How does Fine-Tuning Actually Work?

(And when should you bother?)

Brendan Beh
AISEA - Co-founder
brendan.bytein@gmail.com

Shared under AISEA's learning commons. Built for the community. Please share, not sell.

Introduction

AISEA

What does it mean to fine-tune an LLM?

Conceptually: an LLM is a function that maps token sequences to token distributions.

To fine tune is to change this mapping function f(x).

AISEA
Shared under AISEA's learning commons. Built for the community. Please share, not sell.

The LLM neural network architecture

Key

Query

The
toilet
is

MLP / FFN

...

Attention

Value

Transformer block

More transformer blocks

Output

clogged
nearby
cozy
new
fragrant
clean
dry

Attention Layer
Which tokens matter to which*

Feed-Forward Network
Factual associations*

*Generally as a working model

Shared under AISEA's learning commons. Built for the community. Please share, not sell.

The LLM neural network architecture

Key

Query

The

toilet

is

Attention

Value

MLP / FFN

...

Transformer block

More transformer blocks

Output

clogged

nearby

cozy

new

fragrant

clean

dry

Attention Layer

Which tokens matter to which*

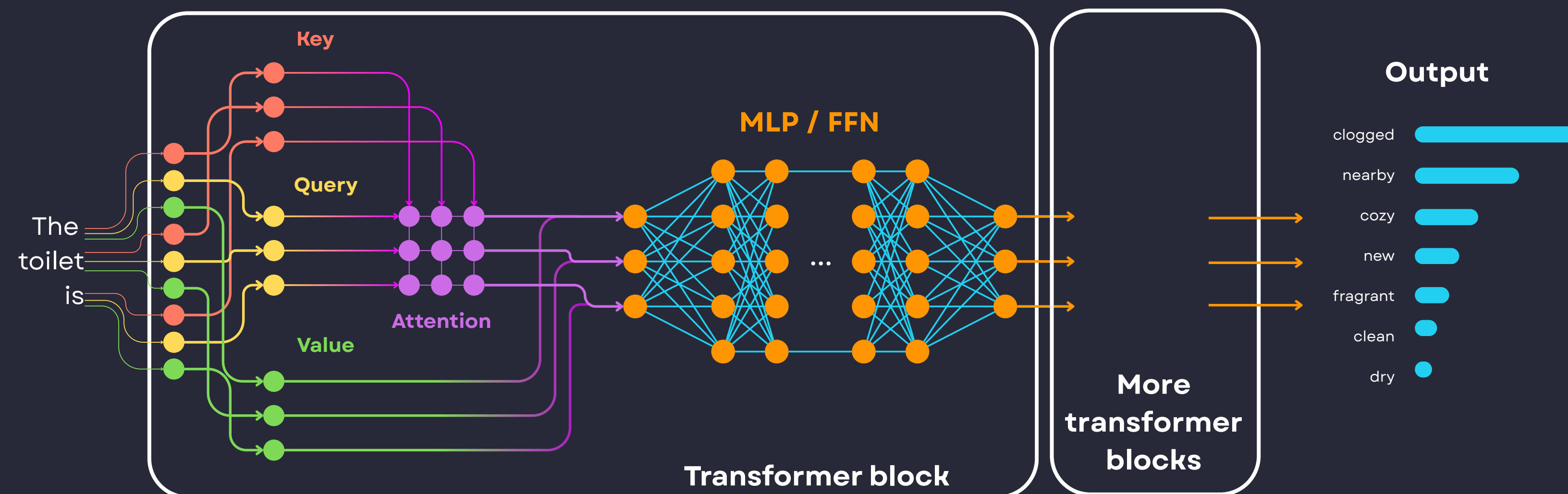
Feed-Forward Network

Factual associations*

Mostly FT this

Shared under AISEA's learning commons. Built for the community. Please share, not sell.

Mathematically...



$$\text{Attention}(Q,K,V) = \text{softmax}\left(\frac{\mathbf{QK}^T}{\sqrt{d_k}}\right)\mathbf{V}$$
$$= \text{softmax}\left(\frac{\mathbf{xQK}^T\mathbf{x}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

$$\mathbf{K} = \mathbf{xW}_k$$
$$\mathbf{V} = \mathbf{xW}_v$$
$$\mathbf{Q} = \mathbf{xW}_q$$

$$\begin{pmatrix} 0.2 \\ 2.1 \\ 3.9 \\ \dots \\ 2.4 \\ 7.7 \end{pmatrix} = \begin{pmatrix} 1.0 & -2.4 & 9.8 & 7.5 & .. & -6.2 & 3.6 \\ -2.5 & -4.4 & -8.7 & 9.6 & \dots & -3.1 & 8.8 \\ -7.3 & 3.5 & 9.3 & -4.4 & \dots & 2.5 & 8.3 \\ \dots & & & & & & \\ 3.2 & 2.4 & -0.6 & 2.5 & \dots & 7.3 & -0.7 \end{pmatrix} \times \begin{pmatrix} 2.9 \\ 8.3 \\ 4.1 \\ \dots \\ 3.3 \\ 6.9 \end{pmatrix}$$

Probabilities (Output)

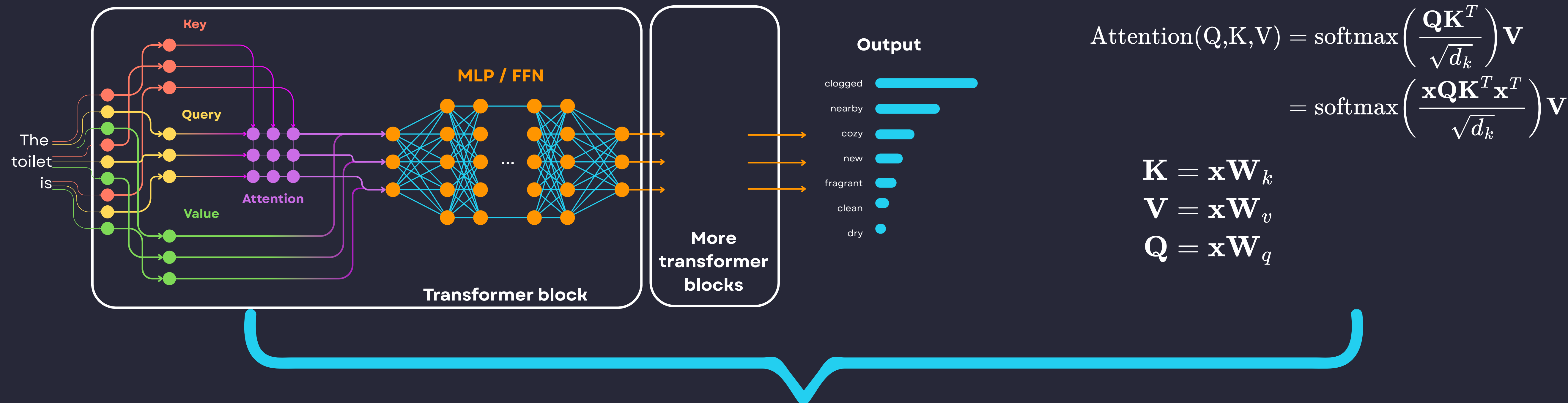
The **matrix** captures the weights → maps input to output

Embeddings (Input)

AISEA

Shared under AISEA's learning commons. Built for the community. Please share, not sell.

A model is dictated by its weights



$$\begin{pmatrix} 1.0 & -2.4 & 9.8 & 7.5 & .. & -6.2 & 3.6 \\ -2.5 & -4.4 & -8.7 & 9.6 & \dots & -3.1 & 8.8 \\ -7.3 & 3.5 & 9.3 & -4.4 & \dots & 2.5 & 8.3 \\ \dots & & & & & & \\ 3.2 & 2.4 & -0.6 & 2.5 & \dots & 7.3 & -0.7 \end{pmatrix}$$

GPT 4o:

~200 Billion

of these numbers

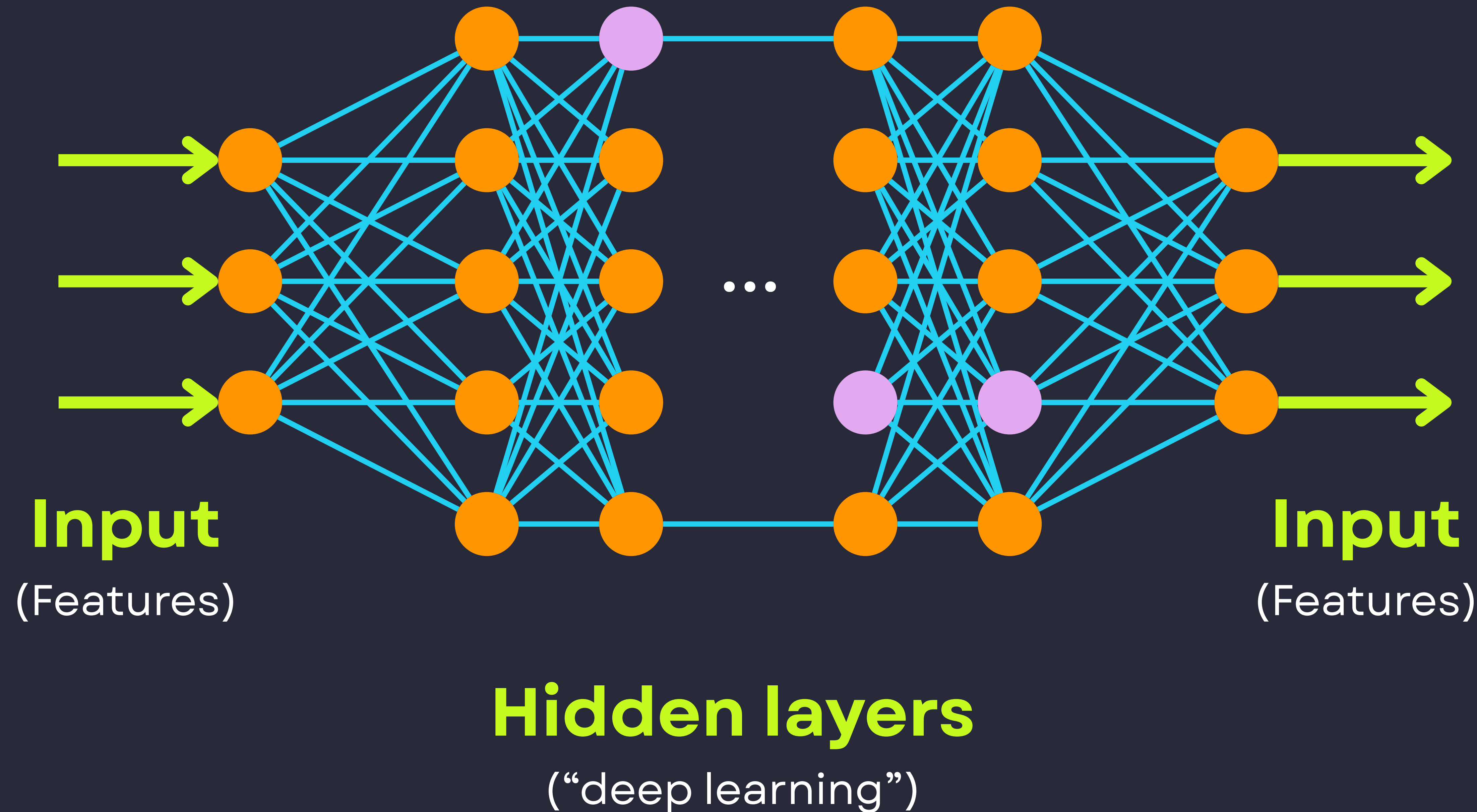
Defined during original training phase

The **matrix** captures the weights
→ maps input to output

AISEA

Shared under AISEA's learning commons. Built for the community. Please share, not sell.

A model is dictated by its weights



Most of this brain is fixed and pre-taught.

Fine-tuning (usually) means changing only parts of the brain.

This means you're doing "brain surgery" (changing parts of an already existing thinking model) .

We cannot change everything.

What's the minimum we can do to get the model to perform better?

AISEA

Shared under AI.SEA's learning commons. Built for the community. Please share, not sell.

How do models learn?

AISEA

How do you train a model in the first place?

To train a model means you want it to be wrong as little as possible. We **measure how wrong it is by the loss function, L.**

AISEA
Shared under AISEA's learning commons. Built for the community. Please share, not sell.

How do you train a model in the first place?

Loss measures the gap between the predicted distribution and what actually came next in the training data.

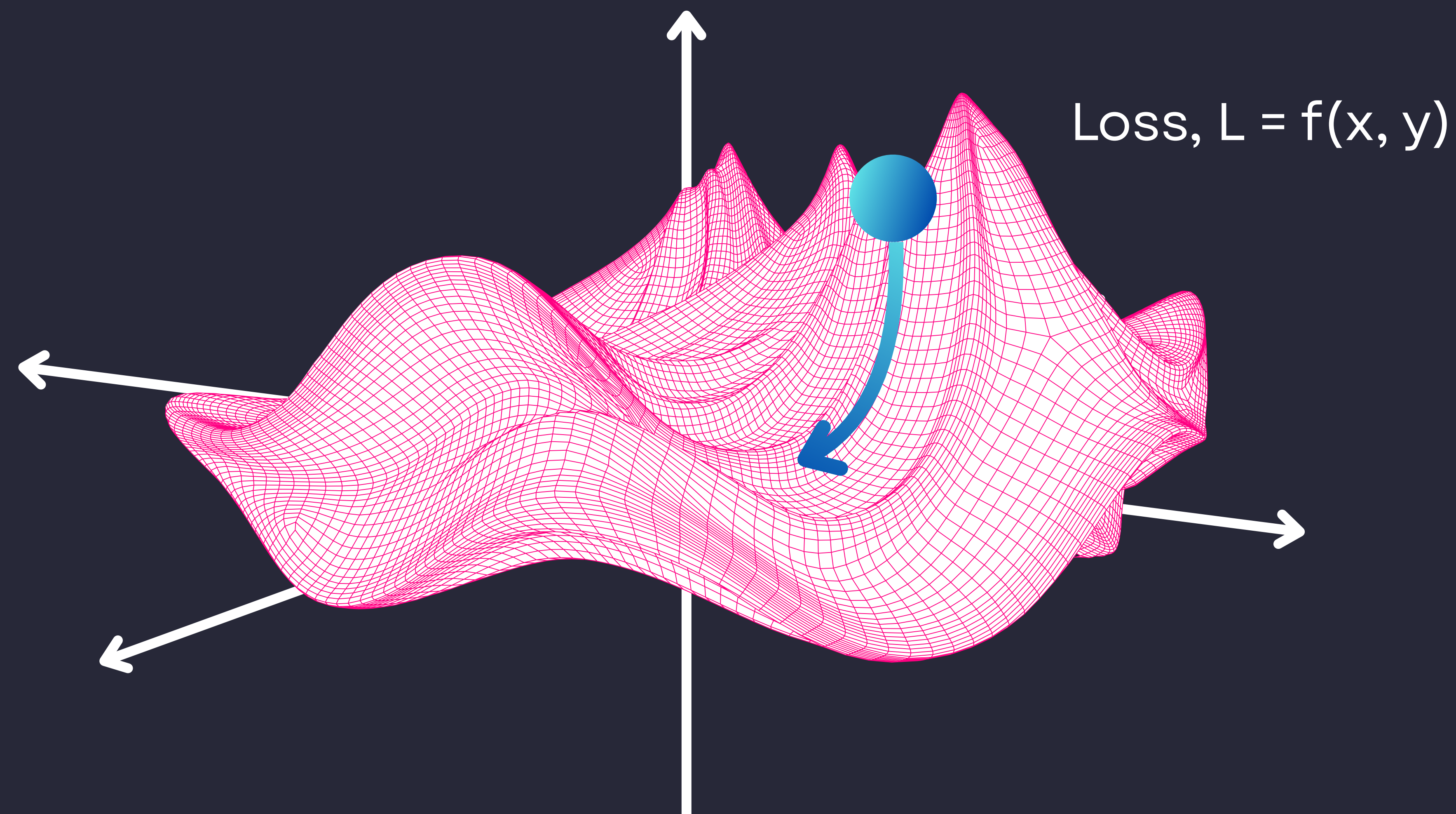
$$L(y, \hat{y}) = -log[P(\text{correct token})]$$

Our end goal is to minimize the loss function.

AISEA

Shared under AISEA's learning commons. Built for the community. Please share, not sell.

Gradient descent



In multiple dimensions:
loss ~ surface in weight space

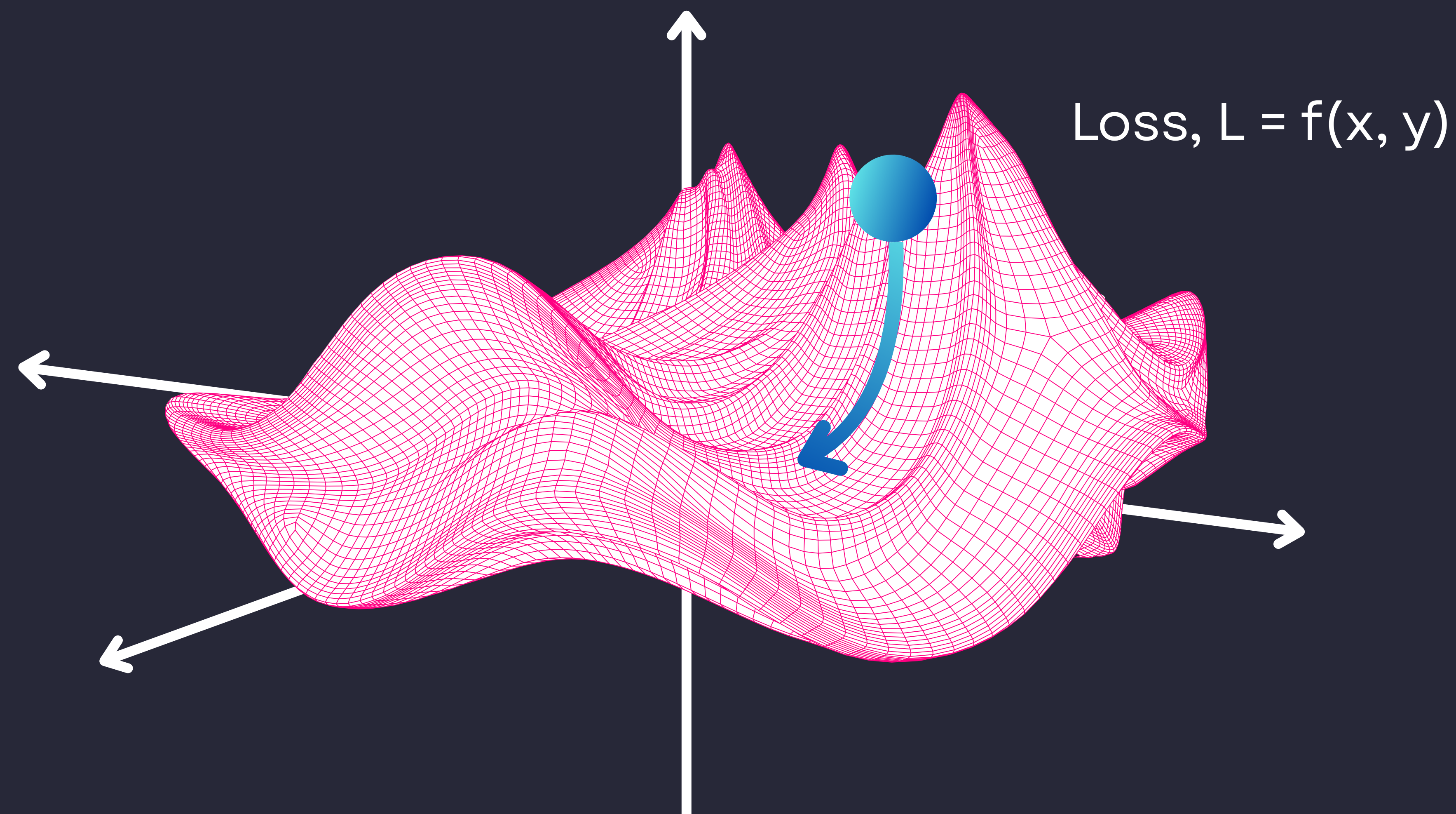
Given a config of weights, we get a given loss. We need to find where that surface is lowest, i.e. the weight combination that is the most accurate.

Do this through **gradient descent**. The gradient tells you the slope at your current position. It points uphill. So you take a step in the opposite direction downhill.

Update rule:
New weight = Old weight − (learning rate × gradient)

Shared under AISEA's learning commons. Built for the community. Please share, not sell.

Gradient descent - more mathematically



Once you have a function, you can ask: if I nudge one of the inputs by a tiny amount, how much does the output change?

For the loss function with respect to one weight W:

$$\frac{\partial L}{\partial W} \approx \frac{L(W+\epsilon) - L(W)}{\epsilon}$$

I.e.

1. perturb that weight by a tiny amount ε
2. recompute the loss, see how much it moved
3. divide by ε.

That ratio is the gradient for that one weight.

But you can't do this naively for 7 billion weights!

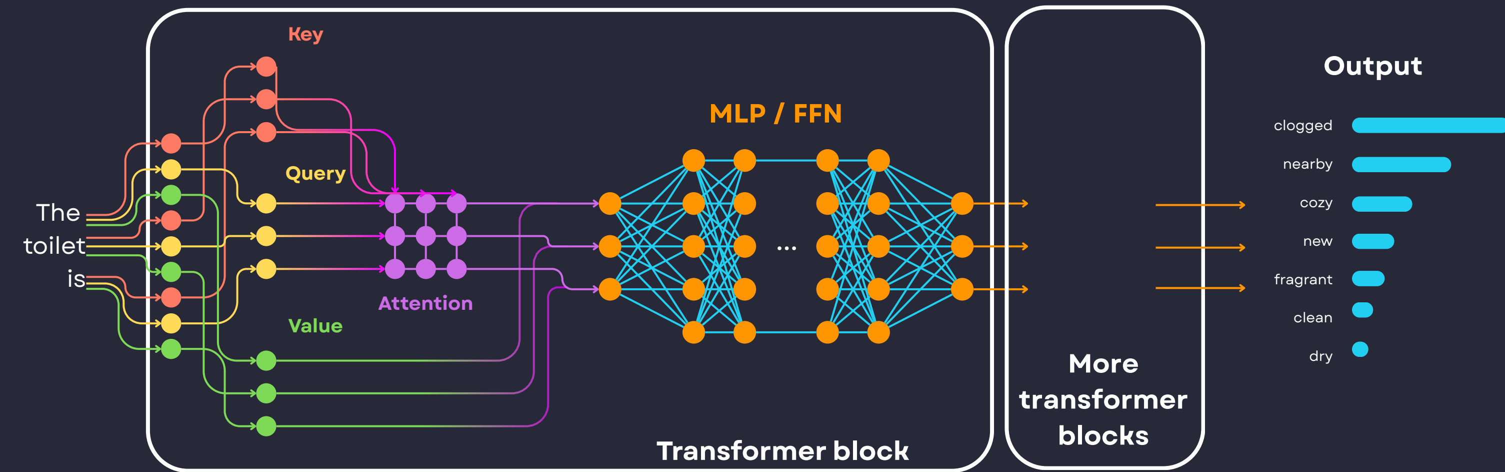
AISEA

Shared under AISEA's learning commons. Built for the community. Please share, not sell.

You can find the loss, but which neurons caused the mistake?

In a model with 7 billion weights, you have 7 billion slopes to compute: which weights caused the loss to be high?

Solve that using **backpropagation**: work backwards.



Which of these neurons...

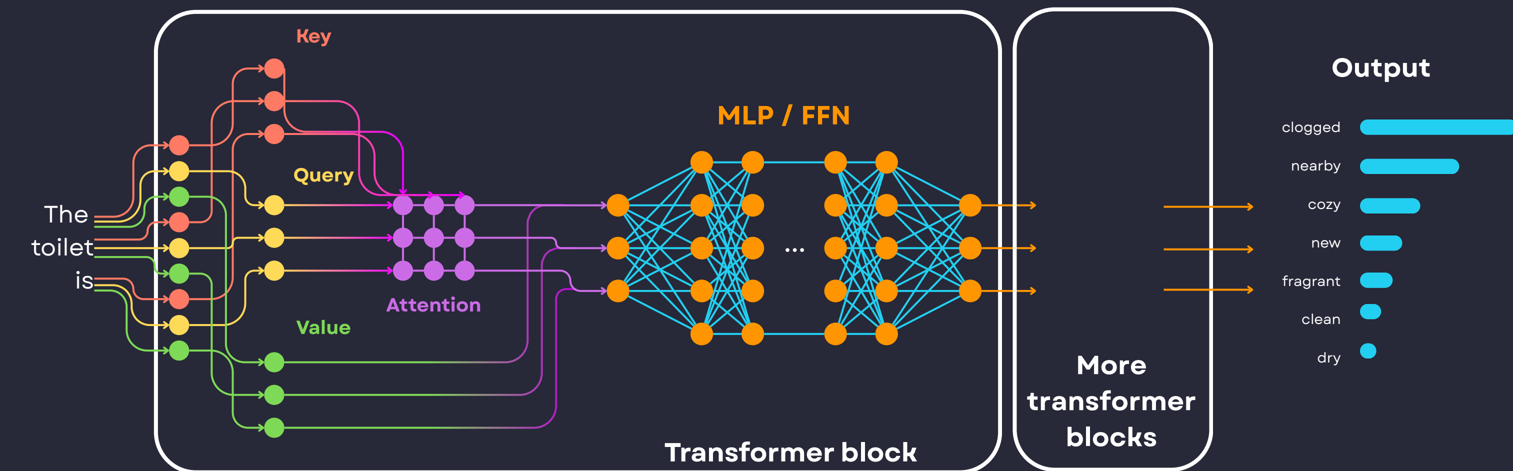
...contributed to this output?

Shared under AISEA's learning commons. Built for the community. Please share, not sell.

You can find the loss, but which neurons caused the mistake?

Start at the output (wrong prediction) and propagate the error signal backwards through every layer back to the input.

For each weight it passes through, it computes a gradient.



Which of these neurons...

...contributed to this output?

- Positive gradient: this weight is making things worse, decrease it
- Negative gradient: decreasing this weight makes things better, increase it.

AISEA

Shared under AISEA's learning commons. Built for the community. Please share, not sell.

The theory behind FT

AISEA

Before we start training, what do we want the model to do?

Goal:

Given an input, predict the correct output: minimise the gap between prediction and ground truth.

You can choose your training objective based on what you want the model to do.

SFT

(Supervised Fine Tuning):

You have input/output pairs with known correct outputs. The model learns to reproduce your outputs. Typically what you do when you FT.

DPO

(Direct Preference Optimisation):

No single correct answer, instead have pairs: a preferred response and a rejected response for the same input. The model learns to favour the chosen one.

This is how you teach nuanced preferences – "be more concise," "be less sycophantic" – things that are hard to express as a single correct output. Used heavily in alignment.

Continued Pretraining:

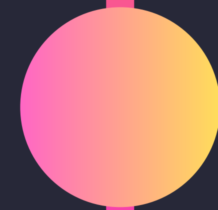
No instruction format at all - just raw text, same as the original pretraining objective. You feed it domain documents e.g. medical literature, legal text, etc. and it absorbs the patterns.

The model doesn't learn to follow instructions; it learns the language and knowledge of a domain. Used when a model fundamentally doesn't know a domain well enough for SFT to work on top of it.

How much do you change?

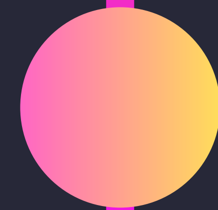
Change nothing

Text prompting



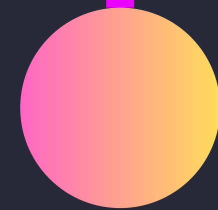
Change almost nothing

Prompt tuning



Change something

LoRA / QLoRA



Change everything

Full FT

Shared under AISEA's learning commons. Built for the community. Please share, not sell.

Text prompting: give the model examples

Don't change the model - put examples in the prompt for the model to pattern match.

```
from openai import OpenAI

client = OpenAI()

SYSTEM_PROMPT = """
You are a customer support assistant for an e-commerce company.

Your responses should match the style demonstrated in the examples below.

Style requirements:
- short, polite, specific
- solution-oriented
- no unnecessary explanation

Example
User: My package is late and tracking has not updated.
Assistant: Sorry about the delay. Please send your order number and I'll check the latest tracking
status for you.

"""

user_message = """
I ordered wireless earbuds two weeks ago and tracking has not updated in 5 days.
"""

response = client.chat.completions.create(
    model="gpt-4.1-mini",
    messages=[
        {"role": "system", "content": SYSTEM_PROMPT},
        {"role": "user", "content": user_message}
    ],
    temperature=0.3
)

print(response.choices[0].message.content)
```

By giving examples, we skew the probability distribution of the output to match the examples.

This is the basic level. The model is unchanged, but examples in the context window bias the prediction.

Shared under AISEA's learning commons. Built for the community. Please share, not sell.

Text prompting: when yo use it

When it works best	When it breaks
You have a capable base model and the task is well within its general knowledge	The behaviour you want is too consistent to maintain across thousands of calls
Format or tone changes that can be demonstrated with 2–3 examples	Your examples eat too much of the context window, leaving little room for actual input
Rapid iteration	The style or format is too subtle to convey in a few examples
One-off or low-volume tasks where training overhead isn't justified	You're paying per token and the system prompt is 2,000 tokens on every call
The task is already close to what the model does naturally	The model pattern-matches the examples but generalises badly to edge cases

AISEA

Shared under AISEA's learning commons. Built for the community. Please share, not sell.

Prompt tuning: the gaslighting method

Conceptually: attach a set of numbers before the message to skew the model, change these numbers to fit the model

User input

"Help me find the best restaurants in Spain"

Embedding

$$\begin{pmatrix} -1.1 \\ 4.2 \\ 3.5 \\ \dots \\ -12.4 \\ -9.4 \end{pmatrix} + \begin{pmatrix} 2.5 \\ -3.1 \\ 7.4 \\ \dots \\ 2.0 \\ 1.8 \end{pmatrix}$$

"Soft prompt"

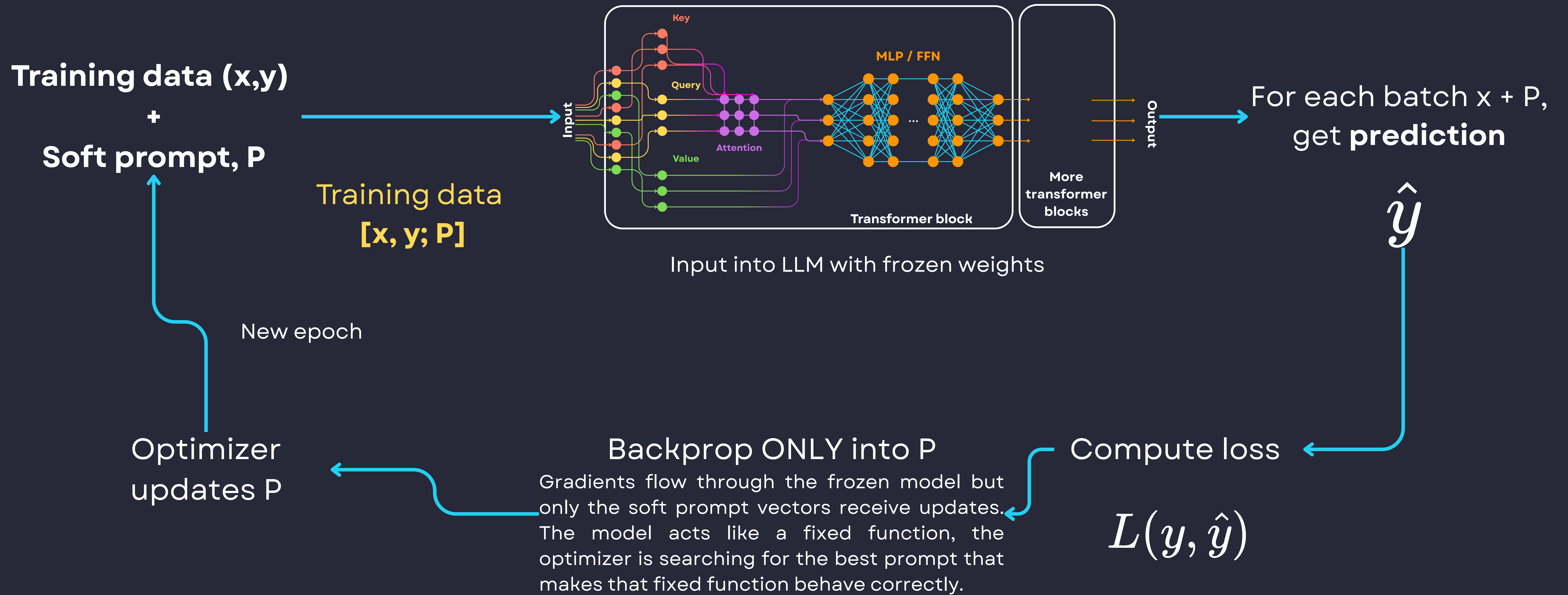
"Help me find the best restaurants in Spain."
"Focus on seafood spots, ignore anything too expensive."

Pass to LLM

AISEA

Shared under AISEA's learning commons. Built for the community. Please share, not sell.

How prompt tuning works mechanically



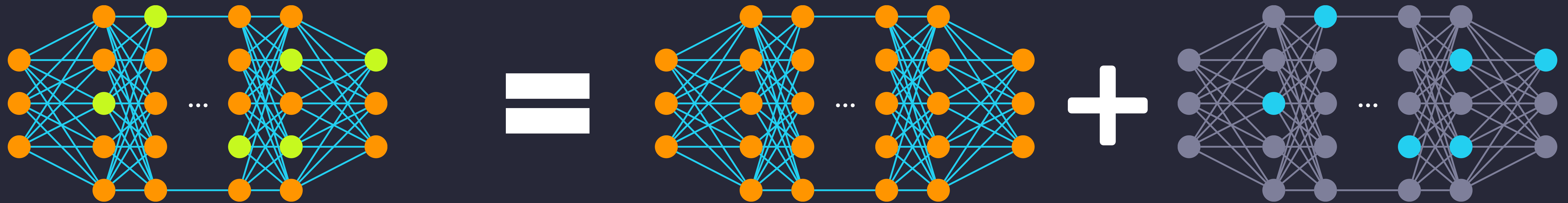
AISEA

Shared under AISEA's learning commons. Built for the community. Please share, not sell.

LoRA - Lowkey Really AI brain surgery

(Not really, it stands for Low-order Rank Adaptation)

The concept of LoRA: even though the weight matrices are huge (say 4096 × 4096), the change you want to make to them during fine-tuning doesn't actually require all that space.



That means we can train two significantly smaller sets of weights and add them as corrections to the core model.

Essentially, we're adding a "layer" on top of an existing model and changing that layer.

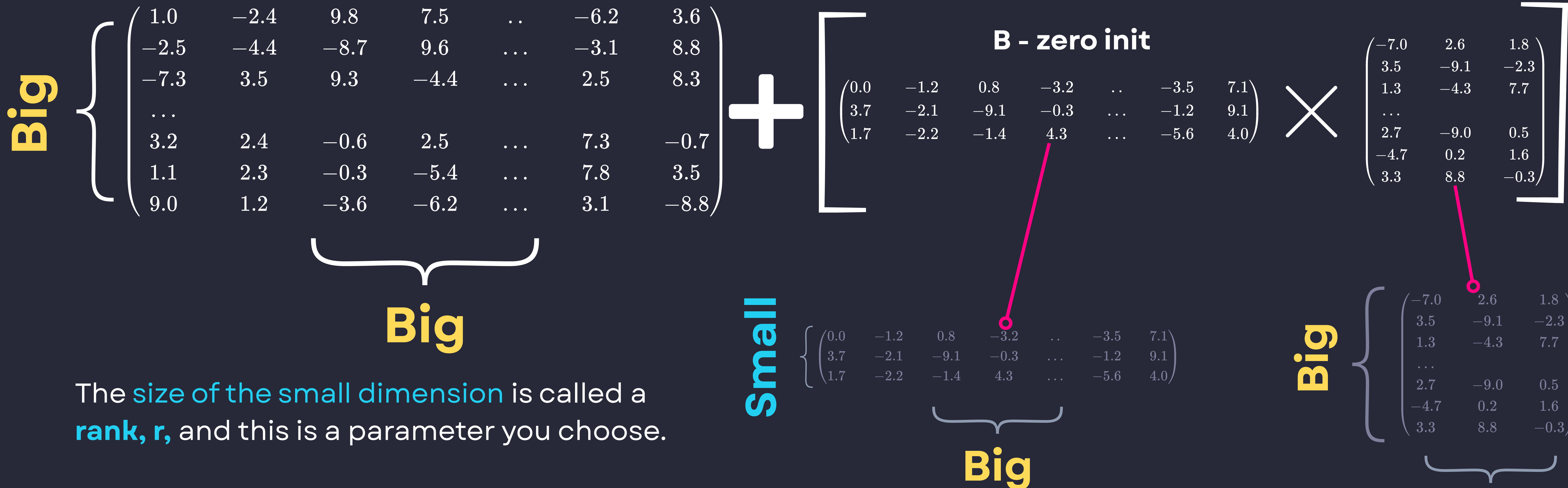
AISEA

Shared under AISEA's learning commons. Built for the community. Please share, not sell.

LoRA - the mechanics

$$W' = W + \triangle W$$

$$\triangle W = B \times A$$



The size of the small dimension is called a **rank, r,** and this is a parameter you choose.

Higher r = better model capacity but need more memory and computation.

AISEA
Shared under AISEA's learning commons. Built for the community. Please share, not sell.

How to LoRA - three knobs to turn

In practice

$$W' = W + \frac{\alpha}{r} \triangle W$$
$$\triangle W = B \times A$$

Rank, r

- The bottleneck dimension. How many independent directions you allow the update to express.
- Higher r = more expressive = more parameters = more memory.

r = 4: minimal, fast
r = 8: common default
r = 64: high capacity
r = 128: approaching full FT

Alpha, α

- A scaling factor applied to ΔW. The effective update is (α/r)·B·A.
- Alpha controls how strongly the adapter influences the output.
- In practice: set alpha = r or alpha = 2r and leave it.

Don’t overthink alpha, set it = r and move on.

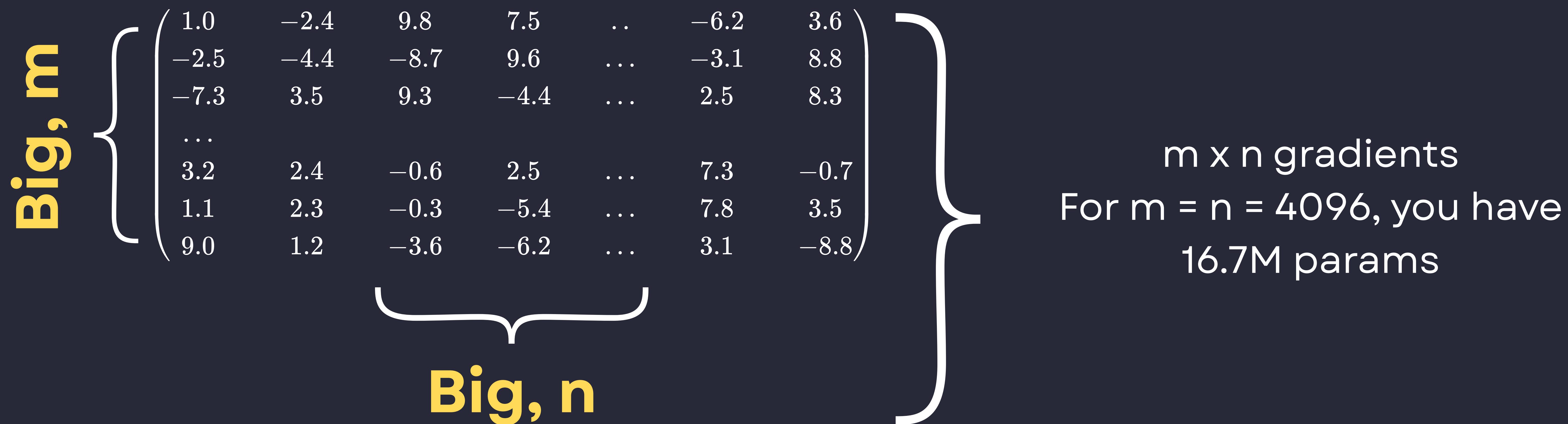
target_modules

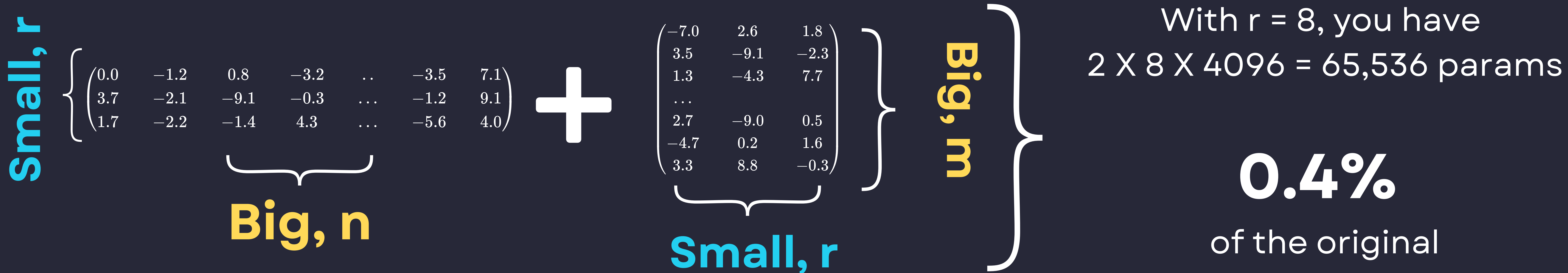
- Which weight matrices inside the model get LoRA applied.
- You don't adapt every layer – just the attention projections.
- Common targets are the query, key, value, and output projections.

q_proj: query
v_proj: value
k_proj: key
o_proj: output

Shared under AISEA’s learning commons. Built for the community. Please share, not sell.

LoRA saves you loads of memory





Shared under AISEA's learning commons. Built for the community. Please share, not sell.

What LoRA cannot do

LoRA cannot inject new data

LoRA reshapes how the model behaves, not what facts it knows. Fine-tuning on your internal docs won't reliably make the model know those docs. That's what RAG is for. LoRA is for style, format, and behaviour.

LoRA can cause catastrophic forgetting

If your dataset is too narrow or training runs too long, the model can overfit and lose general capability. It becomes great at your specific task and bad at everything else. Low rank and early stopping are your defences.

LoRA doesn’t fix bad data

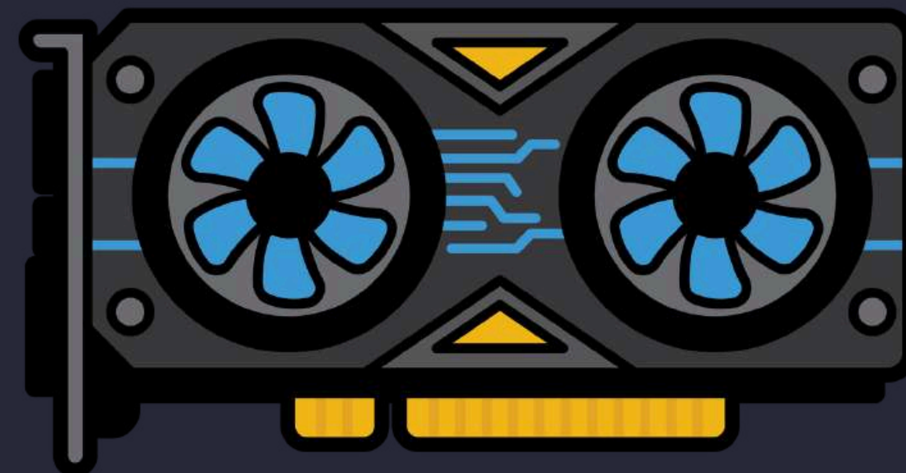
LoRA amplifies whatever signal is in your dataset, including noise and inconsistencies. A fine-tune is only as good as the data it trains on. This is why data prep is the actual hard part.

Shared under AISEA’s learning commons. Built for the community. Please share, not sell.

Memory is important!!!!

When you train a model, data is stored on the VRAM. This is GPU memory separate from system RAM.
Everything running on a GPU has to fit into VRAM, otherwise the job crashes or falls back to system RAM which is much slower.
The VRAM is the bottleneck for training.

When training, the GPU needs to fit 4 items at once:



AISEA

Shared under AISEA's learning commons. Built for the community. Please share, not sell.

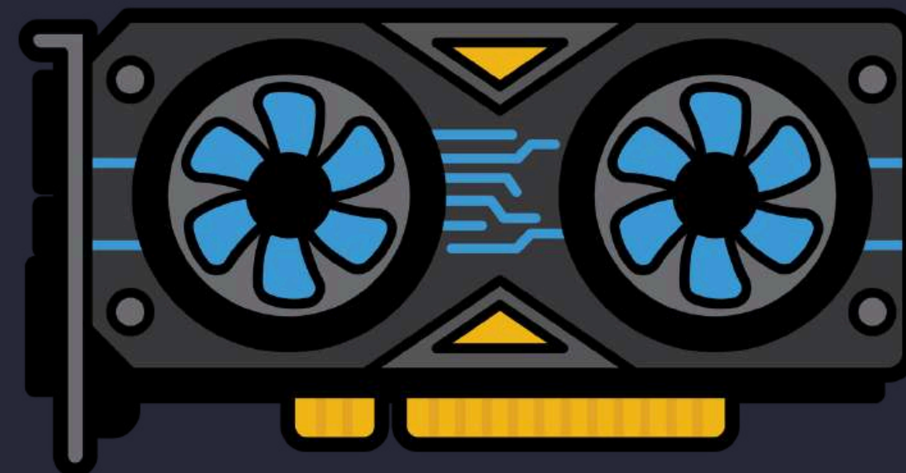
Memory is important!!!!

When you train a model, data is stored on the VRAM. This is GPU memory separate from system RAM.
Everything running on a GPU has to fit into VRAM, otherwise the job crashes or falls back to system RAM which is much slower.
The VRAM is the bottleneck for training.

When training, the GPU needs to fit 4 items at once:

Model weights

The parameters themselves.
A 7B model at 16-bit (bfloat16) = 2 bytes per parameter × 7 billion = ~14GB.



Shared under AISEA's learning commons. Built for the community. Please share, not sell.

Memory is important!!!!

When you train a model, data is stored on the VRAM. This is GPU memory separate from system RAM.
Everything running on a GPU has to fit into VRAM, otherwise the job crashes or falls back to system RAM which is much slower.
The VRAM is the bottleneck for training.

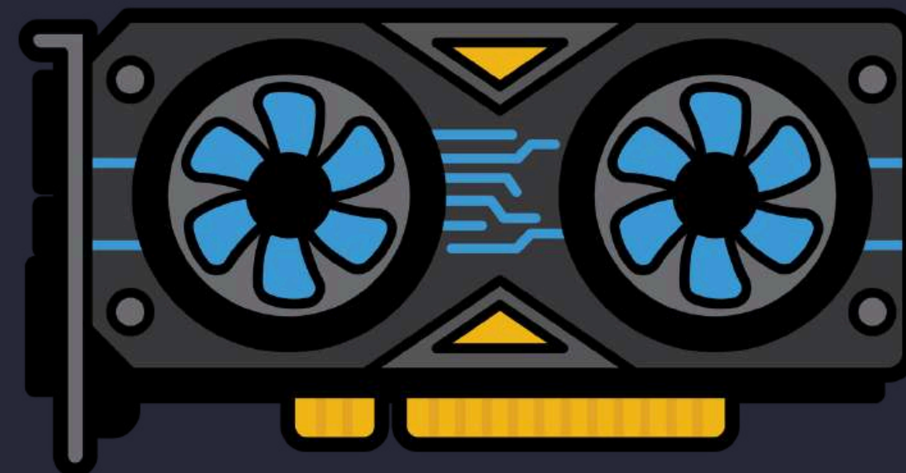
When training, the GPU needs to fit 4 items at once:

Model weights

The parameters themselves. A 7B model at 16-bit (bfloat16) = 2 bytes per parameter × 7 billion = ~14GB.

Gradients

For every weight, you need to store its gradient during backprop. Same size as the weights: another ~14GB.



Shared under AISEA's learning commons. Built for the community. Please share, not sell.

Memory is important!!!!

When you train a model, data is stored on the VRAM. This is GPU memory separate from system RAM.
Everything running on a GPU has to fit into VRAM, otherwise the job crashes or falls back to system RAM which is much slower.
The VRAM is the bottleneck for training.

When training, the GPU needs to fit 4 items at once:

Model weights

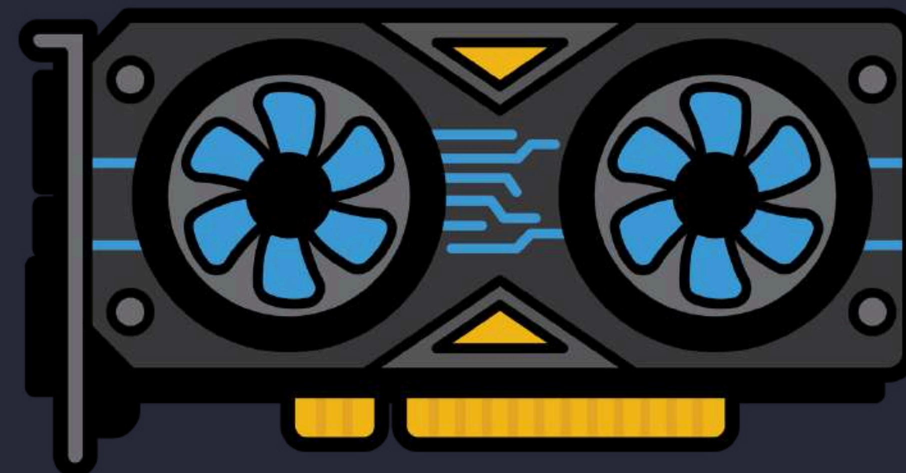
The parameters themselves. A 7B model at 16-bit (bfloat16) = 2 bytes per parameter × 7 billion = ~14GB.

Optimiser states

Adam stores two additional values per parameter: a first moment and a second moment. 2× the weight size = ~28GB.

Gradients

For every weight, you need to store its gradient during backprop. Same size as the weights: another ~14GB.



Shared under AISEA's learning commons. Built for the community. Please share, not sell.

Memory is important!!!!

When you train a model, data is stored on the VRAM. This is GPU memory separate from system RAM.
Everything running on a GPU has to fit into VRAM, otherwise the job crashes or falls back to system RAM which is much slower.
The VRAM is the bottleneck for training.

When training, the GPU needs to fit 4 items at once:

Model weights

The parameters themselves. A 7B model at 16-bit (bfloat16) = 2 bytes per parameter × 7 billion = ~14GB.

Optimiser states

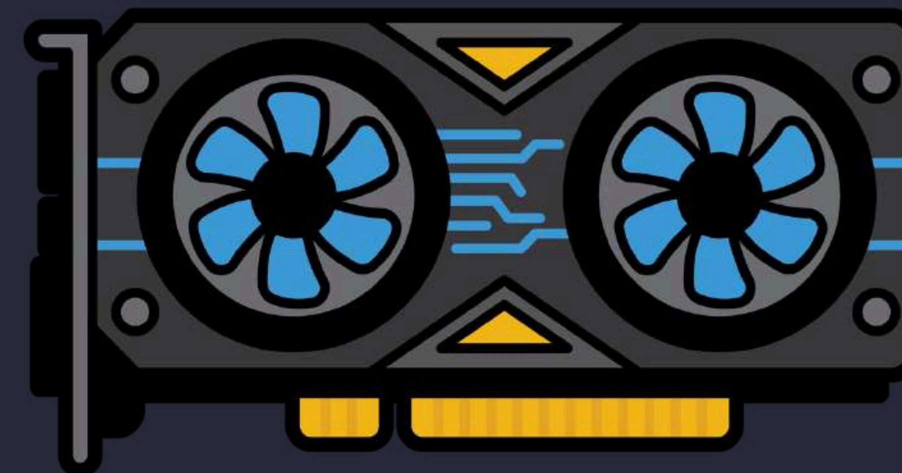
Adam stores two additional values per parameter: a first moment and a second moment. 2× the weight size = ~28GB.

Activations

Intermediate values computed during the forward pass, kept in memory for the backward pass. Size depends on batch size and sequence length.

Gradients

For every weight, you need to store its gradient during backprop. Same size as the weights: another ~14GB.



Shared under AISEA's learning commons. Built for the community. Please share, not sell.

Memory is important!!!!

When you train a model, data is stored on the VRAM. This is GPU memory separate from system RAM.
Everything running on a GPU has to fit into VRAM, otherwise the job crashes or falls back to system RAM which is much slower.
The VRAM is the bottleneck for training.

When training, the GPU needs to fit 4 items at once:

Model weights

The parameters themselves. A 7B model at 16-bit (bfloat16) = 2 bytes per parameter × 7 billion = ~14GB.

Optimiser states

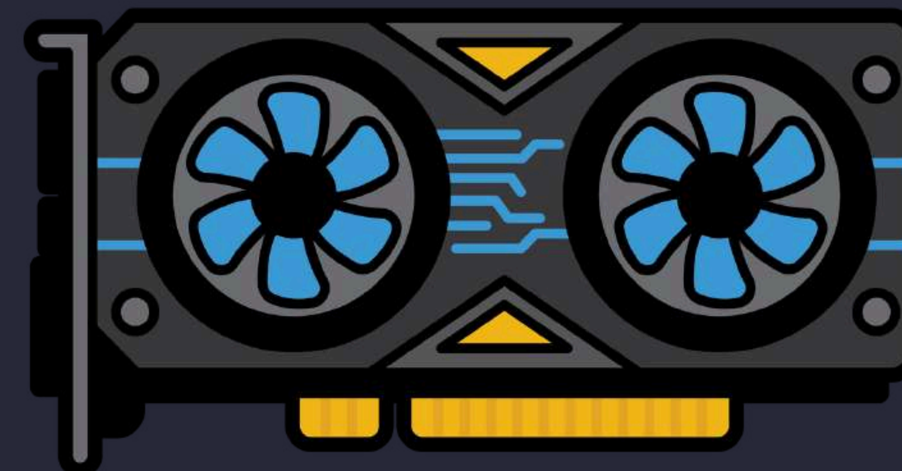
Adam stores two additional values per parameter: a first moment and a second moment. 2× the weight size = ~28GB.

Activations

Intermediate values computed during the forward pass, kept in memory for the backward pass. Size depends on batch size and sequence length.

Gradients

For every weight, you need to store its gradient during backprop. Same size as the weights: another ~14GB.



=

For a 7B model with Adam:
14 + 14 + 28 = **~56GB**
before activations.

Consumer RTX: **24 GB**
Colab T4: **16 GB**

AISEA

Shared under AISEA's learning commons. Built for the community. Please share, not sell.

Memory is important!!!!

LoRA shrinks gradients and optimiser states

Model weights

The parameters themselves. A 7B model at 16-bit (bfloat16) = 2 bytes per parameter × 7 billion = ~14GB.

Optimiser states

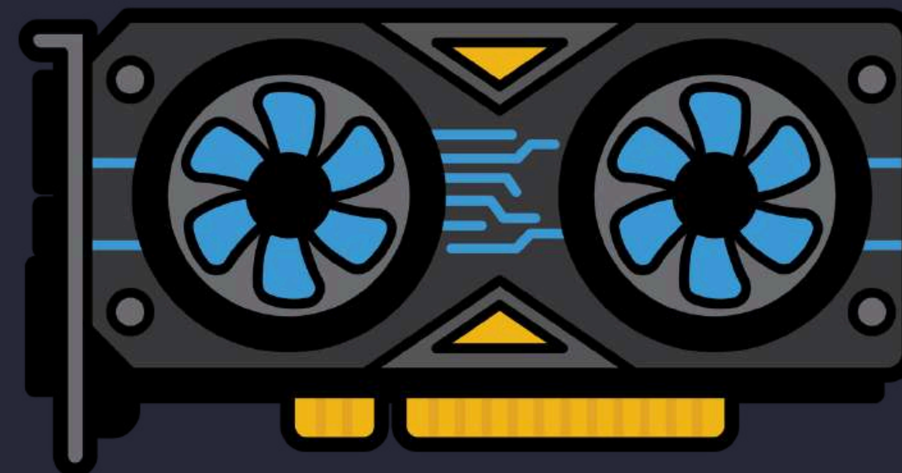
Adam stores two additional values per parameter: a first moment and a second moment. 2× the weight size = ~28GB.

Activations

Intermediate values computed during the forward pass, kept in memory for the backward pass. Size depends on batch size and sequence length.

Gradients

For every weight, you need to store its gradient during backprop. Same size as the weights: another ~14GB.



LoRA shrinks gradients and optimiser states

Shared under AISEA's learning commons. Built for the community. Please share, not sell.

Memory is important!!!!

Still need to handle base model weights, especially for large models

Model weights

The parameters themselves. A 7B model at 16-bit (bfloat16) = 2 bytes per parameter × 7 billion = ~14GB.

LoRA shrinks gradients and optimiser states

Optimiser states

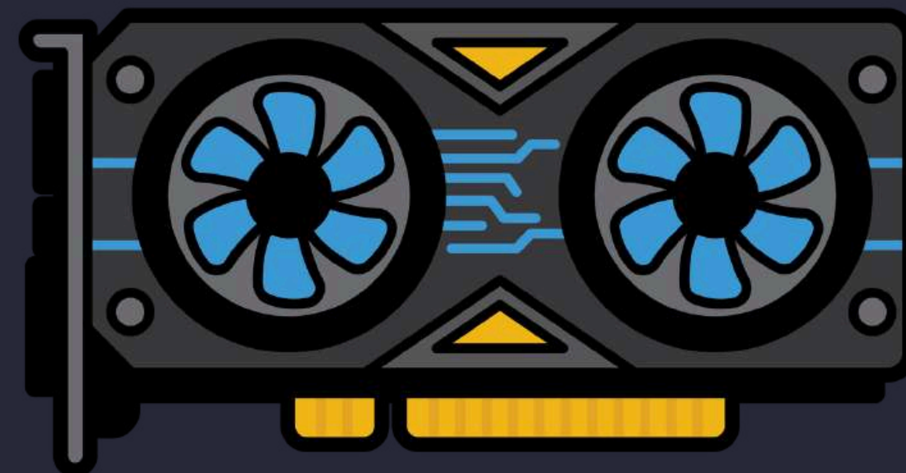
Adam stores two additional values per parameter: a first moment and a second moment. 2× the weight size = ~28GB.

Activations

Intermediate values computed during the forward pass, kept in memory for the backward pass. Size depends on batch size and sequence length.

Gradients

For every weight, you need to store its gradient during backprop. Same size as the weights: another ~14GB.



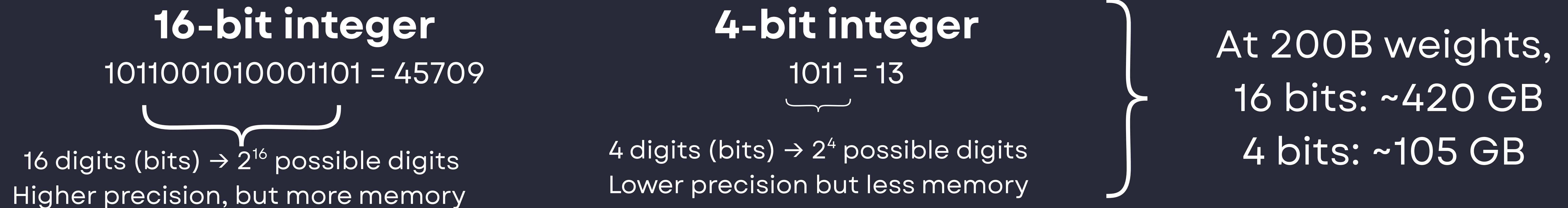
LoRA shrinks gradients and optimiser states

AISEA

Shared under AISEA's learning commons. Built for the community. Please share, not sell.

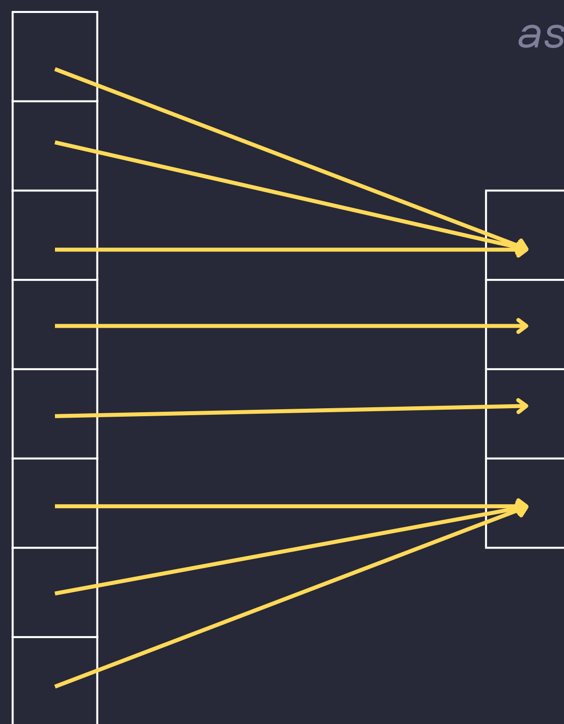
QLoRA: LoRA with better memory management

QLoRA manages the memory problem by quantising weights, i.e. “rounding to the nearest digit”.



Most model weights cluster in a normal distribution. We can save memory by grouping and storing in buckets.

QLoRA uses NF4 (Normal Float 4-bit) as the algorithm to create buckets.



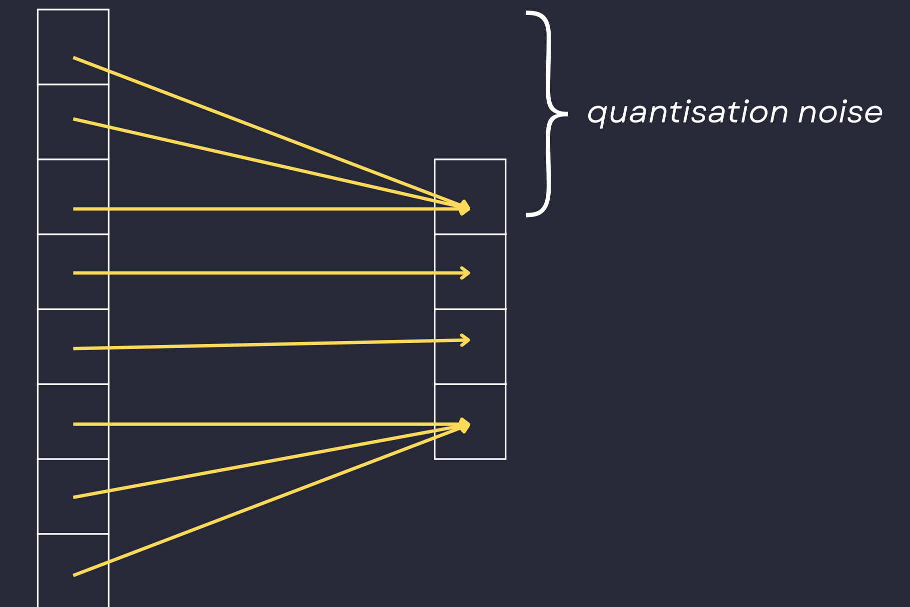
Dequantisation:

To use the weight later, you look up which bucket it was in and use the bucket's representative value.

AISEA

Shared under AISEA’s learning commons. Built for the community. Please share, not sell.

How bad is the precision loss from quantization?



When you map a weight from a precise 16-bit value to the nearest of 16 buckets, you introduce a small error: the distance between the original value and the bucket centre. This is quantisation noise.

Two reasons this doesn't matter much in practice:

1. The base model is frozen.
You're not training these weights. You're just reading them during the forward pass. A small amount of noise in a frozen, already-well-trained model is far less damaging than the same noise during active training.

2. The adapters compensate.
The LoRA adapters that sit on top are in full 16-bit precision and are being actively trained. They learn to work with the slightly-noisy base. In the original QLoRA paper, Dettmers et al. found that QLoRA matched full 16-bit LoRA performance on most benchmarks – the adapters absorb the noise.

The exception is when you have very long inference chains where small errors compound, or tasks requiring extreme numerical precision. For the vast majority of builder use cases, it doesn't matter.

AISEA

Shared under AISEA's learning commons. Built for the community. Please share, not sell.

When do you do what?

Method	When to use it
Harness engineering / RAG	Default starting point. Always try this first. Works for most tasks. Zero training cost, instant iteration. Reach for FT only when this provably fails.
Prompt tuning	Rarely worth it in 2026. Makes sense when you need consistent behaviour across a fixed model you can't modify, and context cost is a real constraint. Mostly of academic interest now.
Distillation	You have a large capable model (teacher) that does the task well but is too slow or expensive to serve at scale. You want a smaller model (student) that mimics its outputs. Requires the teacher to generate your training data. Best when latency or cost is the constraint, not capability.
LoRA / QLoRA	You've confirmed prompting can't solve it. You need consistent style, format, or domain behaviour at scale. You want to own the weights. Your dataset is high quality and > ~500 examples.
Full FT	You have a large, high-quality dataset. You need fundamental behavioural change across the entire model. You have A100-class hardware. You've exhausted LoRA. Almost never the right first move.

So how do you FT?

AISEA

What options do we have for FT?

Layer 1 - Foundation

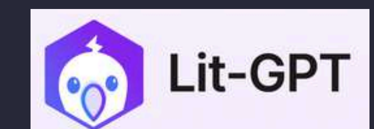
Raw building blocks: model architectures and weights you can interact with, LoRA and adapter math all implemented explicitly, gradient descent via tensor and autograd engine. **The baseline implementation of FT.**

Layer 2 - Optimization

Tools that change how efficiently training happens without touching what you're training, e.g. rewriting low-level GPU operations such as matrix multiplications to use less memory and run faster. Make it possible to train large models on consumer hardware.

Layer 3 - Orchestration

You run training on your own GPUs, but a framework handles the wiring. You write a config file (usually YAML) that says which model, dataset, technique, and hyperparams. The framework assembles the training loop, handles data loading, logging, checkpointing, and calls the layers below. **Basically the Vercel to your FT.**



Layer 4 - Platforms

Someone else owns the infra, you bring your dataset and hyperparams. You never see the layers below. Highest abstraction, fastest to start, but least control, highest cost per run, and least insight into what's happening. **Essentially the Lovable of FT.**

Shared under AISEA's learning commons. Built for the community. Please share, not sell.

SFT + LoRA + QLoRA: The full picture

Each concept answers a different question. A real training run answers all four at once.

What am I teaching?	**SFT**	Input → output pairs. Cross-entropy loss. Minimise the prediction gap.
Which weights update?	**LoRA**	Only the small A and B adapter matrices. Frozen base. 0.4% of parameters.
How do I fit it in memory?	**QLoRA**	Base model compressed to 4-bit NF4. Adapters stay in 16-bit. Paged optimiser.
How do I make it faster?	**Unsloth AI**	Custom CUDA kernels. 2× speed, 70% less VRAM. Same results. One flag to enable.
How do I wire it all together?		One YAML config. Declares model, dataset, technique, hyperparams. Calls the layers below.

In practice: SFT is the objective → LoRA limits what updates → QLoRA fits it in memory → Unsloth speeds it up → Axolotl wires it together.

AISEA

Shared under AISEA's learning commons. Built for the community. Please share, not sell.

Data formats

This is the first wall builders hit. Format matters — Axolotl and TRL parse these differently.

Alpaca

Single-turn instruction → response

```
{
  "instruction": "Summarise this in one sentence.",
  "input": "[article text]",
  "output": "The article discusses..."
}
```

Use when: simple input/output pairs. Most tutorials use this format. Best for classification, summarisation, extraction.

ShareGPT

Multi-turn conversations

```
{
  "conversations": [
    {"from":"human","value":"Fix this bug..."},
    {"from":"gpt","value":"The issue is..."}
  ]
}
```

Use when: teaching dialogue, back-and-forth reasoning, or chat-style behaviour. Closer to how chat models are trained.

JSONL (raw)

Custom or continued pretraining

```
{"text": "The full prompt and completion concatenated as one string, separated by your template tokens."}
```

Use when: continued pretraining on raw text, or when you need full control over the token template. More manual setup.

Rule of thumb: use Alpaca for simple tasks, ShareGPT for chat/dialogue, raw JSONL only when you need full control.

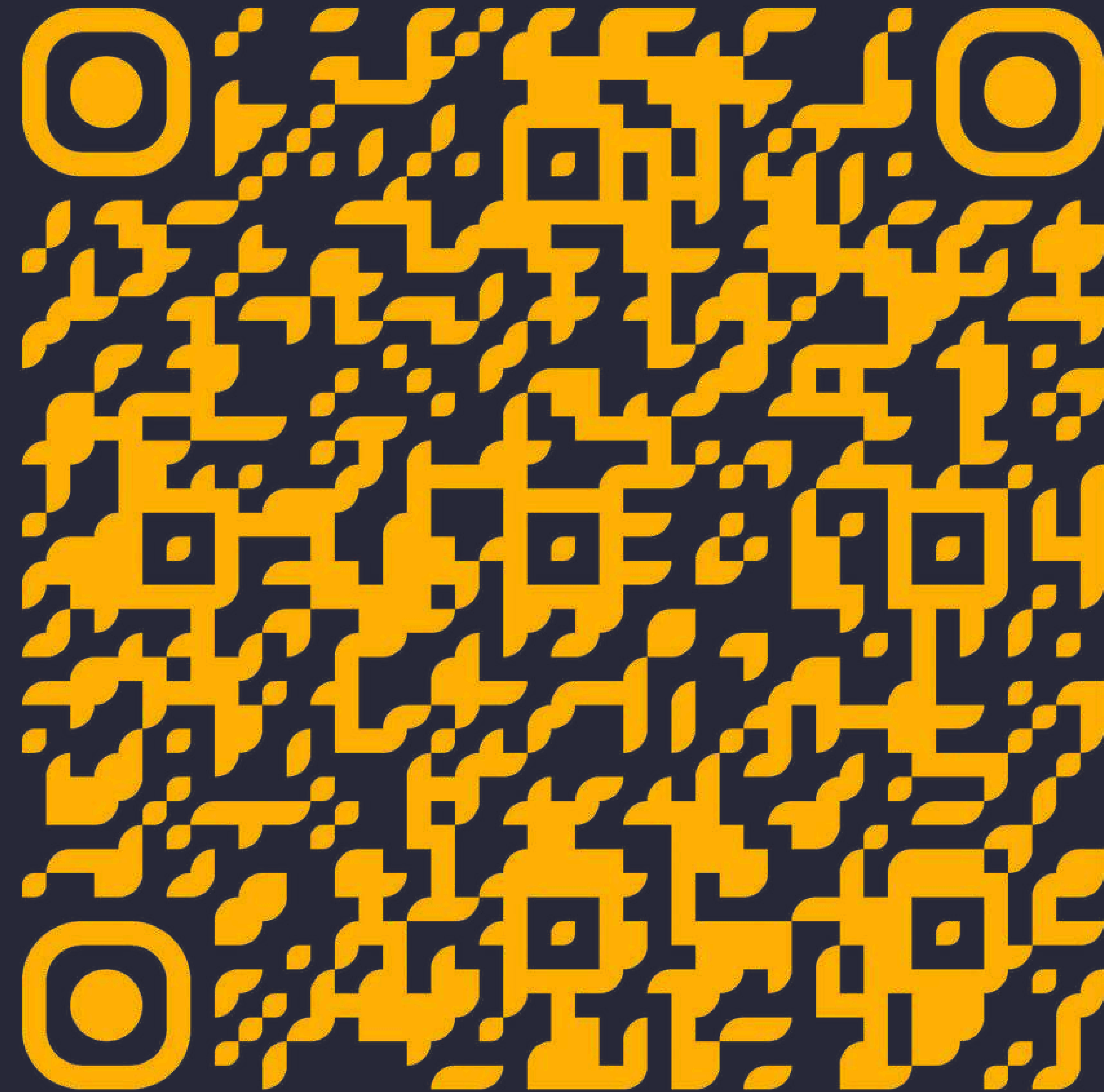
Shared under AISEA's learning commons. Built for the community. Please share, not sell.

How do we get started with FT?

Level	Stack	What you'll see
Beginner's first LoRA run	Google Colab + Unsloth + Qwen/Llama 3B–8B	Dataset formatting, training loss, adapter basics. Best for learning the loop end-to-end.
Tinkerers / engineers who want better control	Hugging Face PEFT + TRL + Unsloth	More control over SFT, LoRA/QLoRA, evals. Best for understanding modern open-source SFT.
Operators running repeatable FT jobs	Axolotl + cloud GPU via RunPod / Modal	Reproducible pipelines, checkpointing, multi-run experiments. Best for running real experiments without hand-wiring everything.
Teams that want to prioritize speed	Together AI / Firework AI	Upload data, pick model, train, deploy. Best for teams that want to ship fast and not manage infra.
Engineers / researchers who want full control	PyTorch + PEFT + DeepSpeed/FSDP + vLLM	Full control, scaling, serving, infra tradeoffs. Best for ML engineers optimizing cost, scale, and deployment.

Shared under AISEA's learning commons. Built for the community. Please share, not sell.

Worked example



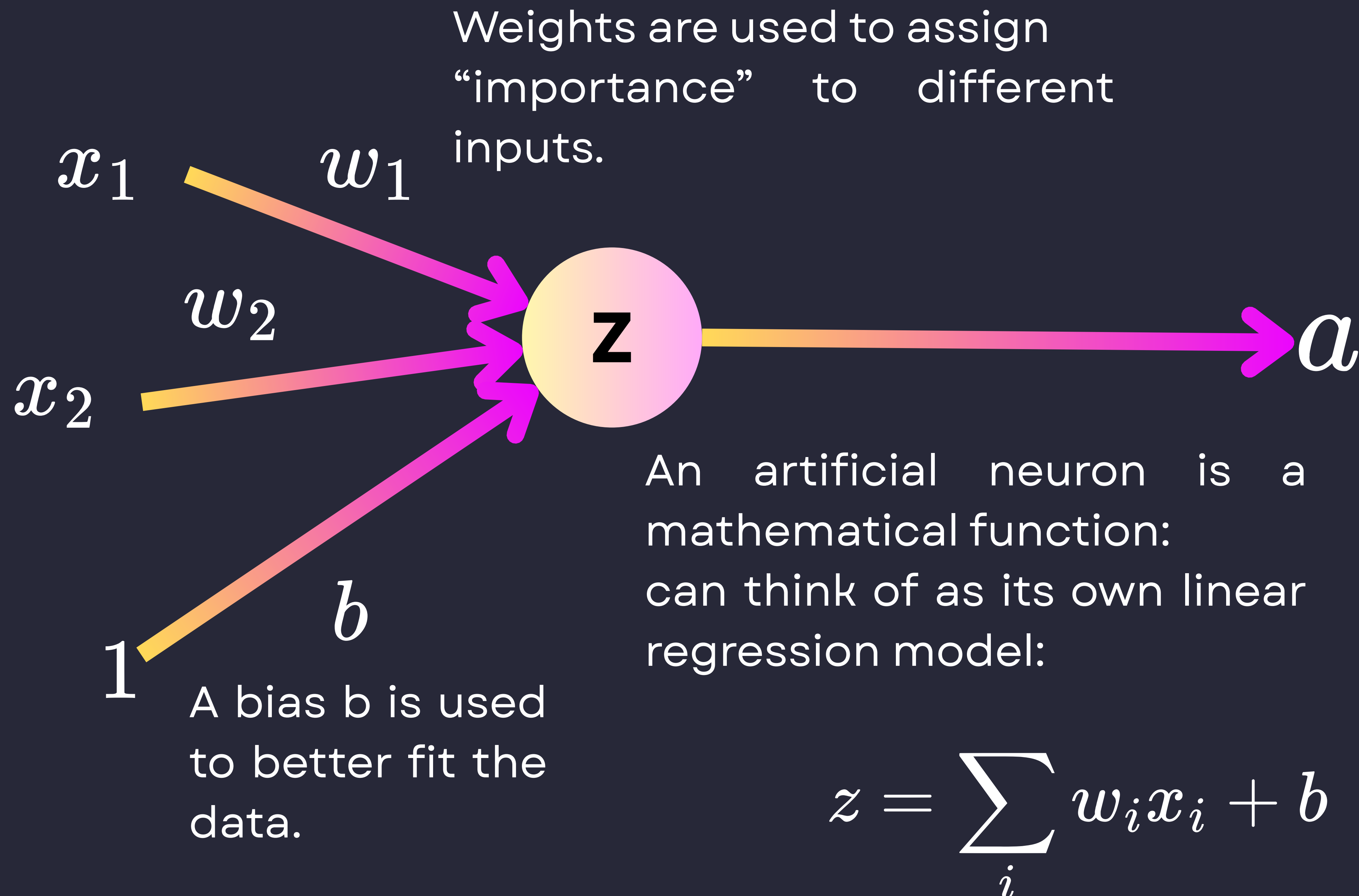
AISEA

Shared under AISEA's learning commons. Built for the community. Please share, not sell.

Appendix

AISEA

How a neural network neuron works



Weights are used to assign "importance" to different inputs.

A bias b is used to better fit the data.

An artificial neuron is a mathematical function: can think of as its own linear regression model:

$$z = \sum_i w_i x_i + b$$

$$a = \sigma(\sum_i w_i x_i + b)$$

The result z is fed into an activation function. Activation functions determine if a neuron should be activated based on output and can impact the training speed and performance of a model.

Importantly, nonlinear activation functions enable backpropagation- their gradients provide important information.

AISEA

Shared under AISEA's learning commons. Built for the community. Please share, not sell.