CS498: Algorithmic Engineering

Lecture 28: Fine-tuning, LoRA & SFT

Elfarouk Harb

University of Illinois Urbana-Champaign

Week 16

Outline

1. Inside Modern LLMs
2. Pre-training vs. Fine-tuning
3. LoRA: Low-Rank Adaptation
4. Supervised Fine-Tuning (SFT)
5. Expert Iteration
6. The Full Pipeline

1. Inside Modern LLMs
2. Pre-training vs. Fine-tuning
3. LoRA: Low-Rank Adaptation
4. Supervised Fine-Tuning (SFT)
5. Expert Iteration
6. The Full Pipeline

Grouped Query Attention (GQA)

Full multi-head attention: each head has its own W_K and W_V.

Grouped Query Attention (GQA)

Full multi-head attention: each head has its own W_K and W_V.

Problem: the KV cache scales linearly with number of KV heads.

Grouped Query Attention (GQA)

Full multi-head attention: each head has its own W_K and W_V.

Problem: the KV cache scales linearly with number of KV heads.

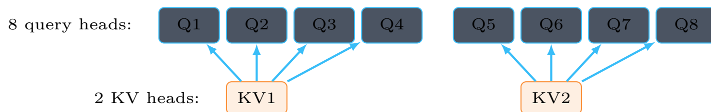
GQA: multiple query heads share the same K and V head.

Grouped Query Attention (GQA)

Full multi-head attention: each head has its own W_K and W_V.

Problem: the KV cache scales linearly with number of KV heads.

GQA: multiple query heads share the same K and V head.

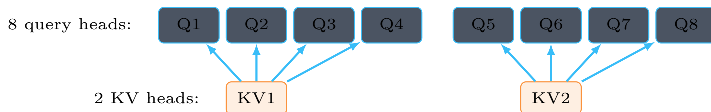


Grouped Query Attention (GQA)

Full multi-head attention: each head has its own W_K and W_V.

Problem: the KV cache scales linearly with number of KV heads.

GQA: multiple query heads share the same K and V head.



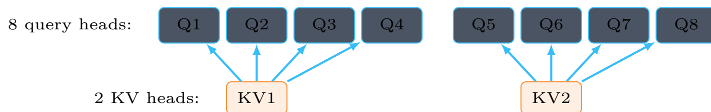
Llama-3 8B: 32 query heads, 8 KV heads. KV cache is $4\times$ smaller.

Grouped Query Attention (GQA)

Full multi-head attention: each head has its own W_K and W_V.

Problem: the KV cache scales linearly with number of KV heads.

GQA: multiple query heads share the same K and V head.



Llama-3 8B: 32 query heads, 8 KV heads. KV cache is $4\times$ smaller.

Quality loss is minimal. Inference speedup is significant.

Mixture of Experts: The Idea

Recall: each transformer block has an FFN that processes every token independently.

Mixture of Experts: The Idea

Recall: each transformer block has an FFN that processes every token independently.

In a standard transformer, every token goes through the same FFN. The FFN has $\sim 8d^2$ parameters and all of them activate for every token.

Mixture of Experts: The Idea

Recall: each transformer block has an FFN that processes every token independently.

In a standard transformer, every token goes through the same FFN. The FFN has $\sim 8\mathrm{d}^2$ parameters and all of them activate for every token.

Observation: not every token needs the same processing. The word “theorem” might benefit from “math expert” neurons, while “DNA” benefits from “biology expert” neurons.

Mixture of Experts: The Idea

Recall: each transformer block has an FFN that processes every token independently.

In a standard transformer, every token goes through the same FFN. The FFN has $\sim 8d^2$ parameters and all of them activate for every token.

Observation: not every token needs the same processing. The word "theorem" might benefit from "math expert" neurons, while "DNA" benefits from "biology expert" neurons.

MoE idea: instead of one big FFN, have E separate FFNs ("experts"). For each token, a small router network picks the top k experts. Only those k experts run.

Mixture of Experts: The Idea

Recall: each transformer block has an FFN that processes every token independently.

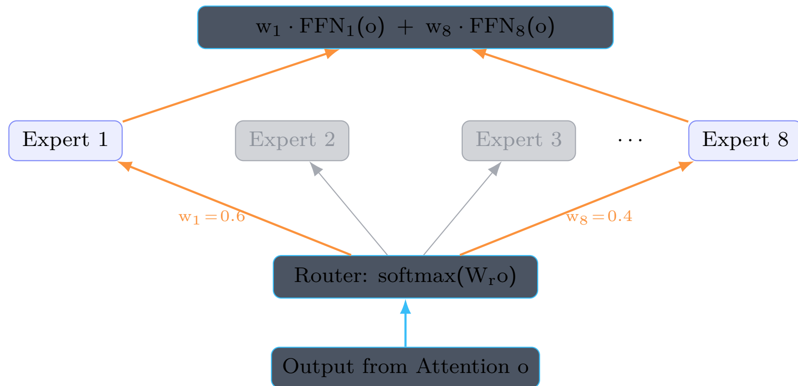
In a standard transformer, every token goes through the same FFN. The FFN has $\sim 8d^2$ parameters and all of them activate for every token.

Observation: not every token needs the same processing. The word "theorem" might benefit from "math expert" neurons, while "DNA" benefits from "biology expert" neurons.

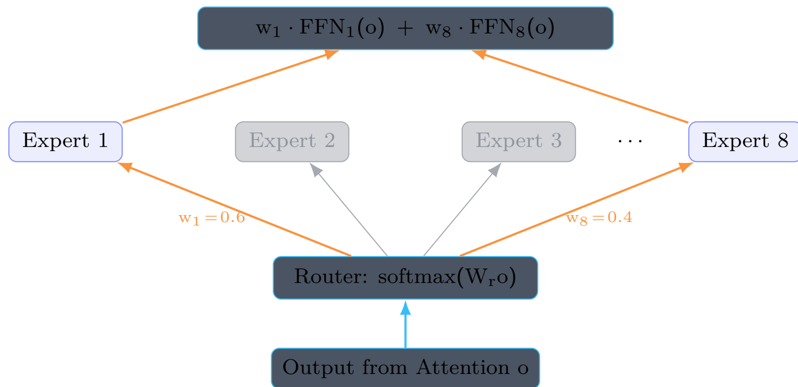
MoE idea: instead of one big FFN, have E separate FFNs ("experts"). For each token, a small router network picks the top k experts. Only those k experts run.

The attention layer is unchanged. Only the FFN is replaced.

Mixture of Experts: How It Works

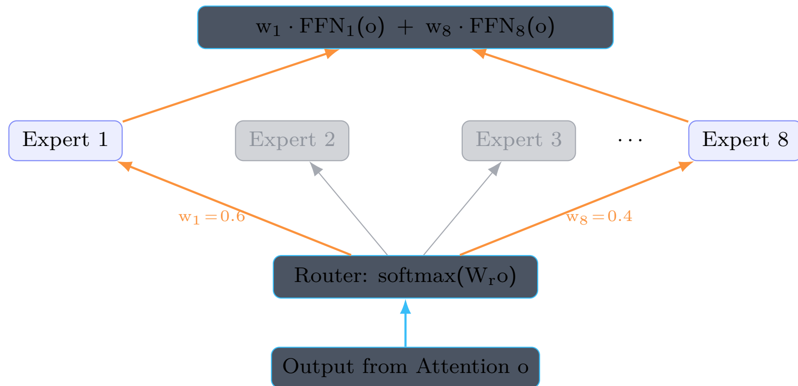


Mixture of Experts: How It Works



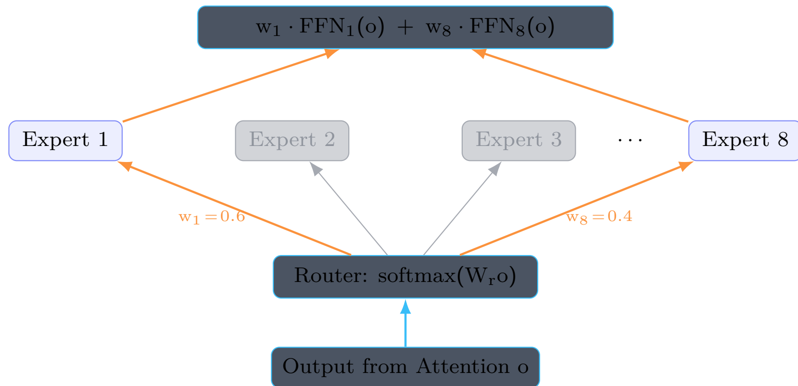
- Router: a tiny linear layer $W_r \in \mathbb{R}^{E \times d}$ that scores each expert. Pick top-k, softmax over those k to get weights $w_1, \ldots, w_k$.

Mixture of Experts: How It Works



- Router: a tiny linear layer $W_r \in \mathbb{R}^{E \times d}$ that scores each expert. Pick top-k, softmax over those k to get weights $w_1, \ldots, w_k$.
- Each expert: a full SwiGLU FFN (same architecture as L26, $\sim 8d^2$ params each).

Mixture of Experts: How It Works



- Router: a tiny linear layer $W_r \in \mathbb{R}^{E \times d}$ that scores each expert. Pick top-k, softmax over those k to get weights $w_1, \ldots, w_k$.
- Each expert: a full SwiGLU FFN (same architecture as L26, $\sim 8d^2$ params each).

MoE: Why Bother?

	Dense (Llama-3 8B)	MoE (Mixtral 8x7B)
Total params	8B	47B
Active params per token	8B	13B
Inference cost per token	$\propto$ 8B	$\propto$ 13B
Quality	Good	Much better

MoE: Why Bother?

	Dense (Llama-3 8B)	MoE (Mixtral 8x7B)
Total params	8B	47B
Active params per token	8B	13B
Inference cost per token	$\propto$ 8B	$\propto$ 13B
Quality	Good	Much better

Mixtral has 47B total parameters. But only 2 experts fire per token, so inference cost is like a 13B model.

MoE: Why Bother?

	Dense (Llama-3 8B)	MoE (Mixtral 8x7B)
Total params	8B	47B
Active params per token	8B	13B
Inference cost per token	$\propto$ 8B	$\propto$ 13B
Quality	Good	Much better

Mixtral has 47B total parameters. But only 2 experts fire per token, so inference cost is like a 13B model.

Memory: you still need to store all 47B params in GPU memory. MoE saves compute, not memory. This is why MoE models need more GPUs than their active parameter count suggests.

Speculative Decoding: The Problem

Decode is slow: one token at a time, each requiring a full forward pass through the big model.

Speculative Decoding: The Problem

Decode is slow: one token at a time, each requiring a full forward pass through the big model.

Observation: most tokens are "easy." After "The capital of France is", the next token is almost certainly "Paris." You don't need 70B parameters to predict that.

Speculative Decoding: The Problem

Decode is slow: one token at a time, each requiring a full forward pass through the big model.

Observation: most tokens are "easy." After "The capital of France is", the next token is almost certainly "Paris." You don't need 70B parameters to predict that.

Idea: use a small, fast draft model (e.g., 1B params) to guess the next k tokens. Then check whether the big model would have generated the same tokens.

Speculative Decoding: The Problem

Decode is slow: one token at a time, each requiring a full forward pass through the big model.

Observation: most tokens are "easy." After "The capital of France is", the next token is almost certainly "Paris." You don't need 70B parameters to predict that.

Idea: use a small, fast draft model (e.g., 1B params) to guess the next k tokens. Then check whether the big model would have generated the same tokens.

If the draft model's guesses are good (they usually are for easy tokens), we skip k forward passes of the big model and do just one.

Speculative Decoding: The Algorithm

1. Draft: small model generates k tokens $t_1, t_2, \ldots, t_k$ autoregressively (fast).

Speculative Decoding: The Algorithm

1. Draft: small model generates k tokens $t_1, t_2, \ldots, t_k$ autoregressively (fast).
2. Score: big model processes all k tokens in one forward pass (parallel, like prefill). This gives us $p_{big}(t_i \mid \text{context})$ for each position.

Speculative Decoding: The Algorithm

1. Draft: small model generates k tokens $t_1, t_2, \ldots, t_k$ autoregressively (fast).
2. Score: big model processes all k tokens in one forward pass (parallel, like prefill). This gives us $p_{big}(t_i \mid \text{context})$ for each position.
3. Accept/reject: for each token t_i in order:
   - Compare $p_{big}(t_i)$ to $p_{draft}(t_i)$
   - If $p_{big} \geq p_{draft}$: accept (big model agrees or is even more confident)

Speculative Decoding: The Algorithm

1. Draft: small model generates k tokens $t_1, t_2, \ldots, t_k$ autoregressively (fast).
2. Score: big model processes all k tokens in one forward pass (parallel, like prefill). This gives us $p_{big}(t_i \mid \text{context})$ for each position.
3. Accept/reject: for each token t_i in order:
   - Compare $p_{big}(t_i)$ to $p_{draft}(t_i)$
   - If $p_{big} \geq p_{draft}$: accept (big model agrees or is even more confident)
   - If $p_{big} < p_{draft}$: accept with probability p_{big}/p_{draft}, otherwise reject and resample from an adjusted distribution

Speculative Decoding: The Algorithm

1. Draft: small model generates k tokens $t_1, t_2, \ldots, t_k$ autoregressively (fast).
2. Score: big model processes all k tokens in one forward pass (parallel, like prefill). This gives us $p_{big}(t_i \mid \text{context})$ for each position.
3. Accept/reject: for each token t_i in order:
   - Compare $p_{big}(t_i)$ to $p_{draft}(t_i)$
   - If $p_{big} \geq p_{draft}$: accept (big model agrees or is even more confident)
   - If $p_{big} < p_{draft}$: accept with probability p_{big}/p_{draft}, otherwise reject and resample from an adjusted distribution
4. Stop at first rejection. Keep all accepted tokens.

Speculative Decoding: The Algorithm

1. Draft: small model generates k tokens $t_1, t_2, \ldots, t_k$ autoregressively (fast).
2. Score: big model processes all k tokens in one forward pass (parallel, like prefill). This gives us $p_{big}(t_i \mid \text{context})$ for each position.
3. Accept/reject: for each token t_i in order:
   - Compare $p_{big}(t_i)$ to $p_{draft}(t_i)$
   - If $p_{big} \geq p_{draft}$: accept (big model agrees or is even more confident)
   - If $p_{big} < p_{draft}$: accept with probability p_{big}/p_{draft}, otherwise reject and resample from an adjusted distribution
4. Stop at first rejection. Keep all accepted tokens.

Key property: this acceptance scheme guarantees the output distribution is exactly the same as the big model alone. No quality loss (rejection sampling).

Speculative Decoding: The Algorithm

1. Draft: small model generates k tokens $t_1, t_2, \ldots, t_k$ autoregressively (fast).
2. Score: big model processes all k tokens in one forward pass (parallel, like prefill). This gives us $p_{big}(t_i \mid \text{context})$ for each position.
3. Accept/reject: for each token t_i in order:
   - Compare $p_{big}(t_i)$ to $p_{draft}(t_i)$
   - If $p_{big} \geq p_{draft}$: accept (big model agrees or is even more confident)
   - If $p_{big} < p_{draft}$: accept with probability p_{big}/p_{draft}, otherwise reject and resample from an adjusted distribution
4. Stop at first rejection. Keep all accepted tokens.

Key property: this acceptance scheme guarantees the output distribution is exactly the same as the big model alone. No quality loss (rejection sampling).

If draft accepts 3–4 of 5 tokens: ∼2–3x speedup.

1 Inside Modern LLMs

2 Pre-training vs. Fine-tuning

3 LoRA: Low-Rank Adaptation

4 Supervised Fine-Tuning (SFT)

5 Expert Iteration

6 The Full Pipeline

The Two Stages of Building an LLM

Stage 1: Pre-training

The Two Stages of Building an LLM

Stage 1: Pre-training

- Predict next token on a massive, general corpus
- Learns language, facts, reasoning patterns
- Extremely expensive: weeks on hundreds of GPUs

The Two Stages of Building an LLM

Stage 1: Pre-training

- Predict next token on a massive, general corpus
- Learns language, facts, reasoning patterns
- Extremely expensive: weeks on hundreds of GPUs

Stage 2: Fine-tuning (this lecture)

The Two Stages of Building an LLM

Stage 1: Pre-training

- Predict next token on a massive, general corpus
- Learns language, facts, reasoning patterns
- Extremely expensive: weeks on hundreds of GPUs

Stage 2: Fine-tuning (this lecture)

- Adapt the pre-trained model to a specific task or behavior
- Much cheaper: hours on a few GPUs

The Two Stages of Building an LLM

Stage 1: Pre-training

- Predict next token on a massive, general corpus
- Learns language, facts, reasoning patterns
- Extremely expensive: weeks on hundreds of GPUs

Stage 2: Fine-tuning (this lecture)

- Adapt the pre-trained model to a specific task or behavior
- Much cheaper: hours on a few GPUs

Analogy: pre-training is a university education (broad knowledge). Fine-tuning is job training (specific skills).

Why Not Just Prompt?

You can get a pre-trained model to do tasks via prompting:

Why Not Just Prompt?

You can get a pre-trained model to do tasks via prompting:

"Solve the following math problem step by step: ..."

Why Not Just Prompt?

You can get a pre-trained model to do tasks via prompting:

"Solve the following math problem step by step: ..."

This works, but has limitations:

Why Not Just Prompt?

You can get a pre-trained model to do tasks via prompting:

"Solve the following math problem step by step: ..."

This works, but has limitations:

- Inconsistent: sometimes follows instructions, sometimes doesn't
- Format: hard to get structured output reliably

Why Not Just Prompt?

You can get a pre-trained model to do tasks via prompting:

"Solve the following math problem step by step: ..."

This works, but has limitations:

- Inconsistent: sometimes follows instructions, sometimes doesn't
- Format: hard to get structured output reliably
- No new knowledge: can't learn from your private data
- Prompt overhead: long system prompts waste tokens every request

Why Not Just Prompt?

You can get a pre-trained model to do tasks via prompting:

"Solve the following math problem step by step: ..."

This works, but has limitations:

- Inconsistent: sometimes follows instructions, sometimes doesn't
- Format: hard to get structured output reliably
- No new knowledge: can't learn from your private data
- Prompt overhead: long system prompts waste tokens every request

Fine-tuning bakes the behavior into the weights, so it works every time without prompting.

Types of Fine-tuning

Method	What changes	Trainable params
Full fine-tuning	All parameters	100%
LoRA	Small adapter matrices	$< 1\%$
Prompt tuning	Learned prompt prefix	$< 0.01\%$

Types of Fine-tuning

Method	What changes	Trainable params
Full fine-tuning	All parameters	100%
LoRA	Small adapter matrices	$< 1\%$
Prompt tuning	Learned prompt prefix	$< 0.01\%$

Full fine-tuning: update every weight in the model. Best quality, but:

Types of Fine-tuning

Method	What changes	Trainable params
Full fine-tuning	All parameters	100%
LoRA	Small adapter matrices	$< 1\%$
Prompt tuning	Learned prompt prefix	$< 0.01\%$

Full fine-tuning: update every weight in the model. Best quality, but:

- Need to store a full copy of all parameters per task
- Risk of catastrophic forgetting: the model forgets what it learned in pre-training

Types of Fine-tuning

Method	What changes	Trainable params
Full fine-tuning	All parameters	100%
LoRA	Small adapter matrices	$< 1\%$
Prompt tuning	Learned prompt prefix	$< 0.01\%$

Full fine-tuning: update every weight in the model. Best quality, but:

- Need to store a full copy of all parameters per task
- Risk of catastrophic forgetting: the model forgets what it learned in pre-training
- Overfits quickly on small datasets

We need a cheaper approach that preserves pre-trained knowledge.

1. Inside Modern LLMs
2. Pre-training vs. Fine-tuning
3. LoRA: Low-Rank Adaptation
4. Supervised Fine-Tuning (SFT)
5. Expert Iteration
6. The Full Pipeline

The Key Insight

Pre-trained weight matrices are $d \times d$ (e.g., $4096 \times 4096 = 16.8$M parameters each).

The Key Insight

Pre-trained weight matrices are $d \times d$ (e.g., $4096 \times 4096 = 16.8$M parameters each).

Observation (Aghajanyan et al., 2021): when you fine-tune, the weight changes ΔW are low-rank.

The Key Insight

Pre-trained weight matrices are $d \times d$ (e.g., $4096 \times 4096 = 16.8M$ parameters each).

Observation (Aghajanyan et al., 2021): when you fine-tune, the weight changes ΔW are low-rank.

The model doesn't need to change in all d^2 directions. It only needs to adjust along a few important dimensions.

The Key Insight

Pre-trained weight matrices are $d \times d$ (e.g., $4096 \times 4096 = 16.8$M parameters each).

Observation (Aghajanyan et al., 2021): when you fine-tune, the weight changes ΔW are low-rank.

The model doesn't need to change in all d^2 directions. It only needs to adjust along a few important dimensions.

LoRA idea (Hu et al., 2022): don't update W directly. Instead, learn a low-rank decomposition of the change:

The Key Insight

Pre-trained weight matrices are $d \times d$ (e.g., $4096 \times 4096 = 16.8$M parameters each).

Observation (Aghajanyan et al., 2021): when you fine-tune, the weight changes ΔW are low-rank.

The model doesn't need to change in all d^2 directions. It only needs to adjust along a few important dimensions.

LoRA idea (Hu et al., 2022): don't update W directly. Instead, learn a low-rank decomposition of the change:

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

where $B \in \mathbb{R}^{d \times r}$ and $A \in \mathbb{R}^{r \times d}$, with $r \ll d$.

The Key Insight

Pre-trained weight matrices are $d \times d$ (e.g., $4096 \times 4096 = 16.8$M parameters each).

Observation (Aghajanyan et al., 2021): when you fine-tune, the weight changes ΔW are low-rank.

The model doesn't need to change in all d^2 directions. It only needs to adjust along a few important dimensions.

LoRA idea (Hu et al., 2022): don't update W directly. Instead, learn a low-rank decomposition of the change:

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

where $B \in \mathbb{R}^{d \times r}$ and $A \in \mathbb{R}^{r \times d}$, with $r \ll d$.

W is frozen. Only A and B are trained.

LoRA: The Parameter Savings

Original weight matrix $W \in \mathbb{R}^{d \times d}$: d^2 parameters.

LoRA: The Parameter Savings

Original weight matrix $W \in \mathbb{R}^{d \times d}$: d^2 parameters.

LoRA adapter $\Delta W = BA$: $d \times r + r \times d = 2dr$ parameters.

LoRA: The Parameter Savings

Original weight matrix $W \in \mathbb{R}^{d \times d}$: d^2 parameters.

LoRA adapter $\Delta W = BA$: $d \times r + r \times d = 2dr$ parameters.

Example: $d = 4096$, rank $r = 16$:

LoRA: The Parameter Savings

Original weight matrix $W \in \mathbb{R}^{d \times d}$: d^2 parameters.

LoRA adapter $\Delta W = BA$: $d \times r + r \times d = 2dr$ parameters.

Example: $d = 4096$, rank $r = 16$:

- Full matrix: $4096^2 = 16.8M$ parameters
- LoRA: $2 \times 4096 \times 16 = 131K$ parameters

LoRA: The Parameter Savings

Original weight matrix $W \in \mathbb{R}^{d \times d}$: d^2 parameters.

LoRA adapter $\Delta W = BA$: $d \times r + r \times d = 2dr$ parameters.

Example: $d = 4096$, rank $r = 16$:

- Full matrix: $4096^2 = 16.8M$ parameters
- LoRA: $2 \times 4096 \times 16 = 131K$ parameters
- That's $128\times$ fewer parameters

For a 7B model with LoRA on all attention matrices:

LoRA: The Parameter Savings

Original weight matrix $W \in \mathbb{R}^{d \times d}$: d^2 parameters.

LoRA adapter $\Delta W = BA$: $d \times r + r \times d = 2dr$ parameters.

Example: $d = 4096$, rank $r = 16$:

- Full matrix: $4096^2 = 16.8M$ parameters
- LoRA: $2 \times 4096 \times 16 = 131K$ parameters
- That's $128\times$ fewer parameters

For a 7B model with LoRA on all attention matrices:

- Full fine-tuning: 7B trainable parameters
- LoRA ($r = 16$): $\sim$20M trainable parameters ($< 0.3\%$)

LoRA: The Parameter Savings

Original weight matrix $W \in \mathbb{R}^{d \times d}$: d^2 parameters.

LoRA adapter $\Delta W = BA$: $d \times r + r \times d = 2dr$ parameters.

Example: $d = 4096$, rank $r = 16$:

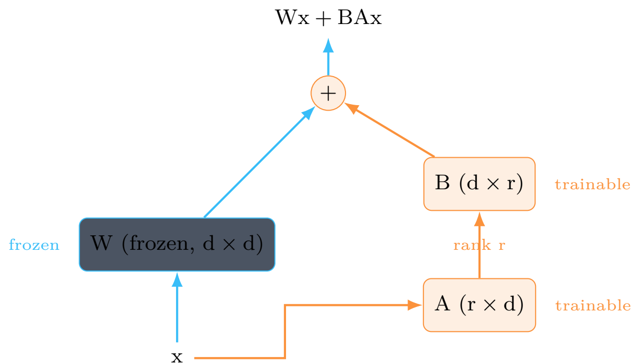
- Full matrix: $4096^2 = 16.8\text{M}$ parameters
- LoRA: $2 \times 4096 \times 16 = 131\text{K}$ parameters
- That's $128\times$ fewer parameters

For a 7B model with LoRA on all attention matrices:

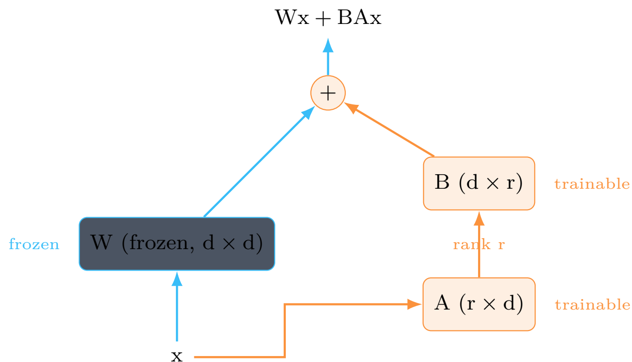
- Full fine-tuning: 7B trainable parameters
- LoRA ($r = 16$): $\sim$20M trainable parameters ($< 0.3\%$)

You can fine-tune a 7B model on a single consumer GPU (16 GB).

LoRA: Visually

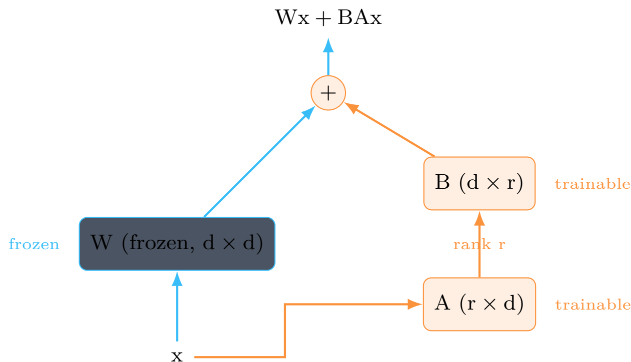


LoRA: Visually



At inference, you can merge: $W' = W + BA$. No extra latency. The LoRA adapter "disappears" into the weights.

LoRA: Visually



At inference, you can merge: $W' = W + BA$. No extra latency. The LoRA adapter "disappears" into the weights.

You can also swap adapters: same base model, different LoRA for different tasks.

How to Choose the Rank r

Rank r	Params (7B model)	Quality	Speed
4	~5M	Good for simple tasks	Very fast
16	~20M	Good for most tasks	Fast
64	~80M	Near full fine-tune	Moderate
256	~320M	Diminishing returns	Slower

How to Choose the Rank r

Rank r	Params (7B model)	Quality	Speed
4	$\sim$5M	Good for simple tasks	Very fast
16	$\sim$20M	Good for most tasks	Fast
64	$\sim$80M	Near full fine-tune	Moderate
256	$\sim$320M	Diminishing returns	Slower

$r = 16$ is the most common default. It works well for:

How to Choose the Rank r

Rank r	Params (7B model)	Quality	Speed
4	∼5M	Good for simple tasks	Very fast
16	∼20M	Good for most tasks	Fast
64	∼80M	Near full fine-tune	Moderate
256	∼320M	Diminishing returns	Slower

$r = 16$ is the most common default. It works well for:

- Instruction following
- Domain adaptation (medical, legal, code)
- Style transfer

How to Choose the Rank r

Rank r	Params (7B model)	Quality	Speed
4	∼5M	Good for simple tasks	Very fast
16	∼20M	Good for most tasks	Fast
64	∼80M	Near full fine-tune	Moderate
256	∼320M	Diminishing returns	Slower

$r = 16$ is the most common default. It works well for:

- Instruction following
- Domain adaptation (medical, legal, code)
- Style transfer

Higher r helps for tasks that require learning a lot of new knowledge (e.g., a new language). For most tasks, $r = 16$ to 64 is sufficient.

1. Inside Modern LLMs
2. Pre-training vs. Fine-tuning
3. LoRA: Low-Rank Adaptation
4. Supervised Fine-Tuning (SFT)
5. Expert Iteration
6. The Full Pipeline

What is SFT?

Supervised Fine-Tuning: train the model on (input, output) pairs.

What is SFT?

Supervised Fine-Tuning: train the model on (input, output) pairs.

The "input" is a prompt or question. The "output" is the desired response.

What is SFT?

Supervised Fine-Tuning: train the model on (input, output) pairs.

The “input” is a prompt or question. The “output” is the desired response.

Examples:

What is SFT?

Supervised Fine-Tuning: train the model on (input, output) pairs.

The “input” is a prompt or question. The “output” is the desired response.

Examples:

- Chat: (user message, assistant response)
- Math: (problem statement, step-by-step solution)

What is SFT?

Supervised Fine-Tuning: train the model on (input, output) pairs.

The "input" is a prompt or question. The "output" is the desired response.

Examples:

- Chat: (user message, assistant response)
- Math: (problem statement, step-by-step solution)
- Code: (function description, implementation)
- Instruction following: ("Translate to French: Hello", "Bonjour")

What is SFT?

Supervised Fine-Tuning: train the model on (input, output) pairs.

The "input" is a prompt or question. The "output" is the desired response.

Examples:

- Chat: (user message, assistant response)
- Math: (problem statement, step-by-step solution)
- Code: (function description, implementation)
- Instruction following: ("Translate to French: Hello", "Bonjour")

This is how base models become chat models. GPT-4 base → ChatGPT. Llama base → Llama-Chat.

Where Does SFT Data Come From?

Option 1: Human annotators

Where Does SFT Data Come From?

Option 1: Human annotators

- Companies like Scale AI, Mercor, Alignerr, and Invisible Technologies hire domain experts to write task-response pairs
- Doctors write medical Q&A, lawyers write legal Q&A, mathematicians write proofs

Where Does SFT Data Come From?

Option 1: Human annotators

- Companies like Scale AI, Mercor, Alignerr, and Invisible Technologies hire domain experts to write task-response pairs
- Doctors write medical Q&A, lawyers write legal Q&A, mathematicians write proofs
- Data then sold to frontier labs (OpenAI, Anthropic, Google) to SFT models

Where Does SFT Data Come From?

Option 1: Human annotators

- Companies like Scale AI, Mercor, Alignerr, and Invisible Technologies hire domain experts to write task-response pairs
- Doctors write medical Q&A, lawyers write legal Q&A, mathematicians write proofs
- Data then sold to frontier labs (OpenAI, Anthropic, Google) to SFT models

Option 2: Distillation from a stronger model

Where Does SFT Data Come From?

Option 1: Human annotators

- Companies like Scale AI, Mercor, Alignerr, and Invisible Technologies hire domain experts to write task-response pairs
- Doctors write medical Q&A, lawyers write legal Q&A, mathematicians write proofs
- Data then sold to frontier labs (OpenAI, Anthropic, Google) to SFT models

Option 2: Distillation from a stronger model

- Ask GPT-4 or Claude to solve 100K problems, use those solutions as training data

Where Does SFT Data Come From?

Option 1: Human annotators

- Companies like Scale AI, Mercor, Alignerr, and Invisible Technologies hire domain experts to write task-response pairs
- Doctors write medical Q&A, lawyers write legal Q&A, mathematicians write proofs
- Data then sold to frontier labs (OpenAI, Anthropic, Google) to SFT models

Option 2: Distillation from a stronger model

- Ask GPT-4 or Claude to solve 100K problems, use those solutions as training data

Option 3: Open datasets

Where Does SFT Data Come From?

Option 1: Human annotators

- Companies like Scale AI, Mercor, Alignerr, and Invisible Technologies hire domain experts to write task-response pairs
- Doctors write medical Q&A, lawyers write legal Q&A, mathematicians write proofs
- Data then sold to frontier labs (OpenAI, Anthropic, Google) to SFT models

Option 2: Distillation from a stronger model

- Ask GPT-4 or Claude to solve 100K problems, use those solutions as training data

Option 3: Open datasets

- GSM8K (8.5K grade school math problems with solutions)
- OpenMathInstruct, DeepSeek R1 traces, UltraChat

Where Does SFT Data Come From?

Option 1: Human annotators

- Companies like Scale AI, Mercor, Alignerr, and Invisible Technologies hire domain experts to write task-response pairs
- Doctors write medical Q&A, lawyers write legal Q&A, mathematicians write proofs
- Data then sold to frontier labs (OpenAI, Anthropic, Google) to SFT models

Option 2: Distillation from a stronger model

- Ask GPT-4 or Claude to solve 100K problems, use those solutions as training data

Option 3: Open datasets

- GSM8K (8.5K grade school math problems with solutions)
- OpenMathInstruct, DeepSeek R1 traces, UltraChat

Data quality matters enormously. 1,000 expert-written examples often beats 100,000 sloppy ones.

SFT Data Format

A typical SFT example for math reasoning:

SFT Data Format

A typical SFT example for math reasoning:

```
<|user|>
Solve: If 3x + 7 = 22, what is x?
<|assistant|>
Let me solve this step by step.
3x + 7 = 22
3x = 22 - 7 = 15
x = 15/3 = 5
The answer is x = 5.
<|endoftext|>
```

SFT Data Format

A typical SFT example for math reasoning:

```
<|user|>
Solve: If 3x + 7 = 22, what is x?
<|assistant|>
Let me solve this step by step.
3x + 7 = 22
3x = 22 - 7 = 15
x = 15/3 = 5
The answer is x = 5.
<|endoftext|>
```

The special tokens (<|user|>, <|assistant|>) tell the model which parts are the prompt and which are the response.

SFT: One Gradient Step (Walkthrough)

Take the example above. How does one training step work?

SFT: One Gradient Step (Walkthrough)

Take the example above. How does one training step work?

Step 1: Tokenize the full sequence (prompt + response):

SFT: One Gradient Step (Walkthrough)

Take the example above. How does one training step work?

Step 1: Tokenize the full sequence (prompt + response):

[<|user|>, Solve, :, If, 3x, +, 7, =, 22, <|assistant|>, Let, me, solve, ..., x, =, 5, ., <|eos|>]

SFT: One Gradient Step (Walkthrough)

Take the example above. How does one training step work?

Step 1: Tokenize the full sequence (prompt + response):

[<|user|>, Solve, :, If, 3x, +, 7, =, 22, <|assistant|>, Let, me, solve, ..., x, =, 5, ., <|eos|>]

Step 2: Build labels. Set label $= -100$ for prompt tokens (ignore them):

SFT: One Gradient Step (Walkthrough)

Take the example above. How does one training step work?

Step 1: Tokenize the full sequence (prompt + response):

[<|user|>, Solve, :, If, 3x, +, 7, =, 22, <|assistant|>, Let, me, solve, ..., x, =, 5, ., <|eos|>]

Step 2: Build labels. Set label $= -100$ for prompt tokens (ignore them):

labels: [-100, -100, -100, ..., -100, Let, me, solve, ..., x, =, 5, ., <|eos|>]

SFT: One Gradient Step (Walkthrough)

Take the example above. How does one training step work?

Step 1: Tokenize the full sequence (prompt + response):

[<|user|>, Solve, :, If, 3x, +, 7, =, 22, <|assistant|>, Let, me, solve, ..., x, =, 5, ., <|eos|>]

Step 2: Build labels. Set label $= -100$ for prompt tokens (ignore them):

labels: [-100, -100, -100, ..., -100, Let, me, solve, ..., x, =, 5, ., <|eos|>]

Step 3: Forward pass. Get logits at every position.

SFT: One Gradient Step (Walkthrough)

Take the example above. How does one training step work?

Step 1: Tokenize the full sequence (prompt + response):

[<|user|>, Solve, :, If, 3x, +, 7, =, 22, <|assistant|>, Let, me, solve, ..., x, =, 5, ., <|eos|>]

Step 2: Build labels. Set label $= -100$ for prompt tokens (ignore them):

labels: [-100, -100, -100, ..., -100, Let, me, solve, ..., x, =, 5, ., <|eos|>]

Step 3: Forward pass. Get logits at every position.

Step 4: Cross-entropy loss, but only on positions where label $\neq -100$. The loss measures: "did the model predict the correct response tokens?"

SFT: One Gradient Step (Walkthrough)

Take the example above. How does one training step work?

Step 1: Tokenize the full sequence (prompt + response):

[<|user|>, Solve, :, If, 3x, +, 7, =, 22, <|assistant|>, Let, me, solve, ..., x, =, 5, ., <|eos|>]

Step 2: Build labels. Set label $= -100$ for prompt tokens (ignore them):

labels: [-100, -100, -100, ..., -100, Let, me, solve, ..., x, =, 5, ., <|eos|>]

Step 3: Forward pass. Get logits at every position.

Step 4: Cross-entropy loss, but only on positions where label $\neq -100$. The loss measures: "did the model predict the correct response tokens?"

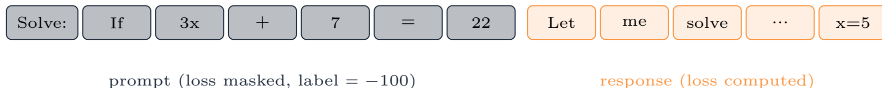
Step 5: Backward + optimizer step. Update only LoRA parameters (or all, if full fine-tuning).

The Key Trick: Masking the Loss

During SFT, we only compute the loss on the response tokens.

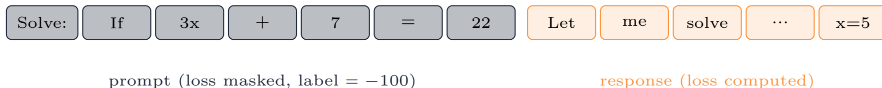
The Key Trick: Masking the Loss

During SFT, we only compute the loss on the response tokens.



The Key Trick: Masking the Loss

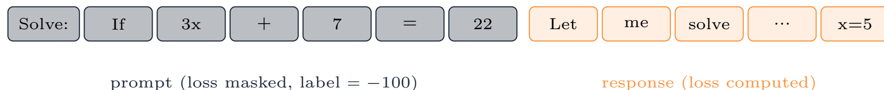
During SFT, we only compute the loss on the response tokens.



Why mask the prompt?

The Key Trick: Masking the Loss

During SFT, we only compute the loss on the response tokens.

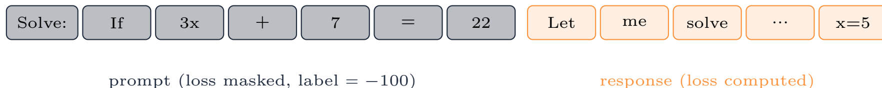


Why mask the prompt?

- We don't want the model to learn to generate questions
- We only want it to learn to answer them

The Key Trick: Masking the Loss

During SFT, we only compute the loss on the response tokens.

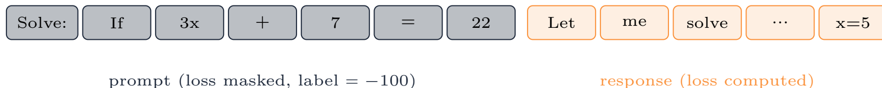


Why mask the prompt?

- We don't want the model to learn to generate questions
- We only want it to learn to answer them
- The prompt tokens still contribute to the forward pass (they provide context), but the gradient only flows from the response

The Key Trick: Masking the Loss

During SFT, we only compute the loss on the response tokens.



Why mask the prompt?

- We don't want the model to learn to generate questions
- We only want it to learn to answer them
- The prompt tokens still contribute to the forward pass (they provide context), but the gradient only flows from the response

SFT Training Loop (Code)

```
def sft_step(model, batch):
    input_ids = batch["input_ids"]        # full sequence
    labels = batch["labels"]              # -100 for prompt, token_id for response

    logits = model(input_ids).logits      # (B, T, vocab_size)

    # Cross-entropy loss, ignoring positions where label == -100
    loss = F.cross_entropy(
        logits[:, :-1].reshape(-1, vocab_size),
        labels[:, 1:].reshape(-1),
        ignore_index=-100,                # skip prompt tokens
    )

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

SFT Training Loop (Code)

```
def sft_step(model, batch):
    input_ids = batch["input_ids"]        # full sequence
    labels = batch["labels"]              # -100 for prompt, token_id for response

    logits = model(input_ids).logits      # (B, T, vocab_size)

    # Cross-entropy loss, ignoring positions where label == -100
    loss = F.cross_entropy(
        logits[:, :-1].reshape(-1, vocab_size),
        labels[:, 1:].reshape(-1),
        ignore_index=-100,                # skip prompt tokens
    )

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

PyTorch's ignore_index=-100 does the masking for us. Set label to -100 for all prompt positions, and the actual token ID for response positions.

LoRA SFT: Full Working Example (1/2)

We'll fine-tune Qwen2.5-Math-1.5B on GSM8K (grade school math, 8.5K problems with step-by-step solutions). This runs on a single A100 in ~30 minutes.

```
# pip install transformers peft trl datasets
from transformers import AutoModelForCausalLM, AutoTokenizer
from peft import LoraConfig, get_peft_model
from datasets import load_dataset  # HuggingFace datasets library
from trl import SFTTrainer, SFTConfig

# 1. Load base model + tokenizer
model = AutoModelForCausalLM.from_pretrained(
    "Qwen/Qwen2.5-Math-1.5B", torch_dtype="bfloat16"
)
tokenizer = AutoTokenizer.from_pretrained("Qwen/Qwen2.5-Math-1.5B")
```

LoRA SFT: Full Working Example (1/2)

We'll fine-tune Qwen2.5-Math-1.5B on GSM8K (grade school math, 8.5K problems with step-by-step solutions). This runs on a single A100 in ∼30 minutes.

```
# pip install transformers peft trl datasets
from transformers import AutoModelForCausalLM, AutoTokenizer
from peft import LoraConfig, get_peft_model
from datasets import load_dataset  # HuggingFace datasets library
from trl import SFTTrainer, SFTConfig

# 1. Load base model + tokenizer
model = AutoModelForCausalLM.from_pretrained(
    "Qwen/Qwen2.5-Math-1.5B", torch_dtype="bfloat16"
)
tokenizer = AutoTokenizer.from_pretrained("Qwen/Qwen2.5-Math-1.5B")
```

datasets is HuggingFace's library for loading open datasets. trl (Transformer Reinforcement Learning) provides SFTTrainer.

LoRA SFT: Full Working Example (2/2)

```
# 2. Add LoRA adapters (freeze base, train only A and B)
lora_config = LoraConfig(
    r=16, lora_alpha=32,
    target_modules=["q_proj", "k_proj", "v_proj", "o_proj"],
)
model = get_peft_model(model, lora_config)
model.print_trainable_parameters()
# => trainable: 3,407,872 (0.22% of 1.5B)

# 3. Load GSM8K from HuggingFace Hub (auto-downloads)
dataset = load_dataset("openai/gsm8k", "main", split="train")
# Each example: {"question": "...", "answer": "...step by step..."}

# 4. Train
trainer = SFTTrainer(
    model=model, tokenizer=tokenizer,
    train_dataset=dataset,
    args=SFTConfig(num_train_epochs=3, per_device_train_batch_size=4,
                   learning_rate=2e-4, max_seq_length=1024,
                   output_dir="./gsm8k-lora"),
)
trainer.train()  # ~30 min on 1 A100
model.save_pretrained("./gsm8k-lora")  # saves only the LoRA weights (~13 MB)
```

What SFT Actually Changes

Before SFT (base model):

What SFT Actually Changes

Before SFT (base model):

- Predicts the next token well on web text
- If you ask a math question, it might continue with more questions (web forum style)

What SFT Actually Changes

Before SFT (base model):

- Predicts the next token well on web text
- If you ask a math question, it might continue with more questions (web forum style)
- No consistent response format

After SFT:

What SFT Actually Changes

Before SFT (base model):

- Predicts the next token well on web text
- If you ask a math question, it might continue with more questions (web forum style)
- No consistent response format

After SFT:

- Given a question, it reliably produces step-by-step reasoning
- Follows the format it was trained on

What SFT Actually Changes

Before SFT (base model):

- Predicts the next token well on web text
- If you ask a math question, it might continue with more questions (web forum style)
- No consistent response format

After SFT:

- Given a question, it reliably produces step-by-step reasoning
- Follows the format it was trained on
- Knows when to stop (produces EOS after the answer)

SFT doesn't add new knowledge. It teaches the model a new behavior: "when you see a question, produce a structured answer."

1 Inside Modern LLMs
2 Pre-training vs. Fine-tuning
3 LoRA: Low-Rank Adaptation
4 Supervised Fine-Tuning (SFT)
5 Expert Iteration
6 The Full Pipeline

SFT Has a Ceiling

SFT learns by imitating training examples.

SFT Has a Ceiling

SFT learns by imitating training examples.

Problem: the model can never be better than its training data.

SFT Has a Ceiling

SFT learns by imitating training examples.

Problem: the model can never be better than its training data.

If your training data was generated by GPT-4:

SFT Has a Ceiling

SFT learns by imitating training examples.

Problem: the model can never be better than its training data.

If your training data was generated by GPT-4:

- Your model learns to imitate GPT-4's style
- But it can't surpass GPT-4's reasoning ability
- It copies the surface patterns, not the deep understanding

SFT Has a Ceiling

SFT learns by imitating training examples.

Problem: the model can never be better than its training data.

If your training data was generated by GPT-4:

- Your model learns to imitate GPT-4's style
- But it can't surpass GPT-4's reasoning ability
- It copies the surface patterns, not the deep understanding

This is the imitation ceiling: SFT can only redistribute existing capability, not create new capability.

SFT Has a Ceiling

SFT learns by imitating training examples.

Problem: the model can never be better than its training data.

If your training data was generated by GPT-4:

- Your model learns to imitate GPT-4's style
- But it can't surpass GPT-4's reasoning ability
- It copies the surface patterns, not the deep understanding

This is the imitation ceiling: SFT can only redistribute existing capability, not create new capability.

How do we go beyond the training data?

Expert Iteration (STaR)

Self-Taught Reasoner (Zelikman et al., 2022): the model generates its own training data.

Expert Iteration (STaR)

Self-Taught Reasoner (Zelikman et al., 2022): the model generates its own training data.

The loop:

Expert Iteration (STaR)

Self-Taught Reasoner (Zelikman et al., 2022): the model generates its own training data.

The loop:

1. Sample: model generates N solutions per problem

Expert Iteration (STaR)

Self-Taught Reasoner (Zelikman et al., 2022): the model generates its own training data.

The loop:

1. Sample: model generates N solutions per problem
2. Filter: keep only the solutions that produce the correct answer

Expert Iteration (STaR)

Self-Taught Reasoner (Zelikman et al., 2022): the model generates its own training data.

The loop:

1. Sample: model generates N solutions per problem
2. Filter: keep only the solutions that produce the correct answer
3. Retrain: SFT on the filtered (correct) solutions

Expert Iteration (STaR)

Self-Taught Reasoner (Zelikman et al., 2022): the model generates its own training data.

The loop:

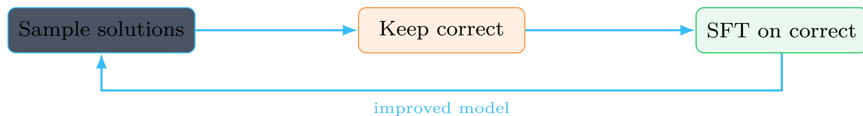
1. Sample: model generates N solutions per problem
2. Filter: keep only the solutions that produce the correct answer
3. Retrain: SFT on the filtered (correct) solutions
4. Repeat: go back to step 1 with the improved model

Expert Iteration (STaR)

Self-Taught Reasoner (Zelikman et al., 2022): the model generates its own training data.

The loop:

1. Sample: model generates N solutions per problem
2. Filter: keep only the solutions that produce the correct answer
3. Retrain: SFT on the filtered (correct) solutions
4. Repeat: go back to step 1 with the improved model



Why Expert Iteration Works

Key insight: the model can sometimes solve problems it can't consistently solve.

Why Expert Iteration Works

Key insight: the model can sometimes solve problems it can't consistently solve.

If you sample 100 solutions to a hard problem:

Why Expert Iteration Works

Key insight: the model can sometimes solve problems it can't consistently solve.

If you sample 100 solutions to a hard problem:

- Maybe 3 out of 100 are correct
- Those 3 correct solutions are valid reasoning traces

Why Expert Iteration Works

Key insight: the model can sometimes solve problems it can't consistently solve.

If you sample 100 solutions to a hard problem:

- Maybe 3 out of 100 are correct
- Those 3 correct solutions are valid reasoning traces
- Train on those 3, and now the model solves it 10/100 times
- Repeat: $10/100 \to 30/100 \to 60/100$

Why Expert Iteration Works

Key insight: the model can sometimes solve problems it can't consistently solve.

If you sample 100 solutions to a hard problem:

- Maybe 3 out of 100 are correct
- Those 3 correct solutions are valid reasoning traces
- Train on those 3, and now the model solves it 10/100 times
- Repeat: $10/100 \to 30/100 \to 60/100$

The model is generating training data that is better than its average performance. It bootstraps itself to higher capability.

Why Expert Iteration Works

Key insight: the model can sometimes solve problems it can't consistently solve.

If you sample 100 solutions to a hard problem:

- Maybe 3 out of 100 are correct
- Those 3 correct solutions are valid reasoning traces
- Train on those 3, and now the model solves it 10/100 times
- Repeat: $10/100 \to 30/100 \to 60/100$

The model is generating training data that is better than its average performance. It bootstraps itself to higher capability.

Limitation: this only works when you can verify correctness. Math problems: check the answer. Code: run the tests. Open-ended text: no automatic verifier.

Expert Iteration: Results

On GSM8K (Zelikman et al., 2022; Singh et al., 2024):

Expert Iteration: Results

On GSM8K (Zelikman et al., 2022; Singh et al., 2024):

Method	Accuracy	Source
GPT-J 6B (base)	12%	Zelikman et al.
+ STaR (expert iteration)	25%	Zelikman et al.
Llama-2 7B + SFT	41%	Singh et al.
+ ReST^{EM} (1 iteration)	49%	Singh et al.
+ ReST^{EM} (3 iterations)	55%	Singh et al.

Expert Iteration: Results

On GSM8K (Zelikman et al., 2022; Singh et al., 2024):

Method	Accuracy	Source
GPT-J 6B (base)	12%	Zelikman et al.
+ STaR (expert iteration)	25%	Zelikman et al.
Llama-2 7B + SFT	41%	Singh et al.
+ ReST^{EM} (1 iteration)	49%	Singh et al.
+ ReST^{EM} (3 iterations)	55%	Singh et al.

Each round of expert iteration pushes past the SFT ceiling.

Expert Iteration: Results

On GSM8K (Zelikman et al., 2022; Singh et al., 2024):

Method	Accuracy	Source
GPT-J 6B (base)	12%	Zelikman et al.
+ STaR (expert iteration)	25%	Zelikman et al.
Llama-2 7B + SFT	41%	Singh et al.
+ ReST$^{\text{EM}}$ (1 iteration)	49%	Singh et al.
+ ReST$^{\text{EM}}$ (3 iterations)	55%	Singh et al.

Each round of expert iteration pushes past the SFT ceiling.

But we're still limited by the binary filter (correct/incorrect). What if we could give the model a continuous reward signal? That's RL, next lecture.

1 Inside Modern LLMs

2 Pre-training vs. Fine-tuning

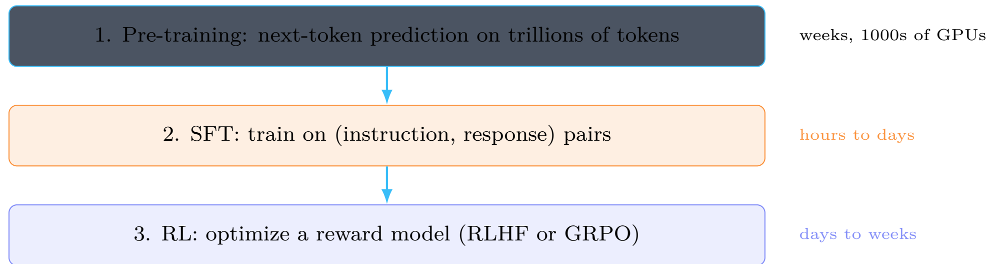
3 LoRA: Low-Rank Adaptation

4 Supervised Fine-Tuning (SFT)

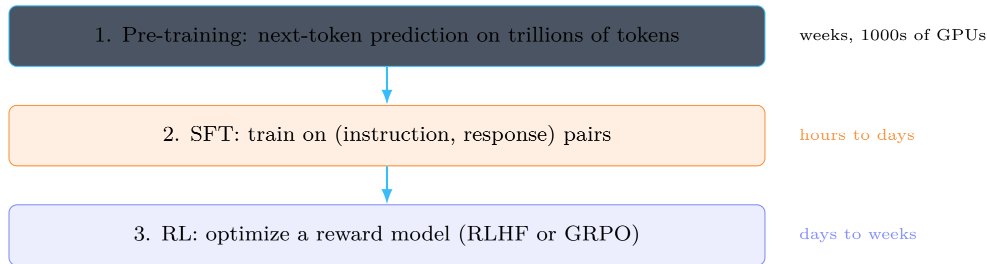
5 Expert Iteration

6 The Full Pipeline

How ChatGPT/Claude Are Built

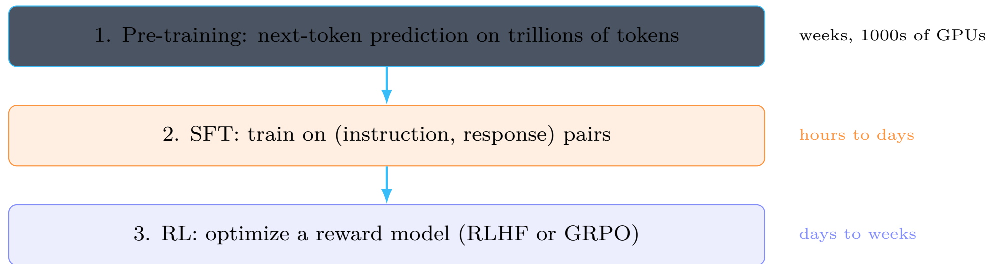


How ChatGPT/Claude Are Built



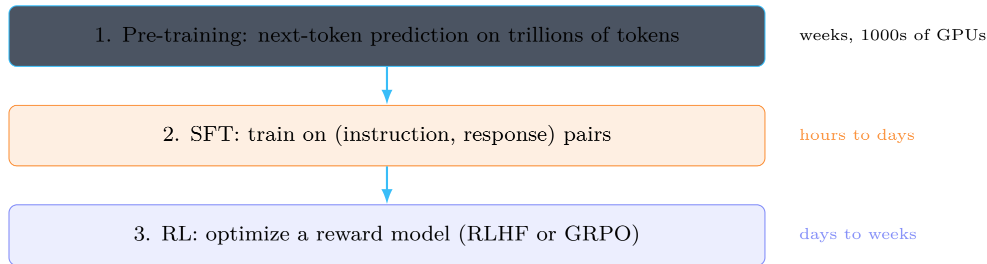
Stage 1 (L26): learn language from raw text.

How ChatGPT/Claude Are Built



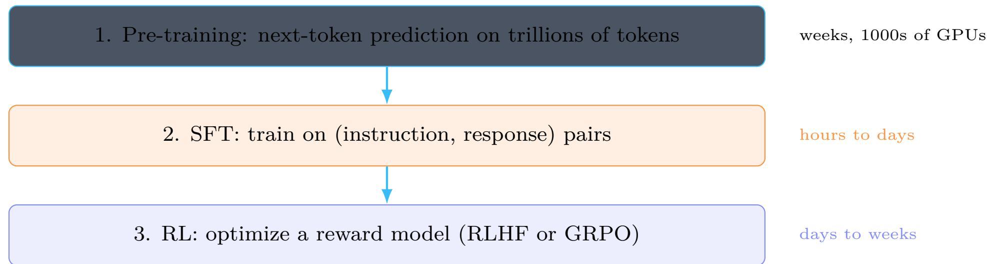
Stage 1 (L26): learn language from raw text.
Stage 2 (this lecture): learn to follow instructions and produce useful output.

How ChatGPT/Claude Are Built



Stage 1 (L26): learn language from raw text.
Stage 2 (this lecture): learn to follow instructions and produce useful output.
Stage 3 (next lecture): learn to reason, be helpful, and avoid harmful output.

How ChatGPT/Claude Are Built



Stage 1 (L26): learn language from raw text.
Stage 2 (this lecture): learn to follow instructions and produce useful output.
Stage 3 (next lecture): learn to reason, be helpful, and avoid harmful output.

Each stage builds on the previous one. You can't skip stages.

Practical Advice

When to use each approach:

Practical Advice

When to use each approach:

Situation	Method
Quick prototype	Prompting
Consistent format	Few-shot prompting
Domain-specific behavior	LoRA SFT
New reasoning capability	Expert iteration
Frontier reasoning	RL (next lecture)

Practical Advice

When to use each approach:

Situation	Method
Quick prototype	Prompting
Consistent format	Few-shot prompting
Domain-specific behavior	LoRA SFT
New reasoning capability	Expert iteration
Frontier reasoning	RL (next lecture)

Start simple. Try prompting first. If that's not enough, try LoRA SFT. Only go to RL if you need the model to discover new strategies.

Practical Advice

When to use each approach:

Situation	Method
Quick prototype	Prompting
Consistent format	Few-shot prompting
Domain-specific behavior	LoRA SFT
New reasoning capability	Expert iteration
Frontier reasoning	RL (next lecture)

Start simple. Try prompting first. If that's not enough, try LoRA SFT. Only go to RL if you need the model to discover new strategies.

Final lecture: GRPO and reasoning RL, where the model discovers reasoning strategies instead of imitating them.