Practical Deep Learning Engineering for Applied ML

ECBS5200 — Week 1

A model is an artifact, not a service endpoint.

What this course IS

Post-training engineering: you take a pretrained model and make it yours.

- **Fine-tune** it on your data
- **Adapt** it cheaply with LoRA
- **Analyze** its errors systematically
- **Compress** it via quantization
- **Distill** its knowledge into a smaller model
- **Justify** your final recommendation with evidence

You will do all of this on **one real task**, building up a single model line over the semester.

What this course IS NOT

- **Not prompt engineering.** You won't be writing system prompts.
- **Not an agents course.** No tool-use, no chains, no orchestration.
- **Not an LLM apps course.** You may have done that already — great.
- **Not a theory course.** We care about what works, what it costs, and why.

We are **inside the model**, not outside it.

If your prior ML experience taught you to call APIs and parse outputs, this course teaches you what's happening on the other side of that API.

The semester arc

Week	Topic
1	Fine-tune a real model, audit the data
2	Improve: learning rate schedules, class weights, error analysis
3	Adapt cheaply with LoRA / PEFT
4	Analyze: confusion matrices, per-class deep dives
5	Compress: quantization (INT8, INT4)
6	Distill + final engineering recommendation

Each week builds on the last. You keep the same dataset. You keep the same model family. The artifact evolves.

The task: consumer complaint classification

Dataset: `determined-ai/consumer_complaints_medium`

Real consumer complaints filed with the US Consumer Financial Protection Bureau.

- **113 classes** (after label cleanup — more on this shortly)
- **57,846 train / 6,430 val / 21,432 test** examples
- Real text, real labels, real class imbalance

Base model: ModernBERT-base (149M parameters, Apache 2.0 license)

You will work with this dataset and this model all semester.

Individual work, one model line

This is **individual work**. Everyone gets the same dataset and base model. Your decisions are yours.

Over the semester you will make choices:

- What learning rate? What schedule?
- Which LoRA rank? Which layers to adapt?
- How aggressively to quantize?
- Whether to distill, and from what teacher?

Different students will make different choices and get different results. That's the point.

Assessment

Component	Weight	What it measures
Weekly memos	60%	Can you analyze results and write clearly about trade-offs?
Weekly quizzes (Weeks 2-6)	15%	Do you understand last week's concepts?
Final exam	20%	Do you understand the concepts, not just the code?
Participation	5%	Are you present and engaged?

The memos are where you demonstrate engineering judgment. Not "I got X accuracy" but "I chose Y because Z, and here's what happened."

Today's plan

Block 1 — Lecture (13:30-15:10)

- Course framing (you are here)
- ModernBERT architecture
- Dataset audit: what are we working with?
- Baseline discipline: TF-IDF + logistic regression
- Anatomy of a fine-tuning loop
- The motivating benchmark: encoder vs. decoder

Block 2 — Lab (15:30-17:10)

- Hands-on: data audit + classical baseline notebook (completable in class)

Homework: Fine-tune ModernBERT on full data (separate notebook)

Why ModernBERT-base?

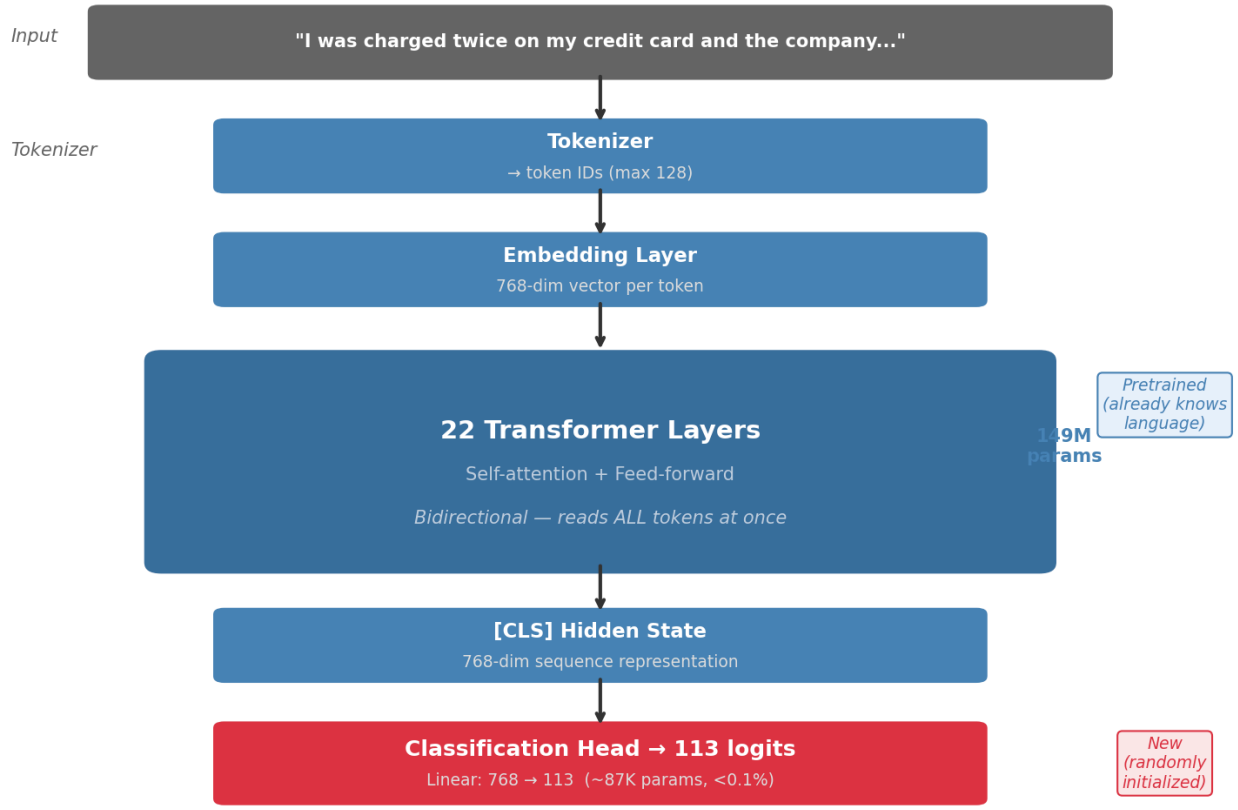
The base model for the semester

Why ModernBERT-base?

- Modern encoder (2024, updated 2025), not legacy BERT from 2018
- 149M parameters, Apache 2.0 license — you can use it anywhere
- Works with LoRA/PEFT, quantization, and distillation on free-tier GPUs
- Uses SDPA (Scaled Dot-Product Attention) — faster and more memory-efficient
- Same tokenizer/architecture patterns as classic BERT, but with modern training

What's inside ModernBERT-base

- **22 Transformer layers** — reads the whole sequence at once (bidirectional, not left-to-right)
- **768 dimensions** per token
- **[CLS] token** represents the full sequence
- **Classification head** — tiny (768 → 113, <0.1% of params)
- Pretrained encoder = 149M params (blue)
- New classification head = ~87K params (red)



Dataset Audit for Supervised NLP

Before you train anything, look at what you're training on.

Why audit your data?

Training a model on data you haven't examined is **malpractice**.

Things that go wrong when you skip the audit:

- Labels that look different but mean the same thing → inflated class count
- Classes with 2 examples → model can't learn them, but they tank your macro-F1
- Redacted or missing text → model learns patterns in the redaction, not the content
- Extreme class imbalance → model ignores minority classes entirely

Every one of these is present in our dataset. We'll deal with them.

The CFPB consumer complaints dataset

Source: US Consumer Financial Protection Bureau (CFPB)

Real complaints from real consumers about financial products and services.

Field	What it contains
complaint_text	The consumer's written complaint (free text)
issue	The category label assigned to the complaint

On Hugging Face as: `determined-ai/consumer_complaints_medium`

This is not a toy dataset. These are real people with real problems.

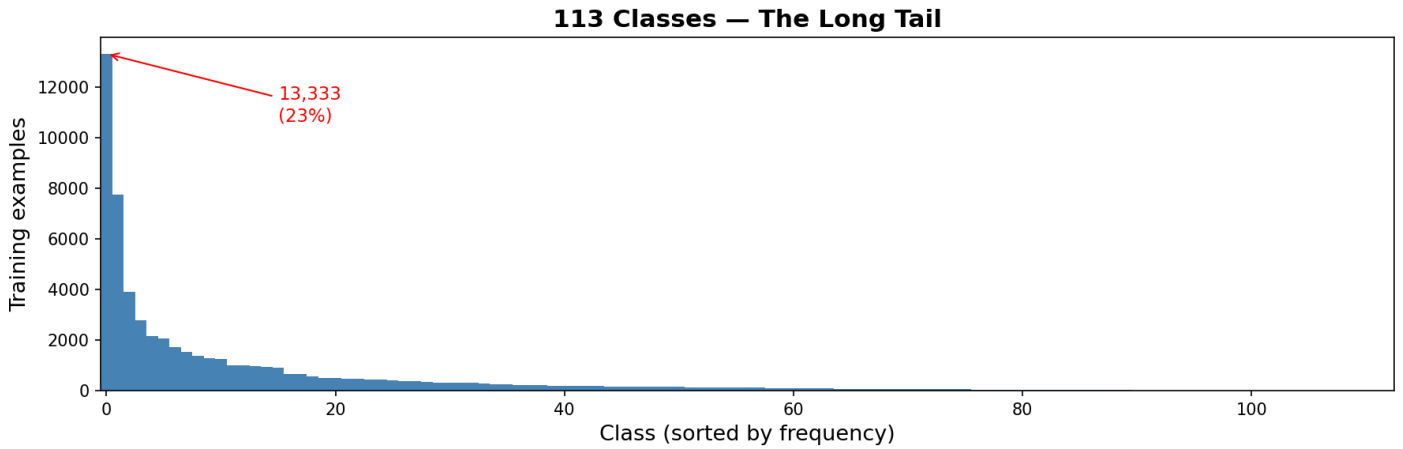
Label distribution: the long tail

113 classes after cleanup.

The top class alone covers nearly **1 in 4** complaints.

Many tail classes have fewer than 50 training examples.

Any model that learns only the common classes will look decent on accuracy and terrible on macro F1.



Why 113 classes? The label merge story

The raw CFPB data has **153 unique issue labels**.

Many are near-duplicates from a CFPB form change:

```
"Incorrect information on credit report"     ← old form
"Incorrect information on your report"        ← new form
```

Same complaint type, different wording. The form changed; the meaning didn't.

What we did:

1. Merged near-duplicate labels → reduced to ~146
2. Dropped classes with fewer than 5 total examples → **113 canonical classes**

This is a normal data-engineering step. Messy labels are the rule, not the exception.

What do complaints actually look like?

Short complaint:

> "There are many mistakes appear in my report without my understanding."

Typical complaint:

> "I have a prior XXXX account that is being reported as a charge off. I have contacted the company numerous times to resolve this matter. The account was paid in full XXXX XXXX, XXXX. Please update my credit report to reflect paid in full."

Redacted complaint:

> "On XXXX XXXX, XXXX I contacted XXXX XXXX XXXX regarding charges of {$XXX} on my account. I was told XXXX XXXX XXXX would investigate..."

Notice the **XXXX** markers. They're everywhere.

Redaction prevalence

63.5% of all complaints contain at least one XXXX redaction marker.

This means for nearly two-thirds of the data, the model is working with **incomplete text**.

What's redacted:

- Names → XXXX
- Dates → XXXX
- Account numbers → XXXX
- Dollar amounts → {$XXX}

Consequence: The model cannot learn from specific dates, names, or amounts. It must learn from the **structure and vocabulary** of the complaint.

This is actually fine for classification — but you need to know it's happening.

Text length distribution

Statistic	Value
Median length	~50 words
85–90th percentile	fits in 128 tokens
Mean length	longer (right-skewed)

Most complaints are **short**: a few sentences describing the problem.

A small fraction are very long — multi-paragraph narratives.

With **max_length = 128 tokens**, we cover 85–90% of complaints without any truncation.

The 10–15% that get truncated lose their tail end — but the complaint type is usually clear from the first few sentences.

Canonical data split

Split	Examples	Purpose
Train	57,846	Model learns from these
Validation	6,430	Tune hyperparameters, monitor overfitting
Test	21,432	Final evaluation only — never train on this

The split is **fixed** for the entire semester. Everyone uses the same split.

This is non-negotiable: you never, ever look at test set performance to make training decisions. That's data leakage.

Dataset audit: key takeaways

1. **Audit before you train.** Know your data before you touch a model.
2. **113 classes** after merging near-duplicates and dropping tiny classes from 153 raw labels.
3. **Extreme long tail** — top class is 23%, many classes < 0.1%.
4. **63.5% of complaints are redacted** — the model learns from structure, not specifics.
5. **Most text fits in 128 tokens** — truncation affects only 10–15% of examples.
6. **Fixed split** — 57,846 / 6,430 / 21,432. Same for everyone, all semester.

This is the terrain. Now let's see what a simple baseline can do on it.

Baseline Discipline

Always know what "dumb" gets you before you try "smart."

Why baselines matter

A model is only impressive **relative to something**.

Without a baseline:

- "58% accuracy" — is that good? Bad? You have no idea.
- "0.30 macro-F1" — better than what?

With a baseline:

- "58% accuracy vs. 54.2% for TF-IDF + logistic regression" — small but real improvement.
- "0.30 macro-F1 vs. 0.132 for TF-IDF" — **the neural model more than doubled F1.**

The baseline turns a number into a **story**.

The simplest baseline: majority class

Always predict the most common class.

In our dataset, "Incorrect information on your report" = 23.0% of all examples.

Metric	Value
Accuracy	**23.0%**
Macro-F1	**~0.003**

This model has zero intelligence. It learns nothing. It ignores every input.

But it gets 23% accuracy. **That's the floor.** If your model can't beat this, it's worthless.

The classical baseline: TF-IDF + logistic regression

TF-IDF: Turn text into a sparse vector of word-importance scores.
Logistic regression: Learn a linear decision boundary in that vector space.

This takes about **30 seconds to train**. No GPU. No pretrained weights. No deep learning.

It's the "can a simple model handle this?" test.

TF-IDF + logistic regression: results

Metric	Value
Val accuracy	**54.2%**
Macro-F1	**0.132**
Weighted F1	**0.479**

At first glance, 54.2% accuracy looks decent — more than double the majority baseline.

But look at that macro-F1: **0.132**.

Something is very wrong.

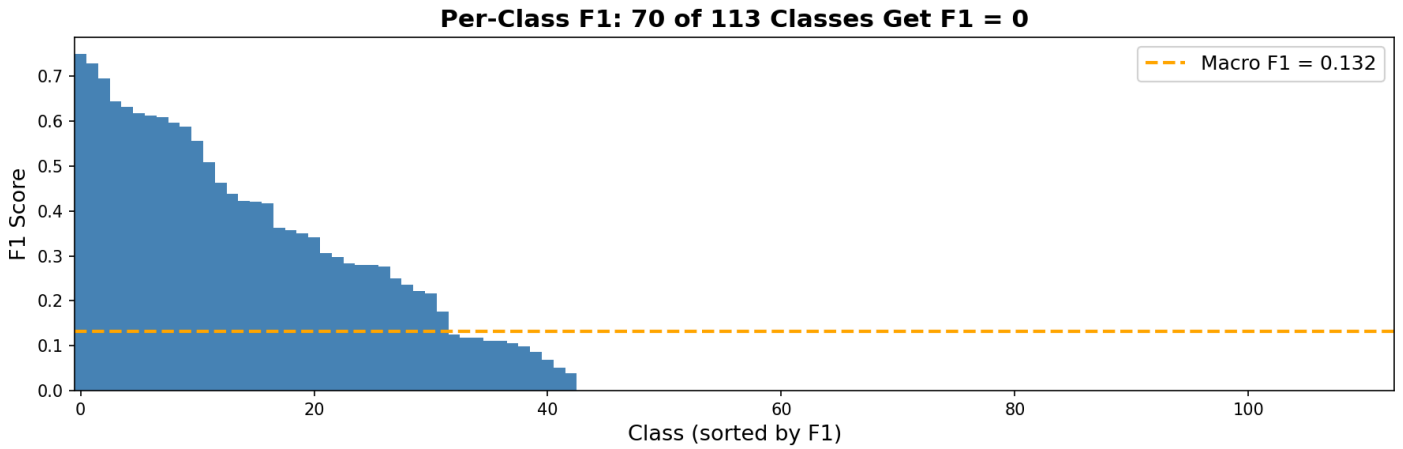
The accuracy vs. macro-F1 gap

54.2% accuracy, 0.132 macro-F1.

The model learned **~40 classes** and completely ignores the other **~70**.

70 out of 113 classes get F1 = 0. The model never predicts them. Not once.

The red bars are classes with zero F1. That's most of the chart.



Why this gap is the core lesson

The accuracy–macro-F1 gap tells you:

"Your model is biased toward common classes and ignoring rare ones."

This is the default failure mode of any model trained on imbalanced data without intervention.

It happens because:

- Common classes dominate the loss function → model optimizes for them
- Rare classes contribute little to the loss → model learns to ignore them
- Accuracy rewards this behavior → you won't notice unless you check macro-F1

Checking only accuracy is how you ship a model that fails on 60% of complaint types.

The scoreboard so far

Model	Accuracy	Macro-F1	Weighted F1	Cost
Majority class	23.0%	~0.003	—	Free
TF-IDF + LogReg	54.2%	0.132	0.479	Free, 30 sec
Fine-tuned encoder	?	?	?	Free (Kaggle T4)

Can a neural model do better?

More importantly: can it do better **where the baseline fails** — on those 70 ignored classes?

Baseline discipline: key takeaways

1. **Always run a baseline first.** Numbers without a reference point are meaningless.
2. **Majority class = 23.0% accuracy.** That's the absolute floor.
3. **TF-IDF + LogReg = 54.2% accuracy, 0.132 macro-F1.** Learns ~40 classes, ignores 70.
4. **The accuracy–macro-F1 gap** is your single most important diagnostic signal for imbalanced classification.
5. **70 out of 113 classes get F1 = 0** from the classical baseline. Those are the classes where a neural model needs to prove its value.

Anatomy of a Fine-Tuning Loop

What actually happens when you fine-tune a pretrained encoder.

The big picture

Fine-tuning = take a model that already understands language and teach it your specific task.

```
Pretrained encoder (ModernBERT-base, 149M params)
  ↓
+ Classification head (new, randomly initialized)
  ↓
Train on your labeled data
  ↓
Model that classifies consumer complaints
```

You're not training from scratch. You're **adapting** a model that already knows English grammar, word meaning, sentence structure. You just need to teach it your 113 labels.

Step 1: Load pretrained weights

```
model = AutoModelForSequenceClassification.from_pretrained(
    "answerdotai/ModernBERT-base",
    num_labels=113
)
```

This does two things:

1. Loads the **pretrained encoder** (149M params) — already knows language
2. Adds a **classification head** (random weights) — needs to learn our task

The encoder is the foundation. The classification head is the part you're really training.

Step 2: Tokenize the data

```
tokenizer = AutoTokenizer.from_pretrained("answerdotai/ModernBERT-base")

def tokenize(example):
    return tokenizer(example["text"],
                     max_length=128,
                     truncation=True,
                     padding="max_length")
```

max_length = 128 — covers ~90% of complaints without truncation.

The tokenizer and the model must use the **same vocabulary**. They were trained together.

You saw this in Pre-Work Module 01. Same concept, now applied to 57,846 training examples.

Step 3: DataLoader — batching

Training processes data in **batches**, not one example at a time.

Batch size	GPU memory	Training speed	Gradient noise
8	Low	Slow	High
16	Moderate	Moderate	Moderate
32	High	Fast	Low
64	Very high	Very fast	Very low

We use **batch_size = 32** — fits comfortably on a free Kaggle T4 GPU (16 GB) with max_length=128.

Each batch: 32 tokenized complaints → model → 32 predictions → one gradient update.

Step 4: Forward pass

For each batch:

```
Input tokens  →  Encoder  →  Hidden states  →  [CLS] embedding  →  Head  →  113 logits
```

1. Tokens enter the encoder (22 transformer layers in ModernBERT-base)
2. The encoder produces a hidden state for every token position
3. We take the **[CLS] token's hidden state** as the sequence representation
4. The classification head maps it to **113 logits** (one per class)

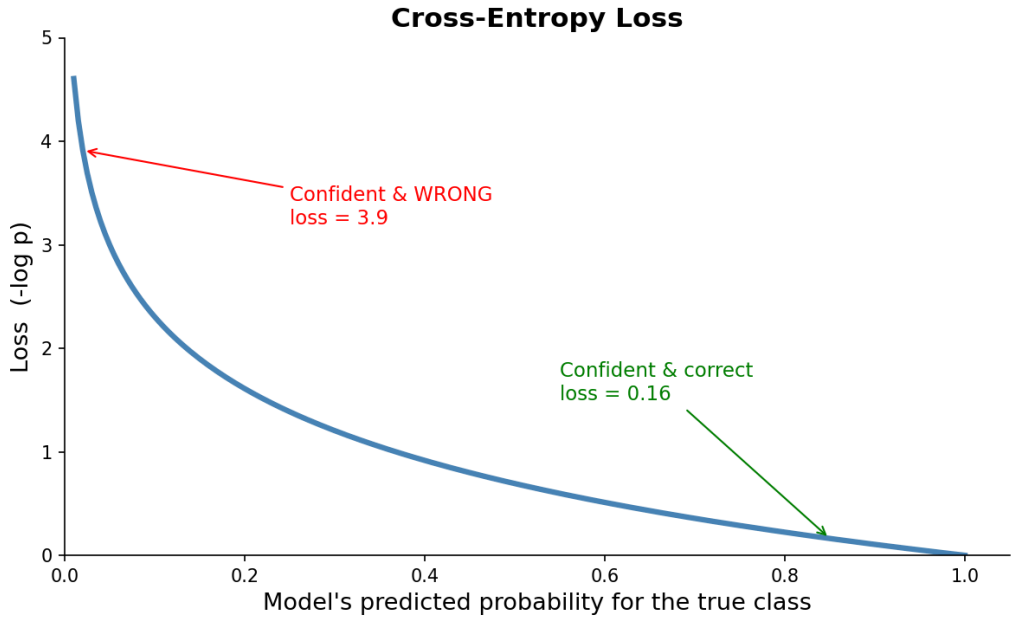
The logits are raw scores — not probabilities yet.

Step 5: Loss — cross-entropy

Cross-entropy loss = −log(p), where p is the model's predicted probability for the **true** class.

Scenario	p(true class)	Loss
Confident & correct	0.85	**0.16**
Uncertain	0.10	**2.30**
Confident & wrong	0.02	**3.91**

The curve is steep on the left — confident wrong answers get **hammered**.



Why cross-entropy explains the 70 ignored classes

Cross-entropy averages over every example in a batch.

Class	Training examples	Share of total loss
"Incorrect info on your report"	14,815	**~25% of loss**
A rare class with 8 examples	8	**~0.01% of loss**

The optimizer follows the gradient. The gradient is dominated by **common classes**.

Getting a rare class right barely moves the loss. Getting it wrong barely hurts. So the model learns to **ignore it**.

This is exactly why TF-IDF ignores 70 classes — and it will happen to our neural model too, unless we intervene (Week 2: class weighting).

Step 6: Backward pass + optimizer

Backward pass (backpropagation): Compute the gradient of the loss with respect to every parameter.

"How should each weight change to reduce the loss?"

Optimizer step: Update every parameter by a small amount in the direction that reduces the loss.

```
new_weight = old_weight − learning_rate × gradient
```

The **learning rate** controls how big each step is.

For fine-tuning: **much smaller than training from scratch** (typically 1e-5 to 5e-5).

The learning rate: why fine-tuning is different

Training from scratch: Large learning rate (1e-3). Weights are random, need big updates.

Fine-tuning: Small learning rate (2e-5). Weights are already good, need gentle nudges.

```
Training from scratch:  lr = 0.001     (100x larger)
Fine-tuning:            lr = 0.00002   (gentle updates)
```

Too high → **catastrophic forgetting**: you overwrite what the model already knows.

Too low → **underfitting**: the model doesn't adapt enough to your task.

This is the single most important hyperparameter in fine-tuning.

Step 7: Validate

After each epoch (one pass through all training data):

1. Switch model to **eval mode** — turns off dropout, freezes batch norm
2. Run the full validation set through the model — **no gradient computation**
3. Compute val loss, val accuracy, val macro-F1
4. Compare to previous epoch — is the model still improving?

Train mode vs eval mode is a real source of bugs.

If you forget to switch to eval mode → dropout is still active → your validation numbers are noisy and wrong.

Step 8: Checkpoint

A **checkpoint** saves the model state so you can resume or deploy later.

What gets saved:

- All model weights (encoder + classification head)
- Optimizer state (momentum, adaptive learning rates)
- Training metadata (epoch number, best validation score)

Why this matters:

- If your GPU crashes at epoch 3, you can resume from the last checkpoint
- At the end of training, you load the checkpoint with the **best validation score** — not the last one

The best epoch is rarely the last epoch.

The complete loop

```
for epoch in range(num_epochs):

    model.train()                          # Train mode ON
    for batch in train_dataloader:
        logits = model(batch)              # Forward pass
        loss = cross_entropy(logits)       # Compute loss
        loss.backward()                    # Backward pass
        optimizer.step()                   # Update weights
        optimizer.zero_grad()              # Reset gradients

    model.eval()                           # Eval mode ON
    validate(model, val_dataloader)        # Measure performance
    save_checkpoint(model)                 # Save state
```

That's it. This is the entire training loop. Everything else is details.

Fine-tuning anatomy: key takeaways

1. **Fine-tuning adapts pretrained knowledge** — you're not training from scratch.
2. **Encoder + classification head** — the encoder knows language, the head learns your task.
3. **The training loop:** tokenize → batch → forward → loss → backward → optimize → validate → checkpoint.
4. **Learning rate is critical** — too high destroys pretrained knowledge (catastrophic forgetting), too low won't adapt.
5. **Train vs eval mode** — forgetting to switch is a real bug with real consequences.
6. **Best checkpoint ≠ last checkpoint** — save checkpoints, use the best one.

What you're building

For the first two weeks, you build the best encoder you can. One model, one dataset, one evolving artifact. You learn the training loop, improve it, analyze where it fails.

In Week 3, you meet a challenger: a decoder that trains on the same data, on the same free GPU, and beats your encoder on rare-class performance.

The rest of the semester: **understand the trade-off and make the recommendation.**

The full picture

Model	Accuracy	Macro-F1	Rare classes rescued	Latency
Majority class	23.0%	~0.003	0 / 113	—
TF-IDF + LogReg	54.2%	0.132	43 / 113	instant
Fine-tuned encoder (149M, 3 ep)	**56.6%**	**0.209**	**67 / 113**	**3 ms**
Decoder + LoRA (494M, 3 ep)	**57.0%**	**0.240**	**76 / 113**	**58 ms**
Opus decoder (zero-shot)	44.0%	0.174	—	2,300 ms

Both the encoder and the decoder train on the same data, on the same free Kaggle T4.

The decoder wins on quality. The encoder wins on speed. **Neither dominates.**

The trade-off

	Encoder (149M)	Decoder (494M)
Accuracy	56.6%	**57.0%**
Macro F1 (rare classes)	0.209	**0.240 (+15%)**
Zero-F1 classes	46	**37 (9 more rescued)**
Latency per example	**3 ms**	58 ms
64K complaints	**~3 min**	~7 min
Training time (T4)	32 min	54 min
Parameters trained	149M (100%)	2.3M (0.46%)

The decoder trains **less than 1% of its parameters** and beats the encoder that trains all of them.

Why doesn't the encoder just win?

The decoder has **3.3x more parameters** trained on **9x more data.**

Scaling laws predict: more parameters + more data = richer representations.

The encoder learns the task from scratch using 58,000 labeled examples. The decoder already understood the domain — it just needed LoRA to learn the 113 label boundaries.

This gap **grows** with decoder size:

Decoder	Accuracy	Macro F1
0.5B	57.0%	0.240
1.5B	58.3%	0.252
3B	58.7%	0.250

The bigger the decoder, the more the encoder falls behind.

The semester question

This course is not about picking the winner.

Both models are useful. The question is: **which one do you deploy, for what use case, and why?**

- A real-time complaint router that must respond in under 10ms? → Encoder.
- A weekly batch analysis where quality on rare classes matters? → Decoder.
- A budget-constrained startup with no GPU? → TF-IDF might be fine.

Your job this semester: **build both, understand both, and make the recommendation.**

Week by week, you'll fine-tune, adapt with LoRA, analyze errors, compress, and distill — on both architectures. At the end, you write the engineering recommendation.

Key takeaways

1. The encoder (56.6%) and decoder (57.0%) are **close on accuracy** — the real gap is on rare classes (macro F1: 0.209 vs 0.240).
2. The encoder is **19x faster** per example (3 ms vs 58 ms) — real but not 1000x.
3. The decoder trains **0.46% of its parameters** and beats the encoder training 100% — scaling laws at work.
4. Bigger decoders are better: 0.5B → 1.5B → 3B, accuracy keeps climbing.
5. **Neither model dominates.** Your semester: build both, understand both, recommend one.

Your homework this week

Block 2 (right now): Open `week1_lab.ipynb` and work through it. Data audit + classical baseline. You should finish in class.

After class — homework notebook: `week1_homework.ipynb`

- Fine-tune ModernBERT-base on the full dataset (2 epochs, ~20 min training on T4)
- Analyze: which of the 70 zero-F1 classes did the neural model rescue?
- Experiment: change one hyperparameter and compare
- Write your memo (prompts are embedded in the notebook)

Due: Wednesday morning before Week 2 class. Submit as **HTML** via Moodle.

Expected time: 5-6 hours outside of class.