LoRA / PEFT: Encoder vs Decoder

ECBS5200 — Week 3

What happens when you only train 2% of the model?

Where we left off

Model	Accuracy	Macro F1	Zero-F1
TF-IDF + LogReg	54.2%	0.132	70
Full fine-tune (3 ep)	56.6%	0.209	46
+ class weighting	~52%	~0.24	~37

Class weighting rescued ~9 classes from zero.

But it cost 4-5 points of accuracy. **And 37 classes still get nothing.**

This week

Two ideas:

1. **LoRA** — train a tiny fraction of parameters and get the same quality
2. **Encoder vs decoder** — a different architecture on the same task

The lecture covers the theory. The lab tests both.

Today's plan

Block 1 — Lecture (after the quiz)

- Full fine-tuning: what's actually expensive?
- LoRA: the idea, the math, the practice
- Encoders vs decoders: what's the difference?
- A decoder for classification: why it might work
- What pretraining scale buys you for rare classes

Block 2 — Lab

- Apply LoRA to the encoder yourself
- Compare encoder LoRA vs a pre-trained decoder

Quick poll

You fine-tuned ModernBERT in Week 1. All 149M parameters.

What fraction of those parameters do you think actually need to change to solve this task?

(a) All of them

(b) About half

(c) About 10%

(d) Less than 5%

The Cost of Fine-Tuning

Why do we care about parameter efficiency?

What full fine-tuning actually requires

ModernBERT-base: **149 million parameters.**

To fine-tune all of them, you need:

- **Memory for the model** — 4 bytes × 149M = ~600 MB (float32)
- **Memory for gradients** — another 600 MB
- **Memory for optimizer states** — AdamW keeps 2 states per param = 1.2 GB
- **Total:** ~2.4 GB just for parameters, before a single example

And if you want 10 fine-tuned models for 10 tasks? **6 GB of checkpoints on disk** — 24 GB of GPU state during training.

The intuition

When you fine-tune a model on a downstream task, how much does each weight actually change?

Aghajanyan et al. (2020) showed: **the weight updates lie in a low-dimensional subspace.**

The full weight matrix is $768 \times 768 = 589{,}824$ values.
But the meaningful change might only occupy a subspace of dimension 16.

LoRA exploits this.

LoRA: Low-Rank Adaptation

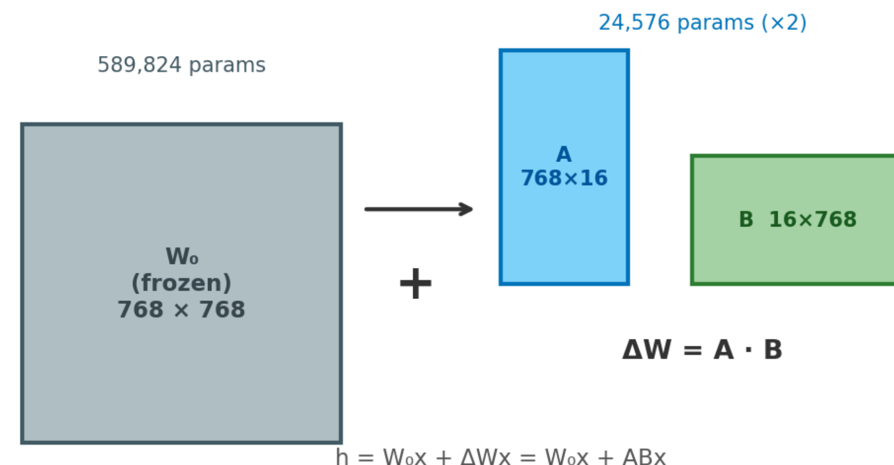
Hu et al. (2021) — the original LoRA paper.

Freeze the pretrained weight W_0.
Learn two small matrices: **A** (768 × 16) and **B** (16 × 768).

$$h = W_0 x + ABx$$

A × B has the same shape as W_0, but only 24,576 trainable parameters instead of 589,824.

`readings/week3/hu2021_lora.pdf`



The numbers: LoRA on GPT-3 175B

From the original paper:

	Full FT	LoRA (r=4)
Trainable params	175B	**17.5M**
GPU memory	1.2 TB	**350 GB**
Checkpoint size	350 GB	**35 MB**
Inference latency	baseline	**same**
GLUE performance	baseline	**same or better**

10,000× fewer parameters. 3× less memory. **Same quality. Same inference speed.**

The "same inference latency" line matters: at deploy time, you compute $W_0 + AB$ once, merge it into the base weights, and the model runs exactly like the original. No extra layers. No added latency. That's why LoRA beat earlier approaches like adapters.

Worked example: the math

ModernBERT's `Wqkv` layer: **2,304 × 768** = 1,769,472 parameters.

With LoRA rank 16:

- A: 768 × 16 = **12,288** parameters
- B: 16 × 2,304 = **36,864** parameters
- Total: **49,152** — that's **2.8%** of the original layer

Each layer has 4 module types (Wqkv, Wo, Wi, Wo2) — different sizes, about 150K trainable params per layer. Across 22 layers + the classifier head: **~3.5M total.**

2.3% of the model.

LoRA in practice

In the lab, you'll configure LoRA on ModernBERT:

```
LoraConfig(
    task_type="SEQ_CLS",
    r=???,                      # Rank — you choose
    lora_alpha=32,
    target_modules=???,         # Which layers — you choose
)
```

You'll explore the model's layer names and decide which modules to target.

The result: ~3.5M trainable parameters out of 149M. About **2.3%**.

Encoders vs Decoders

Two families of transformer. Same building blocks, different training.

Encoder: bidirectional

ModernBERT, BERT, RoBERTa, DeBERTa

- Trained with **masked language modeling** — predict missing tokens
- Every token attends to every other token (bidirectional)
- Designed for understanding: classification, NER, similarity

149M parameters. Pretrained on ~2 trillion tokens of English web text.

Decoder: left-to-right

GPT, Llama, Qwen, Gemma

- Trained with **next-token prediction** — predict what comes next
- Each token attends only to tokens before it (causal mask)
- Designed for generation: chatbots, code, creative writing

Qwen 2.5-0.5B: **494M parameters.** Pretrained on ~**18 trillion tokens** (multilingual).

Why 2.5, not the newer Qwen 3? We tested Qwen 3-0.6B — it scored slightly higher (+0.009 F1) but took 2× longer to train on T4. Not worth the trade-off for a homework you need to complete.

A decoder for classification?

The default instinct is still: **encoders for classification, decoders for generation.**

So why would we use a decoder on a classification task?

One reason: we're betting that more pretraining data buys us better representations of rare concepts — and we're willing to pay more at inference to get them.

Classification head on a decoder

Yousefiramandi & Cooney (2025) tested exactly this.

Two approaches:

1. **Classification head** on the last token embedding → one forward pass → 113 logits
2. **Instruction tuning** → generate the label as text → parse the output

The classification head approach **significantly outperformed** instruction tuning.

And was competitive with fine-tuned BERT baselines.

`readings/week3/yousefiramandi2025_decoder_cls.pdf`

Why might a decoder work for classification?

The decoder reads left-to-right. The **last token** has attended to everything before it.

```
[complaint text tokens...] → last hidden state → classification head → 113 logits
```

The last token's representation is a summary of the entire input.
Analogous to the encoder's `[CLS]` token — but built left-to-right.

But there's a catch: padding must go on the **left**, not the right. Otherwise the "last token" is a pad token.

What the decoder sees

```
Input:   "I was charged a fee for a convenience check I never ordered"

Encoder: every token attends to every other token (bidirectional)
         → pool all tokens → classify

Decoder: each token attends to tokens BEFORE it (causal)
         → last token has seen everything → classify from that token
```

Both produce a single 113-dimensional logit vector. Same loss function. Same evaluation.

The architectural difference is in **how the representation is built**, not what it's used for.

The comparison

In the lab, you'll compare these two models head-to-head:

	Encoder (ModernBERT)	Decoder (Qwen 0.5B)
Parameters	149M	494M
LoRA trainable	~2.3%	~0.46%
Pretraining data	~2T tokens	~18T tokens
Architecture	Bidirectional	Left-to-right
Training	Same data, same epochs, same LoRA rank	

Same task. Same LoRA. Different backbone. What do you expect?

Important: this is an applied comparison, not a controlled experiment. Architecture, model size, and pretraining data all differ. That's realistic — in practice you rarely get to isolate one variable. Your job is to reason about the result given all the differences.

What pretraining buys you

Especially for the long tail.

The rare-class problem

Your encoder gets zero F1 on the rarest ~29 classes.

Those classes have 4-27 training examples each.

Is that enough to learn from?

The encoder was pretrained on ~2T tokens — it knows something about language. But its exposure to concepts like "Convenience checks" or "Applying for a mortgage" may be thin. Fine-tuning on 4-27 examples has to bridge whatever gap remains.

Competing hypotheses for rare-class performance

If the decoder outperforms on rare classes, **why?** At least three explanations:

1. **Pretraining exposure** — 18T tokens vs 2T means more documents about rare concepts
2. **Parameter count** — 494M vs 149M gives more capacity to separate rare classes
3. **Causal representation structure** — the last-token summary may encode tail concepts differently than bidirectional pooling

> **You cannot distinguish these from your data alone.**

The evidence will point in one direction; multiple mechanisms could be responsible.

The paper: Kandpal et al. 2023

"Large Language Models Struggle to Learn Long-Tail Knowledge" (ICML 2023)

Key finding: **model accuracy is strongly correlated with how many relevant documents appeared in pretraining.**

- More pretraining documents → better performance
- Larger models help, but only gradually
- To handle truly rare knowledge competitively, models would need to scale by **"many orders of magnitude"**

`readings/week3/kandpal2023_long_tail_knowledge.pdf`

What this means for your lab

The encoder and decoder differ in architecture, scale, and pretraining exposure.

The question: do those differences show up in your metrics? And if so, **where** — in the aggregates, or somewhere more specific?

Keep Kandpal's finding in mind when you look at rare-class behavior.

The speed question

Quality is half the story. The other half: **how fast does it run?**

	Encoder	Decoder	Ratio
Parameters	149M	494M	3.3×
Batched inference (ms/ex)	~10	~27	**2.8×**
Throughput (ex/sec)	~100	~37	2.7×
64K complaints	~10 min	~29 min	2.9×

At scale: **2.8× means 2.8× the GPUs to serve the same traffic.**

The difference between $1M/year and $2.8M/year in GPU spend is not nothing.

A different question about rare classes

In Week 2, class weighting improved rare-class F1 on the encoder.

Would the same trick work on the decoder?

Think about what class weighting does: it changes the loss function to pay more attention to rare classes.

Will the same trick that helped the encoder also help the decoder? Form a hypothesis before you run it.

You'll test this in the homework.

Practical LoRA decisions

Things you need to know for the lab.

Every transformer layer has the same parts

```
Input
 ↓
Attention block  → (residual + norm)
 ↓
Feed–forward block → (residual + norm)
 ↓
Output
```

Each block is a handful of linear layers stacked together.

Every linear layer is a candidate for LoRA.

Attention: Q, K, V, O

Attention lets each token look at other tokens. To do that, every token gets projected three ways:

- **Query (Q)** — "what am I looking for?"
- **Key (K)** — "what do I offer?"
- **Value (V)** — "what information do I carry?"

Attention = similarity(Q, K) → weights → weighted sum of V.

Then an **Output (O) projection** reshapes the result back into the token stream.

Four linear layers per attention block.

Attention naming: fused vs separate

Same four projections — different names in different models.

ModernBERT (fused):

- `Wqkv` — one matrix does Q, K, and V together (faster: single matmul)
- `Wo` — output projection

Qwen (separate):

- `q_proj`, `k_proj`, `v_proj` — one matrix each
- `o_proj` — output projection

Math is identical. Fusion is a speed optimization.

Feed-forward: expand and contract

After attention, each token goes through a small 2-layer MLP:

- **Up projection** — expand hidden dim (e.g., 768 → 3072)
- **Activation** — a nonlinearity (GeLU, SiLU, …)
- **Down projection** — contract back (3072 → 768)

Each token "thinks" on its own in a wider space, then gets compressed back.

Classic FFN: 2 linear layers. SwiGLU: 3 (adds a gate).

Summary: same roles, different names

Role	ModernBERT	Qwen
Q/K/V projections	`Wqkv` (fused)	`q_proj`, `k_proj`, `v_proj`
Attention output	`Wo`	`o_proj`
FFN expansion	`Wi`	`up_proj` + `gate_proj`
FFN contraction	`Wo2`	`down_proj`

In the lab, run `print(model)` to see the real names.

Then decide: attention only (classic LoRA), or attention + FFN (more capacity, more params).

Rank: how many dimensions?

The rank `r` controls the size of the low-rank matrices.

Rank	Params per layer	Typical use
4	~6K	Light adaptation
16	~25K	Standard choice
64	~100K	Heavy adaptation

Higher rank = more capacity, but diminishing returns.

Biderman et al. (2024) showed: **LoRA learns less than full fine-tuning**, but also **forgets less**. The rank controls the trade-off.

Decoder-specific setup

Three things to get right for the decoder:

1. `padding_side = "left"` — causal LMs read from the left; classify from the right
2. `modules_to_save = ["score"]` — the classification head is randomly initialized; must be saved
3. `dtype = torch.float32` at load, `fp16=True` in training — T4 can't handle bfloat16 GradScaler

Get any of these wrong and training fails silently.

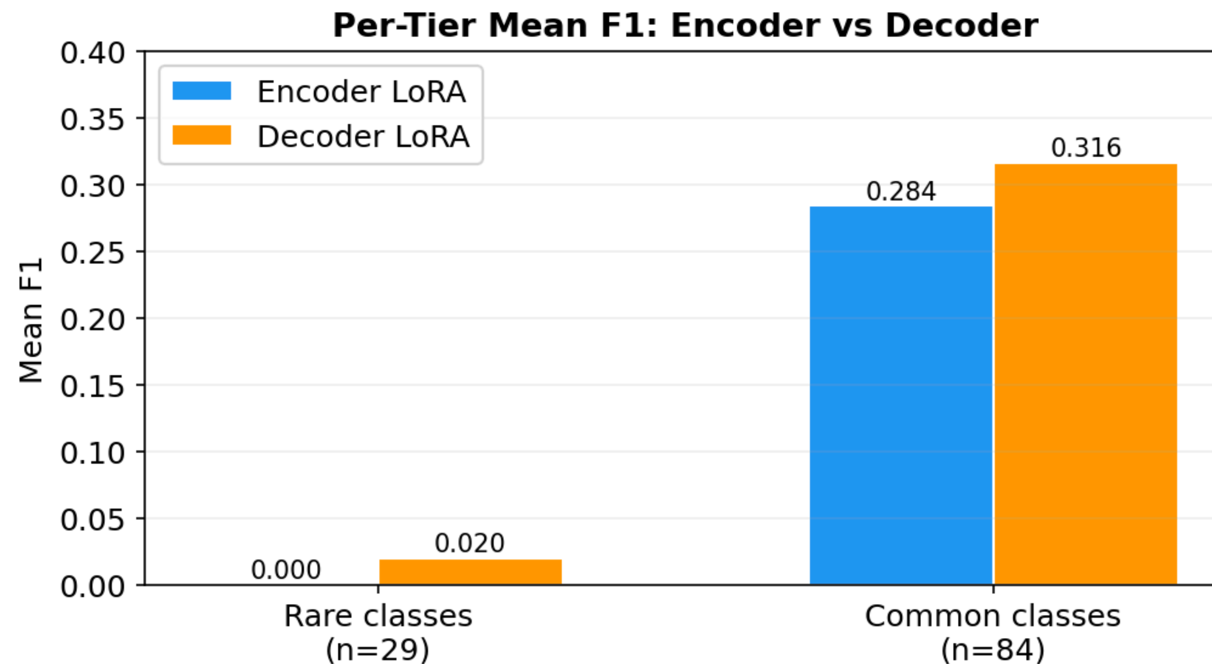
Where to look in the lab

The aggregate metrics will tell you **who** wins.

The per-class analysis will tell you **where** and **why**.

Break the 113 classes into frequency tiers. Compare encoder vs decoder on each tier.

This is the analysis that earns the most points on the rubric.



Make your predictions now

Before the lab, write down:

1. **Will LoRA match full fine-tuning on the encoder?** (your full FT: ~56.6% acc, ~0.209 F1)
2. **Which model wins on macro F1 — encoder or decoder?** By how much?
3. **Where will the decoder's advantage be largest — rare classes or common classes?**

Hold these predictions. You'll check them in 90 minutes.

The lab

Encoder LoRA → Decoder reveal → Per-class comparison

Lab structure (~80 min)

1. **Configure LoRA** on the encoder — you choose the rank and target modules (~5 min)
2. **Train encoder LoRA** — 3 epochs, ~32 min on T4 (you write predictions while it trains)
3. **Evaluate encoder** — compare to your full fine-tune from Week 1 (~5 min)
4. **Load the decoder** — pre-trained, from HuggingFace Hub (~5 min)
5. **The reveal** — side-by-side comparison (~5 min)
6. **Per-class analysis** — rare vs common, scatter plot, interpret (~15 min)

There is a **pre-trained encoder fallback** on HuggingFace Hub if training doesn't finish.

Your homework

You train the decoder yourself.

- Configure LoRA on Qwen 0.5B
- Train with standard cross-entropy (vanilla)
- Apply class weighting to the decoder
- Benchmark latency: encoder vs decoder
- Write your memo (5 sections)

The central question: **given your results, what would you deploy — and in what context?**

Due: Wednesday morning before Week 4 class. HTML via Moodle.

The rubric at a glance

Section	Points	Focus
1. Systems-level comparison	20	Architecture, parameter efficiency, aggregates
2. Class weighting on decoder	20	Does the Week 2 trick transfer? Why or why not?
3. Per-class analysis	**25**	Where do the models differ? Rare vs common
4. Latency + deployment	20	Translate speed into a recommendation
5. What you'd do next	15	Identify the bottleneck, propose an experiment

Section 3 carries the most weight. The aggregate metrics hide the important story.

Week 3 Reading List

Paper	Year	Key idea
Hu et al. — LoRA	**2021**	**Low-rank weight updates = full FT quality**
Yousefiramandi & Cooney — Decoder Classification	**2025**	**Cls heads on decoders beat generation**
Kandpal et al. — Long-Tail Knowledge	**2023**	**Accuracy tracks pretraining document count**
Houlsby et al. — Adapters	2019	The PEFT predecessor
He et al. — Unified View of PEFT	2022	Why LoRA, adapters, and prefix tuning are variations on one idea
Dettmers et al. — QLoRA	2023	LoRA + 4-bit quantization
Weller et al. — Seq vs Seq	2025	Controlled encoder-vs-decoder comparison
BehnamGhader et al. — LLM2Vec	2024	Decoders are secretly good encoders
Valdes Gonzalez — Cost-Aware Selection	2026	Pareto frontier for quality vs latency

All PDFs in `readings/week3/`. **Bold = covered in lecture.**

A mental model for model choice

When you face this decision in the real world, the shape is:

Situation	Reach for
Common classes, tight latency budget	**Encoder + LoRA**
Rare classes matter, latency budget exists	**Decoder + LoRA**
Maximum accuracy, cost is no object	Full fine-tune (or a larger model)
Many fine-tuned variants served concurrently	Adapters (not LoRA)

The homework's deployment question asks you to identify which situation you're in — and justify it with evidence from your own results.

Next week

We go deeper on the encoder-vs-decoder trade-off.

New tools. New interventions. The question sharpens.