Attention and transformers

IMDb

LECTURE 18 | GÉRON CH. 14–15

Applications of Machine Learning

BSc Mathematics of Artificial Intelligence · Third year

Where we are

Lecture 17 classified IMDb reviews: words and subword tokens, an embedding table, and a bidirectional GRU trained on logits with cross-entropy — which lost to a bag-of-words anchor. However well it is trained, a recurrent model carries three limits:

- The computation is **sequential** along the sequence
- Everything remembered must fit through **one fixed-width state**
- Position 400 is four hundred steps from position 2, however good the gates

> **ONE CHANGE REMOVES ALL THREE**
>
> Let every position compute a weighted sum over *every* position directly, with the weights learned from the content. No recurrence, no single state, and every pair of positions one step apart.

Today

1. **The mathematics** — scaled dot-product attention, why the scores are divided by $\sqrt{d_k}$, and why position has to be added
2. **The architecture** — one block, the causal mask, attention in code, and its quadratic cost
3. **The task** — fine-tune a pretrained transformer on the reviews, against a bag of words and Lecture 17's GRU
4. **Beyond classification** — search and grouping from sentence embeddings
5. **Two leaks** — a vectoriser fitted before the split, and the same document on both sides of it
6. **Design exercise** — a social platform's comment-moderation system, in nine steps, with the pitfalls and how to address them

Open Lecture 18 in Colab

The question

Every position needs a summary of the whole sequence, weighted by *relevance* — and relevance has to be learned, not designed.

> **THE SHAPE OF THE ANSWER**
>
> Output at position i is $\sum_j w_{ij} \mathbf{v}_j$: a weighted sum of values, one per position. Everything below is about where w_{ij} comes from and why it takes the form it does.

Note what is already gone: there is no recurrence in that expression, so nothing forces the positions to be computed in order.

Queries, keys and values

1. Project each position three ways, with learned matrices: $\mathbf{q}_i = \mathbf{W}_Q \mathbf{x}_i, \mathbf{k}_j = \mathbf{W}_K \mathbf{x}_j, \mathbf{v}_j = \mathbf{W}_V \mathbf{x}_j$.

   A query is what this position is looking for; a key is what that position offers; a value is what it contributes if chosen

2. Score every pair by how well the query matches the key: $s_{ij} = \mathbf{q}_i^\top \mathbf{k}_j$.

   A dot product, because it is the cheapest similarity that a matrix multiply computes for all pairs at once

3. Turn the scores into weights that sum to one: $w_{ij} = \mathrm{softmax}_j(s_{ij})$.

   Lecture 17's softmax, applied along the sequence rather than across classes

4. And take the weighted sum: $\mathbf{z}_i = \sum_j w_{ij} \mathbf{v}_j$.

   In matrix form for all positions at once: $\mathrm{softmax}(\mathbf{Q}\mathbf{K}^\top)\mathbf{V}$

Why divide by $\sqrt{d_k}$

Take $\mathbf{q}$ and $\mathbf{k}$ to have independent components with mean 0 and variance 1. Then

5 $s = \mathbf{q}^\mathsf{T}\mathbf{k} = \sum_{d=1}^{d_k} q_d k_d$ is a sum of d_k independent zero-mean terms.

Each term has variance $\mathrm{Var}(q_d)\mathrm{Var}(k_d) = 1$

6 So $\mathrm{Var}(s) = d_k$, and s has standard deviation $\sqrt{d_k}$.

The scores grow with the dimension — for $d_k = 64$, typical scores are already of order 8

7 Softmax of scores that large saturates — one weight near 1, its gradient near 0 — so divide by $\sqrt{d_k}$, and the variance is 1 whatever the dimension: $\mathrm{softmax}\left(\mathbf{Q}\mathbf{K}^\mathsf{T}/\sqrt{d_k}\right)\mathbf{V}$.

The scale is not a hyperparameter — it is the number that makes the score distribution independent of d_k

Derive the variance of the score and state what the division fixes.

EXAMINABLE

Positional encoding

THE PROBLEM ATTENTION CREATES

$\mathrm{softmax}(\mathbf{QK}^\top)\mathbf{V}$ is **permutation equivariant**: shuffle the input positions and the outputs shuffle with them, unchanged. So the model literally cannot tell *"the film was not good"* from *"good, the film was not"*.

Recurrence got order for free, because it processed the sequence in order. Attention has to be *told*: add a position-dependent vector to each embedding before the first layer.

- **Sinusoidal** — fixed, and extends to lengths never seen in training
- **Learned** — an embedding table indexed by position; simpler, and bounded by the longest training sequence
- **Relative** — encode the *distance* between two positions in the score, which is what most current models do

One block

A transformer block is attention plus a small feed-forward network, each wrapped the same way:

1. Self-attention over the whole sequence
2. Add the input back (a residual connection) and normalise
3. A position-wise feed-forward network — the same two-layer MLP applied at every position independently
4. Add and normalise again

> **WHY THE RESIDUAL, AND WHY THE NORM**
>
> The residual gives the gradient an additive path through depth — the same argument as an LSTM gate, one level up. Layer normalisation — Lecture 12's batch norm, taken over each position's features instead of the batch — keeps the variance from drifting as blocks stack. A transformer is deep, and it is deep only because of these two.

Masking, and why the decoder needs it

A decoder generating text must not look at what it has not written yet. So set $s_{ij} = -\infty$ for $j > i$ before the softmax.

$$w_{ij} = \mathrm{softmax}_j\left(\frac{s_{ij}}{\sqrt{d_k}} + m_{ij}\right), \quad m_{ij} = \begin{cases} 0 & j \le i \\ -\infty & j > i \end{cases}$$

$e^{-\infty} = 0$, so those positions get exactly zero weight and contribute nothing — and, crucially, no gradient either.

✗ This is Lecture 15's causality condition again, enforced in the architecture rather than in the data split. An unmasked decoder is a model that has read the answer.

Attention in six lines

```
def attention(Q, K, V, mask=None):
    d_k = Q.shape[-1]
    scores = Q @ K.transpose(-2, -1) / math.sqrt(d_k)   # the derivation
    if mask is not None:
        scores = scores.masked_fill(mask, float("-inf"))
    w = scores.softmax(dim=-1)
    return w @ V, w
```

Six lines, and every one of them is a step of the derivation. The rest of a transformer — the residuals, the feed-forward, the normalisation — is arrangement around this function.

Which is worth saying plainly: the architecture that changed the field is, at its centre, a similarity, a softmax and a weighted average.

The quadratic cost, and what is done about it

$\mathbf{QK}^\mathsf{T}$ is $T \times T$. At $T = 512$ that is 262,144 scores; at $T = 8{,}192$ it is 67 million.

- **Sparse patterns** — attend to a local window plus a few global positions, making the cost linear
- **Low-rank or kernel approximations** — approximate the softmax so the product never has to be formed
- **Better implementations** — the exact same computation, tiled so it never writes the $T \times T$ matrix to memory. This is what most systems actually use

✗ Note which of those changes the mathematics and which does not. The third is an engineering result, and it moved the practical limit further than either of the first two.

> **FAILURE CONDITION — A CONTEXT LENGTH THAT RUNS OUT OF MEMORY**
>
> Attention is quadratic in the sequence length, so doubling the context quadruples the score matrix. The failure arrives as an out-of-memory error at a length that worked on the previous run, and no accuracy number anywhere hints at how close you were.

Reading an attention map, carefully

Attention weights are easy to plot and easy to over-read.

> **WHAT THEY ARE NOT**
>
> They are **not an explanation**. A weight says this value contributed to that output, not that the model's decision depended on it: the residual stream carries information around the attention, and different weight patterns can give identical outputs.

Useful for forming a hypothesis; not evidence on its own. The test is the same as everywhere else in this course — remove the thing and measure whether the number moves.

The model we borrow

Property	Value
Parameters	66,955,010
Vocabulary — the same one we used last lecture	30,522
Embedding width	768
Pretraining task	predict masked words

Last lecture we took 30,522 × 768 numbers out of this model and threw the rest away. The rest is where the language is.

Fine-tuning, precisely

```
from transformers import AutoModelForSequenceClassification

model = AutoModelForSequenceClassification.from_pretrained(
    "distilbert-base-uncased", num_labels=2)
# every weight is pretrained EXCEPT the classification head, which is random

opt   = torch.optim.AdamW(model.parameters(), lr=2e-5)
lossf = nn.CrossEntropyLoss()          # logits in, as always
```

- A new linear head, randomly initialised, on top of a pretrained body
- A learning rate fifty times smaller than we used from scratch — large steps destroy what was pretrained
- One pass over the data, not two

How much is the head, and how much is the body?

Run the same backbone with its untrained head and no fine-tuning at all:

Model	Test accuracy
✗ Pretrained body, random head, no training	**✗ 49.9%**
Always one class	50.0%

Measured on 3,000 test reviews sampled at random, 50.1% of them positive. Note it is *no better* than the majority class: an untrained head is an arbitrary hyperplane, and which side it puts each class on is a coin toss. The pretraining on its own predicts nothing about sentiment, because nothing has told it what the two output units mean.

Everything that follows is what one pass of fine-tuning does with a representation it did not have to build.

What it costs

Setting	Value
Reviews used to fine-tune	2,000
Epochs	1
Batch size, learning rate	16, 0.00002
Test reviews scored, sampled at random	3,000

Run in `float32`, and sized so that the notebook runs on a free CPU. One pass, fixed in advance: with more budget the number of epochs is a hyperparameter, chosen on the validation split — never on the test set.

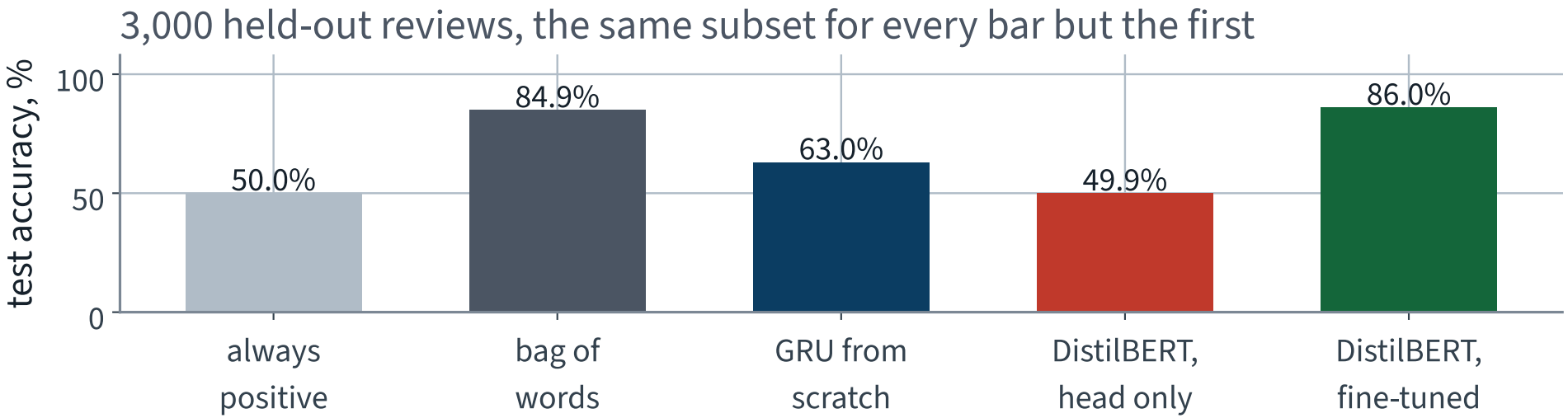
The result

Model	Test accuracy
Always one class	50.0%
tf-idf + logistic regression, the same 2,000 reviews	84.9%
GRU from scratch	63.0%
✓ DistilBERT, fine-tuned	**86.0% ✓**

✓ 22.97 points over the model you built.

And only 1.07 points over a bag of words fitted on the same 2,000 labels — which is the second result of the afternoon, and the less comfortable one. Every row but the first is scored on the same 3,000 test reviews.

Everything, measured the same way



Four of the five bars are on the same 3,000 held-out reviews, with the same metric, computed by the same function.

Say it in errors, not in points

Model	Reviews misclassified, out of 3,000
GRU from scratch	✗ 1,109
✓ DistilBERT, fine-tuned	420 ✓

62%

of the mistakes removed. Percentage points flatter a model near the top of the range; the error count is the number the desk actually cares about, because each one is a review someone has to fix by hand.

Where the pretrained model earns its place

Measured here at one label budget: 2,000 reviews for each model, scored on the same 3,000 test reviews.

Model, fitted on 2,000 labels	Test accuracy
tf-idf + logistic regression	84.9%
✓ DistilBERT, fine-tuned	**86.0% ✓**

Pretraining is knowledge the labels do not have to supply. So the expectation is that its lead *grows* as labels get scarcer and shrinks as they arrive — and the situation every real feedback desk starts in is the scarce one.

This notebook measures one point on that curve, not the curve.

What it would take to measure the curve

> **THE EXPERIMENT, NOT RUN HERE**
>
> Repeat the comparison at several label budgets, from a couple of hundred reviews to several thousand, on one fixed test sample drawn at random — the corpus ships all the positives first.

- Choose each model's budget on the validation split **at every size**: the number of optimisation steps for the transformer, the regularisation strength for the linear model
- Fixing the epochs instead starves the smallest run of steps; fixing the steps undertrains the largest. The two protocols can disagree about the *sign* of the gap

✗ That is several fine-tunes, more than a free CPU runs inside a lecture. Until it is run, "the pretraining is worth thousands of labels" is a hypothesis — not yet the number to quote to whoever is paying for the annotation.

What did the work

Changed	Unchanged
the whole model, not just the table	the corpus, the split, the metric
attention instead of recurrence	the loss — still logits in
a much smaller learning rate	the training loop, line for line
one epoch instead of two	the evaluation function

Every row on the right is a decision you do not have to re-make when the model changes. That is what the discipline of Part I bought.

One vector for a whole review

The desk also wants “find me the ones like this one” and “what do people keep complaining about?”. Neither is classification; both start from one unit-length vector per review, so that a dot product is a cosine.

```
1  m   = AutoModel.from_pretrained("sentence-transformers/all-MiniLM-L6-v2")
2  tk  = AutoTokenizer.from_pretrained("sentence-transformers/all-MiniLM-L6-v2")
3
4  enc  = tk(texts, truncation=True, max_length=256, padding=True,
5            return_tensors="pt")
6  out  = m(**enc).last_hidden_state                  # (B, T, 384)
7  mask = enc["attention_mask"].unsqueeze(-1).float()
8  vec  = (out * mask).sum(1) / mask.sum(1)           # mean over REAL tokens
9  vec  = torch.nn.functional.normalize(vec, dim=1)   # onto the unit sphere
```

✗ Line 8 is last lecture’s padding bug in its other costume. Divide by `out.shape[1]` instead of by `mask.sum(1)` and every short review is scaled toward zero.

Semantic search, in two lines

```
sims = query_vec @ corpus_vecs.T          # (1, N) cosine similarities
top  = np.argsort(-sims[0])[:3]
```

2,000 negative reviews, embedded once, 384 dimensions each. No index, no training, no labels.

> **COMPARE WITH WHAT IT REPLACES**
>
> A keyword search over the same corpus, using the tf-idf vectoriser from the previous lecture. Same corpus, same queries, different notion of “similar”.

One query, two searches

Query: *“the acting was wooden and unconvincing”*

> **NEAREST BY EMBEDDING — COSINE 0.555**
>
> Ed Wood rides again. The fact that this movie was made should give any young aspiring film maker hope…

Nearest by keyword overlap — tf-idf cosine 0.194:

“This film was sheer boredom from beginning to end. Ok, so i salute Boorman for raising the worldwide recognition of events in Burma, but that is all he achieves…”

Two different reviews, and neither opens with the query’s words. The scores are not comparable across the two methods — different spaces — and nothing here says which list is *better*: that takes labelled queries and a ranking metric, which is where the next lecture starts.

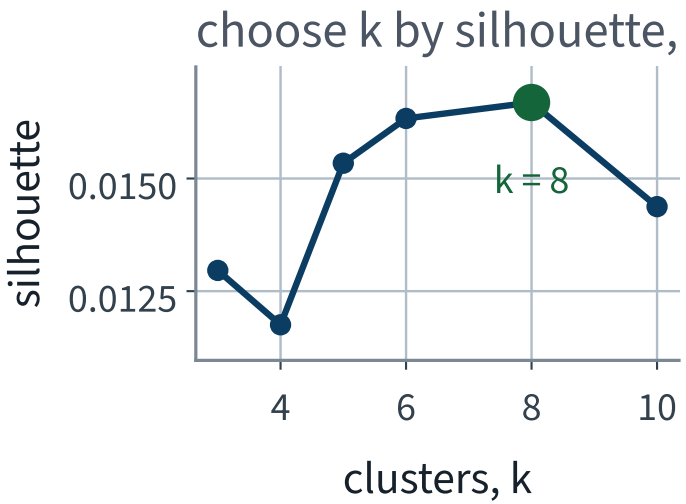
“What do people keep complaining about?”

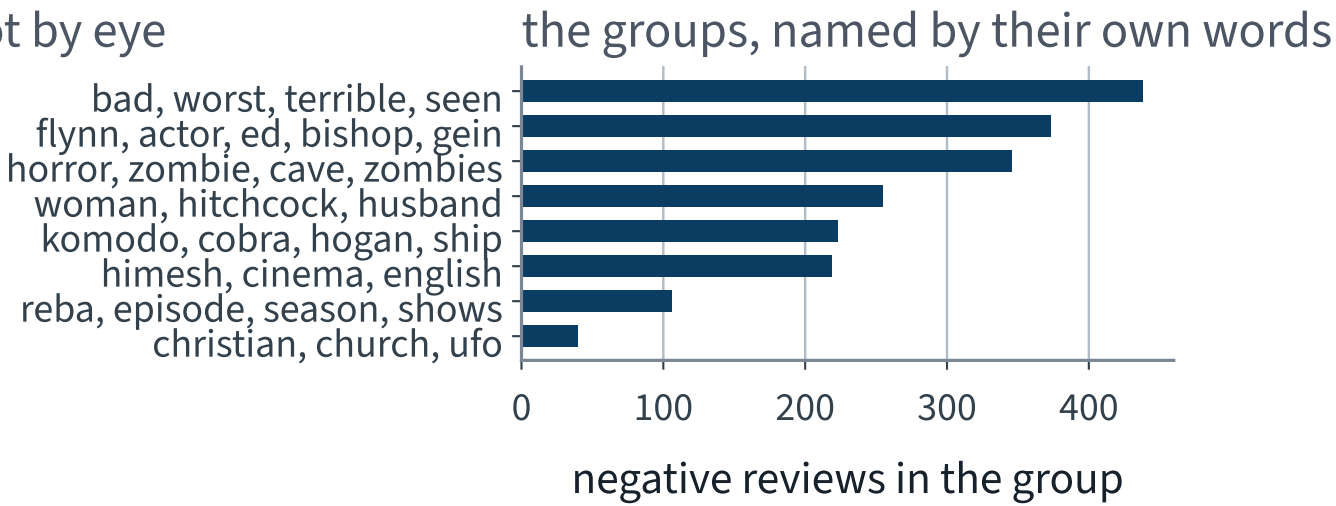
Cluster the 2,000 negative reviews in the same embedding space. This is Lecture 6, unchanged — k-means, and the choice of k by silhouette rather than by eye.

```
for k in (3, 4, 5, 6, 8, 10):
    km = KMeans(n_clusters=k, n_init=10, random_state=42).fit(V)
    print(k, silhouette_score(V, km.labels_, sample_size=1500,
                              random_state=42))
```

The vectors are on the unit sphere, so Euclidean k-means and cosine k-means order the same points the same way.

The groups, and how many





The silhouette is low in absolute terms — complaint text does not fall into well-separated balls — and the largest group is still legible.

The three largest groups

Reviews	Terms most characteristic of the group
438	bad, worst, terrible, seen
373	flynn, actor, ed, bishop
346	horror, zombie, cave, zombies

The terms are not the cluster centres. They are the words whose average tf-idf weight is highest *inside* the group relative to outside it — a group is named by what makes it different, not by what is common everywhere.

The second group is named by proper nouns — a handful of films reviewed many times. That is a *topic*, not a complaint: grouping finds what recurs, and deciding which recurrences the desk cares about is still a person's job.

8 groups in total, chosen by a silhouette of 0.0167.

What the desk actually gets

They asked for	They get
a complaint flag	86.0% accurate: 420 errors in 3,000
“find me ones like this”	cosine search over 2,000 vectors
“what recurs”	8 groups, each named by its own words

Three deliverables, one pretrained representation, and one afternoon. None of the three required a labelled example of the thing being asked for except the first.

Where is the vectoriser fitted?

> **THE TASK**
>
> *“Build a scikit-learn pipeline that vectorises the reviews with tf-idf and classifies them with logistic regression, and report the test accuracy.”*

Stated like that, it is open in the same place as a scaler fitted before the split (Lecture 3), and this time the object being fitted is not a scaler. Read the code and find where the vectoriser is fitted, and on what.

One way to write it

```
1  vec = TfidfVectorizer(min_df=1, ngram_range=(1, 2))
2  X   = vec.fit_transform(all_texts)            # every document
3  y   = all_labels
4
5  X_tr, X_te, y_tr, y_te = train_test_split(
6      X, y, test_size=0.25, random_state=42, stratify=y)
7
8  clf = LogisticRegression(max_iter=2000, C=4.0).fit(X_tr, y_tr)
9  print(f"accuracy {(clf.predict(X_te) == y_te).mean():.3f}")
```

It runs, it splits, it never fits the classifier on test rows, and it reports a plausible number. But line 2 fits the vectoriser on every document, the test rows included.

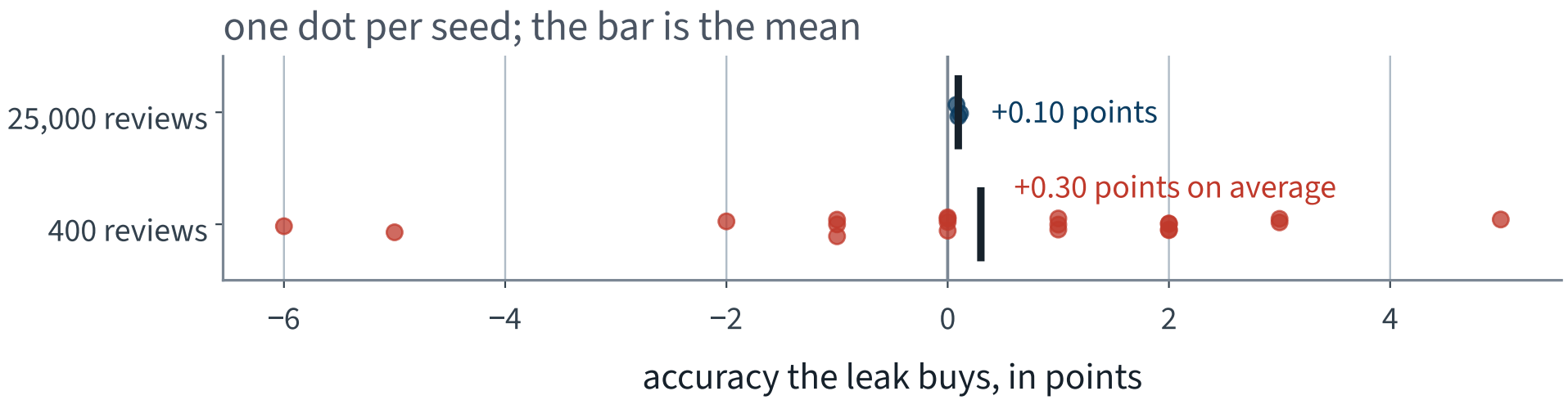
So measure it — twice

The same experiment at two corpus sizes, because the answer depends on the size and that is the lesson:

- **25,000 documents** — the whole of IMDb, 3 seeds
- **400 documents** — the size of a real feedback corpus in its first month, 20 seeds

Leaky and honest are run on identical splits with identical seeds, so the difference is the leak and nothing else.

What the leak buys, per seed



Read each row's spread before its mean. On 25,000 reviews the leak is small and steady; on 400 the seeds fall on both sides of zero and the mean is lost inside that scatter — a null result.

Three leaks, and none of them loud

A scaler fitted before the split (Lecture 3) raises no error and makes the score optimistic by an amount the code does not tell you. Lecture 8's per-batch mean was worth 0.067 accuracy points. Today’s vectoriser leak is worth nothing you can measure.

AND THAT IS NOT A REASON TO RELAX

A leaked score and an honest score can be identical, so you cannot tell from the number which one you are holding. The procedure is what separates them — which is why the procedure is what gets checked in review, and why the next slide matters more than this one.

The text leak that really is expensive

The same document on both sides of the split.

No pipeline protects you from this: every object is fitted correctly, on training rows only, and the test row is still one the model has memorised. On scraped or user-submitted text it is the normal case, not the pathological one — a resubmit button is enough.

```
norm = lambda s: " ".join(s.lower().split())
train_norm = set(norm(s) for s in train_x)
dupes = sum(1 for s in test_x if norm(s) in train_norm)
```

The same pipeline, two splits



✗ A corpus where three reviews in ten were submitted twice, and the same honest pipeline split two ways: at random over rows, and grouped by entry so every copy stays on one side. Nothing was fitted on the test set in either bar; the difference is entirely which rows the split separated.

The silent-failure catalogue so far

Failure	Diagnosed in	Measured cost
Scaler fitted before the split	Lecture 3	an optimistic score, and no error
Accuracy under class imbalance	Lecture 3	a useless model above 90%
Missing `zero_grad()`	Lecture 8	66 points
Random split on a time series	Lecture 15	skill overstated by 9.0%
✗ Summarising at the padded last position	**✗ Lecture 17**	**✗ the state after the padding, and no error**
Softmax before the loss	Lecture 17	the wrong gradient, and no error
Vectoriser fitted before the split	today	0.30 points — not distinguishable from zero
✗ Duplicate documents across the split	**✗ today**	**✗ 7.83 points**

Is 86.0% good?

Against the desk's own criterion: 420 reviews out of the 3,000 scored are routed wrongly, and nobody sees them again. Whether that is acceptable depends on what happens to a missed complaint, and we have not asked.

> **WHAT A DEFENSIBLE REPORT SAYS**
>
> Quote the error count, not the percentage. Say which reviews are misread — short ones, sarcastic ones, ones whose verdict is in the last clause. Offer an abstain band around a probability of one half, and let the desk choose how wide it is.

The model is not the deliverable. The routing decision is.

The notebook for this lecture

Open Lecture 18 in Colab

- **Runs on free CPU.** A small pretrained transformer, a subsampled corpus, and a short fine-tune
- Lecture 17's GRU rebuilt, DistilBERT before and after one pass of fine-tuning, and a bag of words on the same 2,000 labels, all scored on the same 3,000 test reviews
- Sentence embeddings for search and grouping; the vectoriser leak and the duplicate leak measured

> **WHAT TO CHANGE**
>
> Fine-tune at `lr=1e-3`, the from-scratch model's rate. The loss rises within twenty steps and never recovers: a pretrained body is destroyed in tens of steps, not thousands.

Design exercise · in the manner of an ML system-design interview

The task: a social platform’s comment-moderation system

> THE BRIEF
>
> A social platform’s comments are where its users talk and its advertisers sit. Its moderation system decides, as a comment is posted, whether it is **toxic** — harassment, threats, slurs — and outputs a probability, so the comment is posted, held for review, or removed. Every error is read by someone: a missed threat by its target, a held comment by its author. The business measures retention, the review desk’s cost, and fines for a missed removal or denied appeal.

- **Given:** comments labelled by the moderation team — tens of thousands of the millions posted — and its written policy
- **Scale:** millions of comments a day; toxic ones a few in a hundred; a decision in tens of milliseconds
- **Consumers of the output:** the posting pipeline, the review queue, appeals, and the regulator’s report

✓ Nine steps follow, each with the pitfall that catches a first attempt and what to do about it. The model at the

Step 1 — Frame it: a probability, a threshold that is policy, two kinds of error

- The model outputs $p(\text{toxic} \mid \text{text})$; the *decision* — post, hold, remove — is a threshold on it, and the threshold belongs to the trust-and-safety team, not the model
- Count the classes: a few in a hundred are toxic, so the anchor "always post" is right almost always
- Report both errors as counts — toxic comments posted, clean comments held — beside the review desk's capacity, since every held comment is a reviewer's time and a user waiting

> **PITFALL**
>
> Accuracy, and a threshold of one half because that is what `argmax` gives. Under this imbalance accuracy cannot see the toxic class at all, and one half is a policy chosen by nobody. Report both error counts across a threshold sweep, and let the team choose the operating point from what it can review and what a missed threat costs.

Step 2 — The label: what the moderators decided, and what they never saw

- A label is a moderator’s decision under the written policy; moderators disagree, so have more than one read a slice and record the disagreement — it is the ceiling on any model
- Quoting a slur to report it, sarcasm, and reclaimed words inside a community are where the policy is hardest to apply, and where the labels are noisiest

> **PITFALL**
>
> Labelling only what was reported or the old filter caught. The positives are then the toxic comments the old system could find, and every comment that was never reviewed is a negative that may not be one: the model learns the old filter’s blind spots as truth, and the users it failed stay failed. Label a random sample of all posted comments as well, and use that sample for every number reported.

Step 3 — Split by time, and keep every copy of a comment on one side

- Train on earlier comments, validate on later ones, test on the latest: slang, targets and coded words move by the month
- Before splitting, normalise case and whitespace and group identical comments — the same text pasted into fifty threads is the normal case for abuse
- Choose the threshold, the passes and the model on the validation comments; read the test comments once

> **PITFALL**
>
> The same comment on both sides of the split. No pipeline protects you: every object is fitted on training rows, and the test row is still one the model memorised. On user-submitted text it is the leak that costs points, and on pasted abuse it is everywhere — the launch score is one no user sees. Group by normalised text, split by time, and count the duplicates across the cut before trusting a score.

Step 4 — The anchor: a bag of words, fitted where it should be

- tf-idf over words and adjacent pairs, logistic regression on top, trained on the same labels the transformer will get: the number the transformer has to beat, and the model that ships if it does not
- Its regularisation strength chosen on the validation comments, not left at the default — the anchor gets the same tuning budget as the challenger
- Its scoring time per comment written down: posting has a latency budget the metric cannot see

> **PITFALL**
>
> `fit_transform` on every comment and *then* the split. It runs, it splits, and it reports a plausible number — the vectoriser has seen the test vocabulary. The measured cost is small, and that is not the point: a leaked score and an honest one can be identical, so the procedure is what is reviewed before release. The vectoriser is fitted on training comments, inside a pipeline.

Step 5 — The model: a pretrained body, a new head, few labels

- A pretrained transformer with a randomly initialised two-logit head; every weight trained, at a learning rate tens of times smaller than from scratch; cross-entropy on the logits
- Few labels is where pretraining earns its place: the language is already in the body, and the labels only have to say what the two output units mean
- Comments are short, so the quadratic cost is paid on a short context: truncate at a length that keeps nearly every comment whole, and measure the per-comment time against the posting pipeline's budget

> **PITFALL**
>
> The from-scratch learning rate. The loss rises within tens of steps and never recovers: a pretrained body is destroyed in tens of steps, not thousands, and what remains scores like a random head — the majority class, which posts everything. Small steps, few passes, and the number of passes chosen on the validation comments.

Step 6 — What fails without an error: the padding, in its other costume

- Batches are padded to the longest comment; the attention mask keeps the padding out of the scores, and the pooled vector must be an average over the *real* tokens only
- The same vector, normalised, is what finds near-duplicates of a removed comment and groups what users keep reporting

> **PITFALL**
>
> Dividing by the padded length instead of the mask's sum. Every short comment is scaled towards zero, and short comments are where the insults are: they cluster near the origin, look alike to the head, and the near-duplicate search returns them for everything. Divide by the number of real tokens, and assert that padding the same comment two ways gives the same vector — the shapes will not tell you.

Step 7 — Evaluate: against the anchor, at the same label budget, honestly

- Both models on the same labels and the same test comments, scored by one function; say it in errors, not points — each error is a comment someone reads
- Each model's budget chosen on validation at every label size: optimisation steps for the transformer, regularisation strength for the linear model. Fixing either in advance can flip the sign of the gap
- Repeat at several label sizes, since the question the labelling budget pays to answer is how many more labels the pretraining is worth

> **PITFALL**
>
> A comparison in which the transformer's passes were tuned on validation and the anchor's regularisation was left at its default. The lead is then partly the tuning, and the lead over a bag of words on the same labels is small enough that the tuning can be all of it. Tune both on the same split, or the result is a claim about effort, not about models.

Step 8 — In production: rollout, the review loop, and who the errors fall on

- Three bands on the probability — post, hold for review, remove — the hold band sized to the desk's capacity; appeals and overturns flow back as labels
- A share of traffic first, the old filter on the rest: hold rate, overturn rate and latency compared; retrain on a recent window, as coded words appear monthly; model team and desk share on-call
- The model changes what is posted, so what is reported: review a random slice of posts regardless of score

> **PITFALL**
>
> False positives concentrated on a dialect or a community, invisible in the mean: the labels marked words toxic wherever they appeared, and the model silences those who use them among themselves — users who leave, and a regulator's case. Count errors per segment — dialect, language, topic — before launch; route those segments to the human band, not removal; read the overturns per segment as the live measure.

Step 9 — Ship it: the pipeline, and the pitfalls in one table

Moderator labels plus a random sample of posts → duplicates grouped, split by time → bag-of-words anchor, fitted and tuned on training rows → a pretrained body fine-tuned at a small rate, pooled on real tokens → error counts at the same label budget → thresholds set by the team → a share of traffic; post, hold, remove; appeals as labels → retrain.

Pitfall	Address it by
Accuracy, and a threshold of one half	error counts both ways; operating point chosen by the team
Labels only where the old filter looked	a random sample of everything posted, several readers
The same comment on both sides	grouping by normalised text; splitting by time
The vectoriser fitted on all comments	fitted on training comments, inside the pipeline
The from-scratch learning rate	a small rate, few passes, chosen on validation
Pooling over the padding	dividing by the real token count; an invariance test
The anchor left untuned	each model’s budget chosen on validation, at every label size
False positives on a dialect	errors per segment; the hold band; overturns read per segment

What we did

1. Derived attention as three projections, a scaled dot product, a softmax and a weighted sum — with no recurrence in it
2. Derived the $\sqrt{d_k}$: the score variance grows with the dimension, a saturated softmax has no gradient, and the division is exactly the correction
3. Saw that attention is permutation equivariant, so position has to be supplied — and three ways to supply it
4. Read the block — attention, residual, normalise, feed-forward, residual, normalise — the mask that makes a decoder causal, and the cost quadratic in sequence length
5. Fine-tuned a pretrained transformer: 86.0%, against 63.0% for Lecture 17's GRU and 84.9% for a bag of words on the same labels
6. Searched and grouped reviews with sentence embeddings, and measured two leaks: the vectoriser, a null result, and duplicates, 7.83 points
7. Designed a social platform's comment-moderation system in nine steps — a threshold that is policy, labels beyond what the old filter caught, duplicates kept on one side of a split by time, a vectoriser fitted where it should be, a body fine-tuned at a small rate, pooling over real tokens, both models tuned on the same split, a share of traffic first, and the errors counted per dialect before launch

FAILURE CONDITION — POSITION NEVER REACHES THE MODEL

Attention is permutation equivariant: shuffle the tokens and the output shuffles with them. A model given no positional signal trains perfectly well while being unable to represent order at all, and no loss curve shows it.

NOT PRESENTED — FOR AFTERWARDS

Exercises Lecture 18

five, in the style of the written paper

Exercises — Lecture 18

1 Scaled dot-product attention divides by $\sqrt{d_k}$. State what the variance of the unscaled dot product is, and what the softmax does without the division. [5]

2 A decoder must not attend to positions after the one it is predicting. State the mechanism that enforces it, and what would leak without it. [5]

3 A pretrained body is fine-tuned at 10^{-3}, the rate that worked for the from-scratch model. Predict what happens, and state the rate the lecture uses instead. [4]

Solutions on Lecture 19's deck.

Exercises — Lecture 18 (continued)

4 A tf-idf vectoriser is fitted on every document of a 400-review corpus, and the matrix is split afterwards. Measured, the leak is worth 0.30 points. Say where the vectoriser should be fitted, and why the small number is no reason to leave the code as it is. [4]

5 A corpus contains duplicate documents across the train/test split. State why no pipeline fixes it, and say how the split must assign the rows instead. [4]

Solutions on Lecture 19’s deck.

THE FIVE SET AT THE END OF LECTURE 17

Solutions Lecture 17

not part of this lecture

Solutions — Lecture 17

1. Swapping the embedding table moved validation accuracy from 64.4% (our words, random vectors) to 58.4% with…

✓ So that the measurement says which change did the work: the tokenizer alone was worth −6.0 points, the tuned table +15.0. Frozen, the model stayed at chance, so the gain needs the 5,000 labels to adapt the pretrained vectors.

- Changing both at once and reporting the total says nothing about either
- Frozen against tuned is one boolean, and it separates what the vectors bring from what adapting them adds

2. State the gradient of cross-entropy with respect to the logits for a one-hot target, and what each row of…

✓ $\partial L/\partial \mathbf{z} = \mathbf{p} - \mathbf{y}$, prediction minus target; each row sums to zero.

- Both the softmax output and the one-hot target sum to one
- The deck checks it against autograd on one row of five classes

Solutions — Lecture 17 (2)

3. A classifier's forward ends in a softmax, and its output is passed to nn.CrossEntropyLoss. State what happens…

✓ Nothing raises and the model still trains, but the softmax is applied twice, so the gradient it trains on is not $\mathrm{p} - \mathrm{y}$. Return the logits and let the loss apply the softmax.

- `nn.CrossEntropyLoss` applies the softmax itself, so it expects logits
- A failure that trains and reports a number is the kind no shape check catches

4. A padded batch is fed to a GRU without pack_padded_sequence. State what the returned state summarises for a…

✓ The review followed by its padding: the state after the zeros, not after the last word. Pad the same review two different ways and require the logits not to move.

- Packing hands the recurrent layer only the real timesteps
- An output-shape test cannot see this; an invariance test can

Solutions — Lecture 17 (3)

5. Enlarging a word vocabulary drives the share of unseen distinct test words down, but even the full…

✓ A word vocabulary is closed and language is not. Subword tokenisation.

- The next review can always contain a word nobody has written yet, and the unseen words are the rare, informative ones
- Starting from characters covers every string by construction, so the floor is zero rather than 61.3%

LECTURE 19 · LECTURE NOTES

Information retrieval: the lexical foundation

The retrieval problem: a query, a corpus, a ranking — and why it is not classification
BM25, built one correction at a time: term frequency, saturation, inverse document frequency, length
Evaluating a ranking: precision@k, reciprocal rank, average precision, NDCG
Where the relevance judgements come from, on SciFact

Part V begins — beyond the textbook, and examinable