#

SESSION 2

Language Modeling by Counting

Building makemore, part 1 — a working language model with no neural network

Bigrams · probability distributions · sampling · negative log likelihood · the vocabulary of neural networks

Where This Session Sits

We go under the hood once — so everything afterwards is understood, not memorised

1. **Foundation models**
   What AI engineering is
2. **Counting bigrams**
   A model with no network
3. **The same model, learned**
   Gradient descent + embeddings
4. **Self-attention**
   The core of the transformer
5. **The full GPT**
   Blocks, training, scaling
6. **Chapter 2**
   Data, scaling laws, post-training
7. **Chapter 3**
   Evaluation

Say this out loud

Session 1 said you build on top of existing models rather than training them. Sessions 2–5 train one from scratch. That is deliberate — one trip under the hood, then back up the stack for good.

Today's promise

By the end of this session you will have built a complete, working language model — using nothing but counting.

What We Are Building: makemore

It takes a file of things, and makes more things like them

A character-level language model

Given the characters so far, it predicts a probability distribution over the next character. Generate by sampling that distribution, appending, and repeating.

Exactly the task GPT performs

The difference is scale, not kind. GPT predicts the next token from a vocabulary of ~100,000 over thousands of tokens of context. We predict the next character from 27, using one character of context.

Why names?

- Small enough to train in seconds on a laptop — you can iterate live in class
- You can judge the output yourself: does it look like a name? No benchmark needed
- Every concept transfers unchanged to words, tokens and full-scale models

The Dataset

names.txt — one name per line, and nothing else

32,033

names

27

symbols (a–z plus one special)

≈228k

character pairs to count

The only question this model answers

Given the characters seen so far — what character is likely to come next?

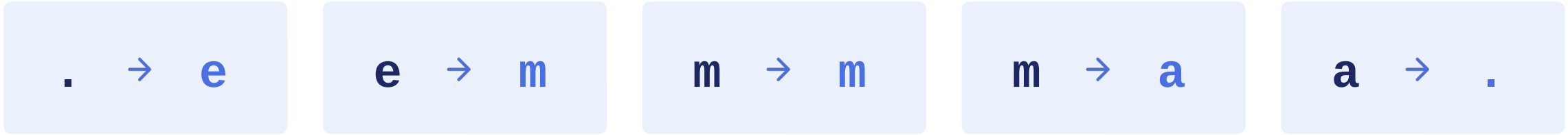
A single name is many training examples

“isabella” does not teach the model one thing. It teaches it that i often starts a name, that s follows i, that a follows b, and that a can end a name — nine separate lessons from eight letters.

The Bigram Simplification

Look exactly one character back — and forget everything before it

emma *becomes five training pairs*



context → next character

One token does two jobs

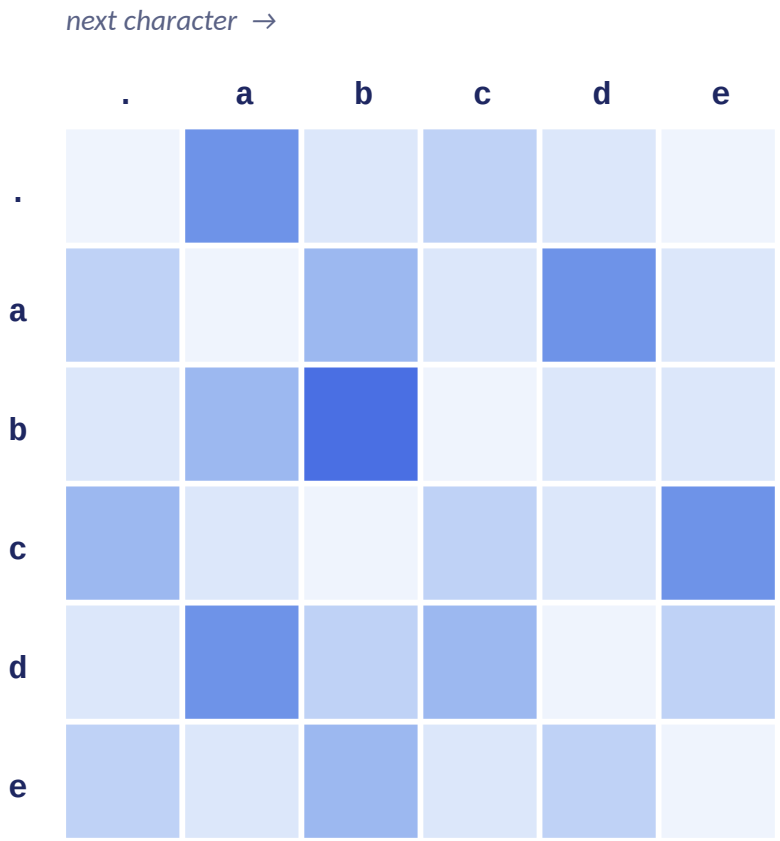
The dot marks both “name starts here” and “name ends here”. One special symbol instead of two keeps the table square: 27 × 27. Without an end symbol the model would have no way to decide when to stop generating.

This throws away almost everything

Predicting the character after the second m in “emma”, the model sees only m. Not that the name began with e, not how long it is. It is a deliberately terrible model — and still a complete, working one.

Training = Tallying

Walk the dataset once, count every pair. That is the entire training procedure.



a 6×6 corner of the real 27×27 table

N[i, j]

how many times character j followed character i, across all 32,033 names

A row is a context; a column is what came next. Each row is one character's story.

The structure is visible: names beginning with a are common, names ending in n are very common.

No gradients, no epochs, no learning rate. One pass over the data and the model is finished.

From Counts to Probabilities

Divide each row by its own total — every row becomes a distribution summing to 1

```
P = N.float()
P /= P.sum(1, keepdim=True)
```

Row 'a' no longer says “n appeared 1,000 times after a”. It says “given a, there is a 12% chance of n”. That is a model.

The bug worth stopping on: keepdim=True

Drop keepdim and the sum collapses from shape (27,1) to (27,). Broadcasting then aligns it along the wrong axis and normalises the columns instead of the rows. No error is raised. The code runs, the shapes are legal, the model is silently wrong.

Worth dwelling on in a CS class: this is a whole category of bug your students have not met before — code that is type-correct, runs cleanly, and produces a wrong answer with no signal that anything happened.

Sampling: Making It Generate

Draw from the distribution, append, feed back, repeat until the model emits a dot

1. **Start at .**
2. **Look up that row**
3. **Sample a character**
4. **Repeat until .**

What it actually produces

```
mor
axx
minaymoyes
kondlaisah
```

Bad — but recognisably name-shaped bad.

Is it better than nothing?

Run the same sampler on a uniform random table and you get unpronounceable strings of consonants. The bigram output has vowels in plausible places and stops at sensible lengths. Rubbish output is not the same as a model that learned nothing — which is exactly why we now need a number.

Teaching note: fix the random seed before class. Students comparing different outputs while you debug live is a guaranteed derailment.

Scoring the Model: Negative Log Likelihood

Turning “is this any good?” into a single number you can minimise

Likelihood

Multiply the probability the model gave to every pair that really occurred

Underflows to 0.0

Log likelihood

Take logs, so the product becomes a sum

Numerically stable

Negative log likelihood

Negate, then average over all pairs

Lower is better

2.45

our counting model

3.29

guessing uniformly at random

The whole game

Everything from here — including GPT — is the same loss, minimised by better means.

A loss function is not an ML concept students need to fear — it is a score, defined so that smaller means better, and differentiable so it can be improved automatically.

One Zero Ruins Everything

Why every language model quietly refuses to call anything impossible

The problem

If a pair never appeared in the data, its probability is exactly 0. The log of 0 is negative infinity. One pair the model considers impossible — and the loss for the entire dataset becomes infinite, no matter how good the other 228,000 predictions were.

The fix: smoothing

Add 1 to every count before normalising. Nothing becomes impossible — only unlikely. Add more than 1 and the model grows steadily more uniform; add nothing and it is brittle. The amount you add is a dial you choose, not something learned from data.

This is your first hyperparameter

A parameter is learned from the data. A hyperparameter is a choice you make about how learning happens — and the only honest way to set it is to try values and measure on data the model was not fitted to. That distinction is worth planting now; it returns in every session that follows, and it is where the train/validation/test split comes from.

Vocabulary: The Shape of a Model

Nothing here appears in today's code — you meet these next week, so meet the words first

Parameter (weight)

A number the model adjusts to fit the data. Today we had none — we counted instead of fitting.

Softmax

Exponentiate the logits, then divide by their sum. Turns any scores into a distribution — exactly what we did to the counts today.

Model

A function with parameters. Input goes in, a prediction comes out.

One-hot encoding

Representing character 13 as 27 numbers: a single 1 and twenty-six 0s. How you feed a symbol to something that only does arithmetic.

Logits

Raw, unbounded scores the model outputs — one per possible next character. Not yet probabilities.

Embedding

A learned, compact vector for each character, replacing the one-hot. Session 3.

Vocabulary: How a Model Learns

The loop that replaces counting — six words, one idea

Forward pass

Run the input through the model and get a prediction out.

Backpropagation

The algorithm that computes every gradient efficiently, working backwards from the loss.

Loss

One number saying how wrong that prediction was. We built one today: NLL.

Learning rate

How big a step to take. Too small and nothing moves; too large and it diverges.

Gradient

For each parameter: which way to nudge it, and how hard, to make the loss smaller.

Epoch / iteration / batch

One pass over the data; one update step; the chunk of examples used for that step.

Be honest with them: today's model had nothing to learn. We tallied and stopped. Every word on this slide describes next week.

So Why Bother With Neural Networks?

Counting works perfectly. It just cannot be made to look further back.

Context length	Table size	Verdict
1 character	`27 rows`	Today's model
2 characters	`729 rows`	Still fine
3 characters	`19,683 rows`	Getting sparse
4 characters	`531,441 rows`	*Most cells empty*
5 characters	`14,348,907 rows`	*Hopeless — and we have only 228k pairs*

Counting scales exponentially with context. A neural network scales gracefully — it compresses the pattern into a fixed set of weights instead of storing a cell for every possible history. That is the entire reason we move to gradient descent next week, and eventually to attention.

Before Next Session

Everything here is doable with what you saw today — no neural networks required

1	**Build the trigram model**	Use two characters of context instead of one. How many rows does the table need? Does the loss improve?
2	**Split the data**	Hold out 10% of the names. Compute the loss on data the model never counted. Does the trigram still win there?
3	**Tune the smoothing**	Try adding 0.01, 1, and 100 to the counts. Plot the held-out loss against the amount added. What shape is it, and why?
4	**Score your own name**	Compute the model's negative log likelihood for your name. Is it a likely name or an unlikely one?

Exercise 2 is the important one — it is where students discover that a model can get better on the data it was built from while getting worse everywhere else.

What To Remember

- A language model is nothing more exotic than a probability distribution over the next symbol.
- You can build a real, working one with counting alone — no neural network, no gradients, no training loop.
- A loss function converts “is this good?” into a number you can minimise. Negative log likelihood is that number, for GPT as much as for this.
- Never assign zero probability to anything. Smoothing is the first hyperparameter, and hyperparameters are chosen on held-out data.
- Counting fails not because it is inaccurate, but because it cannot see further back without exploding in size.
- Next week: the identical model, learned instead of counted — and it will reach the same loss the hard way, for reasons that matter.