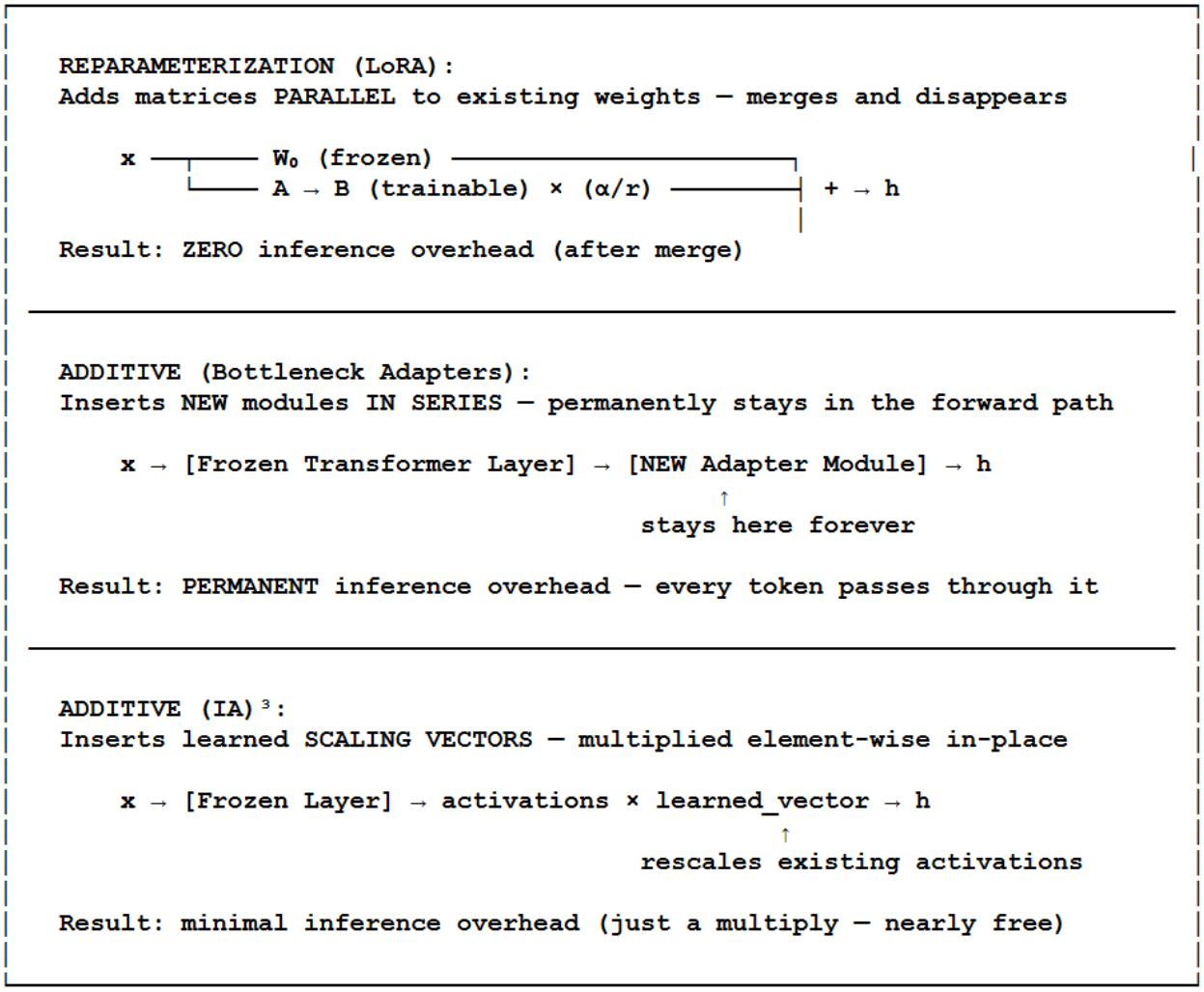


Bottleneck Adapters & $(IA)^3$

Additive PEFT — Complete Breakdown

The Core Idea — What Makes Additive Different

Every PEFT method freezes the base model and trains something tiny. The question is *what* you add and *where* you put it.



The Defining Trade-Off: Additive Methods vs LoRA

Property	**LoRA**	**Bottleneck Adapters**	**(IA)³**
Inference Overhead	Zero (after merge)	Permanent (extra modules remain)	~Zero (just elementwise multiply)
Architecture Change	None (parallel bypass)	Yes (new serial layers added)	Minimal (scale vectors applied)
Expressiveness	High	Highest (has non-linearity)	Lowest
Parameters Added	~0.1–1%	~0.5–3%	~0.01–0.1%
Non-linearity	No	Yes (ReLU/GELU in bottleneck)	No
Mergeable into Base Model	Yes	No	Yes (into weight matrices)

Additive – Part 1: BOTTLENECK ADAPTERS

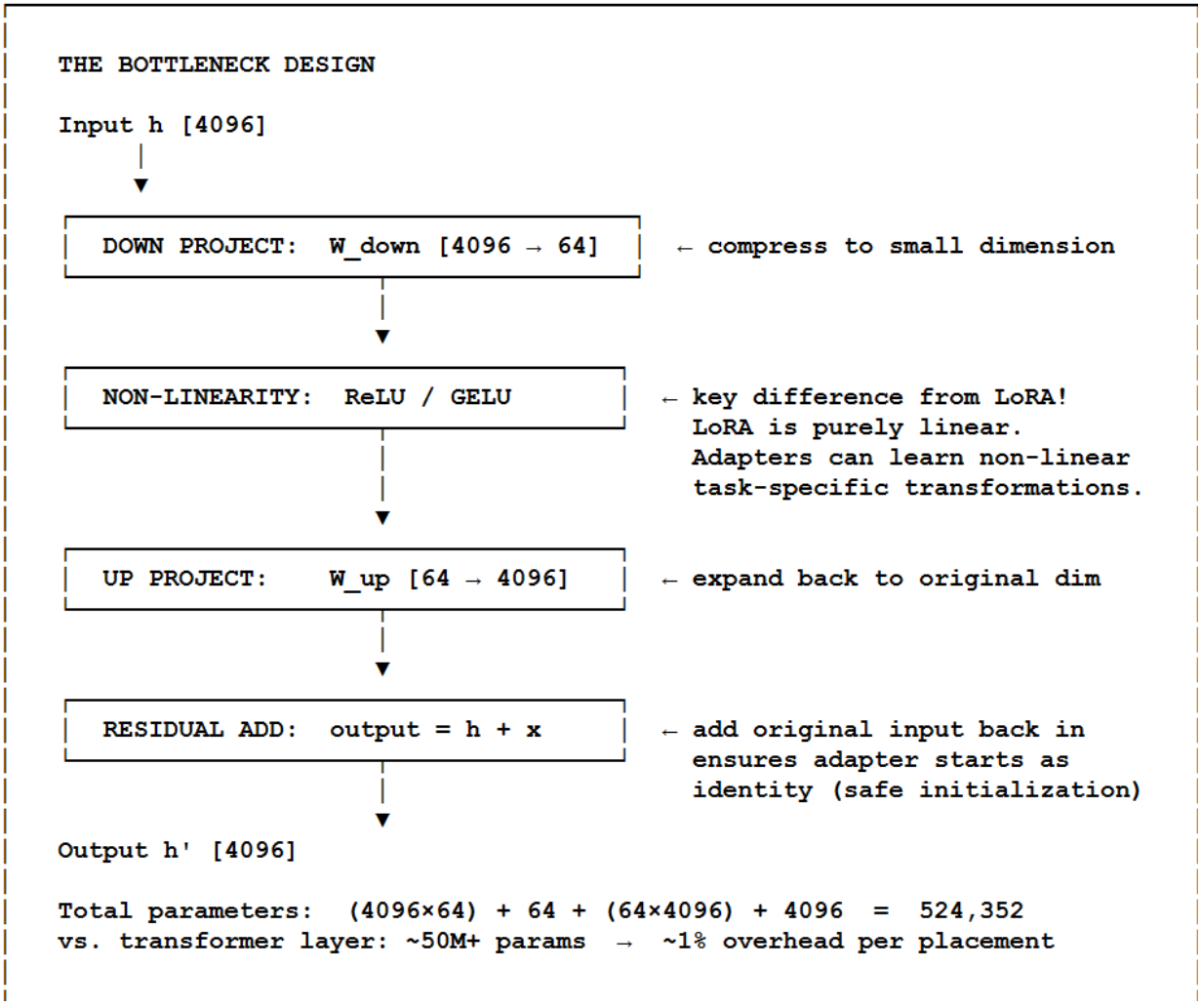
What Problem Bottleneck Adapters Solve

Houlsby et al. (2019) — the original adapter paper — made a simple observation:

"What if we could add a small trainable module after each transformer layer,
leave everything else frozen, and teach only that module the task ?"

The module had to be small (so it doesn't eat all VRAM) but expressive enough
to capture complex task-specific transformations.

The solution: a bottleneck.



The bottleneck dimension (64 in the example) is the key hyperparameter — called reduction_factor or bottleneck_dim. Smaller = fewer params, less expressive.

The Math

For a single adapter module applied to hidden state h:

```
h' = h  +  W_up · activation( W_down · h  +  b_down )  +  b_up
↑             ↑                    ↑
residual    expand                        compress + non-linearity
(original)  back up                       through bottleneck
```

Where:

- W_down : [hidden_dim × bottleneck_dim] e.g. [4096 × 64]
- W_up : [bottleneck_dim × hidden_dim] e.g. [64 × 4096]
- b_down, b_up : bias vectors
- activation : ReLU or GELU

Initialization (the identity trick):

W_up = zeros → W_up · anything = 0
W_down = random → b_up + W_up · act(W_down · h) = 0

Therefore at step 0: h' = h + 0 = h ← perfect identity function

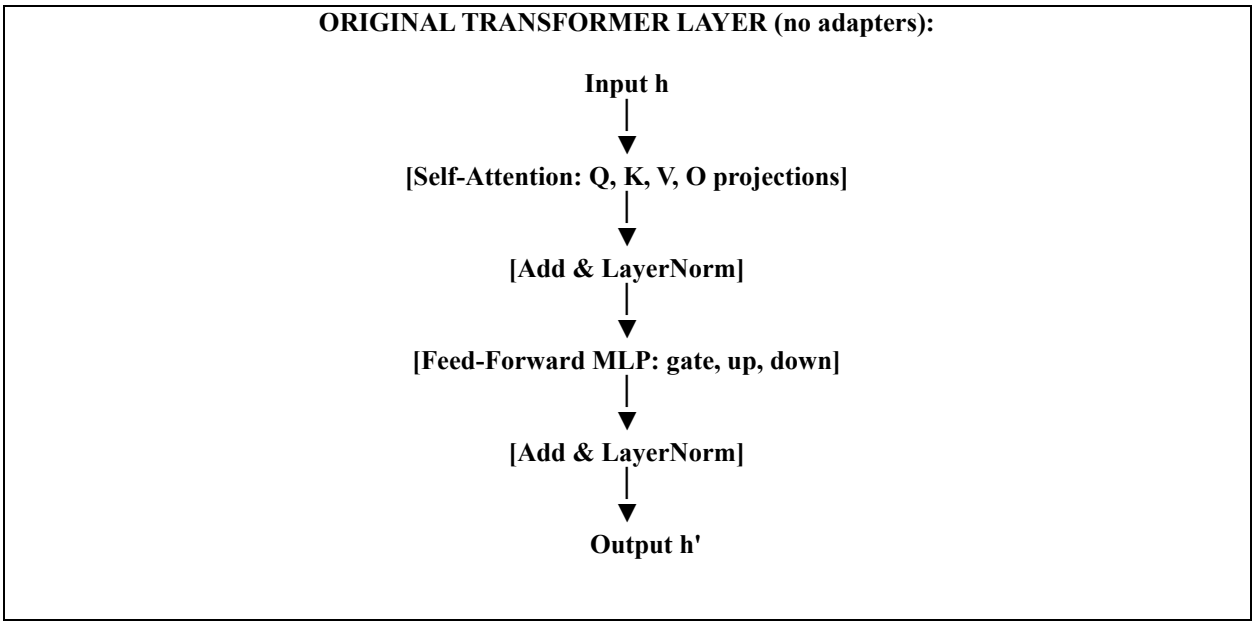
Same principle as LoRA's B = 0

init — the adapter starts transparent and learns to deviate from identity only as training progresses.

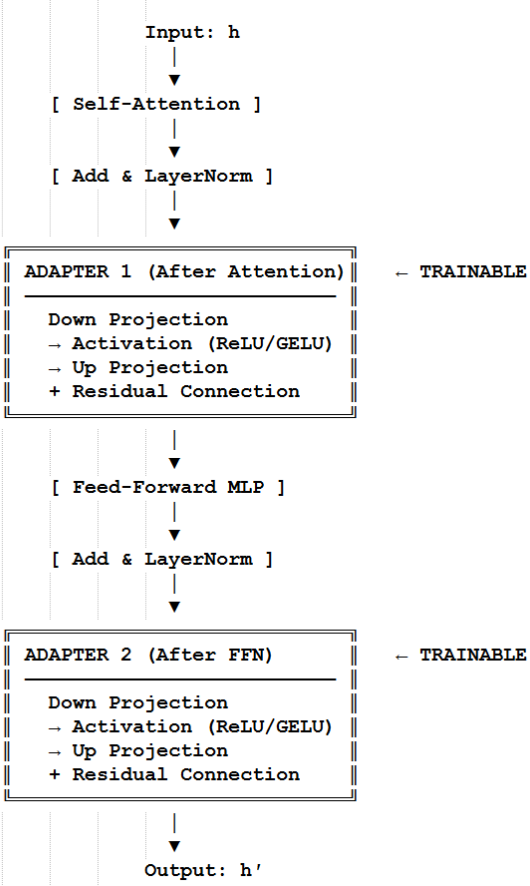
Where Adapters Are Placed in the Transformer

This is where Additive differs fundamentally from LoRA. LoRA adds a parallel bypass to individual weight matrices.

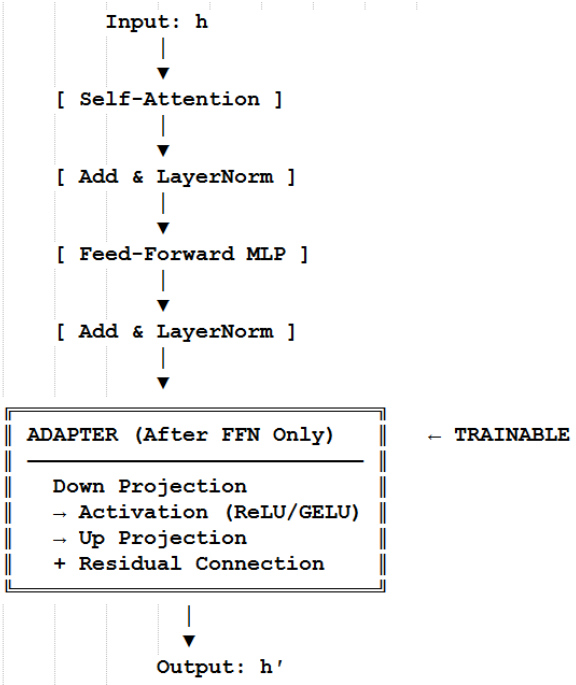
Adapters are inserted as entire new sequential modules between existing layers.



HOULSBY ADAPTER (2019) — 2 Adapters Per Transformer Layer:



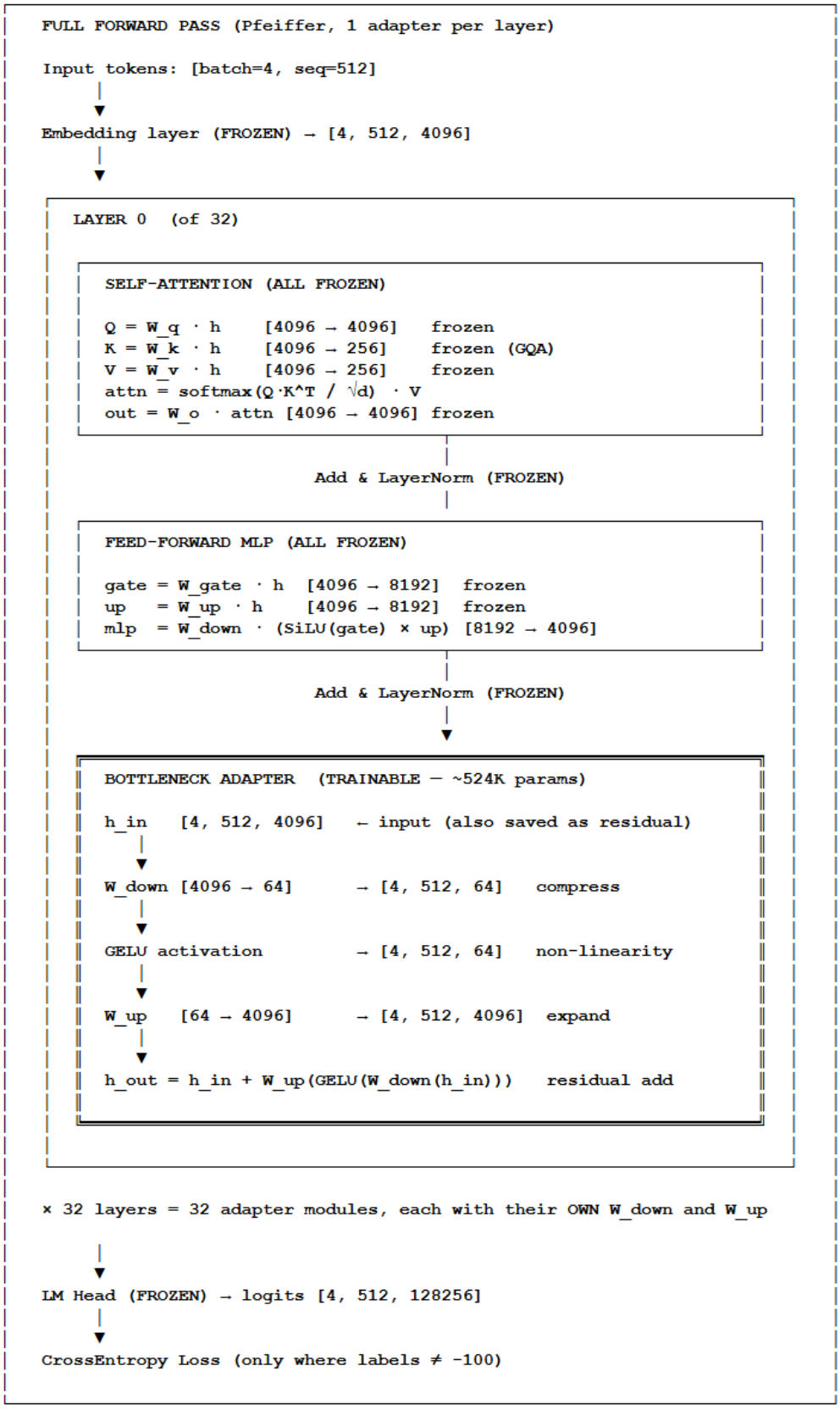
PFEIFFER ADAPTER (2020) — 1 Adapter Per Transformer Layer
(More Parameter Efficient than Houlsby)



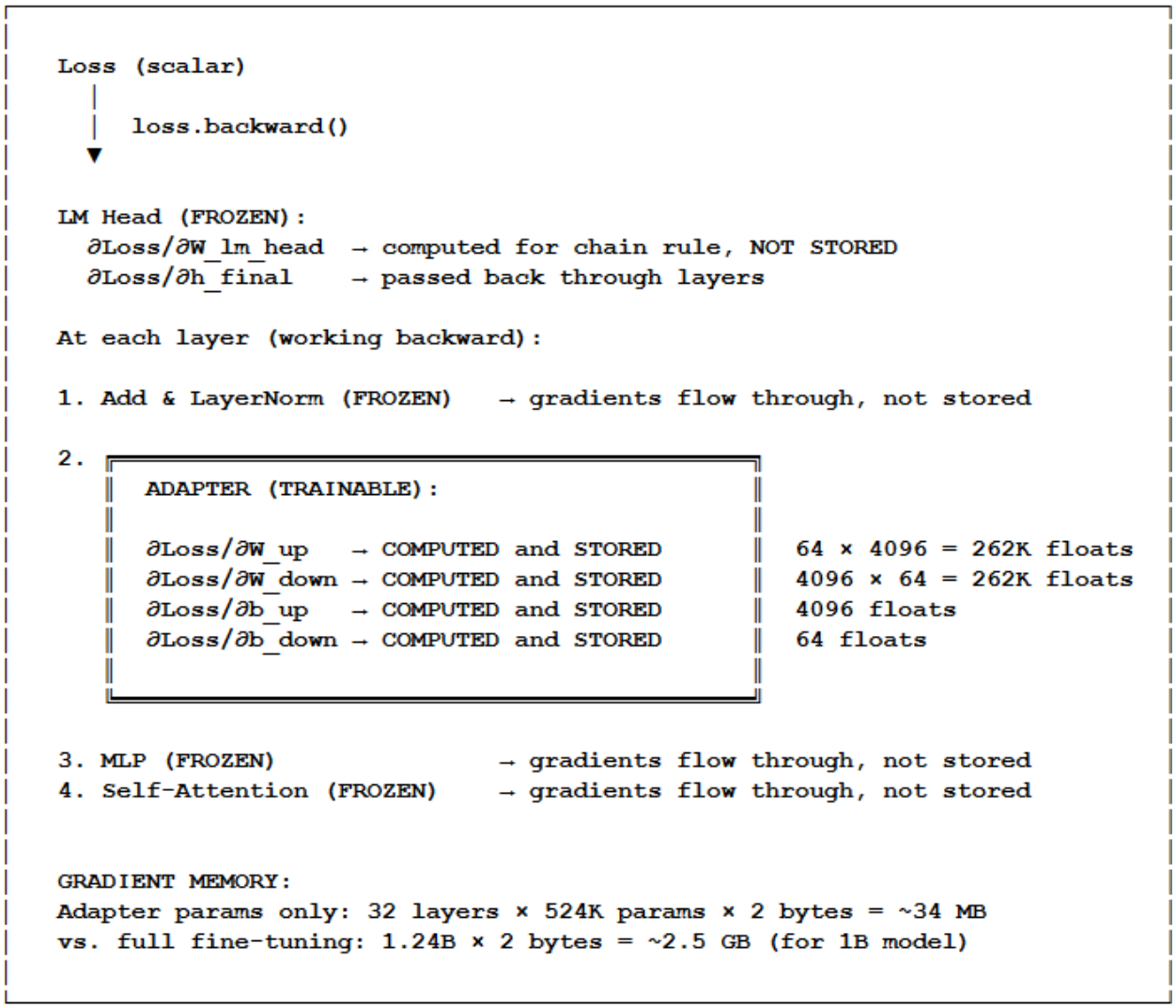
Houlsby = 2 adapters per layer, higher capacity, more parameters.
Pfeiffer = 1 adapter per layer, more efficient, nearly same performance.
Pfeiffer is the standard today.

Houlsby refers to the adapter method introduced in:
Neil Houlsby et al., 2019
Paper: “Parameter-Efficient Transfer Learning for NLP”

The Complete Forward Pass



Backward Pass — Where the Gradients Go



Parameter Count

```
PARAMETER COUNT (Pfeiffer, 1B model, bottleneck=64)

Per adapter module:
  W_down: [4096 × 64]     = 262,144 params
  b_down: [64]            =      64 params
  W_up:   [64 × 4096]     = 262,144 params
  b_up:   [4096]          =   4,096 params
  ------------------------------------
  Per adapter:              528,448 params

× 16 layers (1B model) = 8,455,168 params

vs. base model: 1,240,000,000 params
Trainable: 0.68%  — comparable to LoRA at r=16


Effect of bottleneck size:

Bottleneck   Params/adapter   Total (16 layers)   % of 1B model
---------    -------------    -----------------   -------------
     8         66,624           1,065,984           0.09%
    16        131,200           2,099,200           0.17%
    32        266,240           4,259,840           0.34%
    64        528,448           8,455,168           0.68%  *
   128      1,052,928          16,846,848           1.36%
   256      2,101,248          33,619,968           2.71%

* = good default starting point
```

Why Adapters Can't Merge (the key LoRA advantage they lack)

```
LoRA merge:
  W_merged = W0 + (a/r) · B · A

  This works because LoRA adds a LINEAR transformation in parallel.
  Two linear transforms can always be summed into one.
  W_merged is the same shape as W0.  Architecture unchanged.

-------------------------------------------------------------------------

Adapter "merge" attempt:
  h' = h + W_up · GELU( W_down · h )
                        ?
                   NON-LINEAR

You CANNOT absorb this into any existing weight matrix because GELU
depends on the VALUE of h, not just its linear transformation.
There is no single matrix W_merged such that W_merged · h = h'.

The adapter must stay in the computational graph forever.
Every token at inference time runs through:

  ... ? FFN ? LayerNorm ? W_down ? GELU ? W_up ? residual add ? ...

This adds latency proportional to the bottleneck size at every layer.
For seq_len=512, batch=1, 32 layers: ~5-15% extra inference time.
```

Saving and Loading Adapters

adapter_config.json:

```
{
  "adapter_type": "bottleneck",
  "base_model_name_or_path": "unsloth/Llama-3.2-1B-Instruct",
  "bottleneck_dim": 64,
  "non_linearity": "gelu",
  "adapter_placement": "after_ffn",    // "after_attn_and_ffn" or "after_ffn"
  "num_layers": 16
}
```

Saved weights (per adapter, per layer):
W_down: [4096, 64] ~2 MB per layer
b_down: [64]
W_up: [64, 4096] ~2 MB per layer
b_up: [4096]

Total file: ~16 layers × ~4 MB = ~64 MB
vs. full model: 2.5 GB
vs. LoRA at r=16: ~15 MB ← LoRA is smaller because no biases, no non-linearity

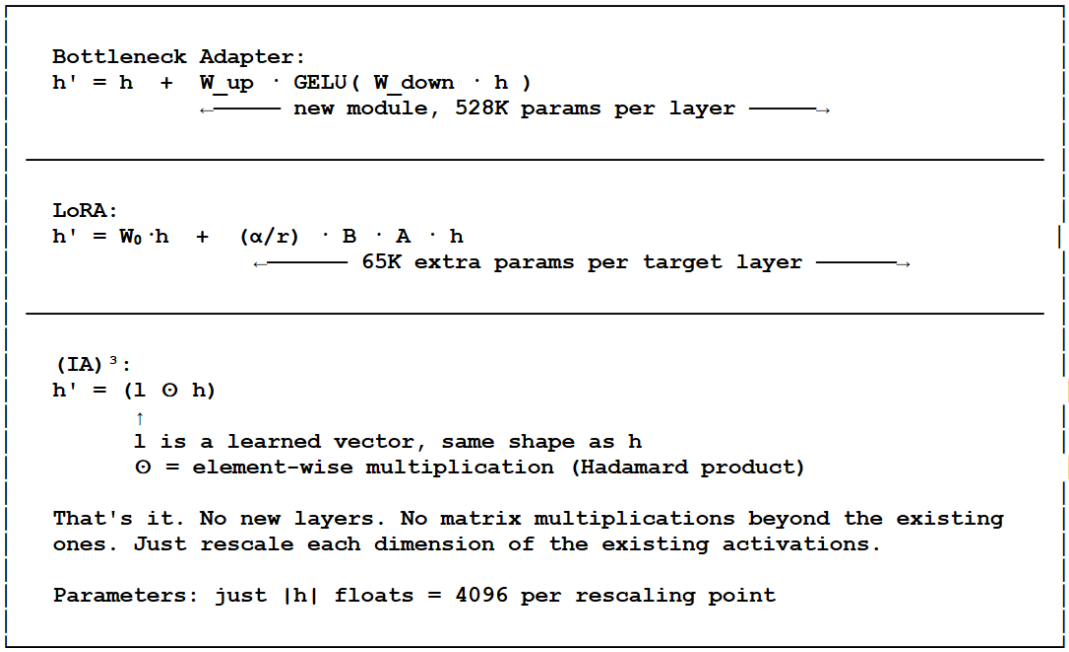
---

Additive - Part 2: (IA)³ — Infused Adapter by Inhibiting and Amplifying Inner Activations

What (IA)³ Is:

Liu et al. (2022). The paper's title is the abbreviation: Infused Adapter by Inhibiting and Amplifying Inner Activations.

The insight is radical simplicity: don't add new modules. Don't add new weight matrices. Just learn vectors that rescale the existing activations.



The Math

(IA)³ learns three sets of vectors — l_k, l_v, and l_ff:

Attention:

$$K = (l_k \odot W_k) \cdot x \quad \leftarrow \text{rescale KEY vectors element-wise}$$
$$V = (l_v \odot W_v) \cdot x \quad \leftarrow \text{rescale VALUE vectors element-wise}$$

The l vectors gate which features in K and V are amplified or suppressed.
High values in l → that dimension matters more for this task.
Low values in l → that dimension is inhibited.

Feed-Forward:

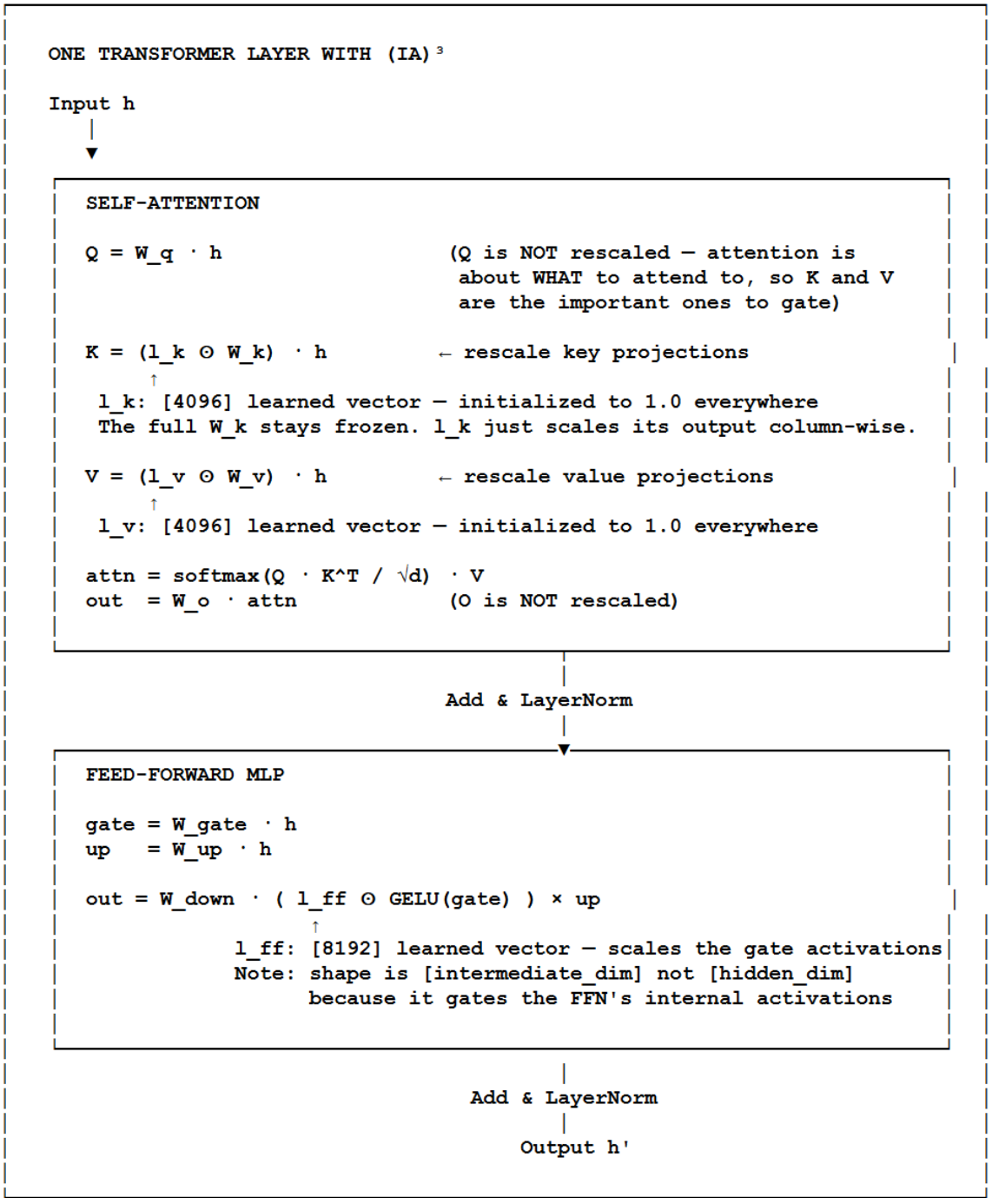
$$\text{output} = (l_{ff} \odot \text{GELU}(\text{gate})) \times \text{up} \quad \leftarrow \text{rescale gate activations}$$

Initialization:

$$l_k = \text{ones} \rightarrow l_k \odot W_k = 1 \times W_k = W_k \quad \text{(identity — unchanged)}$$
$$l_v = \text{ones} \rightarrow l_v \odot W_v = 1 \times W_v = W_v \quad \text{(identity — unchanged)}$$
$$l_{ff} = \text{ones} \rightarrow l_{ff} \odot \text{act} = 1 \times \text{act} = \text{act} \quad \text{(identity — unchanged)}$$

Same principle as B=0 in LoRA and W_up=0 in Adapters:
model starts as identical to base, deviations learned over training.

Where (IA)³ Applies Scaling



Parameter Count — Remarkably Tiny

```
PARAMETER COUNT (IA)³ on 1B model (Llama-3.2-1B: 16 layers, d=2048,
                                   kv_d=256, ffn=8192)

Per layer:
  l_k:  [kv_d] = [256]   ← key head dimension (GQA)
  l_v:  [kv_d] = [256]   ← value head dimension
  l_ff: [ffn]  = [8192]  ← FFN intermediate dimension
  ──────────────────────────────
  Per layer: 256 + 256 + 8192 = 8,704 params

× 16 layers = 139,264 total trainable parameters

vs. base model: 1,240,000,000 params
Trainable: 0.011%  ← this is ~10x fewer than LoRA r=16
                   ← ~60x fewer than Bottleneck Adapters


COMPARISON TABLE (1B model):

Method              Trainable Params    % of Model    File Size
──────              ────────────────    ──────────    ─────────
(IA)³               ~139K               0.011%        ~0.5 MB
LoRA (r=4)          ~4.2M               0.34%         ~16 MB
LoRA (r=16)         ~10.2M              0.82%         ~40 MB
Bottleneck (b=64)   ~8.5M               0.68%         ~64 MB
Full Fine-Tuning    1,240M              100%          ~2.5 GB

(IA)³ is the most parameter-efficient method in all of PEFT.
```

Can (IA)³ Merge?

Yes — and this is a surprise given it's Additive. Because the rescaling is LINEAR (element-wise multiply has no non-linearity), it CAN be folded in:

(IA)³ key rescaling:

$$\mathbf{K = (l\_k \odot W\_k) \cdot h}$$

This is equivalent to:

$$\mathbf{K = W\_k\_merged \cdot h}$$

where

$$\mathbf{W\_k\_merged = diag(l\_k) \cdot W\_k}$$

Or more efficiently (no explicit diag):

$$\mathbf{W\_k\_merged[i, :] = l\_k[i] \times W\_k[i, :]} \quad \textbf{(scale each row by l\_k[i])}$$

After merge: W_k becomes W_k_merged, l_k is discarded.
Zero inference overhead. Same as LoRA.

CONTRAST with Bottleneck Adapters:

h' = h + W_up · GELU(W_down · h) ← GELU makes this non-mergeable

$(IA)^3$ is the only Additive method that is also mergeable.
It sits at the intersection of Additive and Reparameterization.

<u>$(IA)^3$ Visualised — What the l Vectors Actually Do</u>

Think of each dimension of h as a "feature channel."
l is a learned dial for each channel — amplify or suppress it.
Key vector K [dim=8]:

```
Before rescaling:   K = [ 0.8,  0.2, -0.5,  0.9, -0.1,  0.3,  0.7, -0.4]
l_k (learned):          [ 2.1,  0.1,  1.8,  0.05, 1.5, 0.02,  2.3, 1.9 ]
After rescaling:    K = [ 1.68, 0.02, -0.9, 0.045,-0.15, 0.006, 1.61,-0.76]
                          ↑           ↑                    ↑
                    amplified   amplified            amplified
                      by 2.1    by 1.8                by 2.3
                          ↑               ↑
                     suppressed       suppressed
                      (×0.1)           (×0.02)
```

The model learns: for THIS task, features 0, 2, 6 are important (amplify them), features 1, 5 are noise for this task (suppress them).

This is essentially learned feature selection — which dimensions of the pretrained representations matter for the target task.

Head To Head Comparision:

BOTTLENECK ADAPTERS vs $(IA)^3$ vs LoRA

Description	Bottleneck Adapters	$(IA)^3$	LoRA (r=16)
MECHANISM			
Operation type	Non-linear	Linear	Linear
Where applied	Serial (in sequence)	In-place scaling	Parallel bypass
Non-linearity	Yes (GELU/ReLU)	No	No
Residual connection	Yes	No	No (bypass itself)
PARAMETERS (1B model)			
Trainable params	~8.5M (0.68%)	~139K (0.011%)	~10.2M (0.82%)
File size	~64 MB	~0.5 MB	~40 MB
Per-layer overhead	528K	8.7K	640K
EXPRESSIVENESS			
Capture non-linear transformations?	YES	No	No
Best for	Complex domain shift tasks	Lightweight task steering	General-purpose fine-tuning
Feature selection	Implicit (bottleneck)	Explicit (l vectors)	Implicit (rank)
INFERENCE			
Extra compute	Yes (permanent)	~None	None (after merge)
Can merge into base model?	NO	YES	YES
Latency overhead	5–15%	<1%	0% (after merge)
Deployment format	Adapter + base	Merge or keep	Merge or keep
TRAINING			
VRAM overhead	Slightly more (extra activations)	Minimal	Comparable
Typical learning rate	1e-4 to 1e-3	1e-3 to 3e-3 (higher OK — very few params)	1e-4 to 3e-4

WHEN TO CHOOSE EACH:

Choose Bottleneck if: Task requires complex non-linear transformation
Domain shift is large (e.g., English ? chemistry)
Expressiveness matters more than inference speed
You're OK with permanent adapter overhead

Choose $(IA)^3$ if: Extreme parameter efficiency required
Many tasks, tiny storage budget
Hardware is very constrained
Task steering rather than deep domain adaptation

Choose LoRA if: Best general-purpose choice for most tasks
Zero inference overhead required after deployment
Good balance of expressiveness vs. parameter count
Adapter swapping needed (before merge)

COMPLETE DATA FLOW (both methods)

DATA PIPELINE — identical to LoRA up to the forward pass

STAGE 1: Raw JSONL → exactly the same as LoRA / full fine-tuning

STAGE 2: Template formatting → exactly the same

STAGE 3: Tokenization → exactly the same

STAGE 4: Labels + loss mask (-100 for instruction tokens) → same

STAGE 5: Padding + attention mask → same

STAGE 6: Move to GPU + Embedding lookup → same

STAGE 7: Forward pass — THIS is where they diverge:

LoRA:

At each target weight	:	$h = W_0 \cdot x + (\alpha/r) \cdot B \cdot A \cdot x$ (parallel path added)
Adapter modules	:	NOT present — no extra modules exist

Bottleneck Adapters:

At each target weight	:	$h = W_0 \cdot x$ (normal frozen path)
After each FFN block	:	$h = h + W\_up \cdot GELU(W\_down \cdot h)$ (extra module applied)

$(IA)^3$:

At key projection	:	$K = (l\_k \odot W\_k) \cdot x$ (rescaled in-place)
At value projection	:	$V = (l\_v \odot W\_v) \cdot x$ (rescaled in-place)
At FFN gate	:	$out = W\_down \cdot (l\_ff \odot GELU(gate)) \times up$

STAGE 8: Loss → exactly the same (CrossEntropy, -100 masking)

STAGE 9: Backward pass → gradients stored ONLY for adapter/l params

STAGE 10: Optimizer update → updates ONLY adapter/l params

STAGE 11: Save → adapter weights / l vectors only

PRACTICAL DECISION GUIDE

```
+-------------------------------------------------------------------------------+
|                                                                               |
|   START HERE: What matters most to you?                                       |
|                                                                               |
|   Inference latency critical?                                                 |
|   +-- YES:  Must be zero overhead?                                            |
|   |         +-- YES: Use LoRA (merge after training) or (IA)³ (merge-able)    |
|   |         +-- NO:  (IA)³ (<1% overhead) or LoRA                             |
|   +-- NO: Continue...                                                         |
|                                                                               |
|   Parameter budget extremely tight? (< 1M params, tiny storage)               |
|   +-- YES: Use (IA)³  (~139K params for 1B model)                             |
|   +-- NO: Continue...                                                         |
|                                                                               |
|   Large domain shift? (English ? medical, general ? code, etc.)              |
|   +-- YES: Consider Bottleneck Adapters (non-linearity helps capture          |
|   |        complex transformations) or LoRA with larger rank                  |
|   +-- NO: LoRA r=8-16 is the default best choice                               |
|                                                                               |
|   Serving many tasks from one base model?                                     |
|   +-- YES: LoRA (adapter swap) or (IA)³ (tiny files, easy swap)               |
|   +-- NO: Any method works                                                    |
|                                                                               |
|   Simple recommendation for most cases:                                       |
|                                                                               |
|   LoRA (r=16) > (IA)³ > Bottleneck Adapters                                   |
|                                                                               |
|   LoRA wins in practice for general fine-tuning.                              |
|   (IA)³ wins when you need extreme efficiency.                                |
|   Bottleneck wins when expressiveness > inference cost.                       |
|                                                                               |
+-------------------------------------------------------------------------------+
```

One Final Analogy

Think of adapting a musician to play a new genre:

Full Fine-Tuning: Retrain them completely — they learn the new genre but may forget old skills.

LoRA: Give them a cheat sheet that adds parallel notes to their existing playing.
The cheat sheet can be folded into their memory (merged) and they play naturally.

Bottleneck Adapters: Give them a small filter pedal that runs their sound through a compression/expansion effect after they play.
The pedal always stays in the signal chain — you can't remove it without also removing the effect.

(IA)³: Give them a tiny set of dials that turn certain aspects of their playing up or down — more treble here, less vibrato there. Minimal gear, just learned adjustments. The dials can be permanently set and removed once calibrated.

ADAPTER MERGING — COMBINING MULTIPLE ADAPTERS

Merging Multiple LoRA Adapters

When you have trained separate adapters for different capabilities, you can combine them:

Linear Merge (simplest):

$W\_merged = W_0 + \lambda_1 \cdot B_1A_1 + \lambda_2 \cdot B_2A_2 + ...$

Where λ_1, λ_2 are weighting coefficients (how much of each adapter to blend).
Simple weighted average. Works when tasks are related and adapters don't conflict.

TIES Merging (Trim, Elect Sign & Merge):

Step 1: Trim — remove small-magnitude changes (noise)
Step 2: Elect Sign — for each parameter, pick the majority sign direction
Step 3: Merge — average only the values that agree on direction

More robust than linear merge. Handles conflicting adapters better.

DARE (Drop And REscale):

Randomly drop a fraction of adapter parameters (set to zero),
then rescale the remaining ones to compensate.

Combined with TIES or linear merge for better generalization.
Think of it like dropout but applied to the adapter delta weights.

```
ADAPTER MERGING EXAMPLE

Base LLaMA-7B
    + Medical QA adapter     (?=0.5)
    + Code generation adapter (?=0.3)
    + Summarization adapter  (?=0.2)
    --------------------------------
    = Single merged model that can do all three
      (quality depends on task compatibility)

No guarantee of quality — conflicting tasks may degrade each other.
Always evaluate merged models carefully.
```

WHEN TO USE WHICH PEFT METHOD

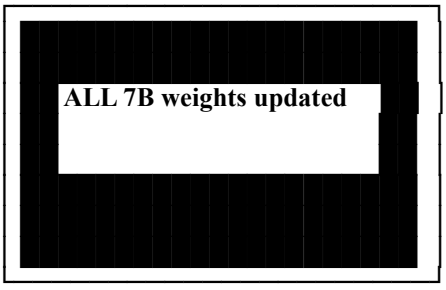
DECISION TREE — CHOOSING A PEFT METHOD

```
Start here: Do you have a GPU?
  |
  +-- No  ?  Use API-based fine-tuning (OpenAI, Anthropic, etc.)
  |
  +-- Yes ?  How much VRAM?
        |
        +-- 8-16 GB   ? QLoRA (7B model, 4-bit)
        |                 or Prompt Tuning if task is simple
        |
        +-- 24 GB     ? QLoRA (7B-13B) or LoRA (7B)
        |
        +-- 48 GB     ? LoRA (7B-13B) or QLoRA (70B)
        |
        +-- 80+ GB    ? LoRA (up to 70B) or Full FT (7B)
```

Model	Budget GPU	Best Method	Approx VRAM
7B	RTX 3090 (24GB)	QLoRA	~10 GB
7B	A6000 (48GB)	LoRA	~18 GB
13B	RTX 3090 (24GB)	QLoRA	~16 GB
13B	A100 (80GB)	LoRA	~30 GB
70B	A100 (80GB)	QLoRA	~40 GB

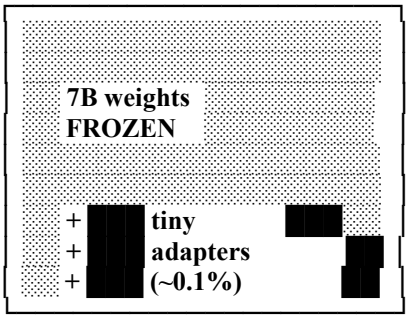
SUMMARY MENTAL MODEL

FULL FINE-TUNING



Memory: 94-114 GB
Storage: 14 GB per task
Forgetting: High risk
Speed: Slow
Quality: Maximum
Use when: Unlimited compute

PEFT (LoRA / QLoRA)



Memory: 8-24 GB
Storage: 33 MB per task
Forgetting: None (base frozen)
Speed: Fast
Quality: 95-99% of full FT
Use when: Everything else

LORA – Low Rank Adaptation:

PEFT's core insight: **you don't need to update all the parameters.**

Research showed that the "intrinsic dimensionality" of fine-tuning is low — meaning the weight changes needed to adapt a model to a new task live in a much smaller subspace than the full parameter space. You can capture most of the adaptation with a tiny fraction of trainable parameters.

Think of it this way: a pre-trained model is a massive building. Full fine-tuning demolishes and rebuilds every room. PEFT just redecorates specific rooms — same structural integrity, fraction of the cost.

The Core Idea: LoRA (Low-Rank Adaptation of Large Language Models, Hu et al. 2021) is built on one key insight:

****The weight changes during fine-tuning have low intrinsic rank.****

What does that mean? When you fine-tune a model, the difference between the original weights and the fine-tuned weights (ΔW) can be well-approximated by a low-rank matrix — meaning it can be decomposed into two much smaller matrices multiplied together.

Instead of learning a full ΔW (which is huge), LoRA learns two small matrices A and B
whose product approximates ΔW.

The Problem LoRA Is Solving:

When you fine-tune a large model the traditional way (full fine-tuning), every weight in the model gets updated. For a single weight matrix W_0 of shape [4096 × 4096], the change you're learning — called ΔW — is itself a full [4096 × 4096] matrix.

That's 16.7 million parameters just for one layer.
Across a 7B parameter model, the gradients and optimizer states (Adam stores two states per parameter) push memory requirements to 94–114 GB. That's inaccessible for most people.

LoRA's starting insight is this: you don't need all that.
Research showed that when you actually analyze what ΔW looks like after fine-tuning, it's almost always low-rank. Most of those 16.7 million changes are redundant — the real adaptation lives in a much smaller subspace.
The matrix is fat, but the information inside it is slim.

The Math — Step by Step

In full fine-tuning:

For a weight matrix W_0 of shape [d_out × d_in] (e.g., [4096 × 4096]):

$W_{new} = W_0 + \Delta W$

ΔW is also [4096 × 4096] = 16,777,216 parameters to learn. That's the problem.

Fine Tuning

In LoRA: The Core Idea

Decompose ΔW Into Two Tiny Matrices

Instead of learning the full ΔW, decompose it - LoRA approximates it as the product of two small matrices:

When you fine-tune a neural network layer, you normally update its weight matrix:

$$W_{new} = W_{original} + \Delta W$$

Where:

W_original → **pretrained weights (frozen in LoRA)**
ΔW → **the learned update during fine-tuning**

Instead of learning the full ΔW, decompose it:

$$\Delta W \approx B \times A$$

Where:

ΔW = Change in the original weight matrix
A is [r × d_in] → e.g., [8 × 4096] = 32,768 parameters
B is [d_out × r] → e.g., [4096 × 8] = 32,768 parameters

In LoRA, we do not learn a full ΔW matrix directly.

Instead we parameterize it as:

$$\Delta W = B \cdot A$$

So fine-tuning becomes:

$$W_{new} = W_{original} + B \times A$$

For the same [4096 × 4096] example with r = 8:

A : [8 × 4096] = 32,768 params
B : [4096 × 8] = 32,768 params

Total: 65,536 vs. 16,777,216 — a 256× reduction for one weight matrix.

Total LoRA parameters **: 32,768 + 32,768 = 65,536**
vs. full **ΔW** **: 16,777,216**

That's a 256× reduction for this single weight matrix.
r is the "rank" — the key hyperparameter. Typical values: 4, 8, 16, 32, 64.

The big question is: **why does this work ?**

Because the hypothesis (backed empirically) is that the "task adaptation" signal can be captured in just a few dimensions of change.
You don't need a full-rank matrix to express "now be a medical assistant instead of a general assistant."

In **LoRA**, we do **not** learn a full ΔW matrix directly.

Instead we parameterize it as:

$$\Delta W = BA$$

So fine-tuning becomes:

$$W_{new} = W_{original} + B \times A$$

The Forward Pass :

During a forward pass through any targeted layer, the computation becomes:

$h = W_0x + (\alpha/r) \cdot BAx$

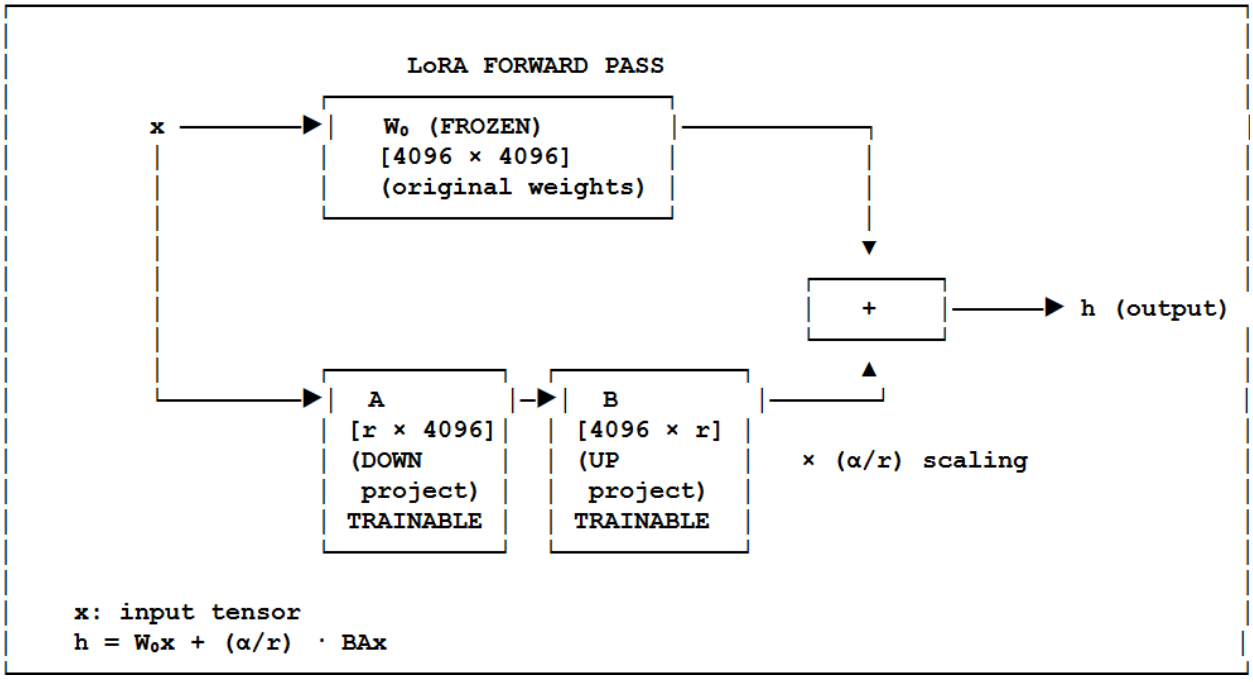
- W_0x — the frozen original weight doing its job, unchanged
- BAx — the small adapter's contribution, running in parallel
- α/r — a scaling factor (more on this below)

Both paths process the same input x simultaneously. Their outputs are just added together.
This is the key structural difference from Adapters (which insert modules in series, adding latency).
LoRA runs in parallel — and can be merged away completely at inference.

The forward pass becomes:

$h = W_0x + (B \times A)x$ (where x is the input)

$= W_0x + BAx$ (just matrix multiplication)



Initialization — Why It Matters:

LoRA uses a very specific initialization strategy:

Matrix A: initialized with random Gaussian (small random values)
Matrix B: initialized to ALL ZEROS

Why zeros for B?

At the start of training: $\Delta W = B \times A = 0 \times A = 0$

This means the model starts EXACTLY where the pre-trained model left off.
The LoRA adapters add zero contribution initially, so training begins from a known-good state — the pre-trained model's behavior is perfectly preserved at step 0.

As training progresses, B learns non-zero values and the adapters gradually steer the model toward the new task.

If both A and B were randomly initialized, the model would start with random perturbations to every adapted layer — potentially destroying pre-trained knowledge immediately.

The Scaling Factor α (Alpha):

Alpha (α) is a scaling constant that acts like a volume knob on the adapter's contribution. The actual scaling applied is α/r:

LoRA includes a scaling factor α (alpha) that controls how much the adapters contribute to the output:

$$h = W_0 x + (\alpha/r) \cdot BAx$$

α is a constant (set before training, typical values: 8, 16, 32)
r is the rank

The ratio α/r controls the "magnitude" of the adaptation.

Think of it like a volume knob:
α/r too low → adapters contribute too little, model barely adapts
α/r too high → adapters dominate, can destabilize training

Common practice:
Set α = r → effective scaling of 1.0 (neutral)
Set α = 2r → effective scaling of 2.0 (stronger adaptation)
Set α = r/2 → effective scaling of 0.5 (gentler adaptation)

Rule of thumb: α = r or α = 2r works well for most tasks.
When you increase r, increase α proportionally to maintain similar scaling.

The reason α exists separately from r is so you can change r without needing to recalibrate the magnitude of updates.
You fix α and tune r.

Which Layers Get LoRA? — Target Modules

Which Layers to Target

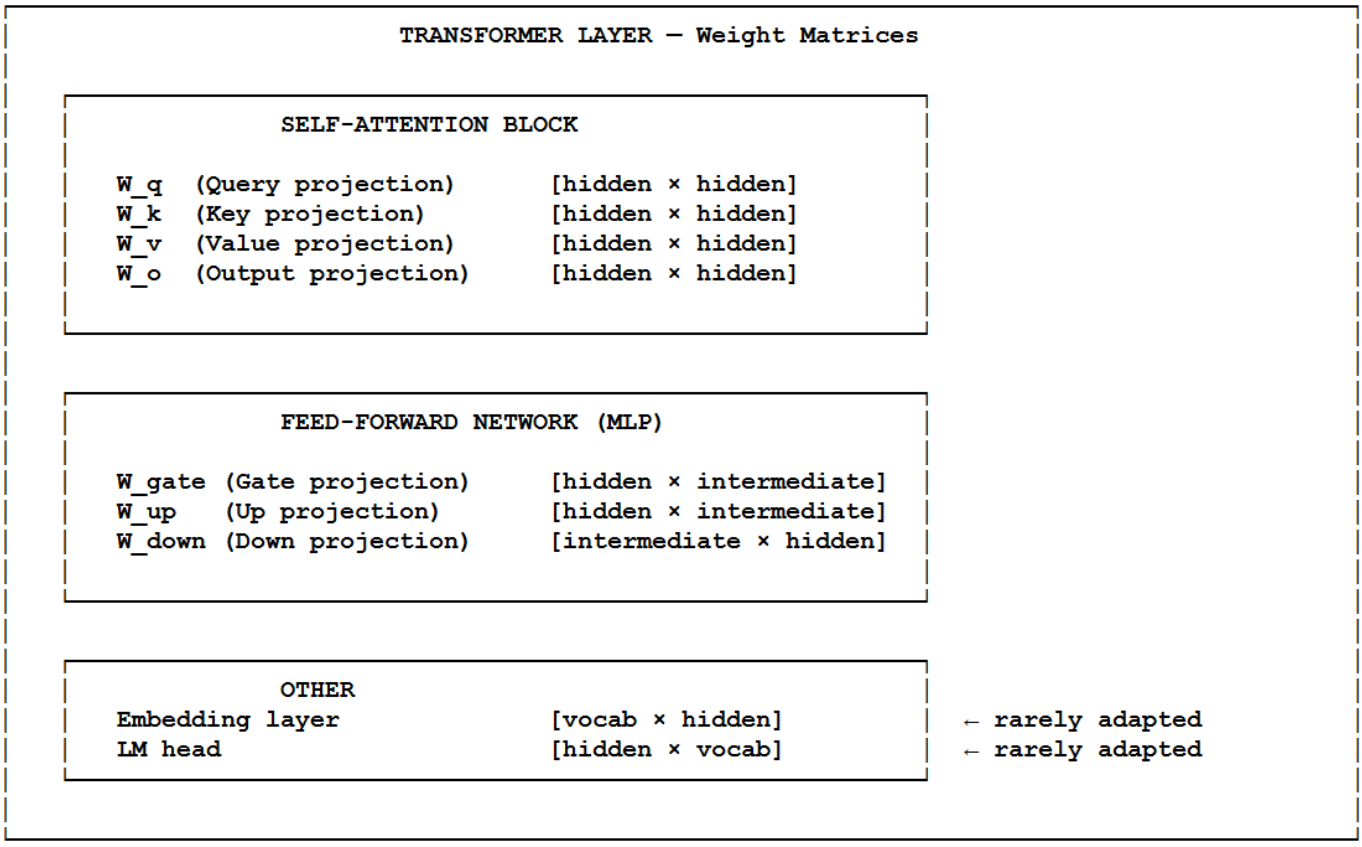
You choose. The original LoRA paper only targeted W_q and W_v in the attention blocks. Modern practice has evolved:

Minimal : W_q, W_v only
Standard : W_q, W_k, W_v, W_o (all attention projections)
Aggressive : all linear layers including the MLP's gate, up, and down projections

For a 32-layer model with standard 4 targets and r=8: that's 256 small matrices,
~8.4 million trainable parameters — roughly 0.12% of the 7B model.

Yet performance on fine-tuning benchmarks is often competitive with full fine-tuning.

You don't have to apply LoRA to every weight matrix. In a transformer, the key weight matrices are:



Common LoRA target configurations:**
Minimal (original paper) : W_q, W_v only
Standard (most popular) : W_q, W_k, W_v, W_o
Aggressive (best performance) : W_q, W_k, W_v, W_o, W_gate, W_up, W_down (all linear layers)

More targets = more trainable parameters = better performance but more memory.

In HuggingFace PEFT config, this looks like:

```
target_modules = ["q_proj", "v_proj"]                          # minimal
target_modules = ["q_proj", "k_proj", "v_proj", "o_proj"]      # standard
target_modules = "all-linear"                                  # aggressive
```

Each target module in EACH transformer layer gets its own A and B matrices.

For a 32-layer model with 4 attention targets and rank 8:

32 layers × 4 targets × 2 matrices (A, B) × (8 × 4096) = ~8.4M parameters
vs. 7B total = ~0.12% of the model

The Rank (r) — How to Choose It

r = 1: **Extremely compressed. Each adapter captures only 1 direction of variation. Works for very simple tasks or when data is tiny.**

r = 4: **Very lean. Good for straightforward classification or single-skill tasks.**

r = 8: **The default sweet spot. Works well for most instruction-tuning and chat tasks. This is where most practitioners start.**

r = 16: **Better for complex tasks requiring nuanced adaptation. Moderate memory increase.**

r = 32: **Approaching diminishing returns for many tasks. Good for very different target domains (e.g., English model → code model).**

r = 64+: **Rarely needed. If you need this much capacity, consider whether full fine-tuning or a larger rank is actually buying you anything.**

RANK vs. PERFORMANCE vs. MEMORY (for a 7B model, 4 attention targets)

Rank	Trainable Params	% of Model	Adapter Size	Quality
4	~4.2M	0.06%	~16 MB	Good
8	~8.4M	0.12%	~33 MB	Very Good ★
16	~16.8M	0.24%	~67 MB	Excellent
32	~33.5M	0.48%	~134 MB	Excellent
64	~67.1M	0.96%	~268 MB	Marginal gain

★ = recommended starting point

Note: these numbers assume LoRA on q_proj, k_proj, v_proj, o_proj only.
Adding MLP targets roughly doubles the parameter count.

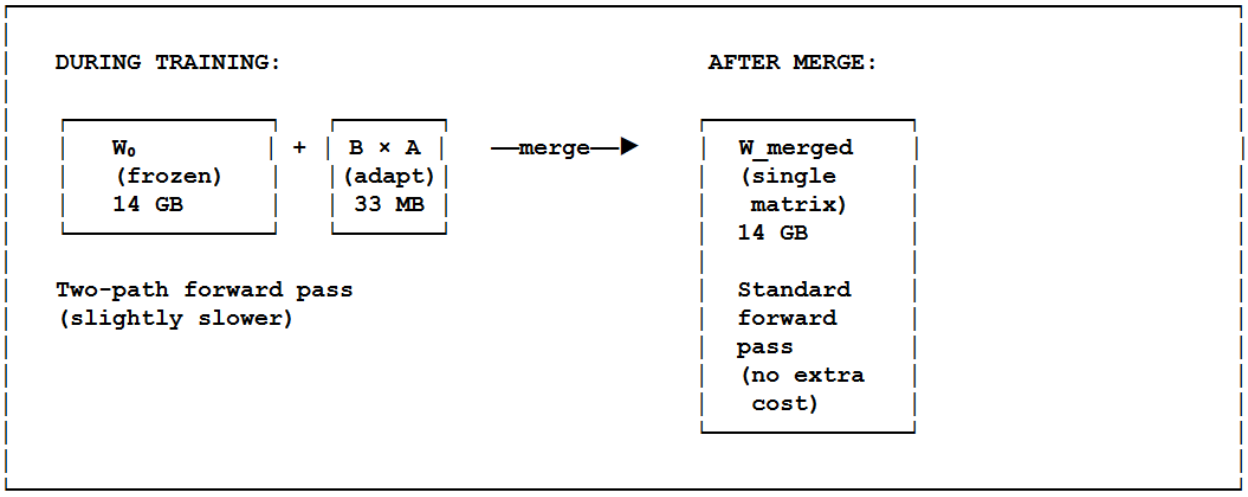
LoRA Merge — Deploying Without Overhead

A beautiful property of LoRA: after training, you can **merge** the adapters back into the base model.

$$W_{merged} = W_0 + (\alpha/r) \cdot B \times A$$

This is just matrix addition. Once merged:

- No inference overhead (no extra computation at runtime)
- No adapter files needed
- Model behaves exactly as if it had been fully fine-tuned
- The merged model is the same size and architecture as the original

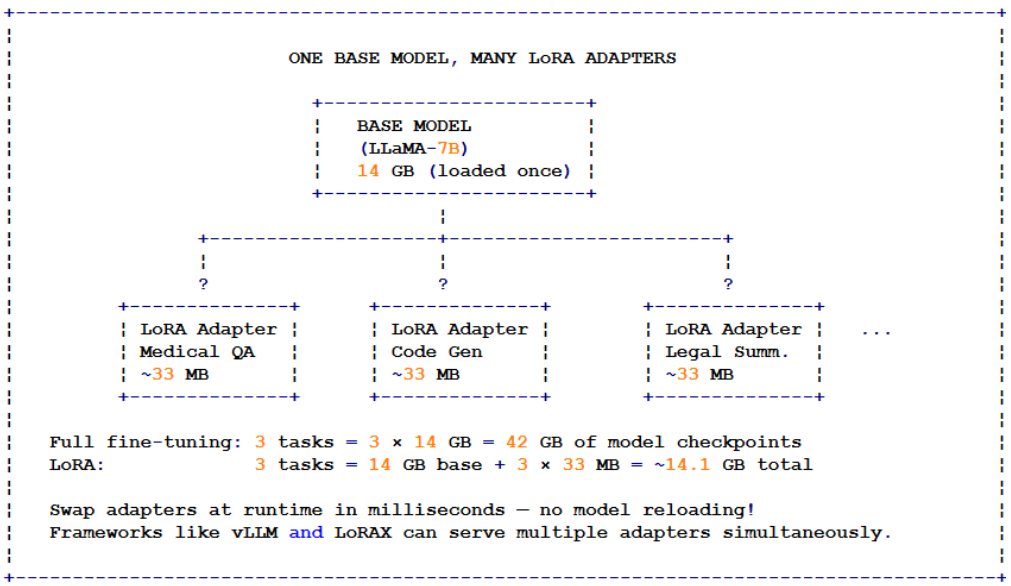


BUT — once merged, you can't "un-merge."

If you want to swap tasks, keep the adapters separate and swap them at inference time (see below).

LoRA Adapter Swapping — Multi-Task Serving

This is one of LoRA's killer features: one base model, many tasks.



Step 0: Understand What Problem LoRA Is Solving

In full fine-tuning, when you update a weight matrix W_0 of shape [4096 × 4096], you're learning a completely new matrix:

$$\mathbf{W\_new = W_0 + \Delta W}$$

ΔW is [4096 × 4096] = 16,777,216 parameters to learn for this ONE matrix.

Researchers discovered that ΔW (the change needed to adapt a model to a new task) has low intrinsic rank. That means most of those 16.7 million changes are redundant — the real adaptation lives in a much smaller subspace.

LoRA exploits this by decomposing ΔW into two small matrices whose product approximates the full update:

$$\mathbf{\Delta W \approx B \times A}$$

A : [r × 4096] (down-projection, compresses)
B : [4096 × r] (up-projection, expands)

With r=8:
A : [8 × 4096] = 32,768 params
B : [4096 × 8] = 32,768 params

Total: 65,536 params

vs. full ΔW: 16,777,216 params

That's a 256× reduction for ONE weight matrix.

Now let's walk through every step of how this actually gets implemented and trained.

Step 1: Load the Pre-Trained Model and Freeze Everything

You start with a pre-trained model — say LLaMA-2-7B. Every parameter gets frozen:

```
model = load_pretrained("meta-llama/Llama-2-7b-hf")

for param in model.parameters():
    param.requires_grad = False     # All 7 billion parameters → read-only
```

At this point the model is in inference mode. No gradients will be computed for any original weight.
This is identical to what Additive PEFT does — the difference comes in Step 2.

Load the pre-trained model. Set requires_grad = False on every single parameter.
The base model becomes read-only. No gradients will flow through it.
This is what eliminates 56 GB of optimizer states and 14 GB of gradients.

Step 2: Attach LoRA Matrices (A and B) to Target Weight Matrices

Here's where LoRA diverges from Adapters. Instead of inserting new modules between layers (in-series), LoRA attaches a small parallel bypass alongside existing weight matrices.

For each target weight matrix W_0 in the model, LoRA creates two new small matrices A and B:

Target : W_q (query projection) in Layer 0
Shape : [4096 × 4096]
Status : FROZEN (requires_grad = False)

LoRA creates:
A: [r × 4096] = [8 × 4096] → requires_grad = True (TRAINABLE)
B: [4096 × r] = [4096 × 8] → requires_grad = True (TRAINABLE)

Which weight matrices get LoRA ? You choose.

This is the target_modules configuration:

Minimal	**:**	**W_q, W_v**	**(original paper)**
Standard	**:**	**W_q, W_k, W_v, W_o**	**(most popular)**
Aggressive	**:**	**W_q, W_k, W_v, W_o, W_gate, W_up, W_down**	**(all linear layers)**

For a 32-layer model with the standard 4 attention targets and rank 8:

32 layers × 4 targets × 2 matrices (A + B) = 256 small matrices

Total trainable params:
256 matrices × (8 × 4096) = ~8.4 million

vs. 7 billion total → 0.12% of the model

For each weight matrix you choose to adapt (e.g., W_q, W_k, W_v, W_o in every attention block), create two new small matrices A and B. These are the only trainable parameters.
They sit alongside W_0 as a parallel bypass.

Step 3: Initialize A and B — This Is Critical

Initialize Carefully (This Is Critical)
A → initialized with small random Gaussian values
B → initialized to all zeros

LoRA uses a very specific initialization:
Matrix A: Random Gaussian (small random values, like normal init)
Matrix B: ALL ZEROS

Why does B start at zero? Because at training step 0:

$$\Delta W = B \times A = 0 \times A = 0 \quad \textbf{(zero matrix)}$$

So: $W_{effective} = W_0 + \Delta W = W_0 + 0 = W_0$

A is initialized with small random Gaussian values. B is initialized to all zeros.
This ensures $\Delta W = B \times A = 0$ at step 0, so the model starts exactly as the pre-trained checkpoint with no disruption to pre-trained knowledge. See Diagram 3 below for the full visual breakdown.

Explanation 2:
The model starts exactly as the pre-trained model. The LoRA adapters contribute nothing initially.
Training begins from a known-good state — the pre-trained model's behavior is perfectly preserved at step 0.
If both A and B were randomly initialized, B × A would produce random noise added to every weight matrix, potentially destroying the pre-trained knowledge immediately.
As training progresses, B learns non-zero values and the product B × A gradually steers the model toward the new task.

Explanation 3:
At training step zero: $\Delta W = B \times A = 0 \times A = 0$. So the effective weight is exactly $W_0 + 0 = W_0$.
The model starts identically to the pre-trained checkpoint. No pre-trained knowledge is disrupted.
As training progresses, B learns non-zero values and the adapters gradually steer behavior.
If both A and B were randomly initialized, you'd be injecting random noise into every layer from the start — potentially catastrophic.

Step 4: Prepare the Data (Identical to Full Fine-Tuning)

The data pipeline is completely unchanged from full fine-tuning.
LoRA doesn't change what goes into the model, only what happens inside it.

4a. Raw data on disk:

train.jsonl:

{"instruction": "Classify the sentiment", "input": "This movie was breathtaking", "output": "positive"}
{"instruction": "Classify the sentiment", "input": "Terrible waste of time", "output": "negative"}
{"instruction": "Translate to French", "input": "The cat sat on the mat", "output": "Le chat était assis sur le tapis"}

Just text. Nothing has happened yet.

4b. Template formatting (text → structured text):

The DataLoader applies a chat template to combine the fields. This is model-specific:

LLaMA format: **"<s>[INST] Classify the sentiment: This movie was breathtaking [/INST] positive</s>"**
ChatML format: **"<|im_start|>user\nClassify the sentiment: This movie was breathtaking<|im_end|>\n<|im_start|>assistant\npositive<|im_end|>"**

Still just text strings.
Using the wrong template degrades performance because the model was pre-trained expecting a specific format.

4c. Tokenization (text → integer IDs):

"<s>[INST] Classify the sentiment: This movie was breathtaking [/INST] positive</s>"
↓ tokenizer
[1, 518, 25580, 29962, 4134, 1598, 278, 19688, 29901, 910, 14064, 471, 4800, 28107, 518, 29914, 25580, 29962, 6374, 2]

That's 20 integer IDs. No vectors yet.

4d. Create labels and loss mask:

Input IDs: [1, 518, 25580, 29962, 4134, 1598, 278, 19688, 29901, 910, 14064, 471, 4800, 28107, 518, 29914, 25580, 29962, 6374, 2]

Labels: [-100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, 6374, 2]

^^

instruction/input tokens: IGNORED by loss output: GRADED

-100 tells PyTorch's CrossEntropyLoss to skip that position.
The model sees the full sequence during forward pass but only gets graded on the response tokens ("positive").

4e. Padding and attention mask:

Different examples have different lengths. GPUs need rectangular tensors, so shorter examples get padded:

Before padding:
Example 1: [1, 518, 25580, ..., 6374, 2] → 20 tokens
Example 2: [1, 518, 25580, ..., 11690, 2] → 29 tokens (longest)

After padding to length 29:
Example 1: [1, 518, 25580, ..., 6374, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0] → 29 tokens
Example 2: [1, 518, 25580, ..., 11690, 2] → 29 tokens

Attention mask (1 = real, 0 = padding):
Example 1: [1, 1, 1, ..., 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Example 2: [1, 1, 1, ..., 1, 1]

4f. Collate into tensors and move to GPU:

```
batch =
{
  "input_ids":      tensor  [batch_size, seq_len]      e.g. [4, 29]
  "attention_mask": tensor  [batch_size, seq_len]      e.g. [4, 29]
  "labels":         tensor  [batch_size, seq_len]      e.g. [4, 29]
}
```

This entire data pipeline (4a through 4f) is identical for full fine-tuning, LoRA, QLoRA, Adapters — all of them. The data doesn't know or care what training method you're using. The difference is entirely in what happens next.

Step 5: Forward Pass — Where LoRA Changes Things

This is where LoRA diverges. In full fine-tuning, the forward pass through a weight matrix is:

$$h = W_0 \times x \qquad \text{(standard matrix multiplication)}$$

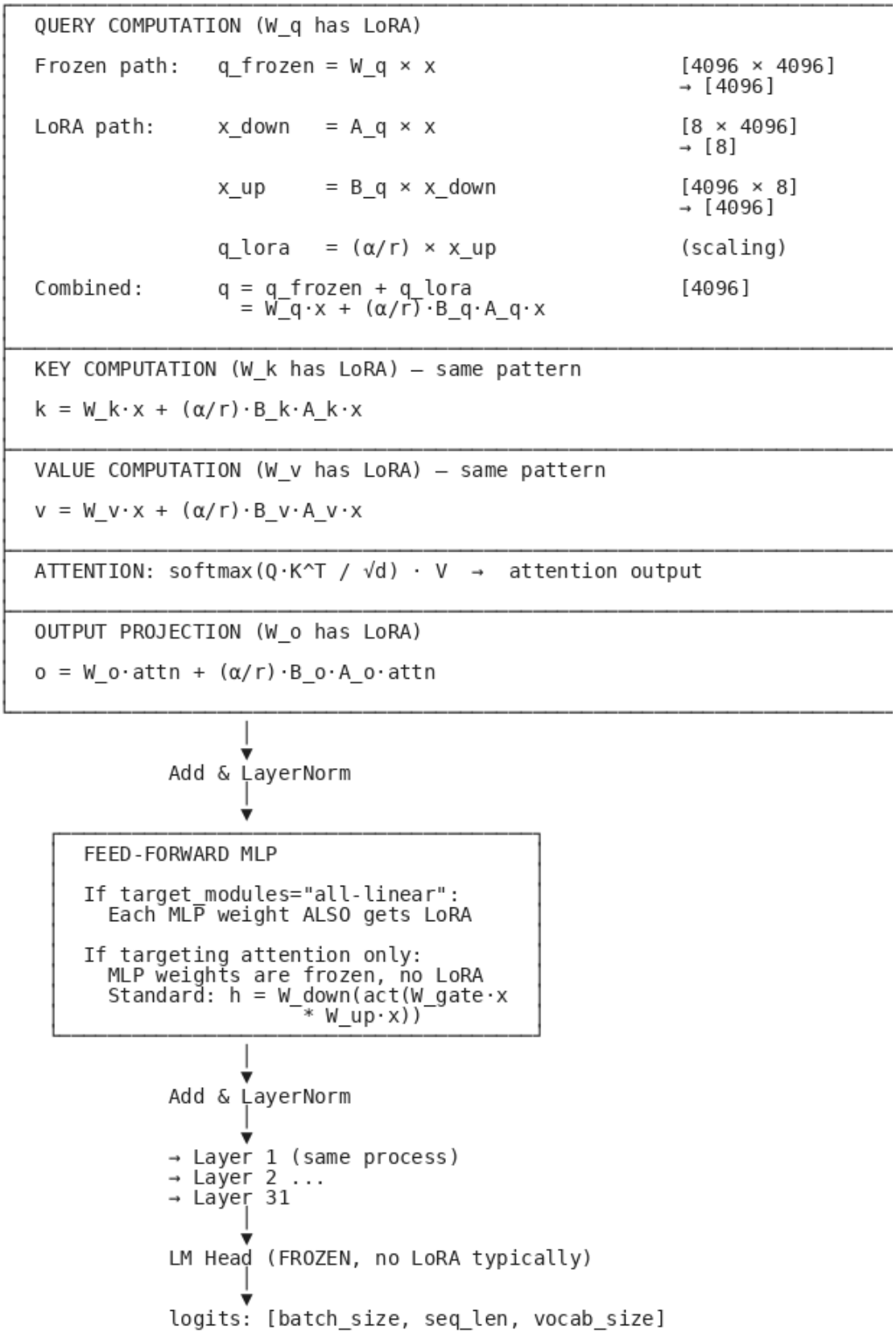
With LoRA, every target weight matrix gets an additional parallel path:

$$h = W_0x + (\alpha/r) \cdot B(Ax)$$

- W_0x: Original frozen path: unchanged
- $(\alpha/r) \cdot B(Ax)$: LoRA bypass: x goes through A (compress), then B (expand) scaled by α/r

Let's trace one token through Layer 0's self-attention with LoRA on W_q, W_k, W_v, W_o:

Input: x = [4096] (one token's hidden state)


QUERY COMPUTATION (W_q has LoRA)
Frozen path: q_frozen = W_q × x [4096 × 4096] → [4096]
LoRA path: x_down = A_q × x [8 × 4096] → [8]
x_up = B_q × x_down [4096 × 8] → [4096]
q_lora = (α/r) × x_up (scaling)
Combined: q = q_frozen + q_lora = W_q·x + (α/r)·B_q·A_q·x [4096]
KEY COMPUTATION (W_k has LoRA) — same pattern
k = W_k·x + (α/r)·B_k·A_k·x
VALUE COMPUTATION (W_v has LoRA) — same pattern
v = W_v·x + (α/r)·B_v·A_v·x
ATTENTION: softmax(Q·K^T / √d) · V → attention output
OUTPUT PROJECTION (W_o has LoRA)
o = W_o·attn + (α/r)·B_o·A_o·attn
Add & LayerNorm
FEED-FORWARD MLP
If target_modules="all-linear":
Each MLP weight ALSO gets LoRA
If targeting attention only:
MLP weights are frozen, no LoRA
Standard: h = W_down(act(W_gate·x * W_up·x))
Add & LayerNorm
→ Layer 1 (same process)
→ Layer 2 ...
→ Layer 31
LM Head (FROZEN, no LoRA typically)
logits: [batch_size, seq_len, vocab_size]

Step 6: Loss Computation (Identical to Full Fine-Tuning)

logits = model output: [batch_size, seq_len, 32000] (32000 = vocab size)
labels = [-100, -100, ..., -100, 6374, 2] (-100 = ignore)

loss = CrossEntropyLoss(logits, labels)

Only positions where labels ≠ -100 contribute.
The model is graded ONLY on predicting the response tokens.

Nothing changes here between LoRA and full fine-tuning. Same loss function, same masking.

Step 7: Backward Pass — Where the Memory Savings Happen

This is the key step. loss.backward() triggers backpropagation.

In full fine-tuning:
Gradients computed for ALL 7B parameters:
∂Loss/∂W_q [4096 × 4096] → stored (67 MB per matrix)
∂Loss/∂W_k [4096 × 4096] → stored
∂Loss/∂W_v [4096 × 4096] → stored
∂Loss/∂W_o [4096 × 4096] → stored
∂Loss/∂W_gate [4096 × 11008] → stored
∂Loss/∂W_up [4096 × 11008] → stored
∂Loss/∂W_down [11008 × 4096] → stored

... × 32 layers = ~14 GB of gradient storage

In LoRA:
Frozen weights (W_q, W_k, etc.):
Gradients flow THROUGH them (needed for chain rule to reach LoRA matrices)
But gradients are NOT STORED (requires_grad = False)
NO memory allocated for their gradients

LoRA matrices (A and B):
∂Loss/∂A_q [8 × 4096] → stored (131 KB per matrix)
∂Loss/∂B_q [4096 × 8] → stored (131 KB per matrix)
∂Loss/∂A_k [8 × 4096] → stored
∂Loss/∂B_k [4096 × 8] → stored
... × 4 targets × 32 layers = ~33 MB of gradient storage

vs. 14 GB in full fine-tuning → 400× less gradient memory

The chain rule detail matters here.
When PyTorch computes gradients for the LoRA matrices, it needs to know how the loss changed with respect to the input of each frozen layer (so it can propagate back further). This means gradients do flow through the frozen weights — the computation happens — but the gradient tensors for the frozen weights themselves are never allocated or stored.
The frozen weights are treated as constants in the computation graph.

The data pipeline, tokenization, loss masking, and training loop are identical to full fine-tuning.
The only difference is which parameters receive gradient updates.
Gradients flow back through the output, through B, through A — and stop there. W_0 never accumulates a gradient.

Step 8: Optimizer Update — Only Adapters Move

```
optimizer = AdamW(model.parameters(), lr=2e-4)
# BUT only params with requires_grad=True are in the optimizer

optimizer.step():
    For each LoRA parameter (A and B matrices):
      momentum[param]  = β₁ × momentum  + (1-β₁) × gradient
      variance[param]  = β₂ × variance  + (1-β₂) × gradient²
      param            -= lr × momentum / (√variance + ε)
```

Adam stores 2 extra values per trainable parameter:
    8.4M LoRA params × 2 × 4 bytes (FP32) = ~67 MB optimizer states

vs. full fine-tuning:
    7B params × 2 × 4 bytes = ~56 GB optimizer states

That's an 800× reduction in optimizer memory.

Frozen parameters:
    No gradient → no momentum → no variance → no update
    They are literally unchanged from the pre-trained values

Step 9: Repeat the Loop

```
for epoch in range(num_epochs):          # typically 1-5 epochs
for batch in dataloader:
    outputs = model(batch)           # forward (frozen + LoRA)
    loss = compute_loss(outputs)     # cross-entropy
    loss.backward()                  # gradients for LoRA only
    optimizer.step()                 # update LoRA only
    optimizer.zero_grad()            # clear gradients
```

After enough batches and epochs, the A and B matrices have learned the task-specific adjustments.
The frozen base model hasn't changed at all.

Step 10: Save — Only the Adapter

After training, you save ONLY the LoRA matrices:

Saved files:

adapter_model.safetensors	~33 MB	(all A and B matrices)
adapter_config.json	~1 KB	(rank, alpha, targets, base model name)

Compare to full fine-tuning:

model.safetensors ~14 GB (entire model)

The adapter_config.json looks like:

```
{
  "base_model_name_or_path": "meta-llama/Llama-2-7b-hf",
  "r": 8,
  "lora_alpha": 16,
  "target_modules": ["q_proj", "k_proj", "v_proj", "o_proj"],
  "lora_dropout": 0.05,
  "bias": "none",
  "task_type": "CAUSAL_LM"
}
```

To use this adapter later, you load the base model (from HuggingFace Hub) and load the adapter on top. The base model is never duplicated — you just store the tiny delta.

Step 11: Deploy — Merge or Swap

You have two deployment options:

Option A: Merge the adapter into the base model

$$W_{merged} = W_0 + (\alpha/r) \cdot B \times A$$

This is just matrix addition. One-time computation.

After merging:

- No adapter files needed at runtime
- No extra computation during inference
- Model behaves as if it was fully fine-tuned
- Same size, same architecture, same speed as original
- But you can't "un-merge" — the adapter is baked in

Option B: Keep adapters separate and swap them

Load base model once (14 GB in GPU memory)
Load medical adapter (~33 MB) → do medical QA
Swap to code adapter (~33 MB) → do code generation
Swap to legal adapter (~33 MB) → do legal summarization

One base model, many tasks. Swap in milliseconds.

Full fine-tuning equivalent : 3 × 14 GB = 42 GB of checkpoints
LoRA equivalent : 14 GB base + 3 × 33 MB = ~14.1 GB total

The Scaling Factor α — What It Actually Does

Every forward pass through a LoRA target computes:

$h = W_0x + (\alpha/r) \cdot BAx$

The α/r ratio is a volume knob controlling how much the adapters contribute:

α = 16, r = 8 → α/r = 2.0 (adapters contribute 2× their raw output)
α = 8, r = 8 → α/r = 1.0 (neutral — adapters contribute as-is)
α = 8, r = 16 → α/r = 0.5 (adapters contribute half their raw output)

Why not just use the learning rate for this? Because α is a structural scaling applied at every forward pass, while learning rate controls how fast the optimizer updates.
They serve different purposes. α controls the magnitude of the adaptation signal, learning rate controls the magnitude of the gradient step.

Common practice: set α = r (neutral scaling of 1.0) or α = 2r (slightly amplified).
When you increase rank, increase α proportionally to keep the effective scaling stable.

```
The Actual Data at Each Stage — Concrete Numbers

Let me trace a single training step with concrete shapes for a

    7B model, batch_size=4, seq_len=512, rank=8, 4 attention targets:

STAGE                              SHAPE                                SIZE IN MEMORY
-----                              -----                                --------------

Raw JSONL                          text strings                         ~few KB per example

After tokenize                     [4, 512] int32                       ~8 KB

After embed (frozen)               [4, 512, 4096] BF16                  ~16 MB

At each attention layer:
  W_q frozen forward               [4096, 4096] × [4, 512, 4096]        ? [4, 512, 4096]
  A_q LoRA down                    [8, 4096] × [4, 512, 4096]           ? [4, 512, 8]
  B_q LoRA up                      [4096, 8] × [4, 512, 8]              ? [4, 512, 4096]
  Sum both paths                   [4, 512, 4096]                       (element-wise add)
  (same for K, V, O)

After all 32 layers                [4, 512, 4096] BF16                  ~16 MB

After LM Head (frozen)             [4, 512, 32000] BF16                 ~128 MB (logits)

Loss                               scalar                               4 bytes

Gradients (LoRA only):
  Per A matrix                     [8, 4096] BF16                       ~64 KB
  Per B matrix                     [4096, 8] BF16                       ~64 KB
  Total: 256 matrices                                                   ~33 MB

Optimizer states (LoRA only):
  Momentum + variance              per LoRA param, FP32                 ~67 MB

Total VRAM for this training step: ~16-24 GB (frozen weights + activations + LoRA overhead).
```

What Makes LoRA Different From Every Other PEFT Method — The One Key Property
After training, you can compute:

$$W_{merged} = W_0 + (\alpha/r) \cdot B \times A$$

This is a permanent merge. The A and B matrices disappear.
The merged weight matrix is the same shape as the original.
The model architecture is completely unchanged from the pre-trained version.

At inference time, there is zero additional computation, zero additional memory, zero additional latency.
The model doesn't know it was LoRA-trained. It's just a standard model with different weight values.

No other PEFT method has this property. Adapters stay in the forward path permanently.
Prompt-based methods need their soft tokens prepended to every input forever.
$(IA)^3$ needs its rescaling vectors applied at every forward pass.

LoRA's mergeability is why it became the dominant PEFT method and why it's the default recommendation for almost every fine-tuning scenario today.

Merge and Disappear (At Inference)

After training, you can compute:

$$W_{merged} = W_0 + (\alpha/r) \cdot B \times A$$

This produces a single weight matrix, identical in shape to the original W_0.
You discard A and B.
The merged model has zero inference overhead — it's structurally identical to the base model, just with slightly different weights. No PEFT library needed at inference.

LORA Diagram :

Textual diagrams covering every aspect of LoRA:
the math, initialization, architecture, forward/backward pass, data flow through every stage, memory layout, merging, adapter swapping, scaling factor, rank selection, and comparison with full fine-tuning and other PEFT methods.

DIAGRAM 1: THE CORE PROBLEM LoRA SOLVES

The core problem: full ΔW (16.7M params) vs LoRA decomposition B×A (65K params), with visual size comparison of the matrices

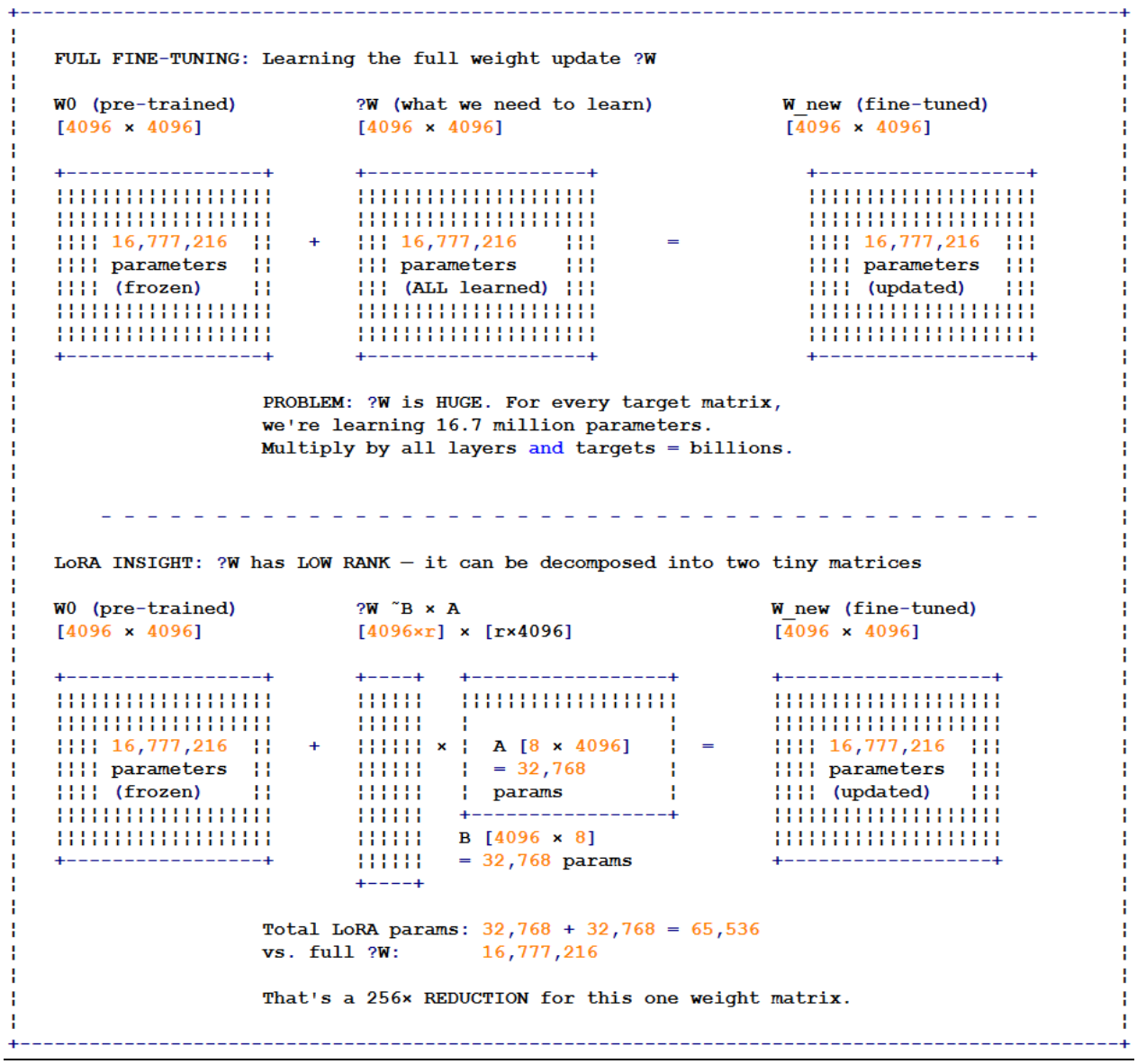


DIAGRAM 2: THE MATH — WHAT HAPPENS TO A SINGLE INPUT VECTOR

The math traced step by step: input x splits into frozen path (W_0x) and LoRA path (A compresses → B expands → scale by α/r → sum), with shapes and compute costs at each step

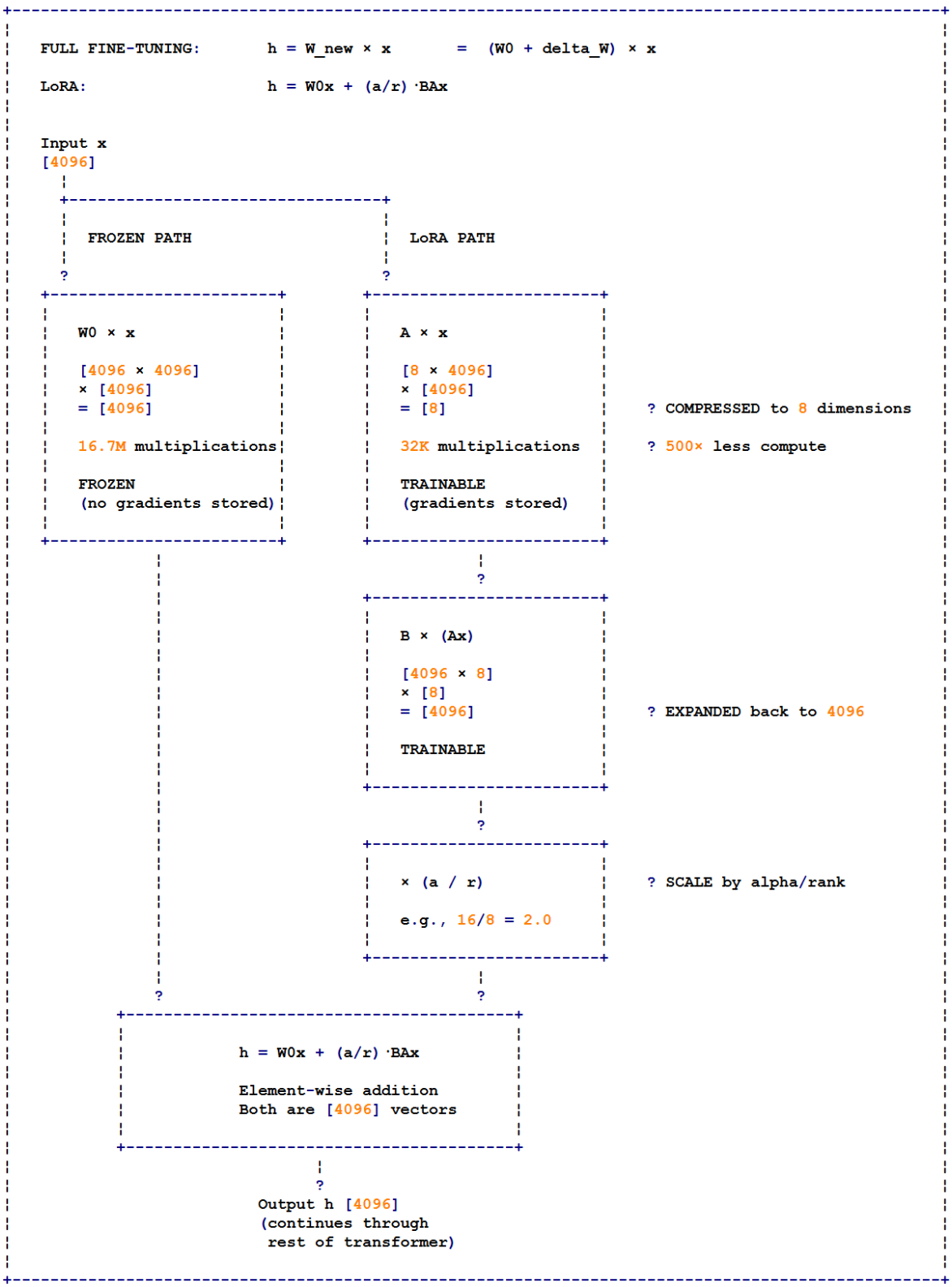


DIAGRAM 3: INITIALIZATION — WHY B STARTS AT ZERO

Initialization: why A gets random values and B gets zeros, what ΔW looks like at step 0 vs after training, and why this preserves pre-trained knowledge

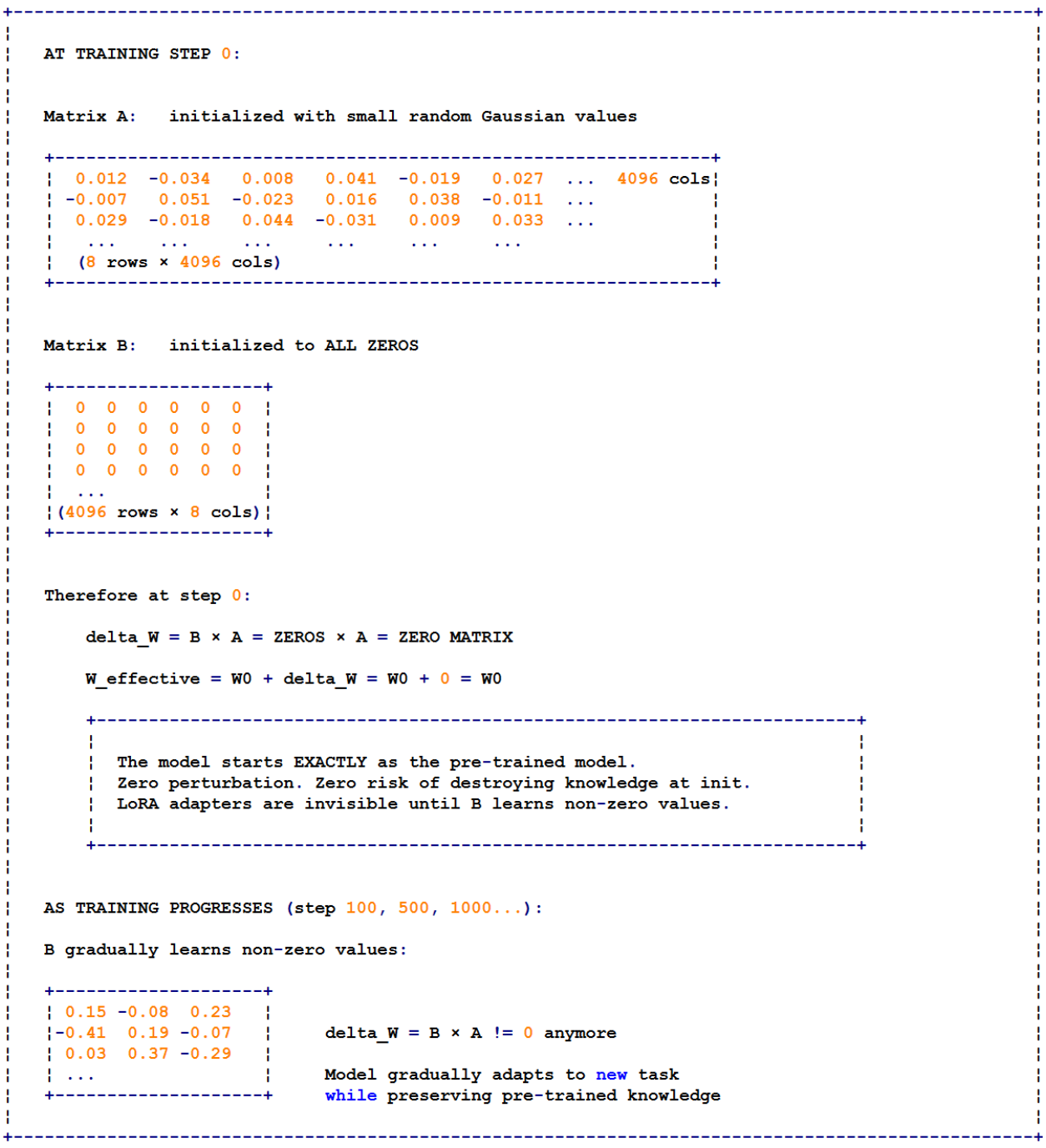


DIAGRAM 4: WHICH LAYERS GET LoRA — TARGET MODULES INSIDE A TRANSFORMER

Target modules: every weight matrix inside a transformer layer labeled with whether it gets LoRA, plus the three common configurations (minimal/standard/aggressive) with parameter counts

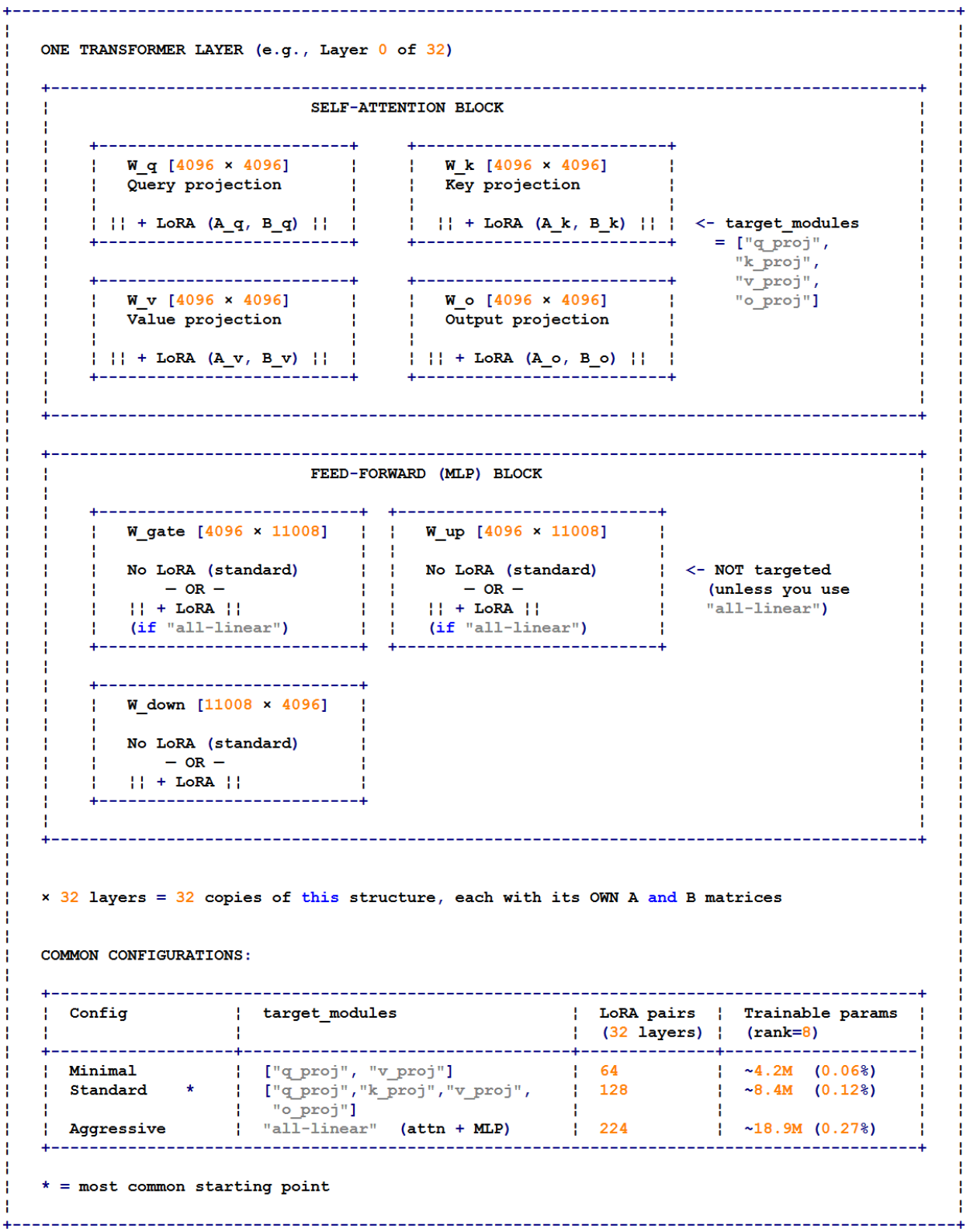


DIAGRAM 5: THE SCALING FACTOR α/r — THE VOLUME KNOB

The **α/r** scaling factor visualized as a volume knob with bar charts showing different ratios

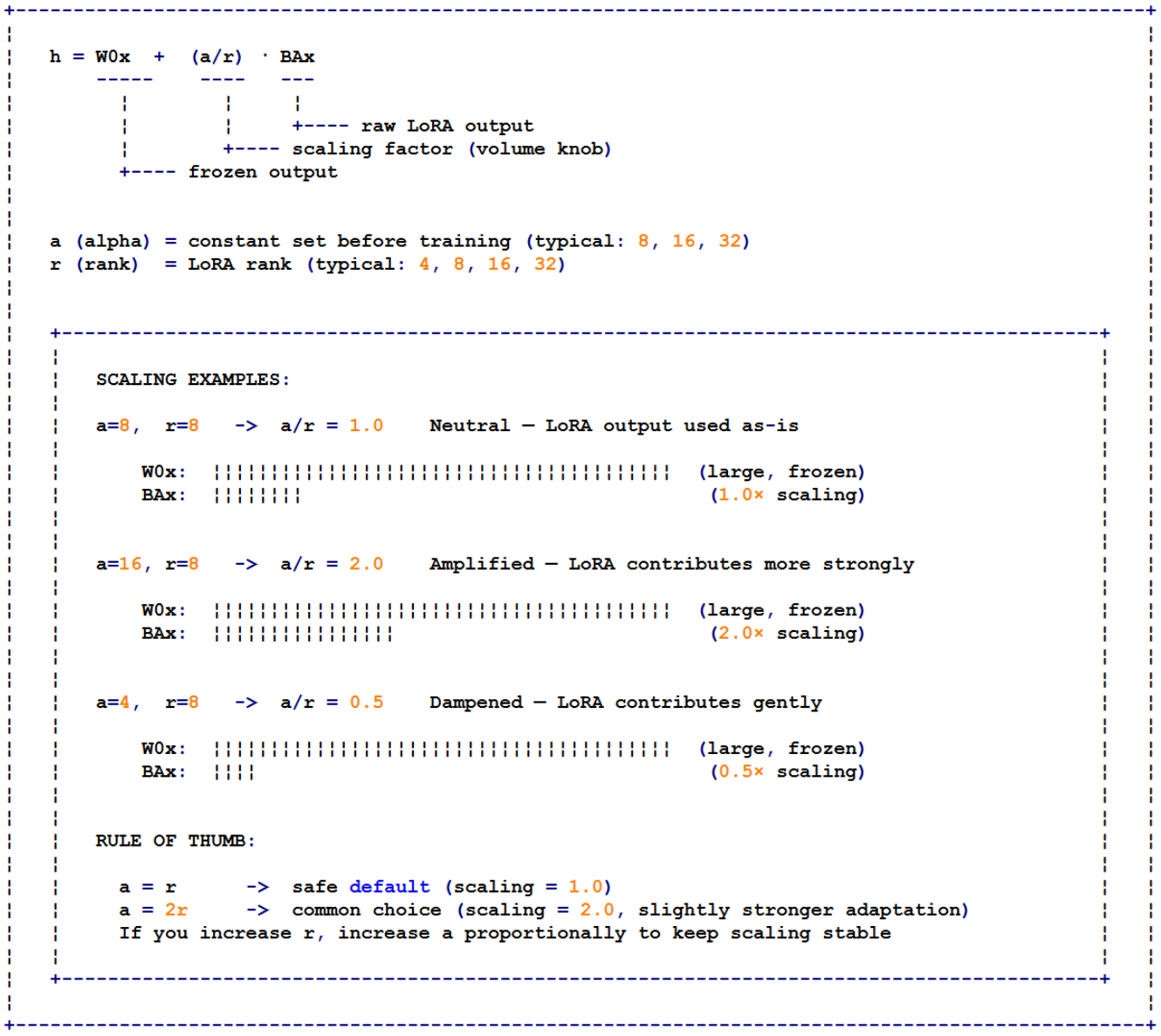


DIAGRAM 6: FULL DATA PIPELINE — FROM RAW TEXT TO LoRA WEIGHT UPDATE

Complete data pipeline:
raw JSONL → template → tokenize → labels/loss mask → padding/attention mask → GPU tensors → embedding,
with concrete examples at every stage

Fine Tuning

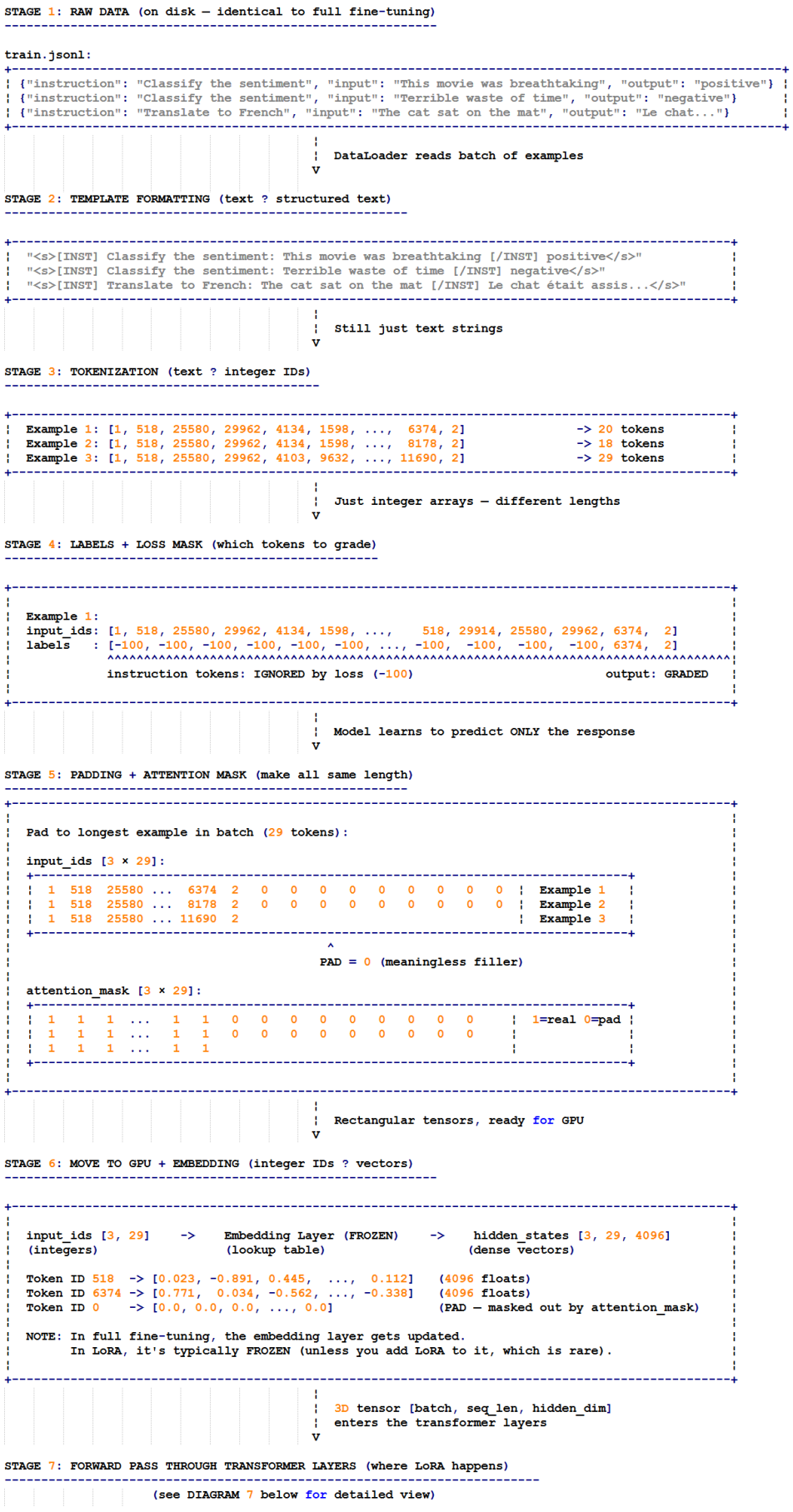

STAGE 1: RAW DATA (on disk — identical to full fine-tuning)
train.jsonl:
{"instruction": "Classify the sentiment", "input": "This movie was breathtaking", "output": "positive"}
{"instruction": "Classify the sentiment", "input": "Terrible waste of time", "output": "negative"}
{"instruction": "Translate to French", "input": "The cat sat on the mat", "output": "Le chat..."}
DataLoader reads batch of examples
STAGE 2: TEMPLATE FORMATTING (text ? structured text)
"<s>[INST] Classify the sentiment: This movie was breathtaking [/INST] positive</s>"
"<s>[INST] Classify the sentiment: Terrible waste of time [/INST] negative</s>"
"<s>[INST] Translate to French: The cat sat on the mat [/INST] Le chat était assis...</s>"
Still just text strings
STAGE 3: TOKENIZATION (text ? integer IDs)
Example 1: [1, 518, 25580, 29962, 4134, 1598, ..., 6374, 2] -> 20 tokens
Example 2: [1, 518, 25580, 29962, 4134, 1598, ..., 8178, 2] -> 18 tokens
Example 3: [1, 518, 25580, 29962, 4103, 9632, ..., 11690, 2] -> 29 tokens
Just integer arrays — different lengths
STAGE 4: LABELS + LOSS MASK (which tokens to grade)
Example 1:
input_ids: [1, 518, 25580, 29962, 4134, 1598, ..., 518, 29914, 25580, 29962, 6374, 2]
labels : [-100, -100, -100, -100, -100, -100, ..., -100, -100, -100, -100, 6374, 2]
instruction tokens: IGNORED by loss (-100)
output: GRADED
Model learns to predict ONLY the response
STAGE 5: PADDING + ATTENTION MASK (make all same length)
Pad to longest example in batch (29 tokens):
input_ids [3 × 29]:
1 518 25580 ... 6374 2 0 0 0 0 0 0 0 0 0 0
1 518 25580 ... 8178 2
1 518 25580 ... 11690 2
Example 1
Example 2
Example 3
PAD = 0 (meaningless filler)
attention_mask [3 × 29]:
1 1 1 ... 1 1 0 0 0 0 0 0 0 0 0 0
1 1 1 ... 1 1 0 0 0 0 0 0 0 0 0
1 1 1 ... 1 1
1=real 0=pad
Rectangular tensors, ready for GPU
STAGE 6: MOVE TO GPU + EMBEDDING (integer IDs ? vectors)
input_ids [3, 29] -> Embedding Layer (FROZEN) -> hidden_states [3, 29, 4096]
(integers) (lookup table) (dense vectors)
Token ID 518 -> [0.023, -0.891, 0.445, ..., 0.112] (4096 floats)
Token ID 6374 -> [0.771, 0.034, -0.562, ..., -0.338] (4096 floats)
Token ID 0 -> [0.0, 0.0, 0.0, ..., 0.0] (PAD — masked out by attention_mask)
NOTE: In full fine-tuning, the embedding layer gets updated.
In LoRA, it's typically FROZEN (unless you add LoRA to it, which is rare).
3D tensor [batch, seq_len, hidden_dim]
enters the transformer layers
STAGE 7: FORWARD PASS THROUGH TRANSFORMER LAYERS (where LoRA happens)
(see DIAGRAM 7 below for detailed view)

DIAGRAM 7: FORWARD PASS — INSIDE ONE TRANSFORMER LAYER WITH LoRA

Full forward pass through one transformer layer:
input splits to Q/K/V/O each with frozen + LoRA paths, through attention, through MLP, out to logits and loss

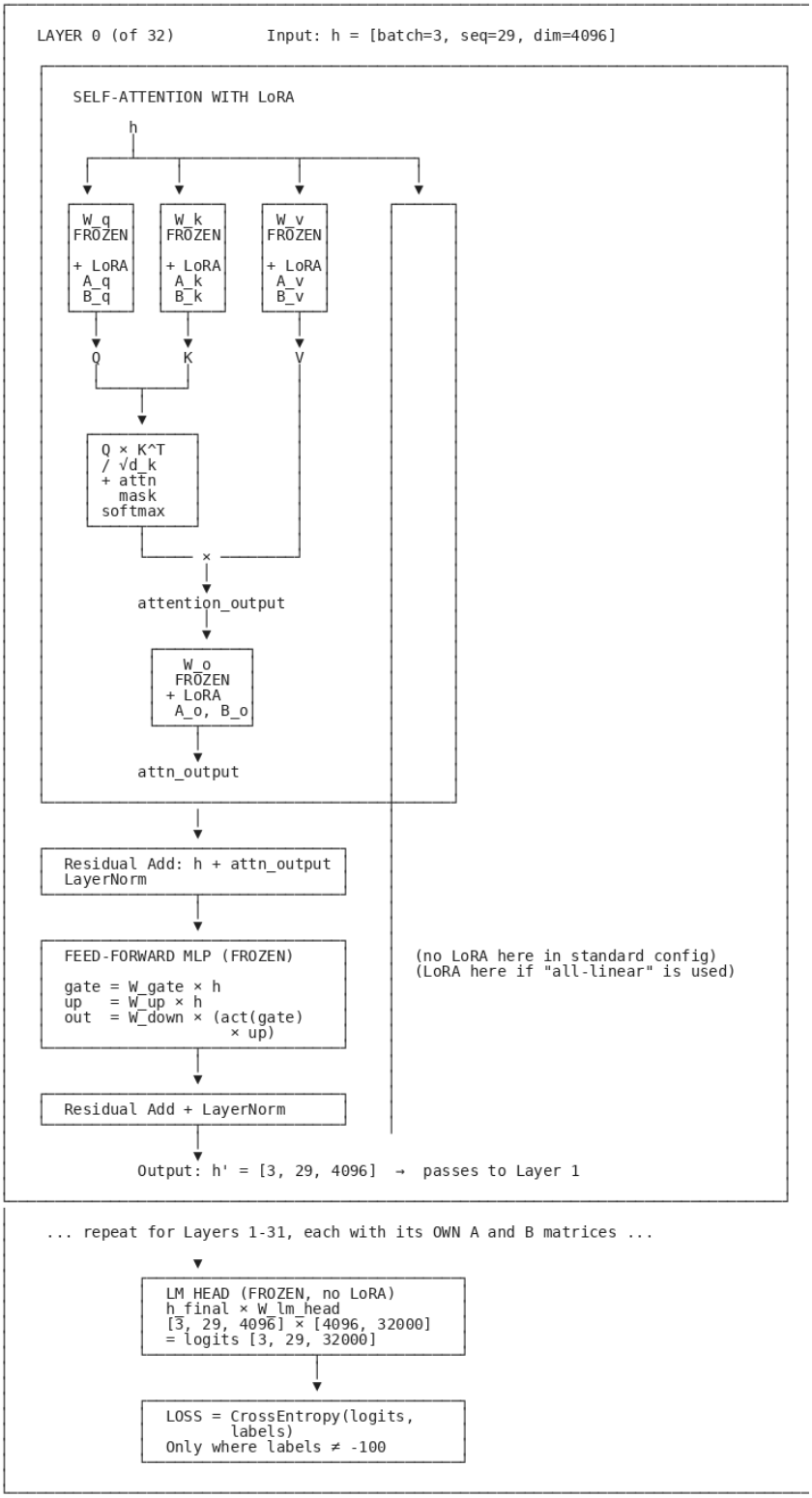


DIAGRAM 8: BACKWARD PASS — GRADIENT FLOW IN LoRA

Backward pass gradient flow: where gradients flow through (frozen layers) vs where they're stored (LoRA A and B only), with memory comparison bars

DIAGRAM 9: OPTIMIZER UPDATE — WHAT ACTUALLY GETS CHANGED

Optimizer update: Adam formula applied only to LoRA params, with optimizer state memory comparison (56 GB vs 67 MB)

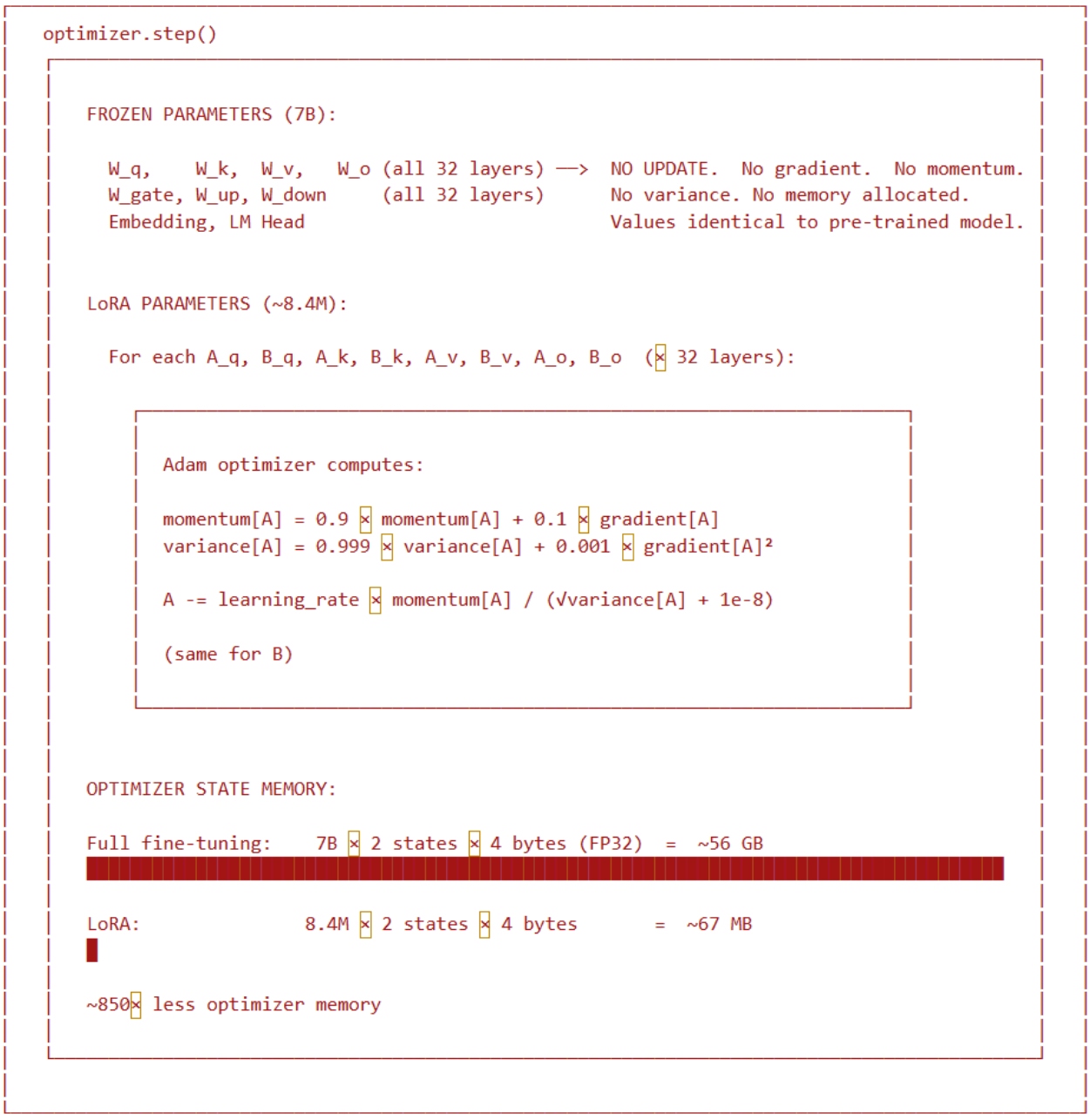


DIAGRAM 10: TOTAL MEMORY LAYOUT — LoRA vs FULL FINE-TUNING

Total memory layout comparison: Full FT vs LoRA vs QLoRA with proportional bar visualizations

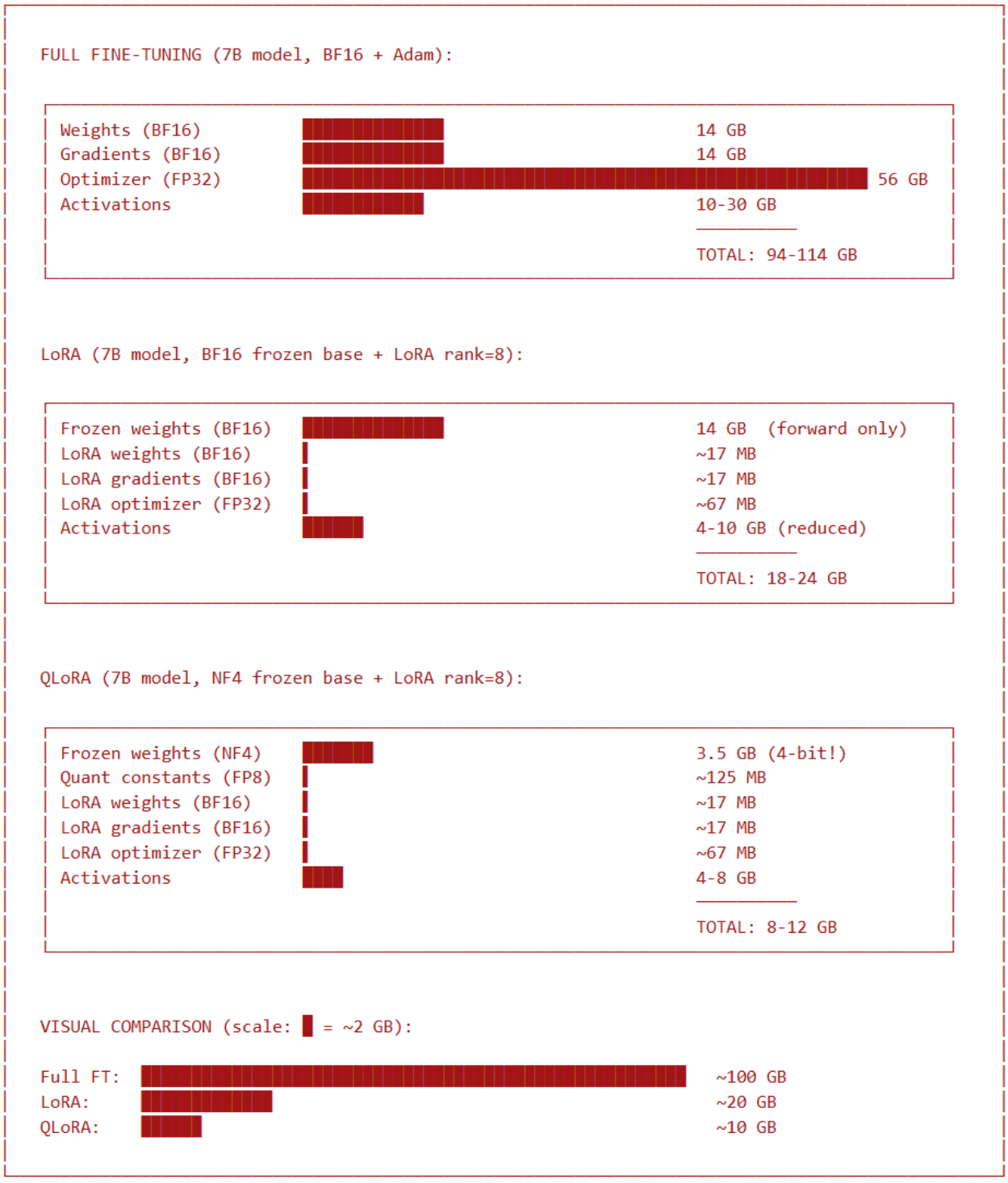
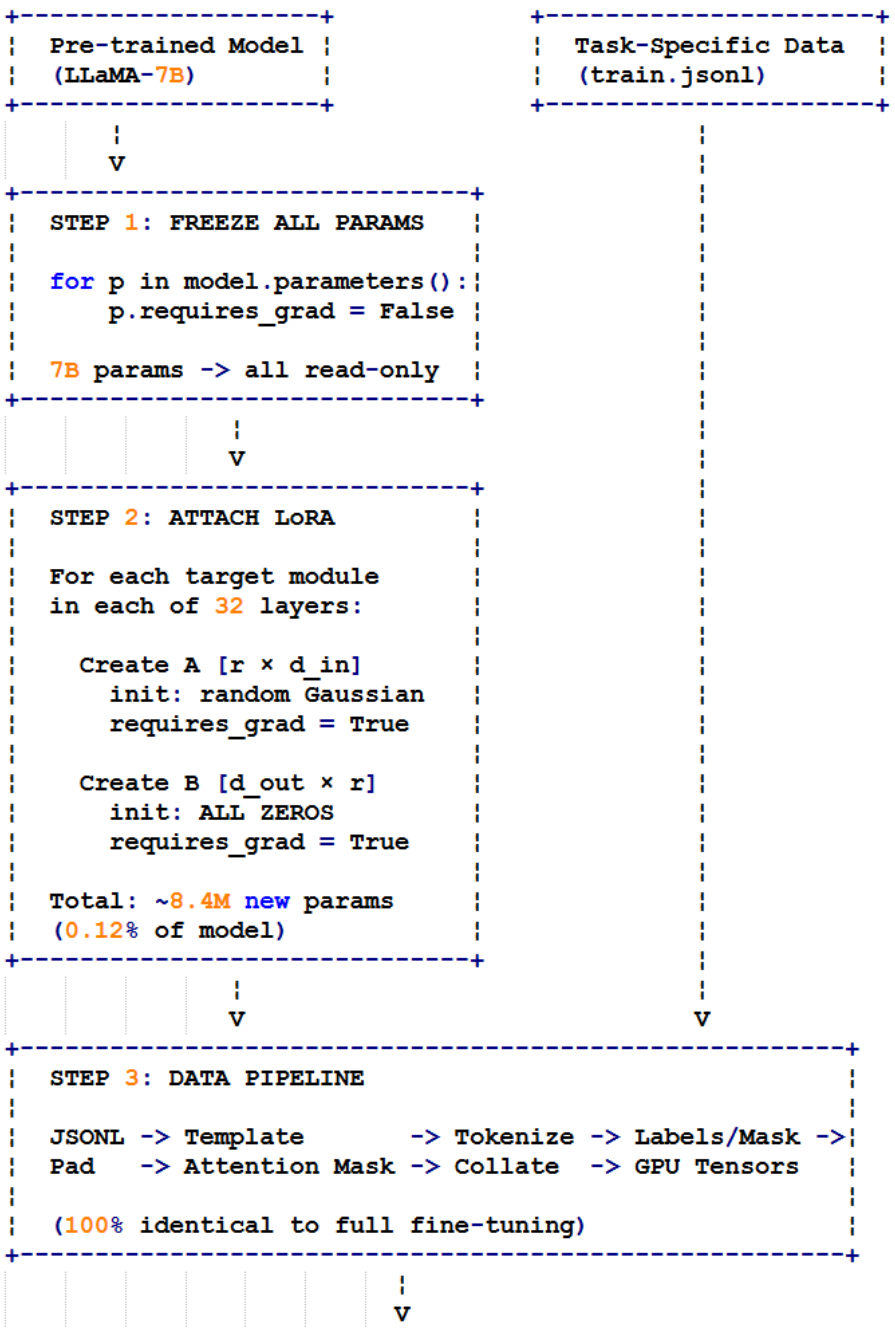


DIAGRAM 11: END-TO-END TRAINING LOOP

End-to-end training loop:

freeze -> attach LoRA -> data pipeline -> forward/backward/update cycle -> save adapter



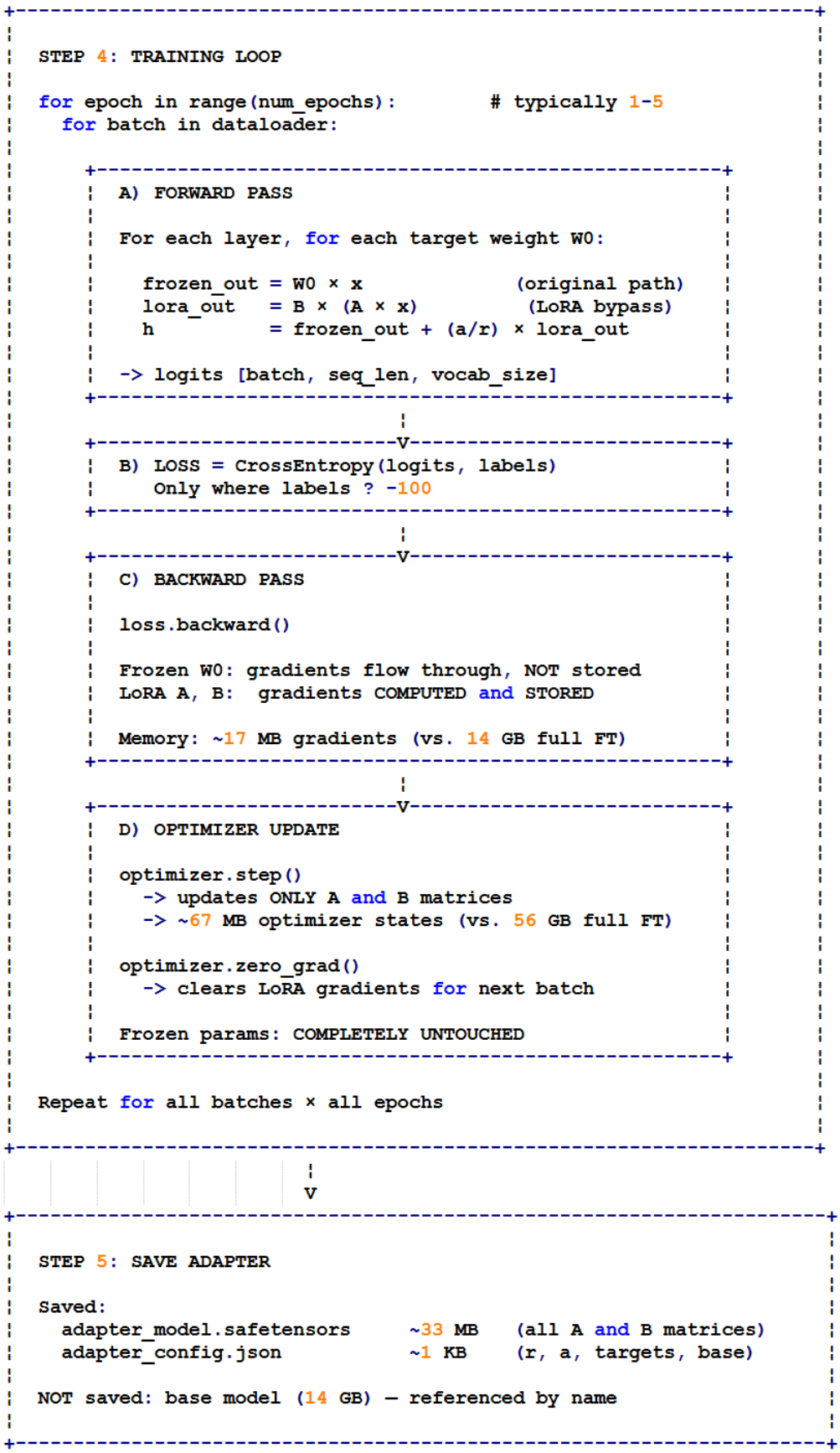
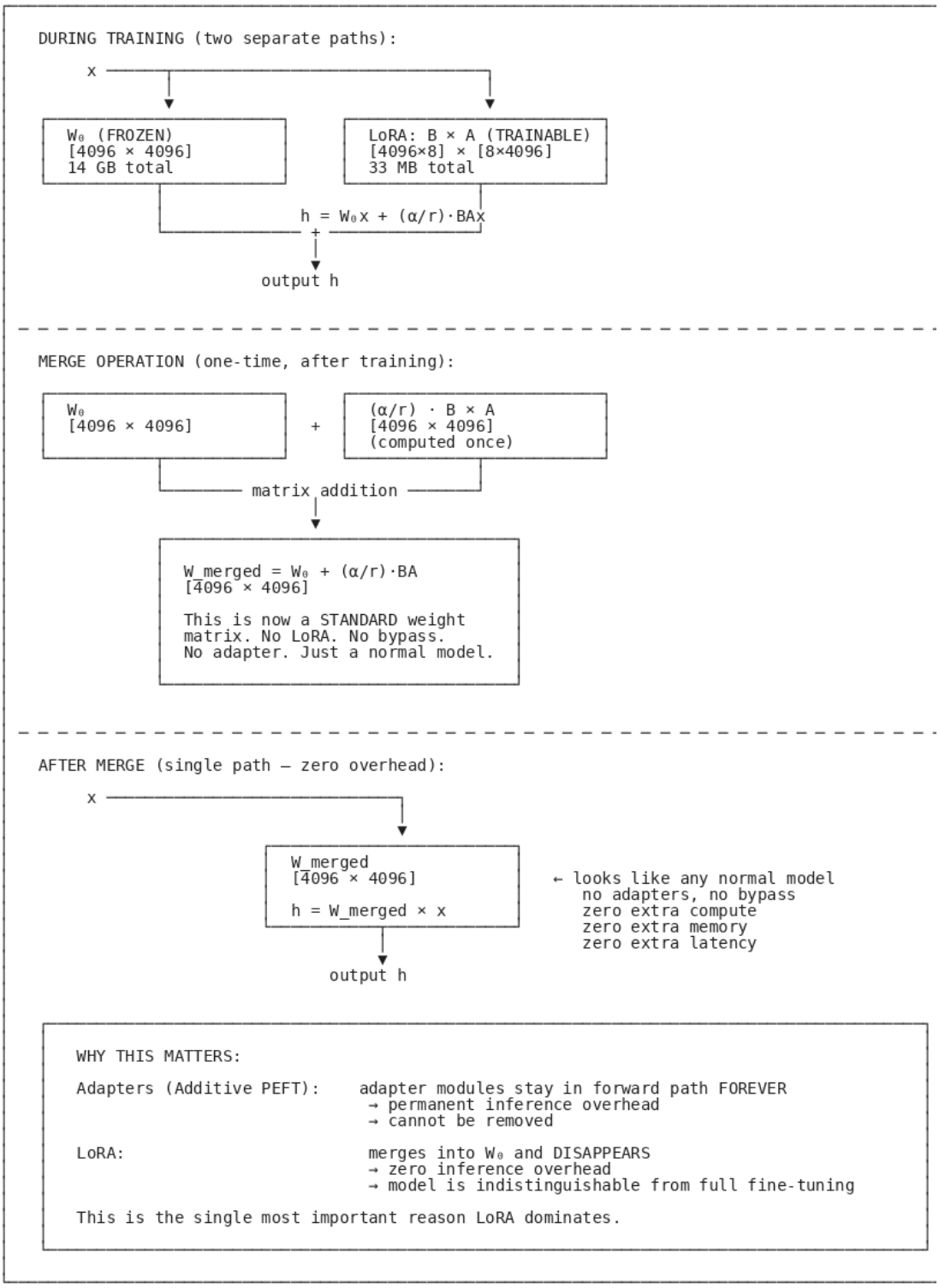

STEP 4: TRAINING LOOP
for epoch in range(num_epochs): # typically 1-5
for batch in dataloader:
A) FORWARD PASS
For each layer, for each target weight W0:
frozen_out = W0 × x (original path)
lora_out = B × (A × x) (LoRA bypass)
h = frozen_out + (a/r) × lora_out
-> logits [batch, seq_len, vocab_size]
B) LOSS = CrossEntropy(logits, labels)
Only where labels ? -100
C) BACKWARD PASS
loss.backward()
Frozen W0: gradients flow through, NOT stored
LoRA A, B: gradients COMPUTED and STORED
Memory: ~17 MB gradients (vs. 14 GB full FT)
D) OPTIMIZER UPDATE
optimizer.step()
-> updates ONLY A and B matrices
-> ~67 MB optimizer states (vs. 56 GB full FT)
optimizer.zero_grad()
-> clears LoRA gradients for next batch
Frozen params: COMPLETELY UNTOUCHED
Repeat for all batches × all epochs
STEP 5: SAVE ADAPTER
Saved:
adapter_model.safetensors ~33 MB (all A and B matrices)
adapter_config.json ~1 KB (r, a, targets, base)
NOT saved: base model (14 GB) — referenced by name

DIAGRAM 12: MERGING — LoRA'S KILLER FEATURE

Merging:
during-training two-path diagram → merge operation (matrix addition) → after-merge single-path diagram, with the key comparison to adapters that can't merge



ADAPTER SWAPPING — ONE MODEL, MANY TASKS

Adapter swapping: one base model serving multiple tasks, with storage comparison scaling to 100 tasks (1.4 TB full FT vs 17.3 GB LoRA)

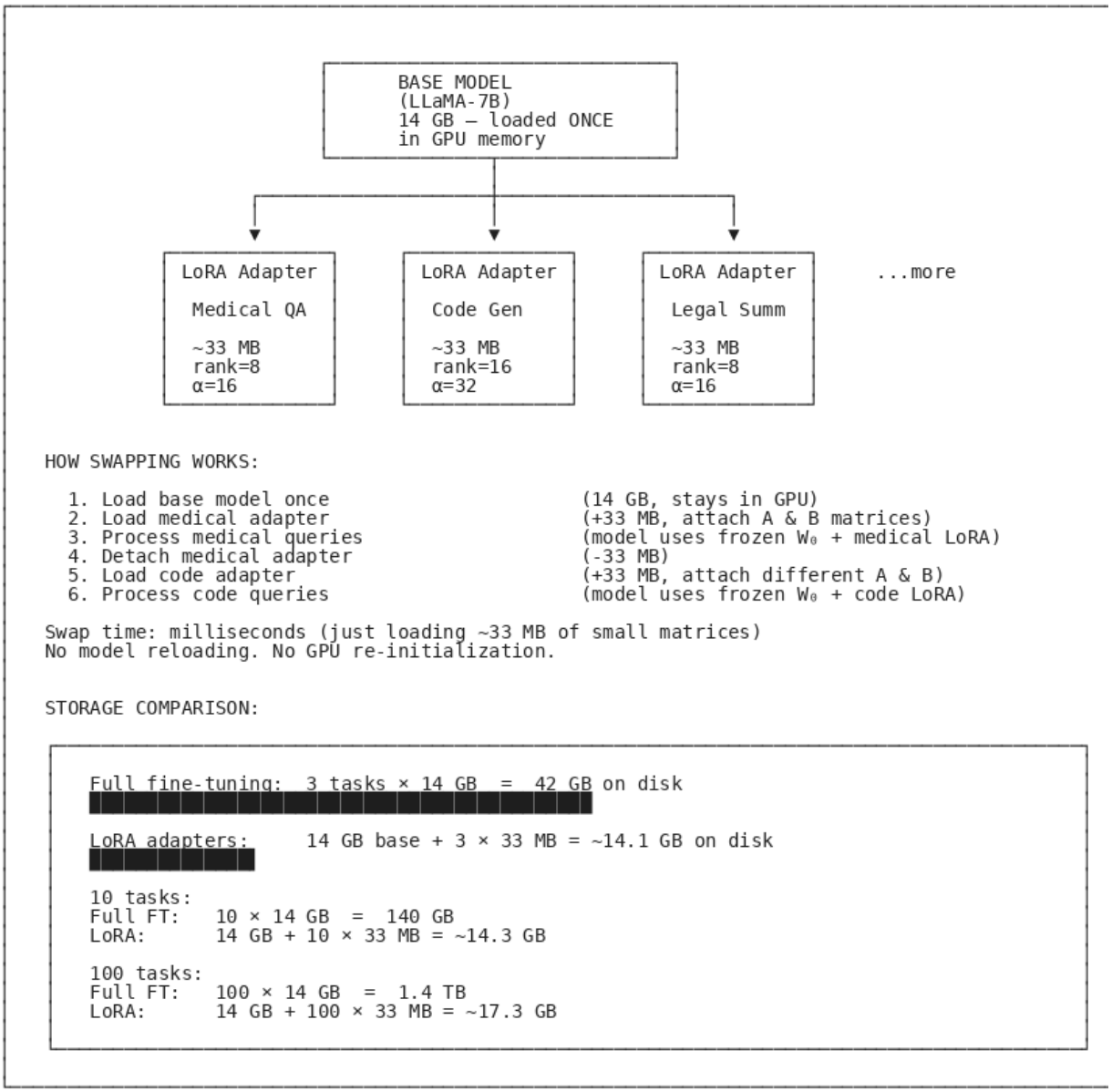


DIAGRAM 14: RANK SELECTION — WHAT DIFFERENT RANKS LOOK LIKE

Rank selection: visual representation of different ranks as "bandwidth," plus a quality-vs-rank curve showing diminishing returns

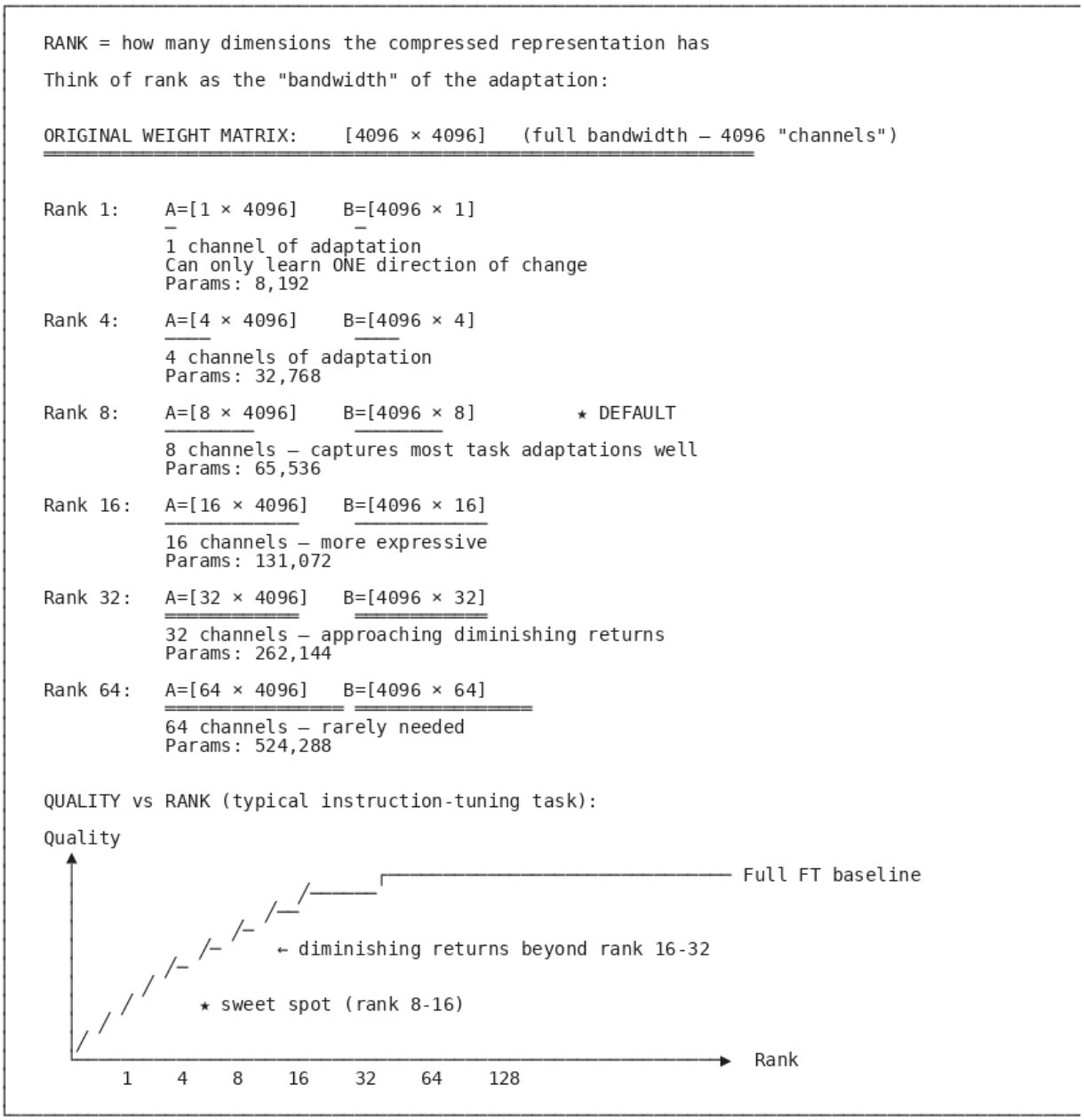
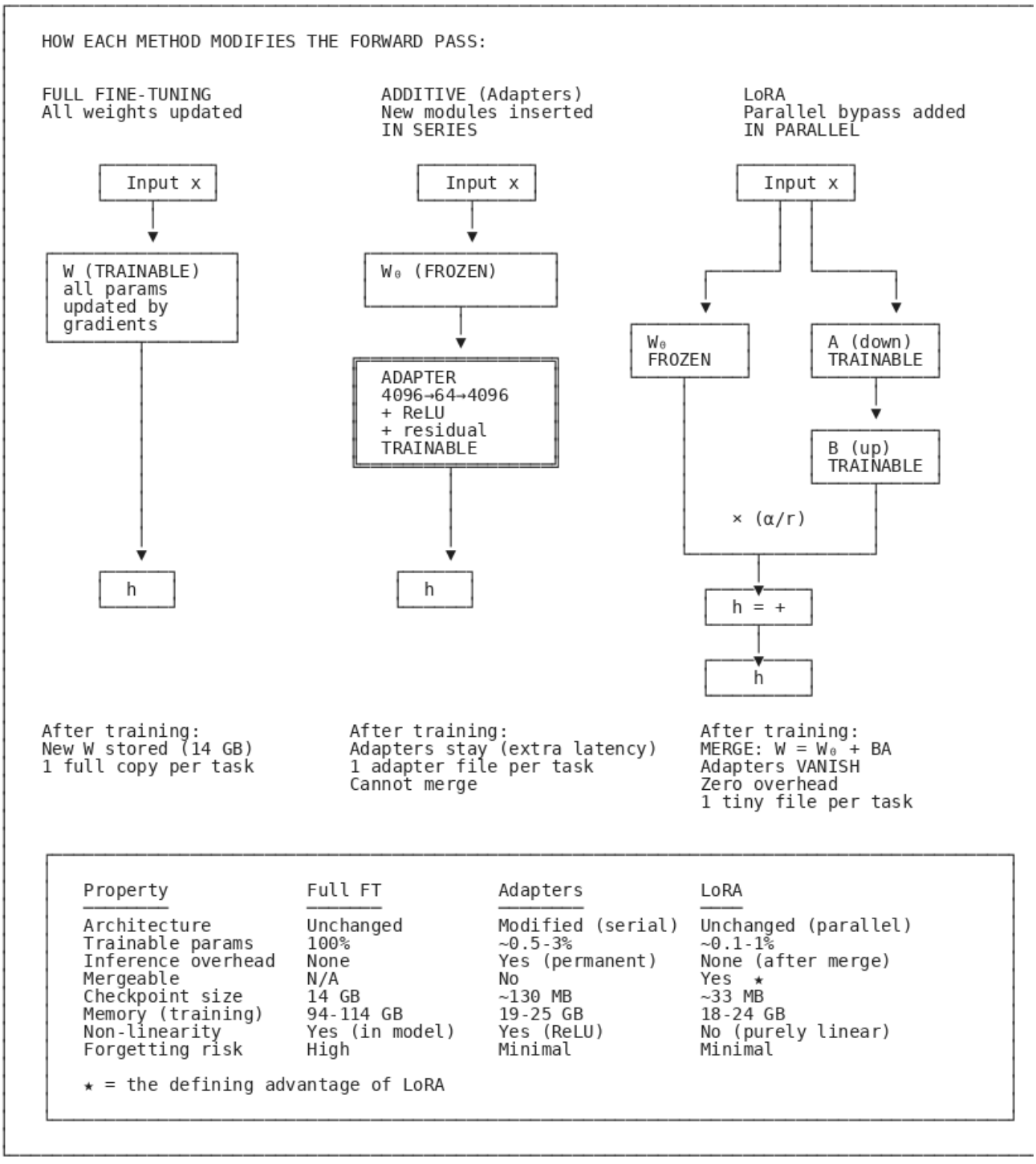


DIAGRAM 15: LoRA vs ADDITIVE ADAPTERS vs FULL FINE-TUNING — SIDE BY SIDE

Three-way architecture comparison: Full FT vs Adapters (serial) vs LoRA (parallel), with full property comparison table



Property	Full FT	Adapters	LoRA
Architecture	Unchanged	Modified (serial)	Unchanged (parallel)
Trainable params	100%	~0.5-3%	~0.1-1%
Inference overhead	None	Yes (permanent)	None (after merge)
Mergeable	N/A	No	Yes ★
Checkpoint size	14 GB	~130 MB	~33 MB
Memory (training)	94-114 GB	19-25 GB	18-24 GB
Non-linearity	Yes (in model)	Yes (ReLU)	No (purely linear)
Forgetting risk	High	Minimal	Minimal

★ = the defining advantage of LoRA

Think of it this way. You have a painting (W_0). LoRA doesn't cut the painting into pieces and try to reassemble it. LoRA keeps the painting perfectly intact and learns a small transparent overlay ($\Delta W = B \times A$) that sits on top of it. The overlay adjusts the colors slightly for your specific task.
The overlay itself is tiny because it's built from two small matrices multiplied together.

The reason $B \times A$ works as an approximation of ΔW (and not of W_0) is the low intrinsic rank insight.
The changes needed to adapt a model to a new task are simple and low-dimensional, even though the original weights themselves are complex and high-dimensional.

The original W_0 has full rank — you could NOT reconstruct it from an 8-dimensional decomposition.
But ΔW (the adjustment) lives in a much simpler subspace that can be captured by rank 8.

A concrete analogy: imagine you have a 4K photograph (W_0). You couldn't compress that entire photo down to 8 pixels and reconstruct it.
But the difference between that photo and a slightly color-corrected version (ΔW) — "make it a bit warmer, boost the blues slightly" — is simple enough to describe with just a few numbers.

LoRA VARIANTS:

DoRA (Weight-Decomposed Low-Rank Adaptation)

DoRA (Liu et al. 2024) decomposes weight updates into magnitude and direction components:

Standard LoRA	**:**	**$W = W_0 + BA$ (single combined update)**
DoRA	**:**	**$W = m \cdot (W_0 + BA) / \|\|W_0 + BA\|\|$**

Where m is a learnable magnitude vector and the rest captures direction.

Intuition: LoRA conflates "how much" and "which way" to adjust weights. DoRA separates them — like adjusting both the brightness (magnitude) and the hue (direction) of a color independently instead of changing both at once.

Result: DoRA consistently outperforms LoRA by 1-3% on benchmarks, at the cost of slightly more parameters (~10% more) and training time.

LoRA+ (Different Learning Rates for A and B)

Standard LoRA	: same learning rate for both A and B matrices.
LoRA+	: uses different learning rates — typically B gets a higher LR than A.

Why?

A (down-projection) maps to a compressed space.
B (up-projection) maps back to the original space.
They play different roles and benefit from different update magnitudes.

Typically: $lr_B = 2\times$ to $8\times lr_A$
Result: faster convergence, slightly better final performance.

rsLoRA (Rank-Stabilized LoRA)

Standard LoRA scaling: $(\alpha/r) \cdot BAx$
rsLoRA scaling: $(\alpha/\sqrt{r}) \cdot BAx$

Problem: as rank r increases, the standard α/r scaling shrinks the adapter contribution, requiring retuning α. rsLoRA replaces r with √r so that the scaling remains stable across different ranks.

Result: you can freely change rank without retuning α. Practical convenience.

AdaLoRA (Adaptive Low-Rank Adaptation)

Standard LoRA uses the same rank r for every target layer.
AdaLoRA dynamically allocates rank across layers based on importance.

Important layers (measured by sensitivity analysis) get higher rank.
Unimportant layers get lower rank or are pruned entirely.

Think of it as a budget allocation problem — given a fixed parameter budget, spend more on the layers that matter most.

Uses SVD-based parameterization ($W = P \times \Lambda \times Q$) instead of the standard $A \times B$ decomposition, which enables pruning by zeroing out singular values.

LoRA-FA (Frozen-A LoRA)

Freezes matrix A after initialization and only trains B.
Cuts trainable parameters roughly in half.

A is initialized with random Gaussian and stays fixed.
B learns to adapt the random projections from A to the task.

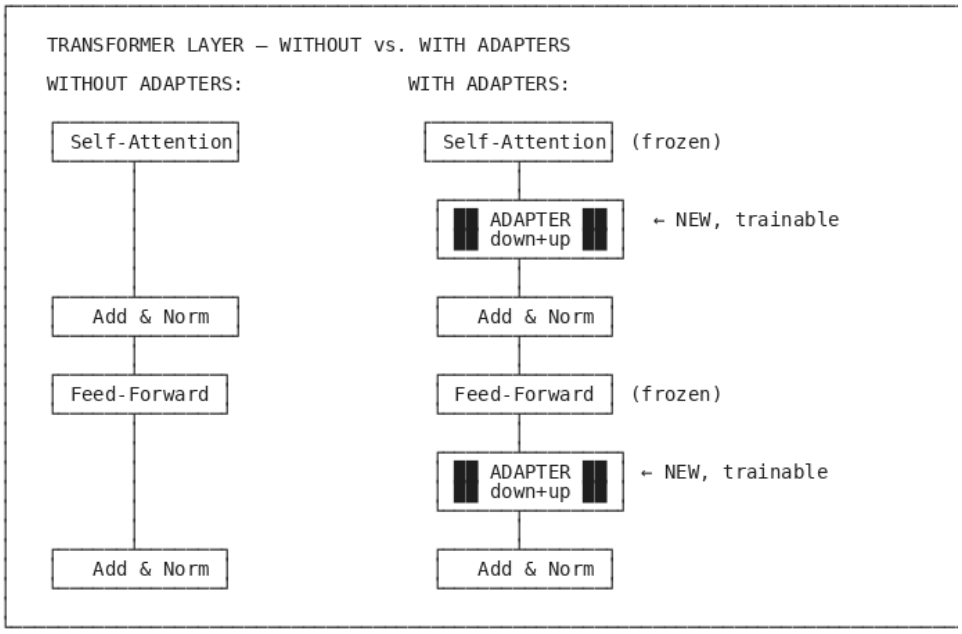
Surprisingly effective — the random projection in A provides sufficient diversity, and B alone can learn task-specific adaptations.

LoRA VARIANTS COMPARISON

Method	Key Change	Params vs LoRA	Quality vs LoRA
LoRA	Baseline	1×	Baseline
DoRA	Magnitude + direction	~1.1×	+1-3%
LoRA+	Different LR for A, B	1×	+0.5-2%
rsLoRA	√r scaling	1×	Same (more stable)
AdaLoRA	Dynamic rank allocation	≤1× (pruned)	+1-2%
LoRA-FA	Freeze A, train B only	~0.5×	-0.5% to same

BOTTLENECK ADAPTERS : (The Original PEFT Method)

Bottleneck Adapters — How They Work
Adapters (Houlsby et al. 2019) were the first major PEFT method. They insert small feed-forward networks (bottleneck modules) between existing transformer layers.



Inside an Adapter Module:

Input (hidden_dim = 4096)
↓
Down-projection: [4096 → bottleneck_dim] (e.g., 4096 → 64)
↓
Non-linearity (ReLU or GELU)
↓
Up-projection: [bottleneck_dim → 4096] (e.g., 64 → 4096)
↓
Residual connection: output = adapter_output + input
↓
Output (hidden_dim = 4096)

The bottleneck_dim controls capacity — like LoRA's rank.
A bottleneck of 64 with hidden_dim 4096 gives a 64× compression.

Adapters vs LoRA:

Property	Adapters	LoRA
Architecture	Sequential (in-series)	Parallel (bypass)
Inference overhead	Yes (extra layers)	None (after merge)
Merge into base model	No	Yes
Non-linearity	Yes (ReLU/GELU)	No (purely linear)
Parameters (typical)	~0.5-3%	~0.1-1%
Popularity (2024+)	Declining	Dominant

LoRA won because: (1) no inference overhead, (2) mergeable, (3) simpler.

(IA)³ — INFUSED ADAPTER BY INHIBITING AND AMPLIFYING INNER ACTIVATIONS

(IA)³ (Liu et al. 2022) is the most parameter-efficient method — it learns three vectors that rescale the keys, values, and feed-forward activations:

Standard attention : Q × K^T × V
(IA)³ attention : Q × (l_k ⊙ K)^T × (l_v ⊙ V)

Standard FFN : FFN(x)
(IA)³ FFN : l_ff ⊙ FFN(x)

Where ⊙ is element-wise multiplication and l_k, l_v, l_ff are learned vectors.

Total trainable parameters: just 3 vectors per layer.
For a 7B model: ~0.01% of parameters (~700K total).

Extremely lightweight, but less expressive than LoRA.
Best for few-shot learning where you need many task-specific adapters and memory is at an absolute premium.

SELECTIVE METHODS

BitFit — Training Only Bias Terms
BitFit (Zaken et al. 2021) freezes all weight matrices and only trains the bias terms.

Standard linear layer : y = Wx + b
BitFit : y = W(frozen)x + b(trainable)

In a typical transformer, bias terms are ~0.05-0.1% of total parameters.
For a 7B model, that's roughly 3.5-7 million trainable parameters.

Surprisingly effective for many NLU tasks (classification, NER, etc.). Less effective for complex generation tasks where the model needs to learn new patterns of output.

Note: many modern LLMs (LLaMA, Mistral) don't use bias terms at all, BitFit inapplicable. Check your model architecture first.

Fish Mask — Learned Binary Masks

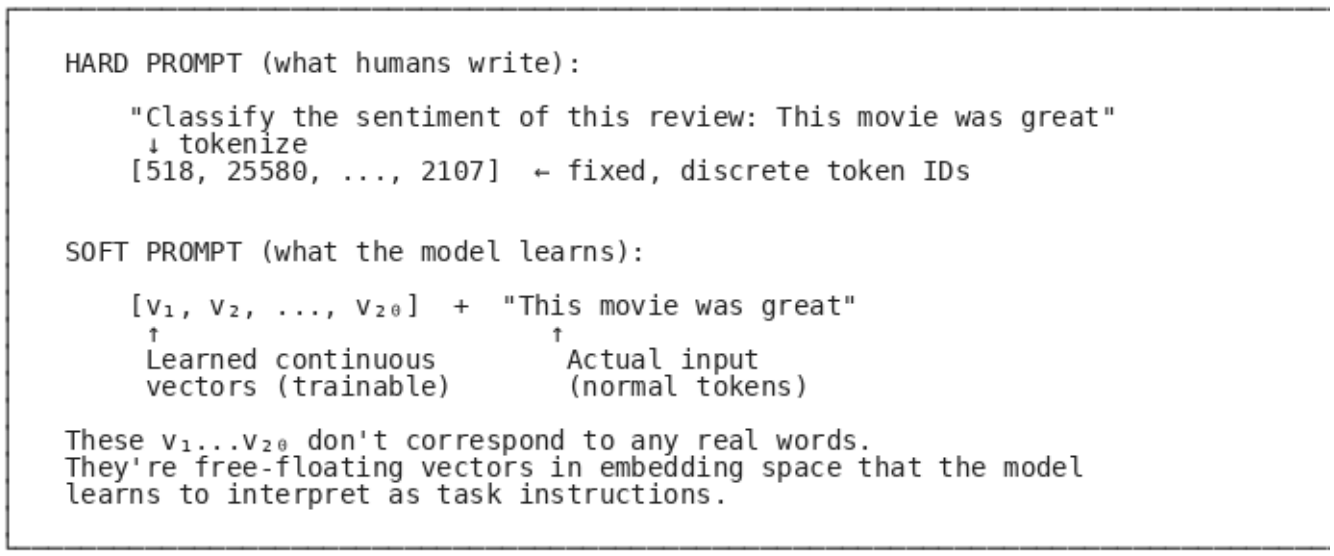
Fish Mask (Sung et al. 2021) uses Fisher information to identify the most important parameters, then creates a binary mask: 1 = train this parameter, 0 = freeze it.

Step 1: Compute Fisher information for each parameter (measures sensitivity)
Step 2: Select top-k% most important parameters
Step 3: Train only those parameters, freeze the rest

Like surgical fine-tuning — identifying exactly which weights matter most for your specific task and only updating those.

PROMPT-BASED METHODS

Prompt-Based Methods — No Weight Changes at All
These methods don't modify any model weights. Instead, they learn "soft prompts" — continuous vectors that are prepended to the input and guide the model's behavior.



Prompt Tuning:

Prompt Tuning (Lester et al. 2021) prepends k learned vectors to the input embedding:

Input: [soft_1, soft_2, ..., soft_k, token_1, token_2, ..., token_n]

Only the soft tokens are trainable. The rest of the model is completely frozen.

Trainable parameters : **k × hidden_dim**
For k=20, hidden_dim=4096: just 81,920 parameters (~0.001% of a 7B model)

Extremely efficient, but performance lags behind LoRA for smaller models. Scales better with model size — at 10B+ parameters, prompt tuning approaches full fine-tuning quality.

Prefix Tuning

Prefix Tuning (Li & Liang 2021) inserts learned vectors at every attention layer, not just the input:
Standard attention at each layer:

Q, K, V from the input sequence

Prefix Tuning:

Q from input, K = [prefix_K; input_K], V = [prefix_V; input_V]
Learned prefix vectors prepended to keys and values at EVERY layer

More expressive than Prompt Tuning because it can influence attention patterns at every depth of the model, not just at the input.

Trainable parameters: 2 × num_layers × prefix_length × hidden_dim
For 32 layers, prefix_length=20, hidden_dim=4096: ~5.2M parameters

P-Tuning v2

P-Tuning v2 (Liu et al. 2022) extends Prefix Tuning with deeper integration: learned prompts are added across all layers with layer-specific parameters.

Essentially Prefix Tuning + learned prompts at the input layer + optional task-specific classification heads.

Designed to close the gap between prompt-based methods and full fine-tuning for smaller models and NLU tasks.

PROMPT METHODS COMPARISON

Method	Where Prompts Live	Params	Quality
Prompt Tuning	Input layer only	~0.001%	Good (at scale)
Prefix Tuning	K, V at every layer	~0.1%	Better
P-Tuning v2	All layers + input	~0.1-1%	Best (prompt-based

All prompt methods: zero inference overhead if prompts are cached.
But none can be "merged" like LoRA — the prompts must always be present.

HYBRID METHODS:

LongLoRA — Efficient Attention for Long Contexts

LongLoRA (Chen et al. 2023) extends LoRA with shifted sparse attention (S^2-Attn)
to efficiently fine-tune models for longer context windows:

Standard attention: $O(n^2)$ memory, where n = sequence length
Shifted sparse attention: splits sequence into groups, applies attention
within groups, then shifts groups to capture cross-group dependencies.

This allows fine-tuning with 8K-100K+ context lengths on limited hardware. The LoRA adapters handle task adaptation while S²-Attn handles the extended context — each solving a different bottleneck.

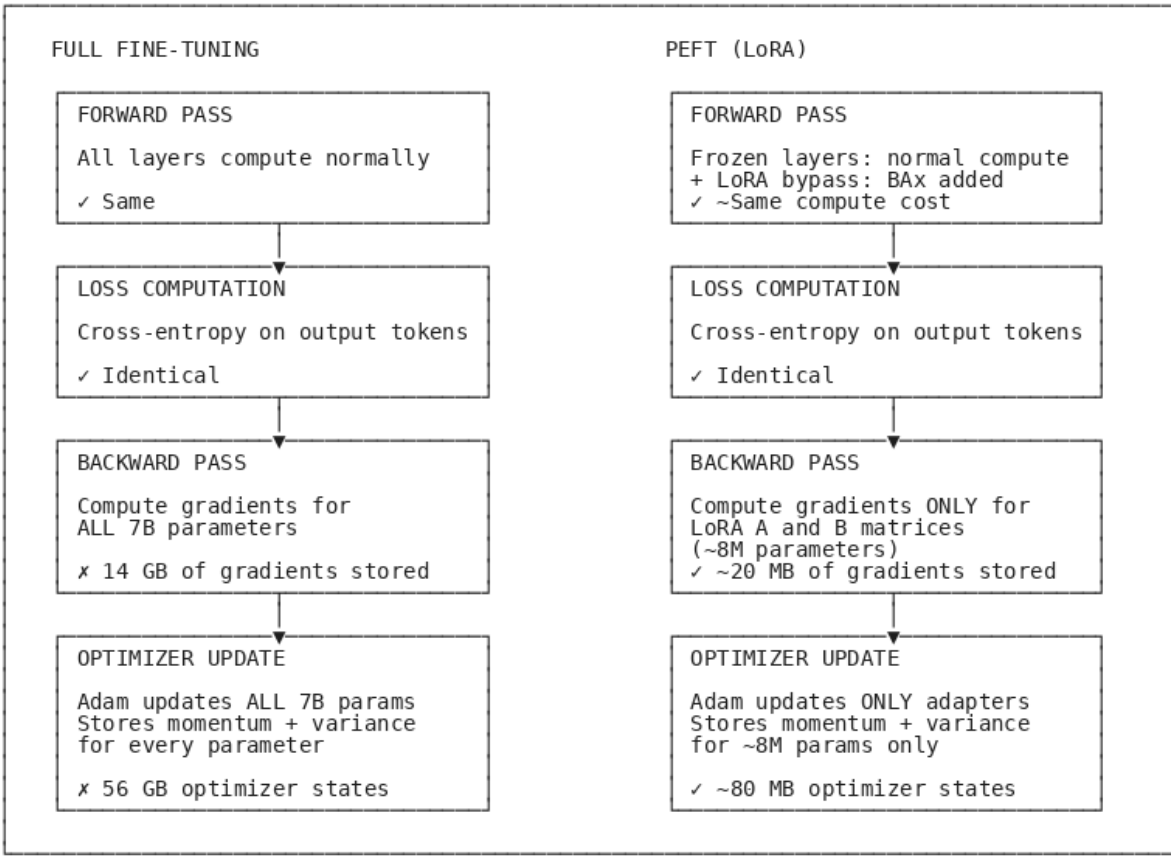
VeRA (Vector-based Random Matrix Adaptation)
VeRA (Kopiczko et al. 2024) takes parameter efficiency even further:

LoRA	**:**	**ΔW**	**=**	**B × A**	**(B and A are trainable)**
VeRA	**:**	**ΔW**	**=**	**Λ_b × B × Λ_d × A**	**(B and A are FROZEN random matrices, Λ_b and Λ_d are trainable diagonal matrices)**

Instead of learning the projection matrices, VeRA shares frozen random projections across all layers and only learns per-layer scaling vectors.
Result: 10× fewer parameters than LoRA with competitive performance.

PEFT TRAINING — UNDER THE HOOD: (The Data Pipeline with LoRA / QLoRA)

The data pipeline (JSONL → tokenize → pad → batch → tensors) is IDENTICAL to full fine-tuning.
The difference is entirely in what happens during the forward pass, backward pass, and weight update.



What Requires Gradients — The requires_grad Flag

In PyTorch, every parameter has a `requires_grad` flag:

```
# Full fine-tuning: ALL parameters have requires_grad=True

    for param in model.parameters():
        param.requires_grad = True     # ALL 7B parameters → compute gradients for all

# PEFT / LoRA: ONLY adapter parameters have requires_grad=True

    for name, param in model.named_parameters():
        if "lora" in name:
            param.requires_grad = True   # ~8M LoRA params → compute gradients
        else:
            param.requires_grad = False  # ~7B frozen params → skip gradient computation
```

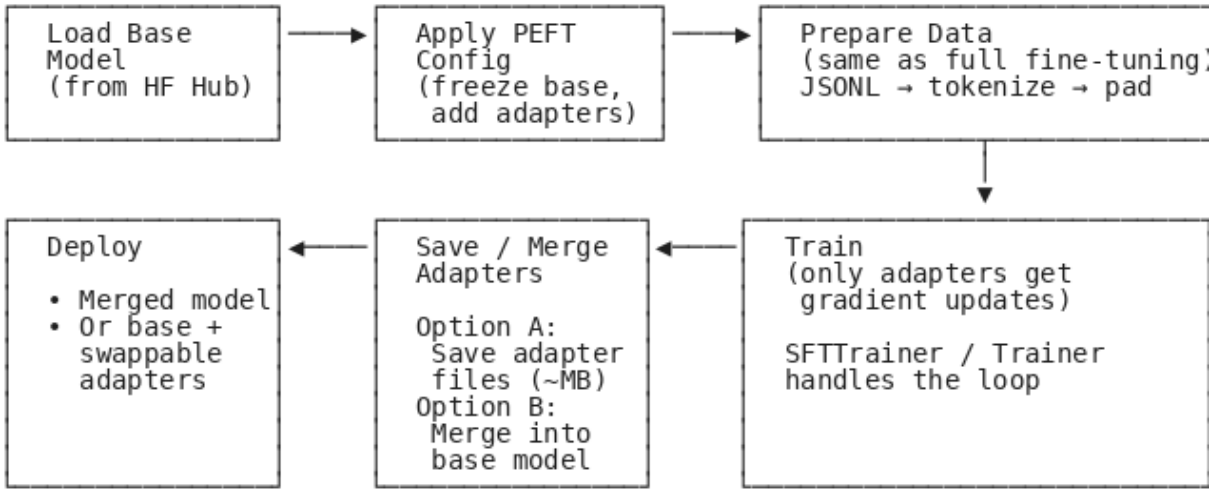
This single flag is what makes PEFT memory-efficient. PyTorch's autograd engine skips gradient computation and storage for any parameter with requires_grad=False.

Learning Rate and Hyperparameters for PEFT

```
HYPERPARAMETER COMPARISON

Hyperparameter          Full Fine-Tuning        LoRA / QLoRA
──────────────          ────────────────        ────────────
Learning rate           1e-6 to 5e-5            1e-4 to 3e-4    * higher
Epochs                  1-5                     1-5             (similar)
Batch size              32-128                  16-64           (similar)
Warmup                  5-10% of steps          3-10% of steps
Weight decay            0.01-0.1                0.0-0.05        * lower
Scheduler               Cosine decay            Cosine decay    (similar)
Gradient accumulation   Often needed            Often needed    (similar)

* LoRA can afford HIGHER learning rates because:
  1. Only updating ~0.1% of params → less risk of catastrophic change
  2. Low-rank adapters are less prone to overshooting
  3. Base model is frozen → built-in regularization

* LoRA needs LOWER weight decay because:
  1. Already heavily constrained by low-rank structure
  2. Few parameters → less need for regularization
```

PRACTICAL PEFT WORKFLOW (End-to-End with HuggingFace PEFT)

The Typical PEFT Pipeline



What Gets Saved — Adapter Files

When you save a LoRA-trained model, you save ONLY the adapter:

Full fine-tuning checkpoint:		LoRA adapter checkpoint:	
model.safetensors	14 GB	adapter_model.safetensors	33 MB
config.json		adapter_config.json	
tokenizer files		(base model referenced by name)	

To use the adapter later:

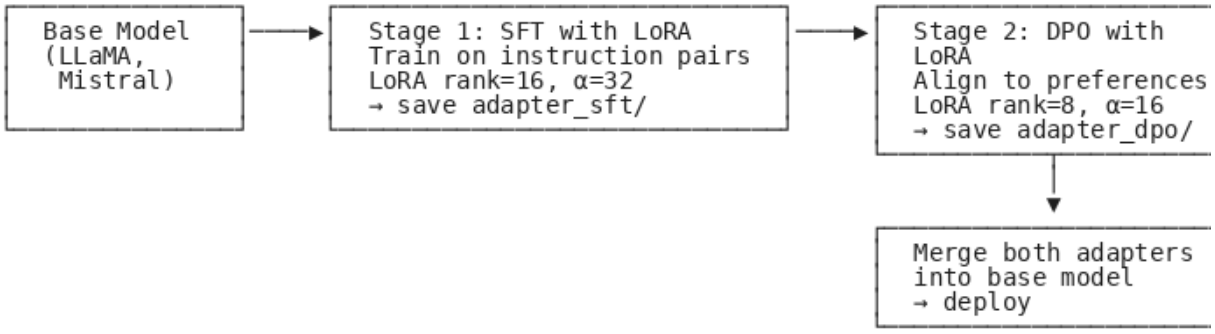
1. Load the original base model (from HuggingFace Hub or local)
2. Load the adapter on top
3. Optionally merge for zero-overhead inference

adapter_config.json contains:

```
{
  "base_model_name_or_path": "meta-llama/Llama-2-7b-hf",
  "r": 8,
  "lora_alpha": 16,
  "target_modules": ["q_proj", "k_proj", "v_proj", "o_proj"],
  "lora_dropout": 0.05,
  "task_type": "CAUSAL_LM"
}
```

PEFT WITH ALIGNMENT — LoRA + DPO / RLHF

The most common production pipeline today uses PEFT at multiple stages:



This entire pipeline can run on a single 24GB GPU with QLoRA. That's remarkable — alignment-tuning a 7B model used to require a GPU cluster.

ADAPTER MERGING — COMBINING MULTIPLE ADAPTERS

Merging Multiple LoRA Adapters

When you have trained separate adapters for different capabilities, you can combine them:

Linear Merge (simplest):

$$W_{merged} = W_0 + \lambda_1 \cdot B_1A_1 + \lambda_2 \cdot B_2A_2 + \ldots$$

Where λ_1, λ_2 are weighting coefficients (how much of each adapter to blend).
Simple weighted average. Works when tasks are related and adapters don't conflict.

TIES Merging (Trim, Elect Sign & Merge):

Step 1: Trim — remove small-magnitude changes (noise)
Step 2: Elect Sign — for each parameter, pick the majority sign direction
Step 3: Merge — average only the values that agree on direction

More robust than linear merge. Handles conflicting adapters better.

DARE (Drop And REscale):

Randomly drop a fraction of adapter parameters (set to zero),
then rescale the remaining ones to compensate.

Combined with TIES or linear merge for better generalization.
Think of it like dropout but applied to the adapter delta weights.

```
ADAPTER MERGING EXAMPLE

Base LLaMA-7B
    + Medical QA adapter    (λ=0.5)
    + Code generation adapter (λ=0.3)
    + Summarization adapter (λ=0.2)
    ──────────────────────────────
    = Single merged model that can do all three
      (quality depends on task compatibility)

No guarantee of quality — conflicting tasks may degrade each other.
Always evaluate merged models carefully.
```

WHEN LoRA DOESN'T WORK WELL (Failure Modes and Limitations)

LoRA is powerful, but it's not a universal solution. Understanding where it breaks down is just as important as knowing where it shines.

1. Tasks requiring vocabulary expansion

LoRA cannot add new tokens to the vocabulary. If your target domain uses specialized terminology that was not in the base model's training data (rare chemical names, proprietary product codes, newly coined terms), LoRA may tokenize them awkwardly and struggle to represent them well.

The LM head (the final projection from hidden states to vocab logits) is typically frozen in LoRA. You can't give the model new output tokens.

Workaround: add new tokens and fine-tune the embedding + LM head layers separately (selectively unfreeze those weights), or use full fine-tuning for vocabulary-heavy domain adaptation.

2. Very large domain gaps

LoRA's low-rank constraint works because task adaptation is assumed to be low-dimensional. But if you are taking a general English model and trying to make it fluent in a technical language with fundamentally different syntax and semantics (e.g., translating between natural language and a domain-specific programming language the base model has rarely seen), the adaptation may not fit within the low-rank subspace.

Rule of thumb: the further your target domain is from pre-training, the higher the rank you need — and eventually, full fine-tuning wins.

3. Small datasets with high rank

Counterintuitively, using too high a rank with too little data causes overfitting. The low-rank constraint is a form of regularization. If you have only a few hundred examples and use rank=64, the adapter has enough capacity to memorize the training set rather than generalize.

With small datasets, start with r=4 or r=8. Let the rank constraint do the regularization work for you.

4. Tasks where non-linearity matters
LoRA is purely linear: ΔW = B × A. No activation function between A and B. Bottleneck Adapters include a ReLU or GELU between the down- and up-projections, which can capture non-linear task-specific transformations.

For most instruction-tuning tasks this doesn't matter — the non-linearity in the frozen transformer itself is sufficient. But for tasks requiring a very different "shape" of computation, adapters may outperform LoRA.

5. Continual learning across many tasks
If you need to sequentially fine-tune a single adapter across many tasks without merging, earlier tasks will be overwritten (catastrophic forgetting within the adapter). LoRA prevents forgetting of the base model, but the adapter itself has no such protection.

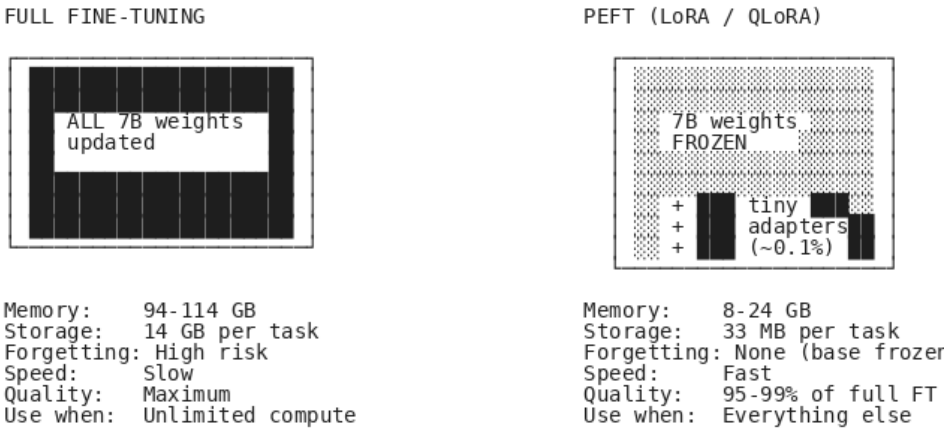
Mitigation: train separate adapters per task and swap at inference time, rather than training one adapter incrementally.

WHEN TO USE WHICH PEFT METHOD

```
DECISION TREE — CHOOSING A PEFT METHOD

Start here: Do you have a GPU?
     ├── No  →  Use API-based fine-tuning (OpenAI, Anthropic, etc.)
     └── Yes →  How much VRAM?
                ├── 8-16 GB   → QLoRA (7B model, 4-bit)
                │               or Prompt Tuning if task is simple
                ├── 24 GB     → QLoRA (7B-13B) or LoRA (7B)
                ├── 48 GB     → LoRA (7B-13B) or QLoRA (70B)
                └── 80+ GB    → LoRA (up to 70B) or Full FT (7B)
```

Model Size	Budget GPU	Best Method	Approx VRAM
7B	RTX 3090 (24GB)	QLoRA	~10 GB
7B	A6000 (48GB)	LoRA	~18 GB
13B	RTX 3090 (24GB)	QLoRA	~16 GB
13B	A100 (80GB)	LoRA	~30 GB
70B	A100 (80GB)	QLoRA	~40 GB
70B	8× A100s	LoRA (FSDP)	~20 GB/GPU

SUMMARY MENTAL MODEL



The Full PEFT Landscape — One Final View

Method	Category	Params Trained	Memory	Quality	Best For
LoRA	Reparameterization	0.1-1%	Low	Very Good	General purpose *
QLoRA	Hybrid	0.1-1%	V. Low	Very Good	Large models *
DoRA	Reparameterization	0.1-1.1%	Low	Excellent	When +1-3% matters
AdaLoRA	Reparameterization	≤0.1-1%	Low	Very Good	Optimal rank alloc
LoRA+	Reparameterization	0.1-1%	Low	Very Good	Faster convergence
Bottleneck Adapt.	Additive	0.5-3%	Medium	Good	Legacy / research
$(IA)^3$	Selective	~0.01%	V. Low	Moderate	Many adapters
BitFit	Selective	~0.05-0.1%	V. Low	Moderate	NLU tasks (w/ bias)
Prefix Tuning	Prompt-based	~0.1%	Lowest	Moderate	Task switching
Prompt Tuning	Prompt-based	~0.001%	Lowest	Moderate	Large models, few-sh
VeRA	Reparameterization	~0.01%	V. Low	Good	Extreme efficiency

* = Recommended starting point for most practitioners

Why It Works — The Intuition

Think of a pre-trained model as having already organized the world's knowledge into a high-dimensional structure. When you fine-tune for a specific task, you're not restructuring the whole thing — you're just rotating a small subspace of that structure to be more relevant to your domain. That rotation can be described with a low-rank matrix.

LoRA is essentially finding that rotation directly, without wasting parameters on the parts of the weight space that don't need to change.

The Inference Advantage Over Adapters:

Bottleneck Adapters (the main Additive PEFT alternative) insert new modules between layers. Those modules are there at inference time, adding extra computation on every forward pass. LoRA's adapters merge back into the original weights and vanish. Net inference overhead: zero.

This is the main practical reason LoRA dominates despite the similar parameter counts.

QLORA – Quantized Low-Rank Adaptation:

QLoRA's core insight: LoRA is already parameter-efficient. Make the base model memory-efficient too.

LoRA solves the parameter problem — instead of training 7 billion weights, you train 8 million.
But the base model still needs to live in GPU memory during training. For a 7B model in BF16,
that's 14 GB just for weights. Add optimizer states, gradients, and activations, and you need 16–24 GB.
A 70B model? 160+ GB — requiring multiple A100s.

QLoRA (Dettmers et al., 2023) asks: what if we could compress the frozen base model down to
4 bits per weight, leaving room for the LoRA adapters to run in full precision alongside it?
The result: a 7B model in ~4.5 GB, trainable with 16-bit LoRA adapters on a single consumer GPU.

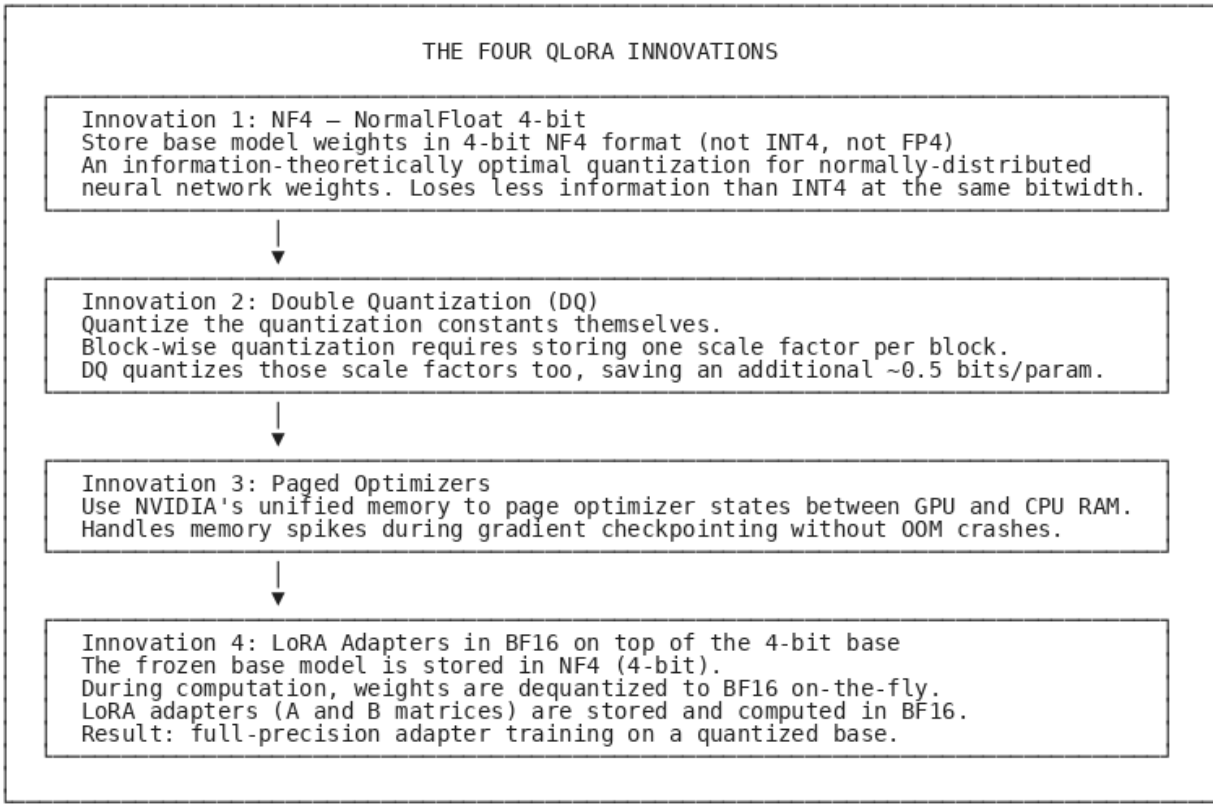
Think of it this way:

LoRA is a small team renovating a single floor of a skyscraper.

QLoRA compresses the entire building into a scale model — the renovation team still works in full scale, but the building they're working around takes up 75% less space.

QLoRA — THE FOUR INNOVATIONS

QLoRA is not a single idea. It's four engineering innovations stacked on top of LoRA, each solving a specific bottleneck in running large model training on limited hardware.



Together, these four innovations allow a 65B parameter model (Guanaco) to be trained on a single 48 GB GPU, achieving 99.3% of ChatGPT's performance on the Vicuna benchmark — results that previously required at least 8× A100s running full fine-tuning.

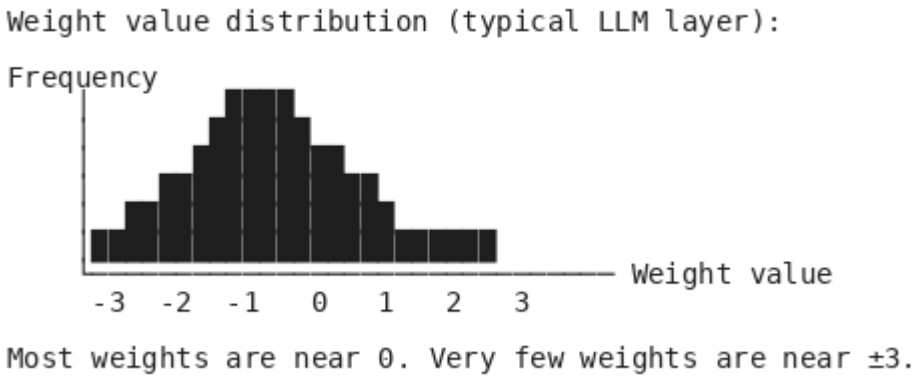
INNOVATION 1 — QUANTIZATION & NF4 IN DEPTH (The Core Memory Compression — Understanding Every Bit)

What Is Quantization?
Quantization is the process of representing numerical values in fewer bits. Every weight in a neural network is a floating-point number. In BF16 (the standard training format), each weight takes 16 bits = 2 bytes.

The goal of quantization is to represent the same weight using only 4 bits = 0.5 bytes.
That's a 4× compression of the weight memory footprint.

The challenge: you lose information. A 4-bit number can only represent 16 distinct values ($2^4 = 16$). A BF16 number can represent ~65,000 distinct values.
The quantization question is: which 16 values should we choose, and how should we map the original weights to them ?

Why Not Just Use INT4?
INT4 maps to 16 evenly spaced integers: {-8, -7, -6, ..., 6, 7}. This looks like a natural choice — it covers the full range uniformly.
But here's the problem: neural network weights are not uniformly distributed.
After pre-training, weight values follow an approximately normal (Gaussian) distribution. Most weights cluster near zero, with fewer at the extremes:



If you use INT4's evenly spaced quantization levels, you waste most of your 16 slots representing extreme values that almost no weight has.
Your quantization grid looks like:

INT4 grid (uniform):

```
| | | | | | | | | | | | | | | | |
-8 -7 -6 -5 -4 -3 -2 -1 0  1  2  3  4  5  6  7  8
```

For a normally distributed weight, most values fall in [-2, 2].
Only 6 of the 16 slots serve the majority of the data. The rest go to waste.

NF4 — Information-Theoretically Optimal for Normal Distributions

NF4 (NormalFloat 4-bit) solves this by **distributing the 16 quantization levels according to the quantiles of the normal distribution** — not uniformly.

Instead of equal spacing between -8 and 7, NF4 places more levels near zero (where most weights cluster) and fewer at the extremes (where weights rarely appear).

The NF4 quantization levels are computed so that each level covers an equal probability mass under the standard normal distribution N(0, 1):

NF4 grid (quantile-based, normalized to [-1, 1]):

{-1.0000, -0.6962, -0.5251, -0.3946, -0.2844, -0.1848, -0.0911, 0.0000,
0.0796, 0.1609, 0.2461, 0.3379, 0.4407, 0.5626, 0.7230, 1.0000}

← sparse dense near 0 sparse →

The levels are NOT evenly spaced. There are 8 levels between -0.4 and +0.4 and only 3 levels below -0.5 and 3 above 0.5.

This matches how neural network weights are actually distributed. level represents roughly the same number of weights (equal probability mass), so no information is "wasted" on empty regions of weight space.

Why "NormalFloat" ?
Because it's designed to be optimal when weights are normally distributed.
The original paper proves that NF4 has lower quantization error than INT4 for Gaussian data
at any given bitwidth — it's the information-theoretically correct format for this task.

Block-Wise Quantization (Absmax Quantization)

Before applying NF4, each weight needs to be mapped to the [-1, 1] range where
the NF4 codebook lives. This is done via absmax quantization, applied block-by-block.

Why block-wise ?
If you quantize a full weight matrix with a single scale factor (the global max), one outlier value could distort the entire matrix. A single weight of magnitude 100 would compress all other weights into a tiny range near 0, destroying precision.

Block-wise quantization divides the weight matrix into small blocks (default: 64 weights per block) and computes a separate scale factor for each block:

Weight matrix W [4096 × 4096] = 16,777,216 weights
│
├── Block 1: weights[0:64] → $scale_1$ = max(|weights[0:64]|)
├── Block 2: weights[64:128] → $scale_2$ = max(|weights[64:128]|)
├── Block 3: weights[128:192] → $scale_3$ = max(|weights[128:192]|)
│ ...
└── Block N: weights[N×64:(N+1)×64] → $scale_n$

Total blocks : 16,777,216 / 64 = 262,144 blocks
Scale factors : 262,144 values (one per block), stored in FP32

Memory for scale factors : 262,144 × 4 bytes = 1,048,576 bytes ≈ 1.0 MB
(small relative to the weight matrix itself)

The quantization process (per block):

Step 1: Find the absolute maximum in the block
absmax = max(|w| for w in block) e.g., absmax = 2.47

Step 2: Normalize all weights in the block to [-1, 1]
w_normalized = w / absmax maps [-2.47, 2.47] → [-1, 1]

Step 3: Map to nearest NF4 quantization level
w_nf4 = nearest_nf4_level(w_normalized) maps float → 4-bit index (0-15)

Step 4: Store the 4-bit index and the scale factor (absmax)
Stored: w_nf4 (4 bits), absmax (float32 per block)

The dequantization process (during forward pass):

Step 1: Look up the NF4 codebook value for the 4-bit index
w_normalized = NF4_CODEBOOK[w_nf4] 4-bit index → float in [-1, 1]

Step 2: Rescale back to original magnitude
w_dequantized = w_normalized × absmax e.g., → ~original value

Step 3: Use w_dequantized (in BF16) for computation

This dequantization happens on-the-fly during each forward pass.
The weights are NEVER stored in BF16 — they live as 4-bit integers in GPU memory and are promoted to BF16 only for the matrix multiplication, then discarded.

The Concrete Memory Math

For a 7B parameter model (LLaMA-2-7B):

WITHOUT quantization (BF16):
7,000,000,000 weights × 2 bytes/weight = 14,000,000,000 bytes = ~13.0 GB

WITH NF4 (4-bit):
7,000,000,000 weights × 0.5 bytes/weight = 3,500,000,000 bytes = ~3.26 GB

Plus scale factors (one FP32 per 64-weight block):

7,000,000,000 / 64 = 109,375,000 blocks

109,375,000 × 4 bytes = 437,500,000 bytes ≈ 0.41 GB

Total: 3.26 + 0.41 = ~3.67 GB
COMPRESSION RATIO: 13.0 GB → 3.67 GB = 3.54 × compression

After double quantization (Innovation 2), the scale factor overhead drops further. Final QLoRA base model: ~3.5 GB for a 7B model.

```
MODEL SIZE COMPARISON (weights only)

Model Size    FP32         BF16         INT8         NF4 (4-bit)
──────────    ────         ────         ────         ──────────
7B            28 GB        14 GB        7 GB         ~3.5 GB
13B           52 GB        26 GB        13 GB        ~6.5 GB
30B           120 GB       60 GB        30 GB        ~15 GB
65B           260 GB       130 GB       65 GB        ~33 GB
70B           280 GB       140 GB       70 GB        ~35 GB

* NF4 includes ~12% overhead for block-wise scale factors
* After double quantization, overhead drops to ~8%
```

INNOVATION 2 — DOUBLE QUANTIZATION (DQ)
(Quantizing the Quantization Constants — Saving 0.5 bits/param)

The Problem with Block-Wise Scale Factors
Block-wise quantization requires storing one FP32 scale factor per block of 64 weights.
FP32 = 32 bits = 4 bytes per scale factor.

Scale factor overhead per parameter = 32 bits / 64 parameters = 0.5 bits/parameter

That's a non-trivial overhead on top of the 4 bits we're using per weight. Effective cost: 4 + 0.5 = 4.5 bits per parameter. We can do better.

Double Quantization — The Solution
Double Quantization quantizes the FP32 scale factors themselves using INT8.

FIRST quantization: Weights (BF16) → NF4 codes (4-bit)
Scale: one FP32 absmax per block of 64 weights

SECOND quantization: Scale factors (FP32) → INT8 (8-bit)
Scale: one FP32 absmax per 256 scale factors (a "super-block")

Let's trace the math:
First level (weights → NF4):
Block size: 64 weights
Scale precision: FP32 (32 bits per block)
Scale overhead: 32 bits / 64 weights = 0.5 bits per weight

Second level (scale factors → INT8):
Super-block size: 256 scale factors (covering 256 × 64 = 16,384 weights)
Super-scale: FP32 (one per super-block)
INT8 scale factors: 256 × 8 bits = 2,048 bits instead of 256 × 32 = 8,192 bits
Savings per weight: (8,192 - 2,048) bits / 16,384 weights ≈ 0.375 bits/weight

Net overhead after double quantization:
NF4 weights: 4.000 bits/param
INT8 scale factors: 8/64 = 0.125 bits/param
FP32 super-scale factors: 32/16384 ≈ 0.002 bits/param

Total: 4.127 bits/param

vs. standard NF4 without DQ:
NF4 weights: 4.000 bits/param
FP32 scale factors: 32/64 = 0.500 bits/param

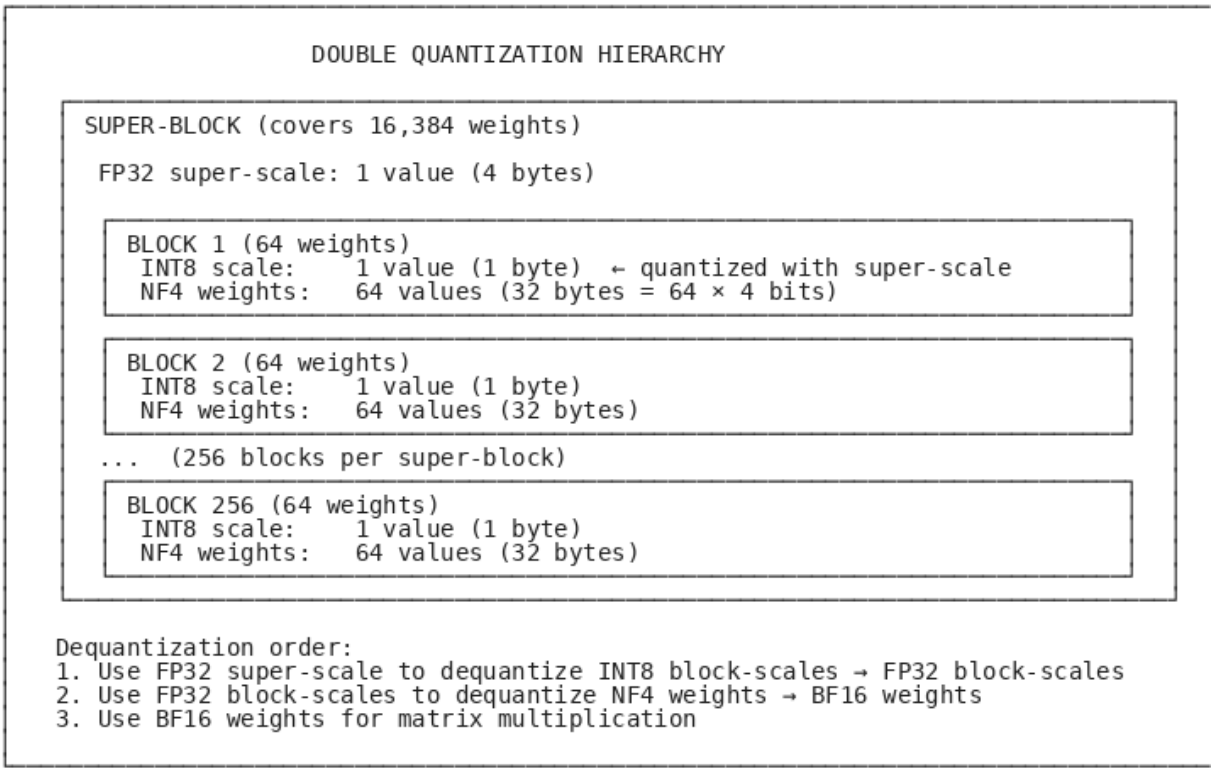
Total: 4.500 bits/param

Savings from Double Quantization: 0.5 - 0.127 = 0.373 bits per parameter

For a 7B parameter model:
Savings = 0.373 bits × 7,000,000,000 / 8 bits/byte ≈ 327 MB

That 327 MB (~0.37 bits/param) is not enormous in absolute terms, but on a 10 GB GPU it represents a meaningful fraction of available memory — potentially the difference between a 7B model fitting at batch_size=4 vs. crashing with OOM.

The Two-Level Hierarchy Visualized



INNOVATION 3 — PAGED OPTIMIZERS
(Handling Memory Spikes Without OOM — GPU ↔ CPU Paging)

The Memory Spike Problem

During training, GPU memory usage is not constant. Two events cause sharp spikes:

1. Gradient checkpointing recomputation:

When recomputing activations during backprop, all activations for a layer are momentarily in memory simultaneously. For long sequences, this can spike memory usage by 2-4 GB for just a few milliseconds.

2. Microbatch boundary processing:

When processing the final tokens in a packed batch, the model may briefly need more memory than the steady-state consumption.

These spikes cause OOM (Out of Memory) crashes at the worst time — deep into a training run.

The rest of the time, there's plenty of memory. The spike is the problem.

Paged Optimizers — The Solution

QLoRA uses NVIDIA's unified memory feature, which allows GPU memory to automatically page to and from CPU RAM when the GPU runs out — exactly like virtual memory in an OS.

Paged optimizers take the AdamW optimizer states (momentum and variance buffers) and allocate them in CUDA unified memory rather than standard GPU memory:

Standard AdamW optimizer states:

momentum buffer : 1 FP32 value per trainable parameter → allocated in GPU VRAM
variance buffer : 1 FP32 value per trainable parameter → allocated in GPU VRAM

For 8.4M LoRA parameters:
momentum : 8.4M × 4 bytes = 33.6 MB (in GPU VRAM, always)
variance : 8.4M × 4 bytes = 33.6 MB (in GPU VRAM, always)

Paged AdamW optimizer states:

Same 33.6 MB each, but allocated in CUDA unified memory.

During normal operation: lives in GPU VRAM (same speed as before)
During memory spikes: automatically pages to CPU RAM (slower, but no OOM)
After spike subsides: pages back to GPU VRAM

The paging is transparent to the training code. You don't have to manage it.
The speed cost is only paid during the rare spike — the rest of training runs at GPU speed.

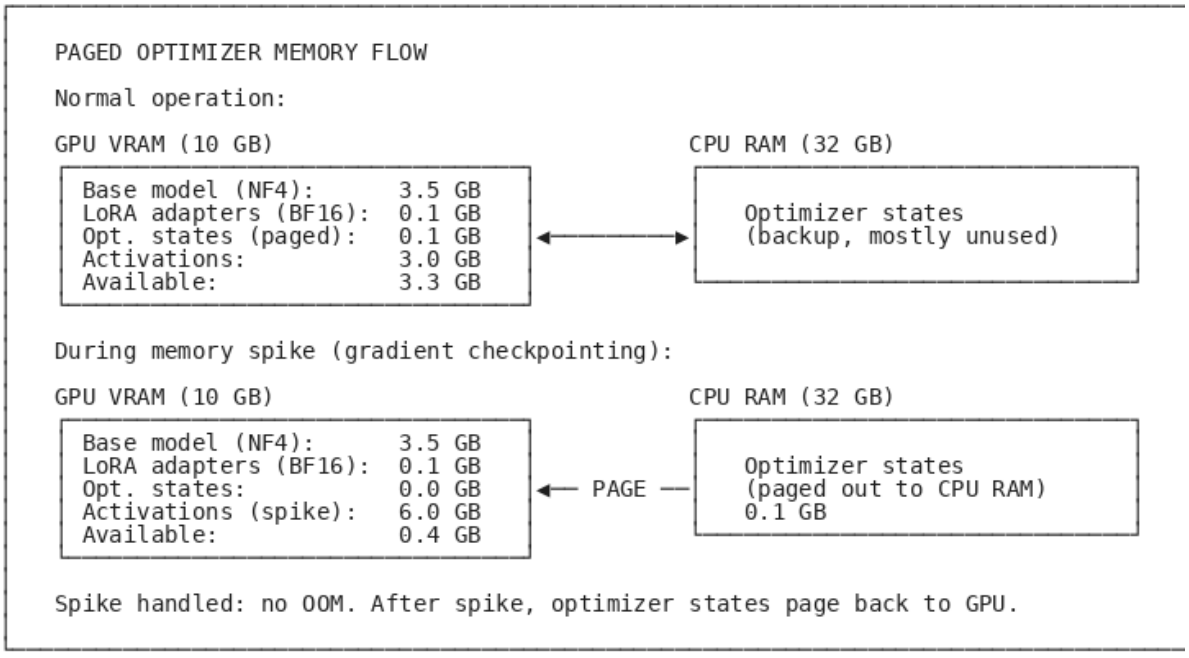
Why This Matters More Than It Sounds

Without paged optimizers, a training run on a 10 GB GPU might proceed perfectly for hours, then crash with OOM on batch 847 because of a single long sequence that caused a temporary memory spike. The entire run is lost.

With paged optimizers:

- The spike gets handled by temporarily using CPU RAM
- Training continues without interruption
- The cost is a brief slowdown on that batch (milliseconds)
- No data loss, no restart

This is particularly important for consumer GPUs (RTX 3090, 4090) where GPU memory is precious and there's typically 32-64 GB of CPU RAM available as a buffer.



INNOVATION 4 — LORA IN BF16 ON TOP OF NF4 BASE
(The Compute Dtype Trick — Why This Works Numerically)

The Storage vs. Compute Dtype Distinction

This is one of the most important and often-misunderstood aspects of QLoRA.

There are TWO distinct dtypes at play:

Storage dtype	**:**	**How weights are stored in GPU memory (NF4 = 4-bit)**
Compute dtype	**:**	**What dtype is used during actual matrix multiplication (BF16 = 16-bit)**

These are different. The 4-bit weights are NOT used directly in matrix multiplications. Modern GPU hardware (CUDA cores, Tensor Cores) doesn't natively support 4-bit arithmetic.

Instead, the workflow is:

1. Retrieve 4-bit NF4 weights from GPU memory (very fast — tiny data transfer)
2. Dequantize to BF16 on-the-fly (cheap — a lookup + multiply per weight)
3. Run matrix multiplication in BF16 (full Tensor Core performance)
4. Discard the BF16 weights (they're NOT stored back — the NF4 remains the source of truth)

This is why QLoRA doesn't sacrifice compute speed as much as you'd expect.
The actual math happens in BF16. The savings come from the storage size.

How LoRA Adapters Sit On Top

The LoRA A and B matrices are stored and computed in BF16 throughout.
They are never quantized. This is deliberate:

The base model (NF4): frozen, read-only, static information
→ aggressively compress it (4-bit is fine, it won't change)

The LoRA adapters (BF16): actively trained, gradients flow through them
→ must preserve precision for gradient descent to work

The frozen model is like old books in a warehouse — compress them.
The adapters are like the active work-in-progress — keep those in full quality.
During a forward pass through a QLoRA-adapted layer:

$$h = (\text{dequant}(W_0\_nf4) \times x) + (\alpha/r) \cdot B(Ax)$$

- Frozen path: W_0 stored in NF4, dequantized to BF16 for compute
  no gradients stored (requires_grad = False)
- LoRA path: A and B in BF16
  gradients flow here

Why NF4 + BF16 Compute Works Numerically

You might worry: if we're dequantizing to BF16 and computing in BF16, aren't we introducing quantization error that corrupts the gradients?

Yes — but it's bounded, and the LoRA structure limits the impact:

1. The quantization error in W_o_nf4 is constant per weight (fixed after quantization).
   It appears as a fixed bias in the frozen path's output, not as noise.

2. The LoRA adapters can partially compensate for systematic quantization errors
   in the frozen base — they "see" the quantized base and adapt accordingly.

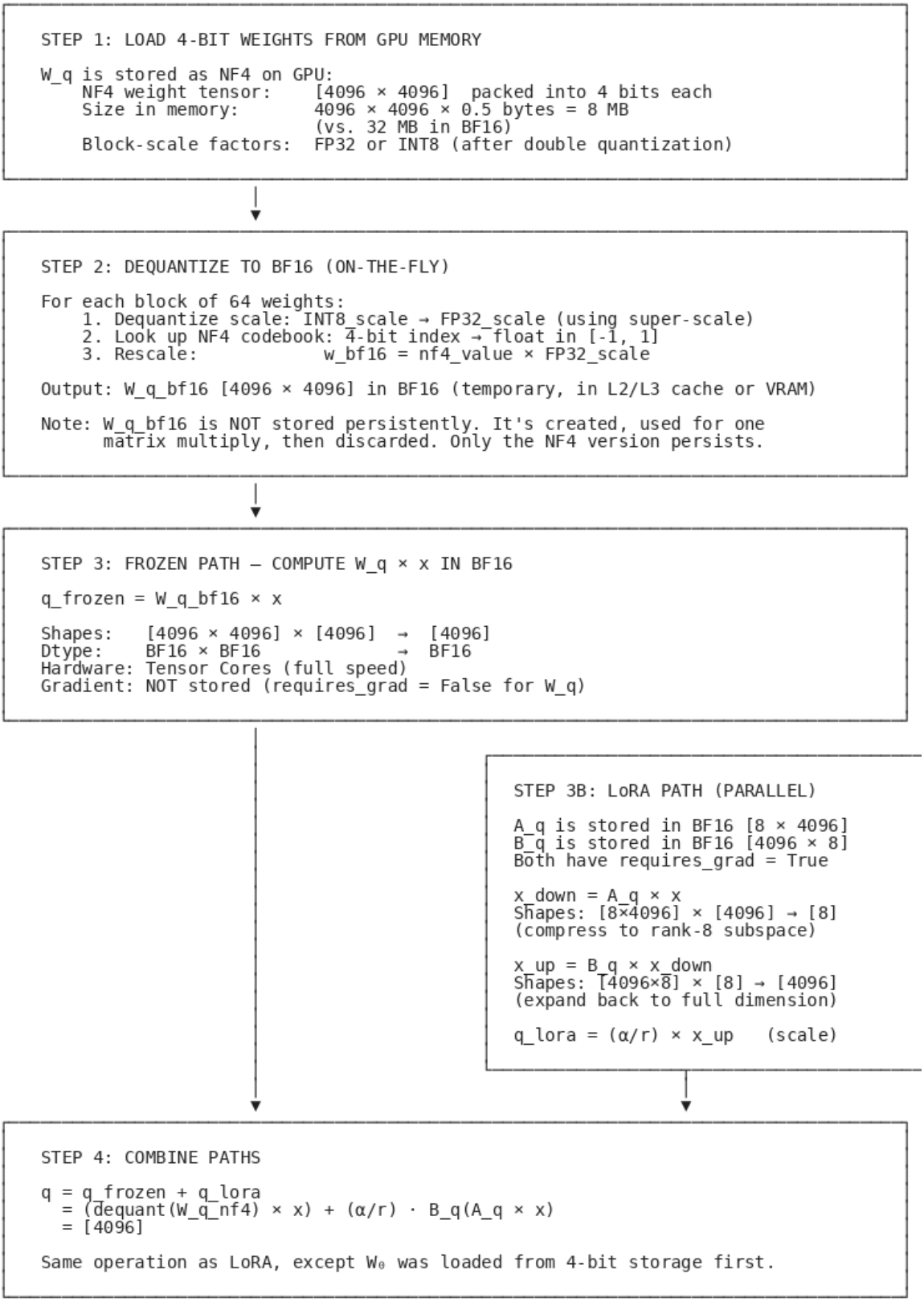
3. The NF4 format is chosen precisely because it minimizes this quantization error
   for normally-distributed weights. The errors are small to begin with.

4. In practice, QLoRA-trained models are within 1-4% quality of LoRA-trained models
   on the same base, which are within 1-5% of full fine-tuning. The compounding
   error is acceptable for most use cases.

THE COMPLETE QLoRA FORWARD PASS — STEP BY STEP
(Tracing a Single Token Through Every Layer of a QLoRA Model)

Let's trace one token through a QLoRA-adapted LLaMA-7B model.
Model specs: 32 transformer layers, d_model=4096, LoRA on q/k/v/o_proj, rank=8.

Input to Layer 0:
x = [4096] (one token's hidden state, BF16)
At every QLoRA attention layer (example: W_q with LoRA):


STEP 1: LOAD 4-BIT WEIGHTS FROM GPU MEMORY
W_q is stored as NF4 on GPU:
NF4 weight tensor: [4096 × 4096] packed into 4 bits each
Size in memory: 4096 × 4096 × 0.5 bytes = 8 MB
(vs. 32 MB in BF16)
Block-scale factors: FP32 or INT8 (after double quantization)
STEP 2: DEQUANTIZE TO BF16 (ON-THE-FLY)
For each block of 64 weights:
1. Dequantize scale: INT8_scale → FP32_scale (using super-scale)
2. Look up NF4 codebook: 4-bit index → float in [-1, 1]
3. Rescale: w_bf16 = nf4_value × FP32_scale
Output: W_q_bf16 [4096 × 4096] in BF16 (temporary, in L2/L3 cache or VRAM)
Note: W_q_bf16 is NOT stored persistently. It's created, used for one
matrix multiply, then discarded. Only the NF4 version persists.
STEP 3: FROZEN PATH – COMPUTE W_q × x IN BF16
q_frozen = W_q_bf16 × x
Shapes: [4096 × 4096] × [4096] → [4096]
Dtype: BF16 × BF16 → BF16
Hardware: Tensor Cores (full speed)
Gradient: NOT stored (requires_grad = False for W_q)
STEP 3B: LoRA PATH (PARALLEL)
A_q is stored in BF16 [8 × 4096]
B_q is stored in BF16 [4096 × 8]
Both have requires_grad = True
x_down = A_q × x
Shapes: [8×4096] × [4096] → [8]
(compress to rank-8 subspace)
x_up = B_q × x_down
Shapes: [4096×8] × [8] → [4096]
(expand back to full dimension)
q_lora = (α/r) × x_up (scale)
STEP 4: COMBINE PATHS
q = q_frozen + q_lora
= (dequant(W_q_nf4) × x) + (α/r) · B_q(A_q × x)
= [4096]
Same operation as LoRA, except W₀ was loaded from 4-bit storage first.

The same process repeats for W_k, W_v, W_o, and optionally the MLP layers.
After 32 such layers, the output hits the LM Head (frozen, no LoRA typically):

logits = LM_head(hidden_state)
= [batch_size, seq_len, vocab_size]

Complete Forward Pass Data Flow (7B model, batch=4, seq=512, rank=8):

```
COMPONENT                          SHAPE                 DTYPE     SIZE IN MEMORY
─────────                          ─────                 ─────     ──────────────

Input token IDs                    [4, 512]              INT32     ~4 KB
Embeddings (frozen, NF4-like)      [4, 512, 4096]        BF16      ~16 MB

Per layer (×32):
W_q (stored NF4)                 [4096, 4096]          NF4       8 MB (persistent)
W_q (dequantized, temporary)     [4096, 4096]          BF16      32 MB (created/discarded)
q_frozen output                  [4, 512, 4096]        BF16      ~16 MB
A_q (LoRA, stored)               [8, 4096]             BF16      0.064 MB (persistent)
B_q (LoRA, stored)               [4096, 8]             BF16      0.064 MB (persistent)
q_lora output                    [4, 512, 4096]        BF16      ~16 MB
q combined                       [4, 512, 4096]        BF16      ~16 MB
(same for k, v, o)

Logits                             [4, 512, 32000]       BF16      ~125 MB
```

7B Model (LoRA rank=8, batch=4, seq=512)
GLOBAL COMPONENTS

Component	Architecture Role	Stored (Persistent)	Temporary (Forward Pass)	Notes
Input IDs	Token indices	INT32 (~4 KB)	—	Minimal memory
Embeddings	Token → vector	NF4 compressed (persistent)	BF16 activation (~16 MB)	Dequantized once per forward
Logits	Final output	—	BF16 (~125 MB)	Largest activation tensor

PER TRANSFORMER LAYER (×32 layers)
Frozen Attention Weights (per projection: q, k, v, o) - Happens **4 times per layer** (q, k, v, o)

Component	Shape	Stored	Temporary	Compute Type
W (NF4)	[4096, 4096]	**8 MB**	—	Stored compressed
W (dequant)	[4096, 4096]	—	**32 MB**	Created for matmul
Frozen output	[4,512,4096]	—	~16 MB	Activation

LoRA Components (per projection)

Component	Shape	Stored	Temporary	Notes
A	[8, 4096]	0.064 MB	—	Trainable
B	[4096, 8]	0.064 MB	—	Trainable
LoRA output	[4,512,4096]	—	~16 MB	BF16
Combined output	[4,512,4096]	—	~16 MB	Frozen + LoRA

THE COMPLETE QLoRA TRAINING LOOP — STEP BY STEP
(Every Stage from Raw Data to Saved Adapter, in Extreme Detail)

Step 0: Understand What Problem QLoRA Is Solving
LoRA on a 7B model requires ~16-24 GB GPU memory (14 GB for BF16 weights + adapters + activations).
This works on an A100 (80 GB) but not on consumer hardware.

QLoRA's target: a 7B model in ~8-12 GB. That's an RTX 3090 or RTX 4090.
A 70B model: ~40 GB. That's a single A100 80 GB instead of 8×A100s.
The recipe:

1. Quantize base model weights to NF4 (4-bit) → 75% memory reduction for weights
2. Double-quantize scale factors → additional ~375 MB savings
3. Use paged optimizers → handle memory spikes without OOM
4. Train LoRA adapters in BF16 on top

Step 1: Install and Configure the 4-Bit Base Model

```
from transformers import BitsAndBytesConfig
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,                  # Enable 4-bit loading
    bnb_4bit_quant_type="nf4",          # Use NF4 (not INT4, not FP4)
    bnb_4bit_compute_dtype=torch.bfloat16,  # Compute in BF16 (not FP32)
    bnb_4bit_use_double_quant=True,     # Enable double quantization
)
```

```
model = AutoModelForCausalLM.from_pretrained
(
    "meta-llama/Llama-2-7b-hf",
    quantization_config=bnb_config,
    device_map="auto",                  # Automatically assign layers to GPU(s)
)
```

What happens during from_pretrained() with 4-bit config:

1. Download model weights (14 GB BF16 checkpoint)
2. For each weight matrix, quantize to NF4:
   a. Divide weights into blocks of 64
   b. Compute absmax per block
   c. Normalize and map to NF4 codebook
   d. Store 4-bit codes + scale factors
3. Load quantized model into GPU VRAM:
   NF4 weights: ~3.5 GB
   Scale factors: ~0.4 GB (or ~0.1 GB after double quant)
4. Register all base model parameters as requires_grad = False

After loading: ~3.6 GB of GPU memory used for the base model (vs. 14 GB in standard BF16 LoRA)

Step 2: prepare_model_for_kbit_training()

```
from peft import prepare_model_for_kbit_training
model = prepare_model_for_kbit_training(model)
```

This function does three specific things:

2a. Enables gradient checkpointing on the model
(saves activation memory at the cost of recomputation during backward)

2b. Casts all non-quantized modules to FP32
(layer norms, embedding layers — these don't get quantized but need FP32 for stability)

Why? Layer norms work on statistics (mean, variance) of activations.
In BF16, these computations lose precision. QLoRA keeps them in FP32
to avoid numerical instability in normalization.

2c. Configures gradient checkpointing to handle the frozen/quantized base
(ensures the checkpointing recomputation correctly dequantizes weights again)

Step 3: Apply LoRA Adapters

```
from peft import LoraConfig, get_peft_model

lora_config = LoraConfig
(
    r=16,
    lora_alpha=32,
    target_modules="all-linear",          # Or ["q_proj", "k_proj", "v_proj", "o_proj"]
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM",
)
model = get_peft_model(model, lora_config)

After this step, the model has:
  - Frozen NF4 base weights (no gradients, ~3.6 GB)
  - Trainable BF16 LoRA A and B matrices (full gradients, ~0.1 GB)

model.print_trainable_parameters()
# trainable params: 83,886,080 || all params: 6,830,690,304 || trainable%: 1.2280
# (varies by target_modules and rank)
```

Step 4: Set Up Paged Optimizer

```
import bitsandbytes as bnb

# Paged AdamW — same as AdamW but optimizer states live in unified memory

    optimizer = bnb.optim.PagedAdamW8bit
(
    model.parameters(),
    lr=2e-4,
)
# This creates momentum and variance buffers in CUDA unified memory:
# They live in GPU VRAM during normal operation
# They page to CPU RAM during memory spikes
# PagedAdamW8bit also quantizes the optimizer states to INT8,
# reducing their memory footprint by 4× compared to FP32 optimizer states

Optimizer state memory comparison:
  Standard AdamW (FP32 states, 8.4M params):
    momentum:  8.4M × 4 bytes = 33.6 MB
    variance:  8.4M × 4 bytes = 33.6 MB
    Total: 67.2 MB

  PagedAdamW8bit (INT8 states):
    momentum:  8.4M × 1 byte  = 8.4 MB
    variance:  8.4M × 1 byte  = 8.4 MB
    Total: 16.8 MB   (+ negligible scale factors per block)
```

Step 5: Data Preparation (Identical to Full Fine-Tuning and LoRA)
The data pipeline is completely unchanged from standard training. QLoRA doesn't affect how you tokenize, format, or batch your data.

5a. Raw JSONL:
{"instruction": "Summarize:", "input": "Long article...", "output": "Summary..."}

5b. Chat template formatting:
"<s>[INST] Summarize: Long article... [/INST] Summary... </s>"

5c. Tokenization:
[1, 518, 25580, 29962, 6264, 279, 675, 29901, ...] (integer IDs)

5d. Label masking:
Input tokens: [-100, -100, ..., -100] (ignored in loss)
Output tokens: [actual token IDs] (graded by loss)

5e. Padding + attention mask, collated into batch tensors

5f. batch = {
"input_ids": [batch_size, seq_len]
"attention_mask": [batch_size, seq_len]
"labels": [batch_size, seq_len]
}

Step 6: Forward Pass — Where NF4 Dequantization Happens

outputs = model(**batch)

During this forward pass, for each QLoRA-adapted linear layer:

6a. The NF4 kernel is invoked (bitsandbytes custom CUDA kernel):
- Reads 4-bit packed weights from GPU memory (fast: small data)
- Reads INT8 block-scales, reads FP32 super-scales
- Dequantizes in a fused GPU kernel: INT8_scale → FP32_scale → BF16_weight
- Performs the matrix multiplication in BF16
- Returns output in BF16

6b. Simultaneously, the LoRA path computes A × x and B × (Ax) in BF16

6c. Both paths' outputs are summed and passed to the next layer

This dequantize-compute-discard pattern repeats for every linear layer in every forward pass.
The 4-bit weights are NEVER stored as BF16 — they're reconstructed transiently.

6d. Output logits: [batch_size, seq_len, vocab_size] in BF16

Step 7: Loss Computation (Unchanged)

loss = CrossEntropyLoss(logits, labels)

Only output positions (labels ≠ -100) contribute to the loss.
The dequantization error in the frozen path shows up as a small constant bias in the loss — it doesn't prevent convergence.

Step 8: Backward Pass — Where Quantization Meets Gradients

loss.backward()

This is where QLoRA's design pays off most clearly:

8a. Gradients flow backward through the output logits to the last layer

8b. At each QLoRA layer:
- Gradients DO flow through the dequantized W_0 (as if it were BF16)
  The backward pass through the frozen path uses the same dequantized weights that were used in the forward pass (held in memory for backward computation)
- But W_0 itself accumulates NO gradient (requires_grad = False)
- Gradients flow to LoRA A and B matrices (requires_grad = True)

8c. The gradient for LoRA A:
$\partial Loss/\partial A = (\alpha/r) \times B^T \times \partial Loss/\partial output \times x^T$
Shape: [r × d_in] (e.g., [8 × 4096])

The $\partial Loss/\partial output$ term comes from the frozen path flowing backward through the dequantized W_0. This is where quantization error can theoretically corrupt gradients — but NF4's low quantization error keeps this manageable.

8d. Memory used during backward:
LoRA gradients: ~33 MB (for rank=16, all-linear targets)
Activation buffer: varies (depends on gradient checkpointing settings)
NO storage for W_0 gradients (they're never computed/stored)

Step 9: Optimizer Step — Only LoRA Parameters Update

optimizer.step()

The paged AdamW8bit optimizer updates only the LoRA A and B matrices:
For each LoRA parameter θ:

$g = \partial Loss/\partial \theta$ (BF16 gradient)
$m = \beta_1 \times m + (1-\beta_1) \times g$ (momentum, stored INT8, dequant for update)
$v = \beta_2 \times v + (1-\beta_2) \times g^2$ (variance, stored INT8, dequant for update)
$\theta \leftarrow \theta - lr \times \hat{m} / (\sqrt{\hat{v}} + \varepsilon)$ (update in BF16)

If a memory spike occurs during this step:
The paged memory for m and v gets offloaded to CPU RAM
The update computation pauses briefly
After the spike, states page back to GPU
Training continues

optimizer.zero_grad() # Clear the ~33 MB of LoRA gradients

Step 10: Gradient Checkpointing Integration

QLoRA strongly recommends enabling gradient checkpointing:

model.gradient_checkpointing_enable()

Without gradient checkpointing:
All layer activations must be kept in memory during the forward pass
so they're available for the backward pass.
For a 7B model: activations can easily reach 10-30 GB for longer sequences.

With gradient checkpointing:
Only checkpoint activations are kept (every N layers, configurable).
During backward pass, the sections between checkpoints are recomputed.
Memory reduction: 4-8× for activations.
Compute cost: ~30-35% more compute per step (each non-checkpointed activation
is computed twice: once in forward, once in backward recomputation).

For QLoRA, the interaction with quantization:
During recomputation, the NF4 dequantization happens AGAIN for those layers.
This is handled correctly by bitsandbytes — the same NF4 → BF16 kernel runs.
The recomputed activations are numerically identical because the NF4 weights
are deterministic (no randomness in dequantization).

Step 11: Save — Only the Adapter (Identical to LoRA)

After training, save ONLY the LoRA adapter:

```
model.save_pretrained("./qlora-adapter/")
```

Saved files:
adapter_model.safetensors ~33-300 MB (all A and B matrices, in BF16)
adapter_config.json ~1 KB

The NF4 base model is NOT saved by default — it's available on HuggingFace Hub.
Only the tiny adapter delta is saved.

adapter_config.json:

```
{
  "base_model_name_or_path": "meta-llama/Llama-2-7b-hf",
  "r": 16,
  "lora_alpha": 32,
  "target_modules": "all-linear",
  "lora_dropout": 0.05,
  "bias": "none",
  "task_type": "CAUSAL_LM"
}
```

Step 12: Inference — Merge or Keep Separate

Option A: Keep adapter separate (for adapter swapping)

```
base = AutoModelForCausalLM.from_pretrained(
    "meta-llama/Llama-2-7b-hf",
    quantization_config=bnb_config,   # Still use 4-bit at inference
)
model = PeftModel.from_pretrained(base, "./qlora-adapter/")
# Inference in 4-bit base + BF16 LoRA: same memory as training, minimal overhead
```

Option B: Merge adapter into base model (for zero-overhead inference)

```
# Step 1: Load model in BF16 (not 4-bit) for merging
base = AutoModelForCausalLM.from_pretrained(
    "meta-llama/Llama-2-7b-hf",
    torch_dtype=torch.bfloat16,       # Full precision for merge
)
model = PeftModel.from_pretrained(base, "./qlora-adapter/")

# Step 2: Merge: W_merged = W₀_bf16 + (α/r) · B × A
model = model.merge_and_unload()
```

```
# Step 3: Optionally re-quantize for deployment
# Or save as BF16 for inference on larger GPUs
model.save_pretrained("./merged-model-bf16/")
```

Note on merge-then-requantize:

It's common to merge the BF16 LoRA into the BF16 base, then re-quantize to GGUF or GPTQ for efficient deployment. The merged model has the adapter's learning "baked in" and can be quantized like any standard model afterward.

MEMORY BREAKDOWN — QLORA VS. LORA VS. FULL FINE-TUNING
(Concrete Numbers for a 7B and 70B Model)

7B Model (e.g., LLaMA-2-7B), batch=4, seq=512, rank=16

MEMORY COMPONENT	Full FT	LoRA (BF16)	QLoRA (NF4)
Base weights	14 GB	14 GB	3.5 GB ✓
Gradients (base)	14 GB	0 GB ✓	0 GB ✓
Optimizer states (base)	56 GB	0 GB ✓	0 GB ✓
LoRA adapter weights	N/A	~0.1 GB	~0.1 GB
LoRA gradients	N/A	~0.1 GB	~0.1 GB
LoRA optimizer states	N/A	~0.3 GB	~0.08 GB ✓
Activations (grad. ckpt)	~5 GB	~5 GB	~3.5 GB ✓
Total	~89-109 GB	~19-23 GB	~7-11 GB ✓
Required hardware	8× A100	1-2× A100	1× RTX 3090/4090

70B Model (e.g., LLaMA-2-70B), batch=2, seq=512, rank=16

MEMORY COMPONENT	Full FT	LoRA (BF16)	QLoRA (NF4)
Base weights	140 GB	140 GB	35 GB ✓
Gradients (base)	140 GB	0 GB ✓	0 GB ✓
Optimizer states (base)	560 GB	0 GB ✓	0 GB ✓
LoRA adapter weights	N/A	~0.5 GB	~0.5 GB
LoRA gradients	N/A	~0.5 GB	~0.5 GB
LoRA optimizer states	N/A	~2.0 GB	~0.5 GB ✓
Activations (grad. ckpt)	~20 GB	~15 GB	~10 GB ✓
Total	~860 GB+	~160 GB	~46 GB ✓
Required hardware	16-20× A100	2× A100 (80G)	1× A100 (80G) ✓

QLoRA is what makes 70B model training accessible to teams without massive GPU clusters.

QLORA VISUAL DIAGRAMS — COMPLETE BREAKDOWN

DIAGRAM 1: HOW NF4 COMPRESSES A WEIGHT MATRIX

```
ORIGINAL WEIGHT MATRIX W_q: [4096 × 4096]
Stored in BF16: 4096 × 4096 × 2 bytes = 33,554,432 bytes = 32 MB

 -0.032  0.018  -0.007  0.041  -0.089  0.003  0.027  ...  (BF16 floats)
  0.011 -0.045   0.033 -0.019   0.056 -0.011  0.008  ...
  ...

                          ↓ QUANTIZE (block-wise, block_size=64)

QUANTIZED WEIGHT MATRIX W_q in NF4:
NF4 codes: 4096 × 4096 × 0.5 bytes  =  8 MB   (4 bits per weight)
Scales:    16,777,216 / 64 blocks × 4 bytes = 1 MB  (FP32 per block)
After DQ:  scales compressed to ~0.25 MB in INT8
Total:     8 + 0.25 = ~8.25 MB  (vs 32 MB original) = 3.88× compression

 Block 0:  scale=0.089  codes=[7, 11, 8, 13, 4, 9, 11, 7, ...]  (64 values)
 Block 1:  scale=0.056  codes=[9, 6, 11, 8, 12, 8, 9, 8, ...]   (64 values)
 ...       (262,144 total blocks for this matrix)

                          ↓ DEQUANTIZE (during forward pass)

RECOVERED W_q in BF16:
 -0.031  0.017  -0.006  0.041  -0.089  0.003  0.027  ...  (BF16, approximate)
  0.011 -0.044   0.033 -0.019   0.057 -0.011  0.008  ...
  ...           (small quantization errors — values ~0.001 off)

Error: each weight ≈ ±0.001 off from original. Small enough to not destroy learning.
```

DIAGRAM 2: INT4 vs. NF4 — WHY NF4 IS BETTER FOR NEURAL NETWORKS

```
Weight distribution (from a real LLM layer, normalized):

                    ▄▄▄
                   ▄▄▄▄▄
                  ▄▄▄▄▄▄▄
                 ▄▄▄▄▄▄▄▄▄
                ▄▄▄▄▄▄▄▄▄▄▄
               ▄▄▄▄▄▄▄▄▄▄▄▄▄
              ▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄
             ▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄▄
  ──────────────────────────────────
     -3.0   -2.0   -1.0    0.0    1.0    2.0    3.0

INT4 quantization levels (16 evenly spaced from -8 to 7, normalized to [-1,1]):

  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
-1.0        -0.5        0.0        0.5        1.0        2.0

Problem: equally-spaced levels waste resolution on extreme values where
almost no weights live. Most weight values are represented by only 4-6 levels.

NF4 quantization levels (quantile-based, more levels near 0):

 | | | | |  |   |       |   |  |  | | | |
-1.0  -0.5          0.0          0.5       1.0

NF4 concentrates quantization levels where most weights actually live.
Result: same 4 bits, dramatically lower average quantization error.

Quantization SNR comparison (higher = better quality):
    INT4:  ~21.0 dB
    NF4:   ~24.5 dB    (+3.5 dB, roughly 1.5× better signal/noise ratio)
```

DIAGRAM 3: QLORA LAYER ARCHITECTURE — FROZEN NF4 BASE + BF16 LORA ADAPTERS

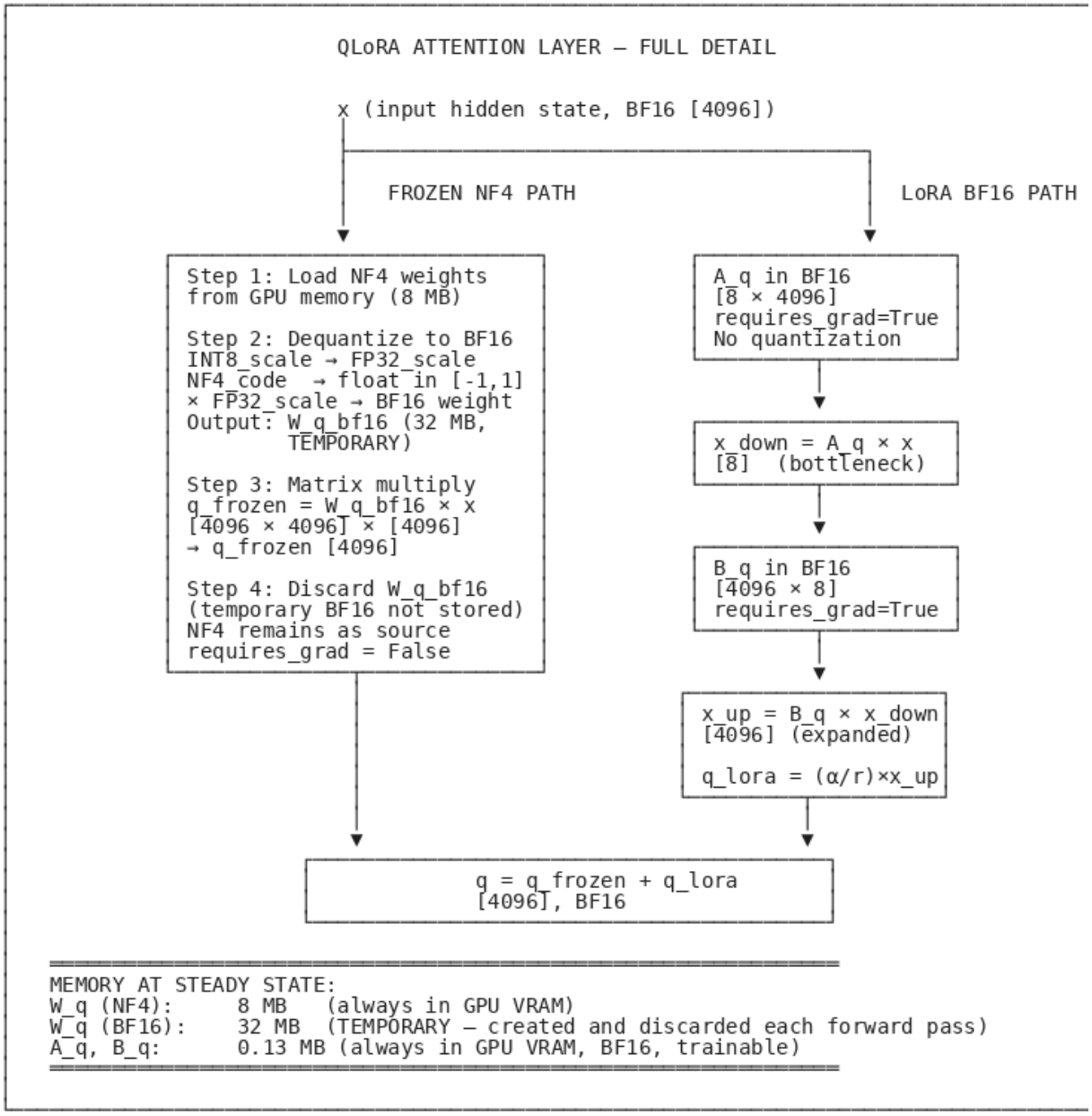


DIAGRAM 4: BACKWARD PASS — GRADIENT FLOW IN QLORA

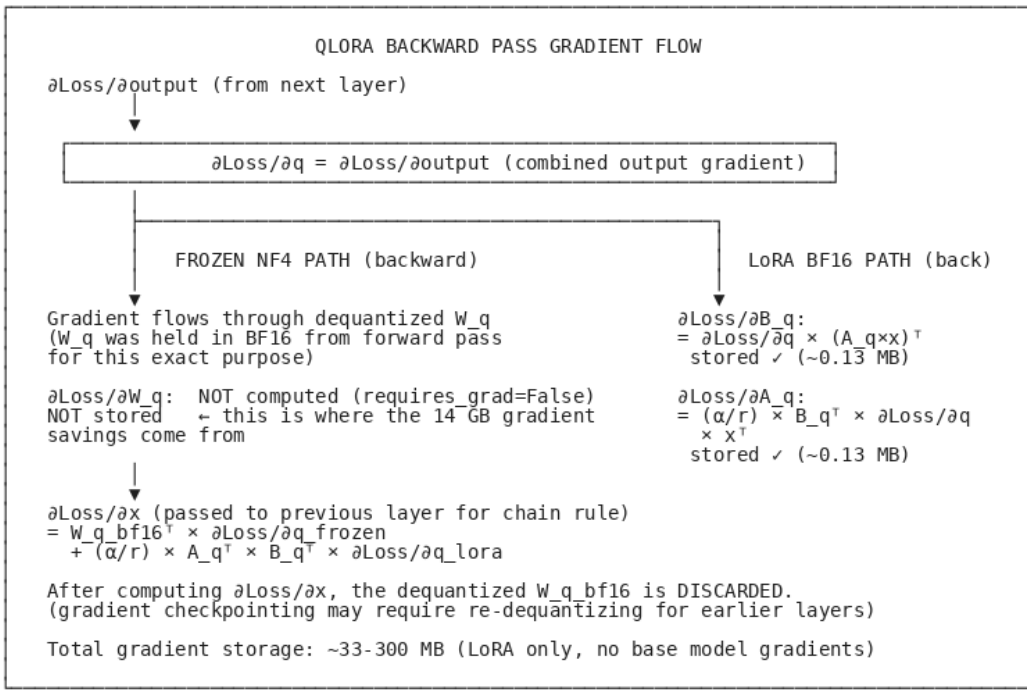


DIAGRAM 5: PAGED OPTIMIZER — MEMORY UNDER PRESSURE

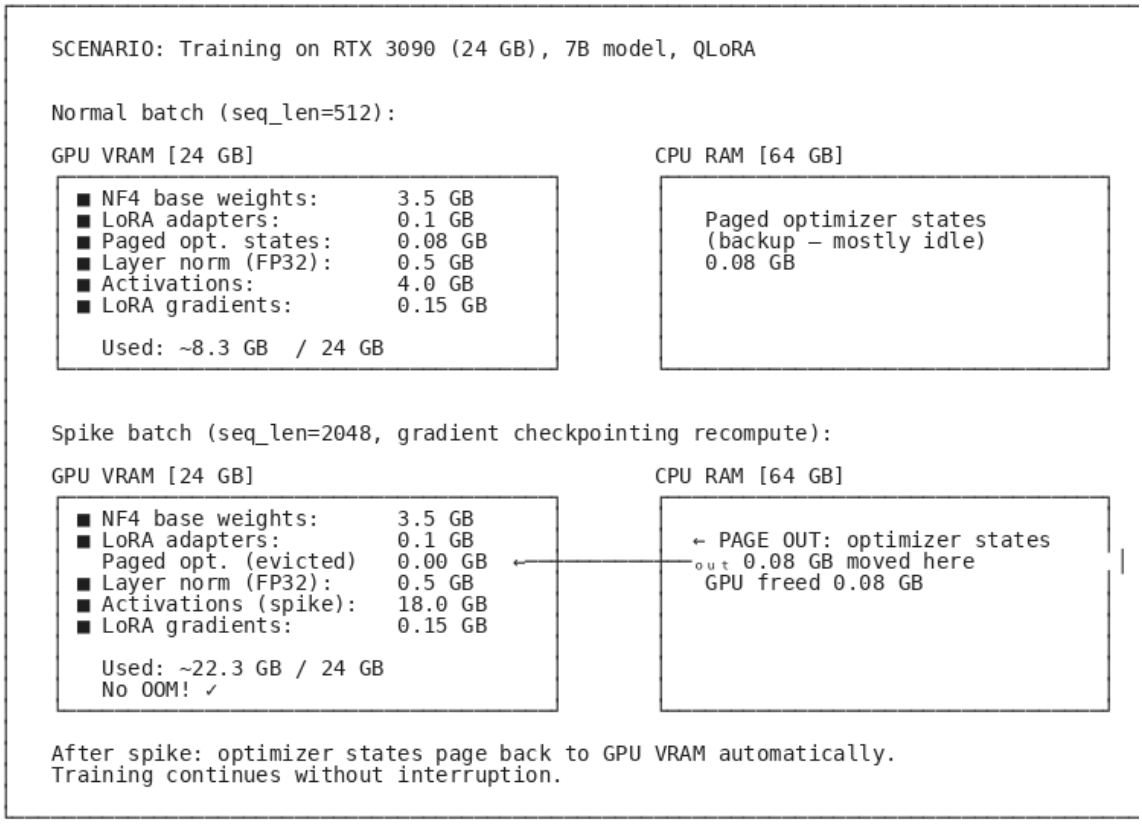
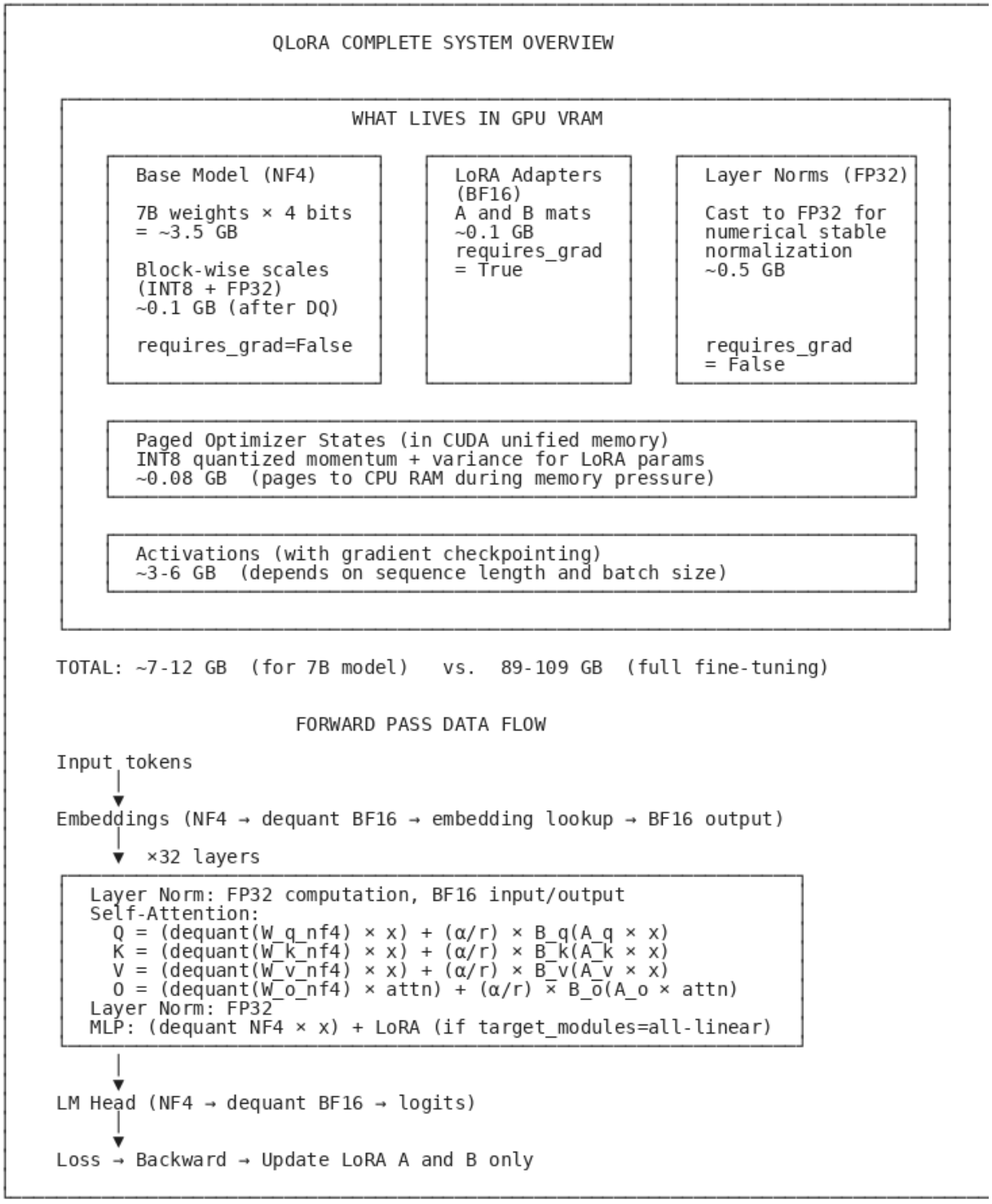


DIAGRAM 6: QLORA COMPLETE SYSTEM — ALL FOUR INNOVATIONS TOGETHER



QLORA HYPERPARAMETERS IN DEPTH (What to Set, Why, and What Breaks If You Get It Wrong)
BitsAndBytes Quantization Config

bnb_4bit_quant_type: "nf4" vs "fp4"

"nf4": NormalFloat4 — use this. Optimal for normally-distributed weights.
Quantile-based, information-theoretically superior to uniform quantization.
The standard for QLoRA fine-tuning.

"fp4": FP4 (floating-point 4-bit) — a floating-point format with tiny mantissa.
Used more in inference scenarios. Slightly lower quality for training.
Avoid unless you have a specific reason to use it.

bnb_4bit_compute_dtype: torch.bfloat16 vs torch.float16

BF16: Wider dynamic range (same exponent bits as FP32), lower precision.
Standard for training on Ampere (A100, RTX 3090) and newer.
Handles gradient explosions better.

FP16: Higher precision, smaller range. Can overflow during training.
Use BF16 unless your hardware doesn't support it (pre-Ampere).

bnb_4bit_use_double_quant: True vs False

True: Enable double quantization — saves ~375 MB on a 7B model.
Recommended: almost always worth it, negligible quality impact.

False: Disable — marginally simpler dequantization, saves a tiny compute cost.
Only disable if you're debugging or have plenty of GPU memory.

LoRA Config for QLoRA
Rank (r): Same considerations as standard LoRA, but start with higher ranks in QLoRA because the quantized base has more noise to overcome.

QLoRA recommended ranges:
r=16: Good starting point for instruction tuning on 7B-13B models
r=32: Better for larger domain gaps or models with more capacity
r=64: Use when r=32 still underperforms; high memory cost
r=8: Only for very simple tasks or extreme memory constraints

Alpha (α): Set equal to rank (α=r) or double (α=2r).
Common for QLoRA: lora_alpha = 2 × r

target_modules:
"all-linear": Most expressive, highest memory cost
Recommended for QLoRA on 7B (still fits easily)
["q_proj","v_proj"] : Minimal — good for memory-constrained setups
["q_proj","k_proj","v_proj","o_proj"] : Standard — good balance

Learning Rate

QLoRA uses the same learning rates as LoRA (higher than full fine-tuning):

Typical range	:	1e-4 to 3e-4
Common default	:	2e-4

Because QLoRA's base model has quantization error, gradients may be noisier.
Some practitioners use slightly lower learning rates (1e-4) for QLoRA vs LoRA.
But this is a minor effect — start with 2e-4 and tune if needed.

Batch Size and Gradient Accumulation

QLoRA enables smaller per-device batch sizes due to memory savings:

BATCH SIZE COMPARISON (7B model, RTX 3090 24GB, seq_len=512)

Method	Max batch/GPU	Grad Accum	Effective Batch
Full FT (BF16)	N/A (OOM)	N/A	N/A
LoRA (BF16)	2-4	8	16-32
QLoRA (NF4)	4-8	4-8	16-64

QLoRA's smaller base model footprint enables larger effective batches
or longer sequences at the same GPU tier.

Gradient Checkpointing

Always enable gradient checkpointing with QLoRA:

```
model.gradient_checkpointing_enable()
```

Without it, activations for all 32 layers must reside in memory simultaneously.
At sequence length 512 with batch size 4, this can be 8-15 GB of activation memory alone.

With gradient checkpointing:

- Activations stored: every N layers (default: every layer, recomputing N-1 layers)
- Memory cost: ~3-5 GB instead of 8-15 GB
- Compute cost: ~30-35% more compute per step

For QLoRA, the recomputation includes re-running the NF4 dequantization.
This is deterministic and correct — no numerical issues.

QLORA QUALITY — WHAT'S LOST, AND WHEN IT MATTERS

The Quality Stack

Full Fine-Tuning on BF16 base	→	Baseline (100% quality)
LoRA on BF16 base	→	95-99% of full FT
QLoRA (NF4 base + BF16 LoRA)	→	94-99% of full FT (~1-3% below LoRA in most benchmarks)

The quality gap between LoRA and QLoRA is small — typically 1-4 points on standard benchmarks depending on model size, task, and rank. The original QLoRA paper showed their Guanaco models trained with QLoRA matched or exceeded ChatGPT on Vicuna benchmarks, despite running on a fraction of the hardware.

Where QLoRA Closes the Gap

- Larger models: 70B QLoRA ≈ 70B LoRA (quantization error is proportionally smaller relative to the model's representational capacity)
- Higher ranks: r=64 QLoRA compensates more for quantization noise
- More training data: longer training allows adapters to compensate for base model noise
- Lower-stake tasks: instruction following, summarization, classification (where exact precision matters less)

Where the Gap Widens

- Smaller models (7B): quantization error is larger relative to model capacity
- Mathematical reasoning: requires exact weight precision for multi-step chains
- Coding tasks with strict syntax: quantization errors can subtly corrupt code logic
- Very low ranks (r=4): adapters have too few parameters to compensate for base noise
- Short fine-tuning runs: not enough gradient steps for adapters to adapt to NF4 noise

Practical Decision Guide:

USE QLoRA WHEN:	USE LoRA (BF16) WHEN:
GPU < 24 GB	GPU ≥ 24 GB for 7B model
Model > 13B	Quality difference matters
Consumer GPU (3090, 4090)	Mathematical/coding tasks
Budget constraints	Production deployment
Experimenting / prototyping	Rank < 8 (less buffer for noise)
Instruction tuning / chat tasks	
70B model on single A100 80G	

QLORA FAILURE MODES & COMMON PITFALLS

1. Using FP16 compute dtype instead of BF16

bnb_4bit_compute_dtype=torch.float16 ← can cause NaN/Inf during training

FP16 has a narrow dynamic range (max ~65,504). Gradients in LLM training can exceed this range. BF16 shares FP32's exponent width and handles this safely.
Always use BF16 as compute dtype with QLoRA.

2. Forgetting prepare_model_for_kbit_training()

Skipping this step means:
- Layer norms stay in BF16 (numerical instability)
- Gradient checkpointing not properly configured for quantized layers
- Potential NaN losses or very slow convergence

Always call prepare_model_for_kbit_training() before applying LoRA.

3. Training with Flash Attention disabled on long sequences

Flash Attention is especially important for QLoRA because:
- Long sequences are often the reason you need QLoRA (memory constraints)
- Standard attention materializes the full attention matrix [seq × seq] in memory
- Flash Attention avoids this entirely, enabling much longer sequences

```
model = AutoModelForCausalLM.from_pretrained(
    ...,
    attn_implementation="flash_attention_2",    # Requires flash-attn package
)
```

4. Merging the adapter while the base is still 4-bit

```
# WRONG — merges adapter into the quantized base (poor quality):
model.merge_and_unload()   ← when model is still in NF4

# CORRECT — reload base in BF16, then merge:
base = AutoModelForCausalLM.from_pretrained(..., torch_dtype=torch.bfloat16)
model = PeftModel.from_pretrained(base, "./adapter/")
model = model.merge_and_unload()   # Now merges cleanly in BF16
```

Merging into a quantized base produces degraded results because the merge operation $W = W_0 + (\alpha/r) \cdot BA$ runs on quantized W_0 values, amplifying errors.

5. Wrong bnb_4bit_quant_type

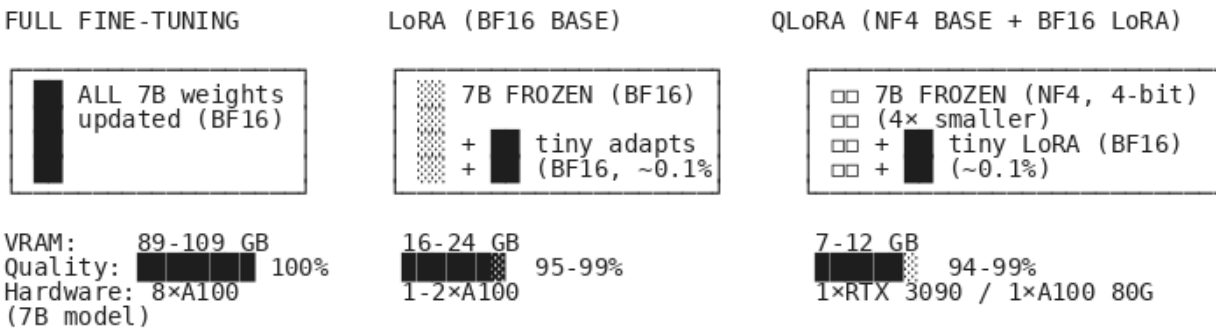
Using "fp4" instead of "nf4" for training is a common mistake.
NF4 was specifically designed and shown to be superior for neural network weight distributions. FP4 is mainly an inference format. Use NF4 for fine-tuning.

6. Not enabling gradient checkpointing

Without gradient checkpointing, activations accumulate across all 32 layers.
This often causes OOM even with QLoRA because activation memory is not reduced by quantization (activations are always in BF16).

```
model.gradient_checkpointing_enable()   # Always include this
```

SUMMARY MENTAL MODEL



The Four Things QLoRA Does:

1. NF4 quantization: pack 7B weights into 3.5 GB instead of 14 GB
   (information-optimal: more quantization levels near zero, where most weights live)

2. Double quantization: quantize the block-scale factors themselves
   (saves an extra ~375 MB, marginal quality impact)

3. Paged optimizers: store optimizer states in unified memory
   (prevents OOM during memory spikes, enables longer sequences)

4. BF16 LoRA adapters on top: train normally in BF16
   (dequantize NF4 → BF16 on-the-fly for compute, gradients only for tiny A and B matrices)

QLoRA didn't invent new model architectures or new training objectives.
It took existing tools — quantization, LoRA, paging — and combined them in exactly the right way to push the frontier of what's trainable on a single GPU.