Text

IMDb

LECTURE 17 | GÉRON CH. 14

Applications of Machine Learning

BSc Mathematics of Artificial Intelligence · Third year

Where we are

Last lecture, recurrent networks on an ordered series. What carries over:

What we found	Why it matters today
A recurrent layer reads a sequence step by step and returns one fixed-size state	today the sequence is a review, and that state is what we classify
Gated cells keep a long sequence trainable; on the ridership forecast the GRU beat the plain cell and the LSTM	today's classifier is a GRU
A ranking that rested on one seed flipped when only the seed changed	every comparison today is one seed, and we say what that can settle
Every model was read against a baseline that needs no learning, on the same rows	today's anchor is a bag of words, measured first

New today: the input is **free text** — variable length, discrete, and with no natural numeric encoding at all.

Today

1. The brief, the corpus, and the anchors a bag of words already gives you
2. **The mathematics** — softmax, cross-entropy, and the gradient a classifier actually trains on
3. **The method** — subword tokenisation, trainable embeddings, and a recurrent classifier
4. The same architecture with pretrained embeddings: change one table, and measure what moves
5. **Design exercise** — a customer-support SaaS's automatic ticket routing, in nine steps, with the pitfalls and how to address them

Open Lecture 17 in Colab

The stakeholder

A streaming service’s customer-feedback desk.

Every title carries free-text reviews. Nobody reads them. The desk wants, for each new review, a single label — *satisfied* or *not* — so that the unhappy ones can be routed to a human within the hour.

- No star rating is available at the point where they need the decision
- The volume is far past what a person can triage
- They are prepared to accept mistakes; they are not prepared to accept silence

What they actually asked for

Their words	What that specifies
“tell us if it is a complaint”	binary classification
“from the text alone”	one variable-length string in, one label out
“within the hour”	inference cost matters; training cost does not
“we will read the ones you flag”	the errors are absorbed by a human

Four sentences, and every one of them is a design decision you would otherwise have made silently.

Free text has no columns

Everything in Chapters 1–13 arrived as a table: a fixed set of features, in a fixed order, each a number or a category.

> **A REVIEW IS NONE OF THOSE**
>
> Variable length. No fixed positions. A vocabulary that is open — the next review can contain a word that has never been written before.

So before any model, there is a question no previous application had to answer: **what is the unit?**

The corpus

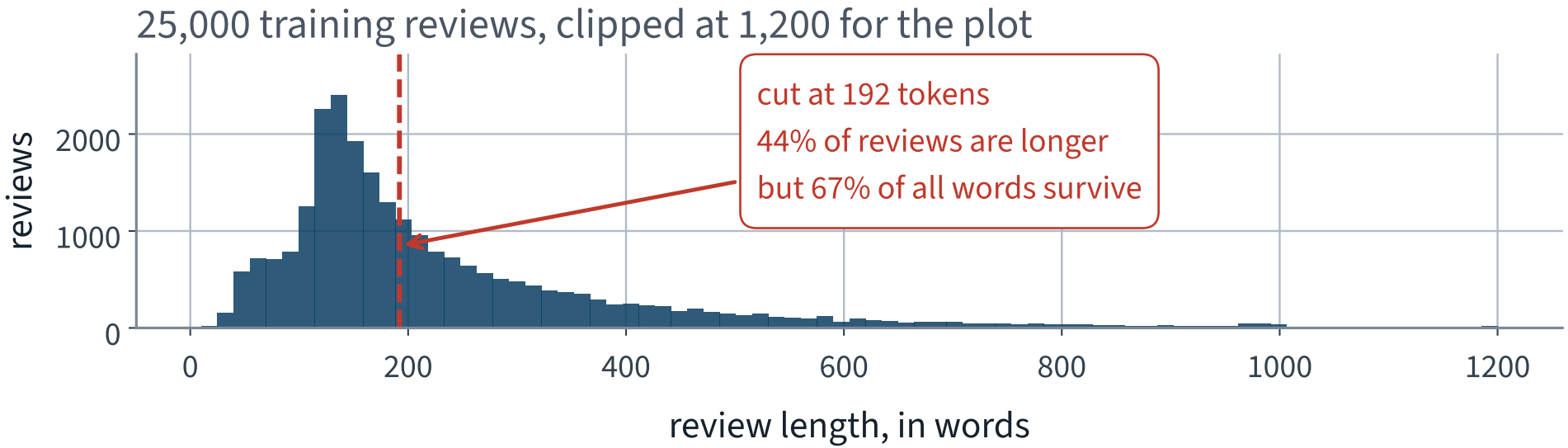
The Large Movie Review Dataset — the corpus the book uses for sentiment. Fetched from Stanford directly, about 80 MB.

```
URL = "https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz"

def load_imdb():
    if not DATA.is_dir():
        urllib.request.urlretrieve(URL, tarball)
        with tarfile.open(tarball) as t:
            t.extractall(path=CACHE, filter="data")
    ...
```

A function, not a manual download — so a cold kernel can rebuild it. The split into train and test ships with the corpus, so every course in the world reports on the same reviews.

How long is a review?



A long tail. Wherever you cut, you cut something — so the cut is a decision to make on purpose and record.

Cutting at 192 truncates 43.8% of the training reviews and keeps 66.7% of all their words.

What the model has to produce

$$\text{string} \longrightarrow \mathbb{R}^2$$

Two numbers per review: one score per class. Everything between the string and those two numbers is what we build today.

- A **tokenizer**, turning a string into a sequence of integers
- An **embedding**, turning each integer into a vector
- A **recurrent layer**, turning the sequence into one vector
- A **linear head**, turning that vector into two numbers

Only the first two are new. The last two are Lectures 8 and 16 unchanged.

Which metric?

From Lecture 3: *count the classes before choosing a metric.* We counted. They are 50.0% to 50.0%.

> **SO ACCURACY IS DEFENSIBLE HERE**
>
> Under a 50/50 base rate, the degenerate classifier scores 50% and has nowhere to hide. That is the condition Lecture 3 said accuracy needs, and this corpus satisfies it.

It is defensible *on this corpus*. Real feedback is not balanced, and the first slide of the next application you build should ask the question again.

Bag of words, precisely

One column per term in the vocabulary; one row per review; the entry is how often the term occurs, weighted down if the term is common everywhere.

$$\text{tf-idf}(t, d) = \underbrace{\text{count}(t, d)}_{\text{term frequency}} \times \underbrace{\log \frac{N}{1 + n_t}}_{\text{inverse document frequency}}$$

N is the number of documents and n_t the number containing t. The second factor is what stops *the* and *movie* dominating the first. Scikit-Learn uses a smoothed variant, $\log \frac{1+N}{1+n_t} + 1$, and normalises each row to unit length.

✗ Note what is not in that formula: position. The bag has no order, so it cannot see that *not* changes the word after it.

The whole anchor, in eleven lines

```
1  from sklearn.feature_extraction.text import TfidfVectorizer
2  from sklearn.linear_model import LogisticRegression
3
4  # fit on the training reviews ONLY — the vectoriser is a fitted estimator
5  vec = TfidfVectorizer(min_df=2, ngram_range=(1, 2),
6                        max_features=200_000)
7  X_train = vec.fit_transform(train_x)
8
9  clf = LogisticRegression(max_iter=2000, C=4.0).fit(X_train, train_y)
10 acc = (clf.predict(vec.transform(test_x)) == test_y).mean()
11 print(f"tf-idf + logistic regression: {acc:.1%}")
```

Line 7 is `fit_transform` on the training reviews; line 10 is `transform` on the test reviews. Different verbs. Hold on to that — it is the last thing we do in the next lecture.

The anchors, measured

Model	Features	Test accuracy
Always predict one class	—	50.0%
tf-idf, single words	44,798	88.5%
✓ tf-idf, words and pairs	200,000	**90.1% ✓**

90.1%, from counting words.

Adding pairs of adjacent words buys 1.6 points, and pairs are the cheapest way a bag can see *not good*. The brief's *other* requirement — "within the hour" — is one the metric cannot see: a bag of words scores the whole test set long before the hour is up.

Softmax

The last linear layer returns K unbounded numbers, the **logits** $\mathbf{z}$. Softmax maps them to a probability vector:

$$\sigma(\mathbf{z})_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}} \qquad k = 1, \ldots, K$$

- Every entry is positive, because $e^x > 0$
- They sum to 1, because the denominator is the sum of the numerators
- It is order-preserving: $z_i > z_j \iff \sigma(\mathbf{z})_i > \sigma(\mathbf{z})_j$

The third point already tells you that `argmax` does not need it. A prediction never requires softmax.

Cross-entropy

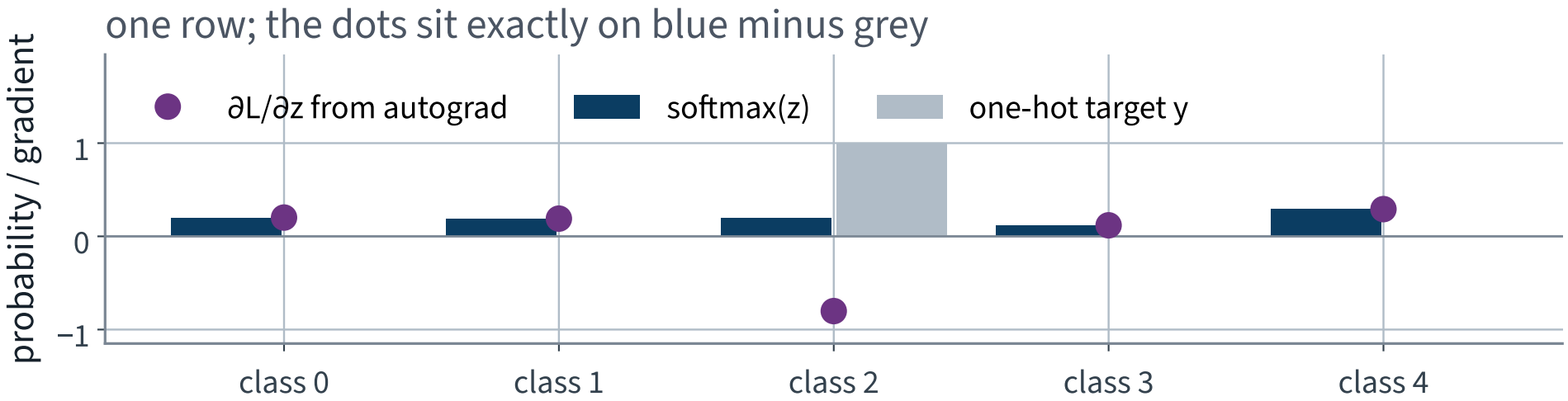
Between a target distribution $\mathbf{y}$ and a predicted $\mathbf{p}$:

$$H(\mathbf{y}, \mathbf{p}) = -\sum_{k=1}^{K} y_k \log p_k$$

A hard label is one-hot — $y_c = 1$, the rest 0 — so every term but one dies:

$$L = -\log p_c$$

The gradient is $\mathbf{p} - \mathbf{y}$



Differentiate $-\log p_c$ with respect to the logits and the result is prediction minus target. The dots are what PyTorch computed; the bars are what the algebra predicts. They agree to 5.551e-17.

The softmax applied twice

FAILURE CONDITION

Apply a softmax before a loss that expects logits and nothing raises, the model still trains, and the gradient it trains on is the wrong one. `nn.CrossEntropyLoss` applies the softmax itself, so it must be given logits: $\mathbf{p} - \mathbf{y}$ is the gradient only when the softmax is applied once.

The plan, in four boxes

Stage	In	Out
Tokenize	one string	`(192,)` integers
Embed	`(B, 192)`	`(B, 192, 128)`
Recur	`(B, 192, 128)`	`(B, 128)`
Head	`(B, 128)`	`(B, 2)`

Keep those four shapes in view. *What is the shape here?* is the question that catches most of what goes wrong from here on.

Step 1 · what is a token?

The unit the model counts as one thing.

- **Character** — tiny vocabulary, sequences six times longer, and every fact about a word has to be relearned from letters
- **Word** — the obvious choice, and the one that breaks
- **Subword** — the compromise the field settled on, and the subject of the next slides

This is a modelling decision, not a preprocessing detail. It fixes the vocabulary, the sequence length, and what the model can possibly represent.

Our word tokenizer, in full

```
WORD_RE = re.compile(r"[a-z0-9']+")

def word_tokens(s):
    return WORD_RE.findall(s.lower().replace("<br />", " "))
```

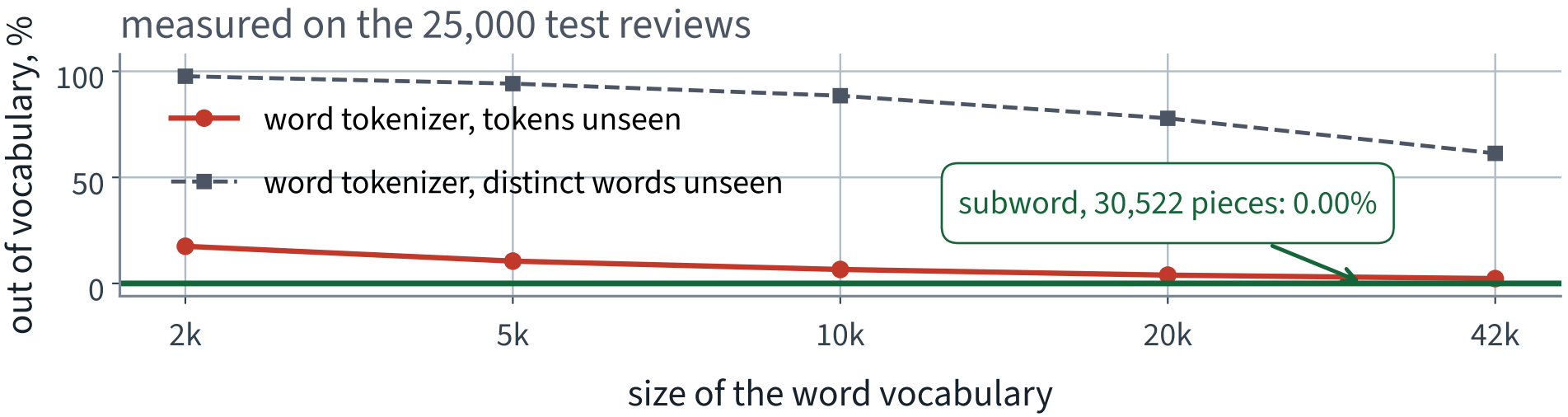
That is the whole specification. Three decisions are already made in it, and every one is arguable:

- Case is destroyed — *Terrible* and *terrible* become one token
- Punctuation is destroyed — so is the exclamation mark
- The corpus is HTML, and `<br />` is not a word

✗ If you had not looked at the raw text you would have learned `br` as one of the commonest words in the corpus.

> **FAILURE CONDITION — A VOCABULARY THAT CAN MISS**
>
> Word-level tokenisation has an open vocabulary: the next review can always contain a word the table has never seen, and the rate over distinct words is far worse than the rate over

What a word vocabulary costs you



✗ The token rate looks survivable. The rate over *distinct* words never does — and those are the informative ones.

Even at the full 42,370-word vocabulary of the *fit* split — every word in the 5,000 reviews a model fits on — 61.3% of the distinct words in the test set have never been seen. That is the floor of the upper curve.

Step 1b · subword tokenisation

Keep whole words for the common ones. Break the rest into pieces that *are* in the vocabulary.

- Start from characters, which cover everything by construction
- Repeatedly merge the commonest adjacent pair into a new token, until the vocabulary reaches the size you asked for
- Common words survive whole; rare words end up as two or five pieces

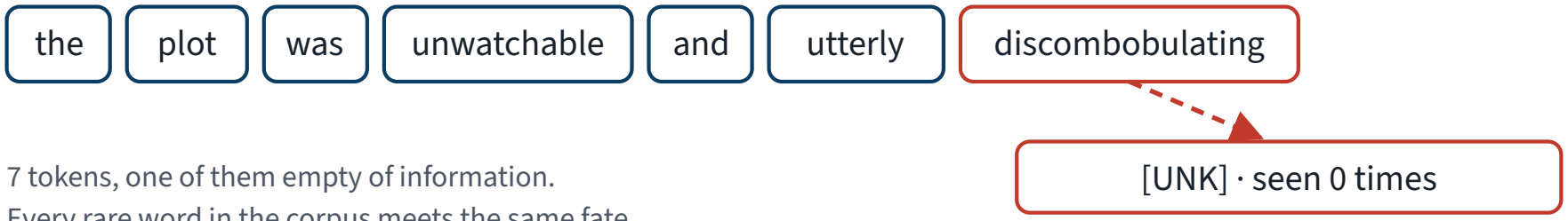
Byte-pair encoding, and WordPiece, which differs only in the criterion for which pair to merge. Chapter 14 names both.

The merges are learned **once**, on some corpus, and then shipped. That is the first pretrained thing we use today.

The same sentence, twice

The plot was unwatchable and utterly discombobulating.

our word tokenizer · 20,000 words, ranked by frequency in the training split



7 tokens, one of them empty of information.
Every rare word in the corpus meets the same fate.

WordPiece · 30,522 pieces, learned once on a corpus we never saw



13 pieces, none unknown. The three pieces of unwatchable keep the negation the word tokenizer threw away, and an unseen word falls back to pieces, a piece to characters. The vocabulary cannot miss.

Above, the rarest and most informative word became nothing. Below, it became four pieces the model has vectors for.

Why that is a structural fix, not a patch

> **THE VOCABULARY CANNOT MISS**
>
> An unseen word falls back to pieces; an unseen piece falls back to characters; the character set is closed. There is no input for which the tokenizer has to produce `[UNK]`.

- Morphology survives: *un* + *watch* + *able* keeps the negation that `[UNK]` destroyed
- Misspellings degrade rather than vanish
- The vocabulary is a fixed size you choose, whatever the corpus

Step 2 · from integers to vectors

Token 4271 is not four thousand of anything. The integer is a name, and arithmetic on names is meaningless.

The obvious fix is one-hot: 20,000 columns, one of them 1. That is a 20,000-dimensional vector per *token*, and 192 of them per review.

> **THE EMBEDDING IS THE FIX**
>
> A learned table with one row per token and d columns. Looking up row i is exactly multiplying the one-hot vector by that table — done without ever forming the one-hot vector.

Step 3 · the classifier

```
class GRUClassifier(nn.Module):
    def __init__(self, vocab):
        super().__init__()
        self.emb  = nn.Embedding(vocab, 128, padding_idx=0)
        self.rnn  = nn.GRU(128, 64, batch_first=True, bidirectional=True)
        self.head = nn.Linear(2 * 64, 2)

    def forward(self, x, lengths):
        e = self.emb(x)
        packed = nn.utils.rnn.pack_padded_sequence(
            e, lengths.cpu(), batch_first=True, enforce_sorted=False)
        _, h = self.rnn(packed)
        return self.head(torch.cat([h[0], h[1]], dim=1))
```

Thirteen lines, and eleven of them are Lecture 8. The two new ones are the embedding and the packing.

Three choices in that model

Choice	Why
`bidirectional=True`	a review's verdict can be in the first sentence or the last
the final hidden state	one fixed-size summary of a variable-length input
GRU, not LSTM	fewer gates and parameters, and on Lecture 16's forecast it beat the LSTM

`h[0]` is the forward direction's last state, `h[1]` the backward direction's — which, read as a sequence, is the *first* token. Concatenating them gives the head both ends.

`pack_padded_sequence`, and what it prevents

It hands the recurrent layer only the real timesteps, so the state returned for a review of 40 words is the state after its 40th word — not after 152 zeros.

> **TWO THINGS IT BUYS**
>
> Correctness, because the summary is of the review rather than of the padding. And speed, because the layer stops early on short sequences instead of stepping through the whole rectangle.

This slide exists because the version without it also runs, also trains, and also reports a plausible number.

The training loop is unchanged

```
1  opt   = torch.optim.Adam(model.parameters(), lr=1e-3)
2  lossf = nn.CrossEntropyLoss()
3
4  for epoch in range(2):
5      model.train()
6      for xb, lb, yb in batches:
7          opt.zero_grad()
8          out  = model(xb, lb)
9          loss = lossf(out, yb)
10         loss.backward()
11         opt.step()
```

Exactly the five lines from Lecture 8, in the same order, with the same two silent failures waiting if any of them is deleted. Nothing about text changes the loop.

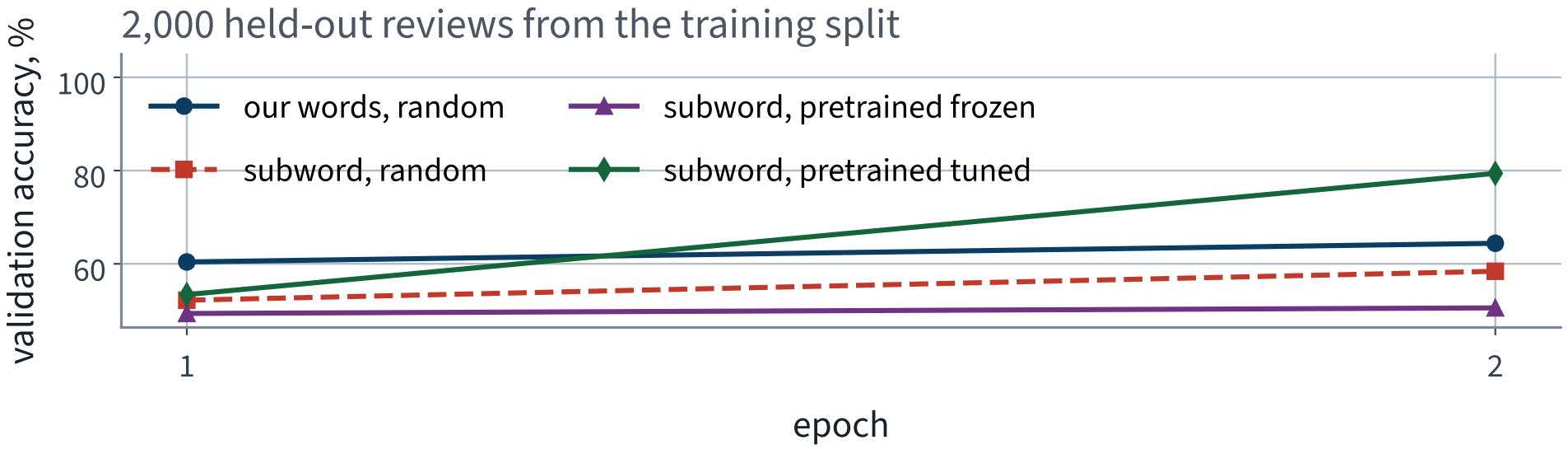
Note line 2. What `CrossEntropyLoss` is actually given, and why, is the mathematics you have just been through — softmax, cross-entropy and logits, earlier in this lecture.

What this costs to run

Setting	Value
Reviews used to fit	5,000
Reviews held out for validation	2,000
Reviews in the test set, untouched	25,000
Epochs, batch size, learning rate	2, 64, 0.001
Parameters	2,634,754

The budget is small on purpose, so that the run fits a free CPU, and every number from here on is from it; Lecture 8 measured why the device rarely earns its place at this scale.

Two epochs



Ignore three of the four curves for now. The blue one is the model we have just built.

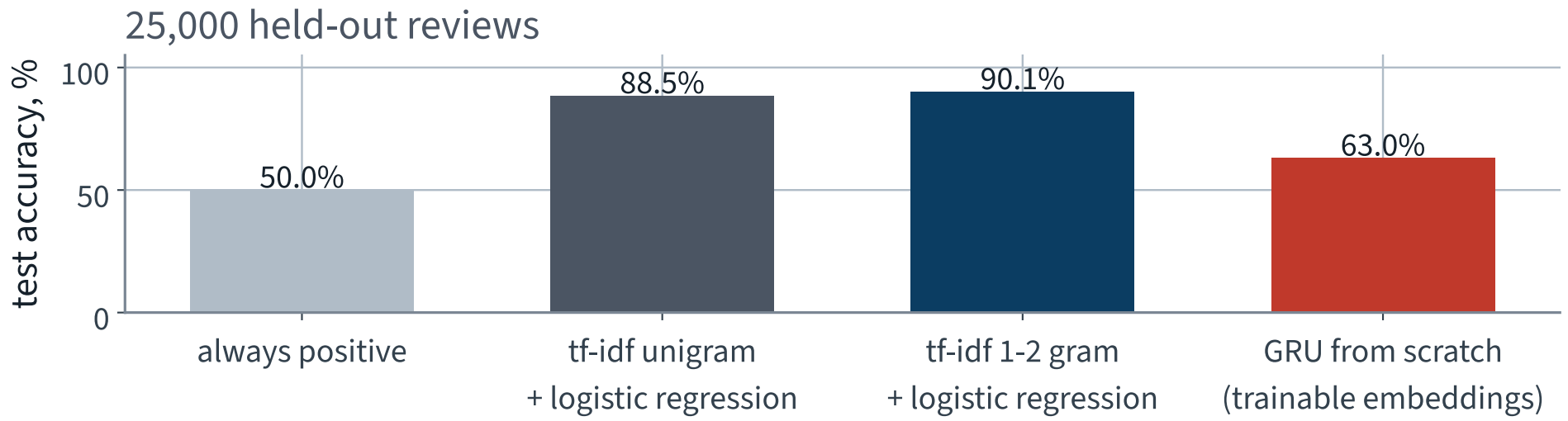
The result

Model	Test accuracy
Always one class	50.0%
tf-idf + logistic regression	90.1%
GRU, our tokenizer, trainable embeddings	63.0%

Hold on to the third number.

Then look at the second one for a moment before turning the page.

Against the anchors



✗ The recurrent network, with an embedding it learned itself, loses to counting words — by 27 points.

Why it loses

- The bag of words starts with 200,000 features that are already meaningful, fitted on all 25,000 training reviews. *Terrible* is its own column
- Our network starts with 20,000 rows of **random numbers** and has to discover from 5,000 labelled reviews, in two passes, that *terrible* and *awful* are related
- Most of its words are rare in 5,000 reviews, and one example is not enough to learn a 128-dimensional vector

> **THE HYPOTHESIS — AND IT IS ONLY A HYPOTHESIS**
>
> The architecture is not the bottleneck. The **embedding table** is, and it is being asked to learn the English language from 5,000 sentiment labels.

Written as a hypothesis on purpose. What follows is an experiment that can refute it, and you should decide now what result would.

The diagnosis said the embedding table

So replace the table. Nothing else.

- Same GRU, same width, same head, same optimiser, same learning rate
- Same two epochs, same seed, same 5,000 reviews
- Only the tokenizer and the initial contents of `emb.weight` change

Two changes, applied one at a time, so that the measurement says which one did the work. Changing both at once and reporting the total is how you learn nothing from an experiment.

Swap 1 · the pretrained tokenizer alone

DistilBERT's subword pieces (`distilbert-base-uncased`) instead of our words — and the embedding rows still **random**, exactly as before.

Tokenizer	Vocabulary	Validation accuracy
our words	20,000	64.4%
subword pieces, random vectors	30,522	58.4%

Worse, not better.

-6.0 accuracy points on the 2,000 validation reviews, from one seed each — not enough to call a ranking, but certainly not the gain the fix promised. Solving the out-of-vocabulary problem, on its own, **bought nothing**. The swap is read on validation, so the test set chooses nothing.

Swap 2 · pretrained vectors, frozen or tuned

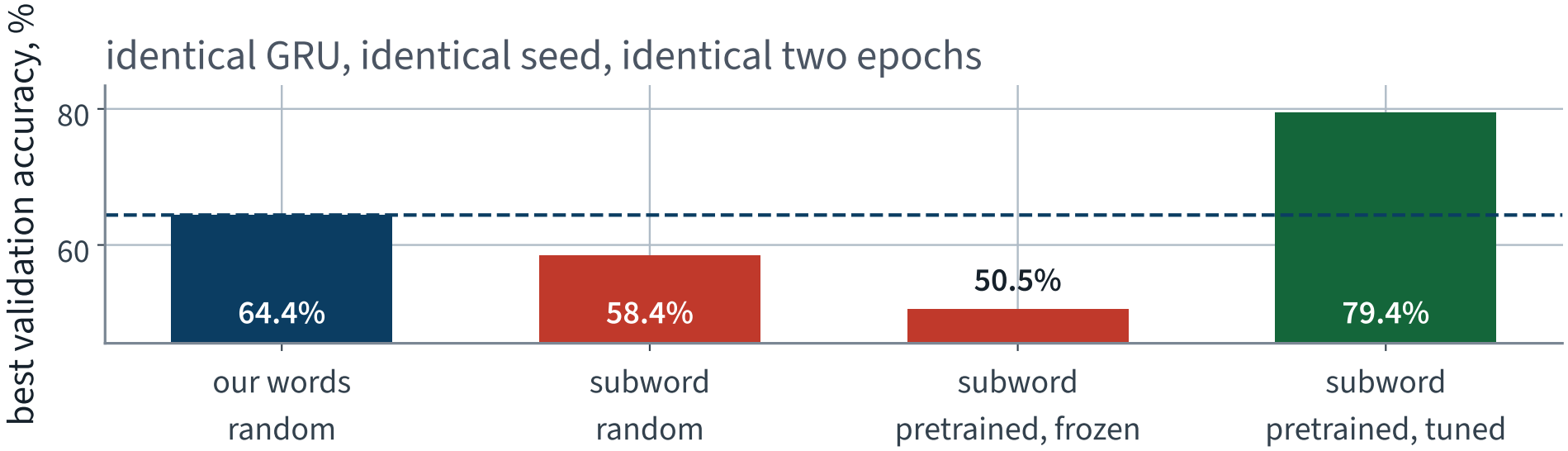
The same subword vocabulary already has a trained embedding table, 768 columns wide. Projected to our 128 with PCA, as `Z`, it goes into `emb.weight` and the architecture does not change.

```
model.emb.weight.data.copy_(torch.from_numpy(Z))
model.emb.weight.requires_grad = False       # frozen  — or True, tuned
```

	Frozen	Fine-tuned
What is learned	the GRU and head only	everything
Risk	the vectors may not suit the task	5,000 labels can overwrite them

Which is better is not decidable from the armchair. Run both; it is one boolean.

Four runs, one architecture



Two things differ across these four bars: which tokenizer produced the ids, and what was in the embedding table at step zero.

Tuned, the pretrained table reaches 79.4%, +15.0 points over our words. Frozen, at 50.5% after two epochs, the model is still at chance: the table is a starting point that the 5,000 labels then *adapt*.

What moved, and what did not

Unchanged	Changed
GRU width, depth, direction	the tokenizer
the head, the loss, the optimiser	the initial values in one tensor
epochs, batch size, learning rate, seed	whether that tensor is trainable
the data, the split, the metric	

Everything on the left is what you would ordinarily spend the afternoon tuning. Everything on the right is a few lines of code — and with 5,000 labels and two epochs it was worth 15.0 validation points, more than any of the left-hand column is likely to buy.

Text checklist — keep this

- Was the vocabulary built on the training split only?
- Is the tokenizer the same object at fit time and at inference?
- Are true lengths carried alongside the padded batch?
- Does padding change the output? Assert that it does not
- What fraction of the test tokens is `[UNK]`? Count it
- How much of each review survives truncation? Count that too

Six questions. The first is the one that will cost you the most, and it is the one the next lecture ends on.

The notebook for this lecture

Open Lecture 17 in Colab

- **Runs on free CPU.** IMDb is subsampled and the model is small
- The corpus, the review lengths and the cut at 192, and both bag-of-words anchors
- The gradient of cross-entropy with respect to the logits, verified against autograd: it really is prediction minus target
- The word vocabulary against subword pieces, the embedding, the packed GRU classifier — and the swap, one change at a time

> **WHAT TO CHANGE**
>
> Concatenate `h[0]` with itself instead of `h[1]`. The shapes and parameter count are unchanged, nothing raises, and the backward direction is dead weight. What happens to the accuracy?

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

The task: a customer-support SaaS's automatic ticket routing

> **THE BRIEF**
>
> A customer-support SaaS sells the helpdesk other companies run support on, and is shipping auto-routing. For each ticket — free text from a customer's customer — the system outputs the **team that should handle it**, from the account's own teams — billing, product, security — so it lands in the right queue unread. Customers measure the time to first response their service agreements promise; the SaaS measures hand triage, and renewals.

- **Given:** each account's past tickets, with the team that closed each; its current teams
- **Scale:** thousands of accounts with thousands to millions of tickets each; a handful to dozens of teams each, two or three taking most traffic
- **Consumers of the output:** the queueing system, the team leads' dashboards, and triage for what the model will not route

✓ 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: one team per ticket, and count the teams first

- One string in, one score per team out: softmax over the teams, cross-entropy on the logits — the class is the team that resolved the ticket
- Count the classes before choosing a metric: they are nowhere near balanced; the largest team alone takes most of the traffic
- So the anchors are always-the-largest-team, then a bag of words with logistic regression — measured first, same tickets

> **PITFALL**
>
> Accuracy. A router that sends everything to the largest team scores the base rate and looks fine, while the small teams — where a misroute costs most: a security report in the billing queue past the promised response time — are invisible in the number. Report the confusion matrix and each team's recall beside the base-rate anchor, and ask the account what a misroute costs per team before optimising.

Step 2 — The label: which team, decided by whom

Pitfall	What happens	What to do
The first assignment as the label	that is the account's old rule, or a tired agent's; the model learns to imitate it, mistakes included	label with the team that *closed* the ticket, and only tickets that are closed
The ticket that bounced	three teams touched it; the closer is one of them by accident	keep the closing team, and flag bounced tickets so their share can be read separately
The ticket that belongs to two teams	a refund *and* a login failure; one softmax forces one answer	agree the rule with the account — the team that must act first owns it — and apply it when the label is read back

✓ **Write the label's definition down before the first fit. Every later number is a number about *that* label.**

Step 3 — Split by time: the test set is later tickets

- Train on the tickets up to a cut, validate on the months after it, test on the latest months — the order the router will face in the account's helpdesk, with products, teams and words moving
- Choose everything — tokenizer, cut length, epochs, which model — on the validation tickets; read the test tickets once

> **PITFALL**
>
> A random split. An outage generates thousands of near-identical tickets in one afternoon, and a shuffle puts half of them in training and half in test: the model learns one afternoon's wording and is scored on the same afternoon. The score climbs, and the next outage, about a new product, is the first the deployed model has ever seen. Split by date, and look at what the test months contain.

Step 4 — What is a token, and whose text is it fitted on

- Look at the raw text first: quoted replies, signatures, mail headers, pasted logs, ticket identifiers — or the commonest tokens in the vocabulary will be *wrote*, *regards* and a date format
- Subword pieces, so that the account's product names, error codes and misspellings become pieces with vectors, not an unknown token
- Cut each ticket at a fixed length and count what survives: the subject plus the opening of the body, because the problem is usually stated first and the pasted log comes after

> **PITFALL**
>
> The vocabulary built on every ticket, the test months included. Nothing raises; the model simply has a row for a product name that did not exist when it was supposedly fitted. Build the vocabulary on the training tickets only, keep the tokenizer as one serialised object per account used at fit time and at inference, and count the unknown-token rate on the later tickets — the first sign the language has moved.

Step 5 — The model: an embedding table, a GRU, and where the table comes from

- Tokens → embedding rows → a bidirectional GRU packed to the true lengths → a linear head with one logit per team; the verdict can be in the subject or the last line, so both ends reach the head
- The embedding table starts **pretrained** and is fine-tuned, because the small teams have a few hundred labels each and a random table would have to learn the language from those
- Change one thing at a time against the bag-of-words anchor: the tokenizer, then the table, then whether it trains

> **PITFALL**
>
> A table of random numbers learned from the tickets alone. It trains, the loss falls, and it loses to counting words by a wide margin, because every rare product name is one row that one example cannot teach. If the pretrained table, tuned, does not beat the bag of words on the validation tickets, ship the bag of words: it is cheaper to serve for thousands of accounts, and nobody bought a neural network.

Step 6 — What fails without an error

Failure	What you see	What to do
Softmax before the loss	it trains, on the wrong gradient	logits into the loss; the loss applies the softmax itself
The state after the padding	a plausible number, slightly worse	carry the lengths, pack the batch, assert the logits ignore the padding
Truncation that removes the problem	long tickets routed by their greeting	count how much of each ticket survives the cut; keep the subject whole

PITFALL

The version without packing also runs, also trains, and also reports a number: the summary of a short ticket is the state after a hundred zeros, and short tickets are the commonest kind. A shape test cannot see it. An invariance test can — pad the same ticket two ways and require the same logits — and it belongs in the release's test suite, not in a notebook.

Step 7 — Evaluate on the validation months, and stop at the best epoch

- Score the validation tickets after every epoch; the curve rises, peaks, and falls as the few labels of the small teams are memorised
- Keep the checkpoint of the **best validation epoch**: that is the model, and the number of epochs is a result, not a setting
- Then the confusion matrix per team on the test months, read once, with the error count beside the anchors; a small gap from one seed is not a ranking

> **PITFALL**
>
> The last epoch shipped, because that is what the loop leaves in memory — or the number of epochs chosen by watching the test set, which then chooses nothing. Early stopping is a selection, and selections are made on the validation tickets; the test set is read after every decision has been taken, and it is later in time than all of them.

Step 8 — In production: opt-in rollout, and the label that moves

- Behind a per-account switch, opt-in first: route when the top team's probability is high; the rest go to triage, whose decisions are the freshest labels
- Monitor per account and team: volume routed, share re-routed by hand, unknown-token rate — the live error, and the sign the language has moved
- Retrain each account on a recent window at a fixed cadence; the data team is on call for accounts whose re-routes jump

PITFALL

A customer reorganises in the admin panel: one team splits, another is merged, a product retired. The label set the model was trained on no longer exists, and tickets land in a queue nobody reads until the customer's customers complain. Version each account's label set, map history through each reorganisation before retraining, and treat an agent's re-route as the correction it is.

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

Closed tickets, labelled by the closing team → split by date → raw text inspected → subword vocabulary on the training tickets → bag-of-words anchor → pretrained embeddings, a packed GRU, logits into the loss → best validation epoch → confusion matrix on the test months → opt-in rollout, triage, monitor re-routes → retrain.

Pitfall	Address it by
Accuracy under imbalance	per-team recall and the confusion matrix, beside the base rate
First assignment as label; two-team ticket	the closing team; a written rule for who acts first
A random split	splitting by date; testing on the latest months
Vocabulary built on all tickets	training tickets only; one tokenizer per account; unknown rate counted
Random embeddings on few labels	a pretrained table, tuned, one change at a time
The state after the padding	lengths carried, batches packed, an invariance test

What we did

1. Framed a free-text brief on a balanced corpus, so accuracy is defensible, and measured a bag-of-words anchor at 90.1% before building any network
2. Softmax turns logits into probabilities; cross-entropy against a hard label is $-\log p_c$, and its gradient with respect to the logits is exactly **prediction minus target** — so the loss is given logits, never a softmax
3. A word vocabulary is closed: 61.3% of the distinct test words are unseen even at its full size. A subword tokenizer cannot miss
4. An embedding is a trainable table; a packed bidirectional GRU on top of it lost to counting words, 63.0% against 90.1%
5. Swapped one table and nothing else: the subword vocabulary alone bought nothing, pretrained vectors tuned bought 15.0 validation points
6. Designed a customer-support SaaS's automatic ticket routing in nine steps — the teams counted before the metric, the closing team as the label, a split by date, a vocabulary built on each account's training tickets only, a pretrained table tuned against the bag-of-words anchor, packing asserted, the best validation epoch kept, an opt-in rollout, and a label set versioned when a customer reorganises its teams

NOT PRESENTED — FOR AFTERWARDS

Exercises Lecture 17

five, in the style of the written paper

Exercises — Lecture 17

1 Swapping the embedding table moved validation accuracy from 64.4% (our words, random vectors) to 58.4% with the subword tokenizer alone, and to 79.4% with pretrained vectors, tuned. Why were the two changes applied one at a time, and what does the frozen run add? [5]

2 State the gradient of cross-entropy with respect to the logits for a one-hot target, and what each row of that gradient must sum to. [5]

3 A classifier's `forward` ends in a softmax, and its output is passed to `nn.CrossEntropyLoss`. State what happens in training, and the fix. [5]

Solutions on Lecture 18’s deck.

Exercises — Lecture 17 (continued)

4 A padded batch is fed to a GRU without `pack_padded_sequence`. State what the returned state summarises for a review shorter than the cut, and a test that detects it. [5]

5 Enlarging a word vocabulary drives the share of unseen distinct test words down, but even the full 42,370-word vocabulary leaves 61.3% of them unseen. Explain why a larger vocabulary is not the fix, and name what is. [4]

Solutions on Lecture 18’s deck.

THE FIVE SET AT THE END OF LECTURE 16

Solutions Lecture 16

not part of this lecture

Solutions — Lecture 16

1. A recurrent network is trained on 56-day windows drawn from one series. Explain why shuffling the windows is…

✓ A window is a training example; the series is the thing the example is cut from.

- The order inside a window is the signal, and shuffling windows leaves it intact
- Shuffling the series destroys the very structure the model exists to read

2. A one-layer RNN beats a three-layer RNN on test and on train. Re-run with only the seed changed, the same…

✓ On one seed. The seed moves both the recipe each model's selection chooses and the fit itself, by more than the gap between the two models. Settle it with several seeds and a gap larger than their spread, or report it as unresolved.

- The recipe is chosen on a slice of the training period, and two scores a few per cent apart can swap under another seed
- A ranking whose gap is smaller than its spread over seeds is not a finding

Solutions — Lecture 16 (2)

3. Gates were introduced after a plain recurrent cell failed over long windows. State the mechanism, in terms of…

✓ The gradient carries 56 copies of the same matrix, and powers of one matrix vanish or explode; gates provide an additive path whose derivative is near one.

- Repeated multiplication through a deep stack, one factor per step, is what makes a gradient vanish or explode
- A gate lets information pass without being multiplied at every step

4. A colleague builds the multivariate forecaster with day_type.shift(-1) as a feature and objects that it reads…

✓ It is not a leak. Ask whether the value is available at prediction time: the next day's type is on the public calendar, its ridership is not.

- The sign of the shift decides nothing; the timing of the information does
- No static check can answer it: it is a property of the world, not of the code

Solutions — Lecture 16 (3)

5. Report the training MAE beside the test MAE. Name the two failures the pair distinguishes, and say why the…

✓ Underfitting, where both are poor, and overfitting, where train is far better. They have opposite fixes.

- A single bad test number is consistent with either
- More capacity fixes one and makes the other worse

LECTURE 18 · GÉRON CH. 14–15

Attention and transformers

Scaled dot-product attention — queries, keys and values, and why the scores are divided by $\sqrt{d_k}$
Fine-tuning a pretrained transformer on these reviews, against today's models
Sentence embeddings for search and grouping — and where a fitted step leaks

Run the Lecture 17 notebook first