Australian National University

Weighted Randomness

Learn how to make random choices where some options are more likely than others — an operation at the core of all generative AI.

You will need

- 10-sided dice (d10)
- coloured marbles or beads in a bag

Your goal

To randomly choose from a fixed set of outcomes according to a given probability distribution.

Key idea

Sometimes we need to make random choices where some outcomes are more likely than others. There are ways to do this which ensure certain relationships *on average* between the outcomes (e.g. one outcome happening twice as often as another one).

Algorithm 1: beads in a bag

- **materials**: coloured beads, bag
- **setup**: count out a number of beads corresponding to the desired weights for each outcome
- **sampling procedure**: shake the bag, then draw one bead without looking

Example

You want to choose an ice cream flavour: `vanilla` 50% of the time, `chocolate` 30%, and `strawberry` 20%.

- add 5 white beads to the bag (corresponding to `vanilla`)
- add 3 brown beads to the bag (corresponding to `chocolate`)
- add 2 red beads to the bag (corresponding to `strawberry`)

Draw a bead from the bag — that's your ice-cream choice for today.

Algorithm 2: dice with ranges

- **materials**: d10 (or d6, d20 as alternatives)
- **setup**: assign number ranges proportional to weights (see table, right)
- **sampling procedure**: roll the die, then look up the corresponding outcome

Example

- for 60% vanilla/40% chocolate, roll a d10: 1-6 means `vanilla`, 7-10 means `chocolate`
- for 50% vanilla/30% chocolate/20% strawberry, roll a d10: 1-5 means `vanilla`, 6-8 means `chocolate`, 9-10 means `strawberry`

You can use different dice (d6, d10, d20, d120, etc.), it will just change the number ranges corresponding to each outcome.

d10 dice roll → outcome mapping table

	1	2	3	4	5	6	7	8	9	10
2	0 4	5 9								
3	0 3	4 6	7 9							
4	0 2	3 5	6 7	8 9						
5	0 1	2 3	4 5	6 7	8 9					
6	0	1	2	3	4	5	6	7	8	9
7	0	1	2	3	4	5	6	7	8	9
8	0	1	2	3	4	5	6	7	8	9
9	0	1	2	3	4	5	6	7	8	9

Australian National University

LLMs Unplugged | Cybernetic Studio

Training

Build a bigram language model that tracks which words follow which other words in text.

You will need

- some text (e.g. a few pages from a kids book, but can be anything)
- pen, pencil and grid paper

Your goal

To produce a grid that captures the patterns in your input text data. This grid is your *bigram language model*. **Stretch goal**: keep training your model on more input text.

Key idea

Language models learn by counting patterns in text. “Training” means building/constructing a model (i.e. filling out the grid) to track which words follow other words.

Algorithm

1. **preprocess your text**:
   - convert everything to lowercase
   - treat words, commas and full stops as separate “words” (and ignore all other punctuation and whitespace)
2. **set up your grid**:
   - take the first word from your text
   - write it in both the first row header and first column header of your grid
3. **fill in the grid** one *word pair* at a time:
   - find the row for the first word (in your training text) and the column for the second word
   - add a tally mark in that cell (if the word isn’t in the grid yet, add a new row *and* column for it)
   - shift along by one word (so the second word becomes your “first” word)
   - repeat until you’ve gone through the entire text

Example

Original text: *“See Spot run. See Spot jump. Run, Spot, run. Jump, Spot, jump.”*

Preprocessed text: `see` `spot` `run` `.` `see` `spot` `jump` `.` `run` `,` `spot` `,` `run` `.` `jump` `,` `spot` `,` `jump` `.`

After the first two words (`see` `spot`) the model looks like:

	see	spot
see		|
spot		

After the full text the model looks like:

	see	spot	run	.	jump	,
see		||				
spot			|		|	||
run				||		|
.	|		|		|	
jump				||		|
,		||	|		|	

Australian National University

Generation

Use a pre-trained model to generate new text through weighted random sampling.

You will need

- your completed bigram model (i.e. your filled-out grid) from *Grid Training*
- d10 (or similar) for weighted sampling
- pen & paper for writing down the generated “output text”

Your goal

To generate new text from your bigram language model. **Stretch goal**: keep going, generating as much text as possible. Write a whole book!

Key idea

Language models generate text by predicting one word at a time based on learned patterns. Your trained model provides the “next word” options and their relative probabilities; dice rolls provide the randomness to choose one of those options (and this process can be repeated indefinitely).

Algorithm

1. **choose a starting word** — pick any word from the first column of your grid
2. **look at that word's row** to identify all possible next words and their counts
3. **roll dice weighted by the counts** (see the *Weighted Randomness* lesson)
4. **write down the chosen word** and use that as your next starting word
5. **repeat** from step 2 until you reach the desired length *or* a natural stopping point (e.g. a full stop `.`)

Example

Using the same bigram model from the example in *Grid Training*:

	`see`	`spot`	`run`	`.`	`jump`	`,`
`see`		||				
`spot`			|		|	||
`run`				||		|
`.`	|		|		|	
`jump`				||		|
`,`		||	|		|	

- choose (for example) `see` as your starting word
- `see` (row) → `spot` (column); it's the only option, so write down `spot` as next word
- `spot` → `run` (25%), `jump` (25%) or `,` (50%); roll dice to choose
- let's say dice picks `run`; write it down
- `run` → `.` (67%) or `,` (33%); roll dice to choose
- let's say dice picks `.`; write it down
- `.` → `see` (33%), `run` (33%) or `jump` (33%); roll dice to choose
- let's say dice picks `see`; write it down
- `see` → `spot`; it's the only option, so write down `spot`... and so on

After the above steps, the generated text is *"see spot run. see spot"*

Australian National University

Trigram

Extend the bigram model to consider *two* words of context instead of one, leading to better text generation.

You will need

- same as *Grid Training* lesson
- additional paper for the three-column model
- pen, paper & dice as per *Grid Generation*

Your goal

To train a trigram language model (a table this time, not a grid like your bigram model from *Grid Training*) and use it to generate text. **Stretch goal**: train on more data, or generate more text.

Key idea

More context leads to better predictions. A trigram model considers two previous words instead of one, demonstrating the trade-off between context length and data requirements that shapes all language models.

Algorithm (training)

1. **create a four-column table** (see example on right)
2. **extract all word triples**: for each (overlapping) *word 1*/*word 2*/*word 3* "triple" in your text increment the **count** column for that triple, or create a new row if it's a triple you've never seen before and set the count to 1 (note: row order doesn't matter)

Example (training)

After the first four words (`see` `spot` `run` `.`) the model is:

word 1	word 2	word 3	count
`see`	`spot`	`run`	|
`spot`	`run`	`.`	|

After the full text (`see` `spot` `run` `.` `see` `spot` `jump` `.`) the model is:

word 1	word 2	word 3	count
`see`	`spot`	`run`	|
`spot`	`run`	`.`	|
`run`	`.`	`see`	|
`.`	`see`	`spot`	|
`see`	`spot`	`jump`	|
`spot`	`jump`	`.`	|

Note: the order of the rows doesn't matter, so you can re-order to group them by **word 1** if that helps.

Algorithm (generation)

1. pick any row from your table; write down **word 1** and **word 2** as your starting words
2. find *all rows* where **word 1** and **word 2** are exact matches for your two starting words, and make note of their **count** columns
3. as per *Grid Generation* roll a d10 weighted by the counts and select the **word 3** associated with the chosen row
4. move along by *one* word (so **word 2** becomes your new **word 1** and **word 3** becomes your new **word 2**) and repeat from step 2

Example (generation)

1. from the table above, choose `see` (**word 1**) and `spot` (**word 2**) as your starting words
2. find all rows with **word 1** = `see` and **word 2** = `spot`; in this case rows 1 and 5 (both have *count* == 1)
3. roll a d10 and write down the **word 3** from the row chosen by the dice roll
4. move along by *one* word (so **word 1** is `spot` and **word 2** is either `run` or `jump` depending on your dice roll) and repeat from step 2

Australian National University

LLMs Unplugged | Cybernetic Studio

Training (bucket version)

Build a bigram language model using physical tokens and buckets to track which words follow which other words.

You will need

- some text (e.g. a few pages from a kids book) printed or handwritten on paper (big enough that you can cut it into individual words)
- several containers (buckets are great, but could also cups, bowls, envelopes, or labelled areas on a table)
- scissors, pen and sticky notes or paper for bucket labels

Your goal

To produce a collection of labelled buckets containing tokens from your text.
Stretch goal: keep training your model on more input text.

Key idea

Language models learn by counting patterns in text. “Training” means building a model that tracks which words follow other words. In this version, the “following” relationship is captured physically — each bucket contains the tokens that appeared after its label in the original text.

Algorithm

1. **prepare your tokens**:
   - print or write out your training text on paper
   - use scissors to cut the text into individual words (called "tokens"); commas & full stops into separate pieces as well (and disregard all other punctuation)... but **keep them in order**
2. **build the model** one token at a time, starting with the first:
   - if this token doesn't have a bucket yet, create one and label it with this word
   - take the *next* token from your pile and put it *into* the current token's bucket
   - now apply the same process to that next token (create its bucket if needed)
   - repeat until all tokens are in buckets

Example

Original text: *"See Spot run. See Spot jump."*

Prepared tokens: `see` `spot` `run` `.` `see` `spot` `jump` `.`

After processing the first two tokens (`see` `spot`):

Bucket	Tokens inside
`see`	`spot`

After all tokens have been processed:

Bucket	Tokens inside
`see`	`spot` `spot`
`spot`	`run` `jump`
`run`	`.`
`.`	`see`
`jump`	`.`

Notice that the "see" bucket contains two `spot` tokens because "spot" followed "see" twice in the original text. This captures the same information as a grid with tally marks, but in a physical form you can touch and manipulate.

Australian National University

LLMs Unplugged | Cybernetic Studio

Generation (bucket version)

Use your bucket-based bigram model to generate new text by picking tokens at random.

You will need

- your completed bucket model from *Bucket Training*
- pen & paper for writing down the generated “output text”

Your goal

To generate new text from your bucket language model. **Stretch goal**: keep going, generating as much text as possible. Write a whole book!

Key idea

Language models generate text by predicting one word at a time based on learned patterns. Each bucket contains all the tokens that could come next — and some tokens appear multiple times, making them more likely to be picked. Choosing randomly from a bucket and repeating word by word creates new text.

Algorithm

1. **choose a starting bucket** — pick any bucket and write down its label as the first word
2. **close your eyes and pick a random token** from inside that bucket
3. **write down the token** you picked
4. **put the token back** in the bucket (so you can use it again later)
5. **find the bucket** whose label matches the token you just picked
6. **repeat** from step 2 until you reach the desired length *or* a natural stopping point (e.g. an empty bucket)

Example

Using the same bucket model from the example in *Bucket Training*:

Bucket	Tokens inside
`see`	`spot` `spot`
`spot`	`run` `jump`
`run`	`.`
`.`	`see`
`jump`	`.`

- choose `see` as your starting bucket; write down “see”
- pick from “see” bucket: both tokens are `spot`, so we get `spot`; write it down
- pick from “spot” bucket: randomly between `run` and `jump`
- let’s say we pick `run`; write it down
- pick from “run” bucket: only `.` inside; write it down
- pick from “.” bucket: only `see` inside; write it down
- pick from “see” bucket: get `spot`; write it down
- pick from “spot” bucket: this time we pick `jump`; write it down
- pick from “jump” bucket: only `.` inside; write it down

After the above steps, the generated text is *“see spot run. see spot jump.”*

The randomness comes from physically picking tokens without looking. Buckets with more tokens of the same type are more likely to produce that token.

Australian National University

Trigram (bucket version)

Extend the bucket bigram model to consider *two* words of context instead of one, leading to better text generation.

You will need

- the same materials as *Bucket Training*
- additional small containers for two-word label buckets
- sticky notes or paper for bucket labels (you'll need to write two words on each label)

Your goal

To build a trigram language model using buckets where each bucket is labelled with *two* words instead of one. **Stretch goal**: train on more data or generate longer outputs.

Key idea

Trigrams show how more context improves prediction quality. Instead of asking "what follows this word?", we ask "what follows these *two* words?". This means more buckets to manage, but better predictions.

Algorithm (training)

1. **prepare your tokens** as per *Bucket Training* (print, cut into tokens, keep in order)
2. **build the model** using word *pairs* as bucket labels:
   - take the first *two* tokens — these form your bucket label
   - create the bucket if needed, then put the *third* token inside it
   - shift along by one (new pair = old second word + token just placed)
   - repeat until all tokens are in buckets

Example (training)

Original text: *"See Spot run. See Spot jump."*

Prepared tokens: `see` `spot` `run` `.` `see` `spot` `jump` `.`

After processing all tokens:

Bucket label	Tokens inside
`see spot`	`run` `jump`
`spot run`	`.`
`run .`	`see`
`. see`	`spot`
`spot jump`	`.`

The "see spot" bucket has two tokens because different words followed that pair. Compare to bigram where "see" would just contain `spot` `spot`.

Algorithm (generation)

1. **choose a starting bucket**; write down its two-word label
2. **close your eyes and pick a random token** from that bucket
3. **write down the token**, then put it back in the bucket
4. **find the bucket** whose label matches your last *two* words (second word of old label + the token you just picked)
5. if no bucket exists, use any bucket starting with the *first* word instead
6. **repeat** from step 2 until you reach a stopping point

Example (generation)

Using the bucket model from training:

1. start with `see spot` bucket; write "see spot"
2. pick randomly: `run` or `jump` — say we get `run`; write it
3. find bucket "spot run"; pick `.`; write it
4. find bucket "run ."; pick `see`; write it
5. find bucket ". see"; pick `spot`; write it
6. find bucket "see spot"; this time pick `jump`; write it
7. find bucket "spot jump"; pick `.`; write it

Generated text: *"see spot run. see spot jump."*

Australian National University

Pre-trained Model Generation

Use a (slightly larger) pre-trained model to generate new text through weighted random sampling.

You will need

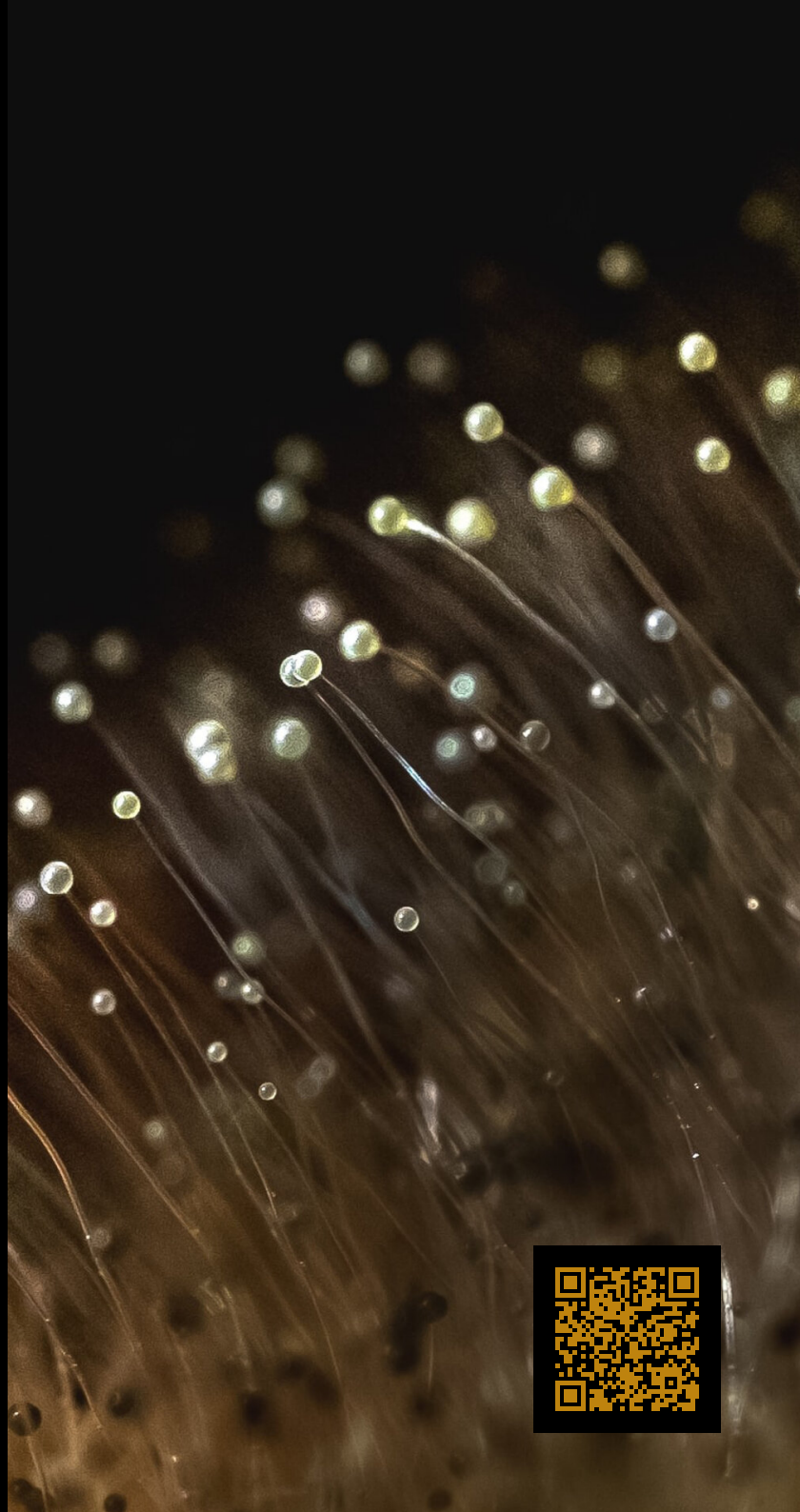
- a pre-trained model booklet
- d10 for weighted sampling
- pen & paper for writing down the generated “output text”

Your goal

To generate new text using a pre-trained language model without having to train it yourself. **Stretch goal**: without looking at the title, try and guess which text the booklet model was trained on.

Key idea

You don’t need to train your own model to use one. Pre-trained models capture patterns from large amounts of text and can be used to generate new text just like your “hand-trained” model from *Grid Training*.



Algorithm

Full instructions are at the front of the pre-trained model booklet, but here's a quick summary:

1. **choose a starting word** — pick any bold word from the booklet and write it down
2. **look up the word's entry** (i.e. use the booklet like a dictionary) to find all possible *next* words according to the model
3. **roll your d10s** (if required): check for diamonds next to the word — this shows how many d10s to roll (e.g. ◆◆◆ means roll 3 d10s). If there are no diamonds, there's only one possible next word — skip to step 5. Read the dice from left to right as a single number (e.g. rolling 2, 1 and 7 means your roll is 217)
4. **find your next word**: scan through the followers until you find the first number ≥ your roll, or just use the single word if no dice were rolled (write it down)
5. repeat from step 2 using this word as your new word, continuing this loop until you reach a natural stopping point (like a period) or reach your desired text length

Example 1: single d10

Your current word is **"cat"** and its entry shows:

cat ◆ 4|sat 7|ran **10**|slept

- one diamond (◆) means roll 1 d10
- roll your dice: roll a 6
- find the next word: first number ≥ 6 is 7|ran, so next word is "ran"
- write it down, look it up and continue the process

Example 2: multiple d10s

Your current word is **"the"** and its entry shows:

the ◆◆ 33|cat 66|dog 99|end

- two diamonds (◆◆) means roll 2 d10s
- roll your dice: roll 5 and 8 → combine them to get 58
- find the next word: first number ≥ 58 is **66**|dog, so next word is "dog"
- write it down, look it up and continue the process

Australian National University

LLMs Unplugged | Cybernetic Studio

Sampling

When generating text the language model gives several different options for which word could come next in the generated text — which one to choose?

You will need

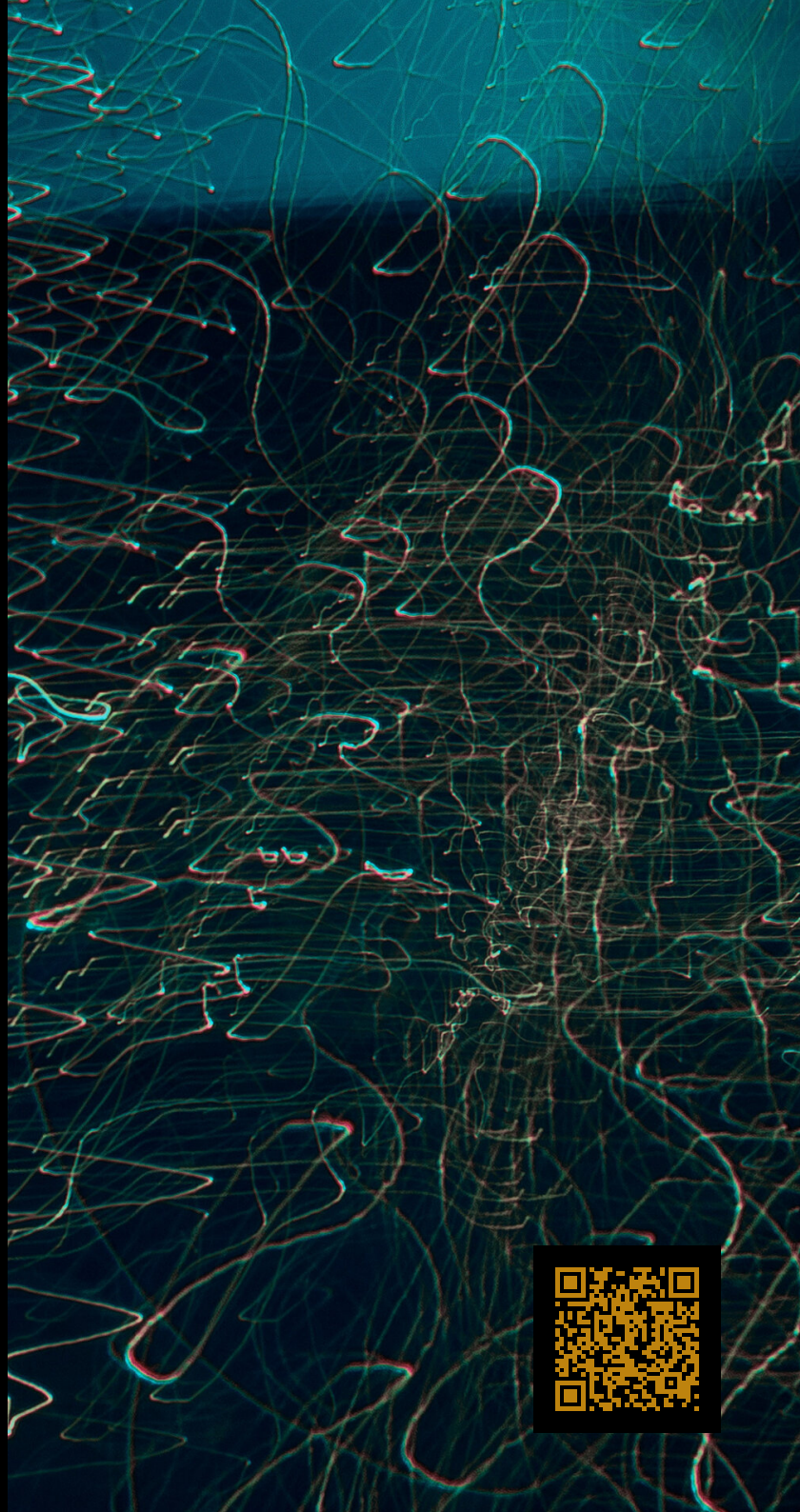
- a completed model from an earlier lesson
- pen, paper & dice as per *Grid Generation*

Your goal

To generate text (with the same model) using at least two different temperature values and at least two different truncation strategies. **Stretch goal**: design and evaluate your own truncation strategy.

Key idea

There are lots of different sampling algorithms — ways to select the next word during text generation. Each strategy has different strengths and weaknesses, and can significantly influence the generated text even if the rest of the model is identical.



Temperature control

The temperature parameter (a number) controls the randomness by adjusting the relative likelihood of probable vs improbable words. The higher the temperature, the more uniform the distribution becomes, increasing randomness and allowing more sampling from unlikely words.

Algorithm

1. when sampling the next word, divide all counts by temperature value (round down, min 1)

Example

If the counts in a given row are:

	`spot`	`run`	`jump`	`.`
`see`	4	2	1	1

1. if **temperature = 1**: use counts as-is (4, 2, 1, 1)
   - `spot` is 2x as likely as `run`, 4x as likely as `jump` or `.`
2. if **temperature = 2**: divide counts by 2 → (2, 1, 1, 1)
   - `spot` still most likely, but only 2x as likely as others
3. if **temperature = 4**: divide counts by 4 → (1, 1, 1, 1)
   - all words equally likely

Truncation strategies

Truncation narrows the viable “next word options” by ruling out some options. Any truncation strategy can be combined with temperature control.

Greedy sampling

1. find current word’s row
2. select the word with the highest count
3. if there’s a tie, roll dice to choose equally among the most likely options

Haiku sampling

1. track syllables in current line (5-7-5 pattern)
2. roll dice to select next word as normal
3. if selected word exceeds line’s syllable limit, re-roll
4. start new line when syllable count reached

Non-sequitur sampling

1. find current word’s row
2. pick the column with the lowest (non-zero) count
3. if there’s a tie, roll dice to choose equally among the least likely options

No-repeat sampling

1. track all words used in current sentence
2. roll dice to select next word as normal
3. if word already used, re-roll
4. if no valid options remain, insert `.` and continue

Alliteration sampling

1. note first letter/sound of previous word
2. if any next-word options start with same letter/sound, sample only from those alliterative options
3. otherwise use standard sampling

Australian National University

Context Columns

Enhance the bigram model with context columns that capture grammatical and semantic patterns.

You will need

- your completed bigram model from *Grid Training*
- pen, paper & dice as per *Grid Generation*

Your goal

To add new “context” columns to an existing bigram model and generate text from your newly context-aware model. **Stretch goal**: add and evaluate your own new context columns.

Key idea

The concept of attention — selectively focusing on relevant context — is a key innovation in Large Language Models. Adding context columns to your model gives it more information about which previous words matter most for prediction, leading to better generated text (with the trade-off being a slightly larger grid and more complex algorithm).

Algorithm (training)

1. **add context columns** to your existing bigram model: *after verb*, *after pronoun* and *after preposition*
2. proceed as per *Grid Training*, but each time after updating the cell count for a word pair:
   - if the first word is a verb, increment the value in the second word's *after verb* column
   - if the first word is a pronoun (I/you/they etc.), increment the value in the second word's *after pronoun* column
   - if the first word is a preposition (in/on/at/with/to etc.), increment the value in the second word's *after preposition* column

This is a little tricky to get the hang of, but the key point is that you're updating two different rows each time — once for the "word follows word" cell, and once for the "context column" cell.

Example (training)

For text *"I run fast. You run to me."* the model with context columns is:

	i	you	run	fast	to	me	.	after verb	after pronoun	after preposition
i			|							
you			|							
run				|	|			||		
fast							|			
to						|		|		
me							|			|
.	|									

Algorithm (generation)

1. **choose a starting word**
2. check its row to identify the "normal" transition counts, but *also* check if the starting word is a verb/pronoun/preposition and if so add the values from the relevant "context" column before using a d10 to choose the next word
3. **repeat** from step 2 until you reach the desired length *or* a natural stopping point (e.g. a full stop `.`)

If you like, you can add your own context columns (based on patterns which *you* think are important).

Example (generation)

Starting word: `run` (a verb):

1. check `run` row: potential next words are `fast` (1) or `to` (1)
2. check all context columns: for `to` the **after verb** column has a count of 1 (appears after verbs)
3. combine both counts: roll a dice to choose either `fast` (1) or `to` ($1 + 1 = 2$)
4. repeat from step 1 until you reach the desired length *or* a natural stopping point (e.g. a full stop `.`)

Australian National University

Word Embeddings

Transform words into numerical vectors that capture meaning, revealing the semantic relationships between words in your model.

You will need

- your completed bigram model grid (including context columns if you have them)
- another empty grid (same size as your bigram model)
- pen, paper & dice as per *Grid Generation*

Your goal

To create a similarity matrix (another square grid) which captures how similar (or different) all the words in your bigram model are. **Stretch goal**: create a visual representation of this similarity matrix.

Key idea

Each word's row in your model is its embedding under that model — a numerical fingerprint that captures meaning through context. Distances between words reveal grammatical and semantic relationships. Similar words have similar embeddings.

Algorithm

For this algorithm you'll need two grids: your original *bigram model* grid and a new *embedding distance* grid (with the same words as row/column headers, but otherwise blank to start with).

1. for the first row and second row in the bigram model, add up the total of the absolute differences between corresponding cells in the two rows and write it in the empty cell for that word pair in the embedding distance grid
2. fill out the embedding distance grid by repeating step 1 for all different pairs of rows in the bigram model grid

Example

Original text: *"See Spot. Spot runs."*

Completed bigram model grid:

	`see`	`spot`	`.`	`runs`
`see`		|		
`spot`			|	|
`.`		|		
`runs`			|	

The embedding distance between the first two rows (`see` and `spot`) is the sum of the absolute differences between corresponding elements (0 for blank cells):

$$\begin{aligned} d(\text{see}, \text{spot}) &= |0-0| + |1-0| + |0-1| + |0-1| \\ &= 0 + 1 + 1 + 1 \\ &= 3 \end{aligned}$$

Put this distance in the embedding distance grid (note diagonals are already pre-filled with 0 as well):

	`see`	`spot`	`.`	`runs`
`see`	0	**3**		
`spot`		0		
`.`			0	
`runs`				0

Complete embedding distance grid (no need to fill out the bottom triangle — the embedding distance is symmetric):

	`see`	`spot`	`.`	`runs`
`see`	0	3	0	2
`spot`		0	3	2
`.`			0	2
`runs`				0

The distances show that `see` and `.` have identical embeddings (distance = 0), while `see` and `spot` are quite different (distance = 3).

Australian National University

LLMs Unplugged | Cybernetic Studio

LoRA

Efficiently adapt a trained language model to a new domain or style without retraining the entire model from scratch.

You will need

- a completed bigram model from an earlier lesson
- pen, pencil and grid paper
- some new text in a different style or domain

Your goal

To create a lightweight “adaptation layer” that modifies your existing model’s behaviour for a new domain. **Stretch goal**: combine the base model and LoRA layer with different mixing ratios.

Key idea

Low-Rank Adaptation (LoRA) allows you to specialise a language model by adding small adjustments rather than retraining everything. The LoRA layer is typically much smaller than the base model because you only track the *changes* from the base model, not the full weights.

Algorithm

1. **choose an existing bigram model as the "base model"**
2. **train a LoRA layer**:
   - start with a new grid (same columns as the base model)
   - process your new domain-specific text using the same algorithm as *Grid Training*, but only include rows for words that appear in your new text
3. **apply the adaptation**:
   - as per *Grid Generation*, but add the counts from both grids (if current word is in the LoRA grid)
   - optionally scale the LoRA values up or down to control adaptation strength

Example

Base model trained on general text:

	saw	they	we	the	a	red
saw		2		4	2	1
they	1			2	1	
we				3		
the			1			
a						2
red		1				

LoRA layer trained on "I saw a red cat. I saw the red dog." (smaller — only 1 row):

	saw	they	we	the	a	red
saw				1	1	2

Combined model (add counts):

	saw	they	we	the	a	red
saw		2		5	3	3
they	1			2	1	
we				3		
the			1			
a						2
red		1				

- `saw` row:
  - `[−,2,−,4,2,1]` (base)
  - `[−,−,−,1,1,2]` (LoRA)
  - `[−,2,−,5,3,3]` (base + LoRA)
- `red` now equally likely as `the` after `saw`
- other rows: base + zero = unchanged
- LoRA is smaller: only 1 row vs 6 in base model

Australian National University

LLMs Unplugged | Cybernetic Studio

Synthetic Data

Use your language model to generate new training data, then train a new model on that synthetic data to see how patterns degrade or change.

You will need

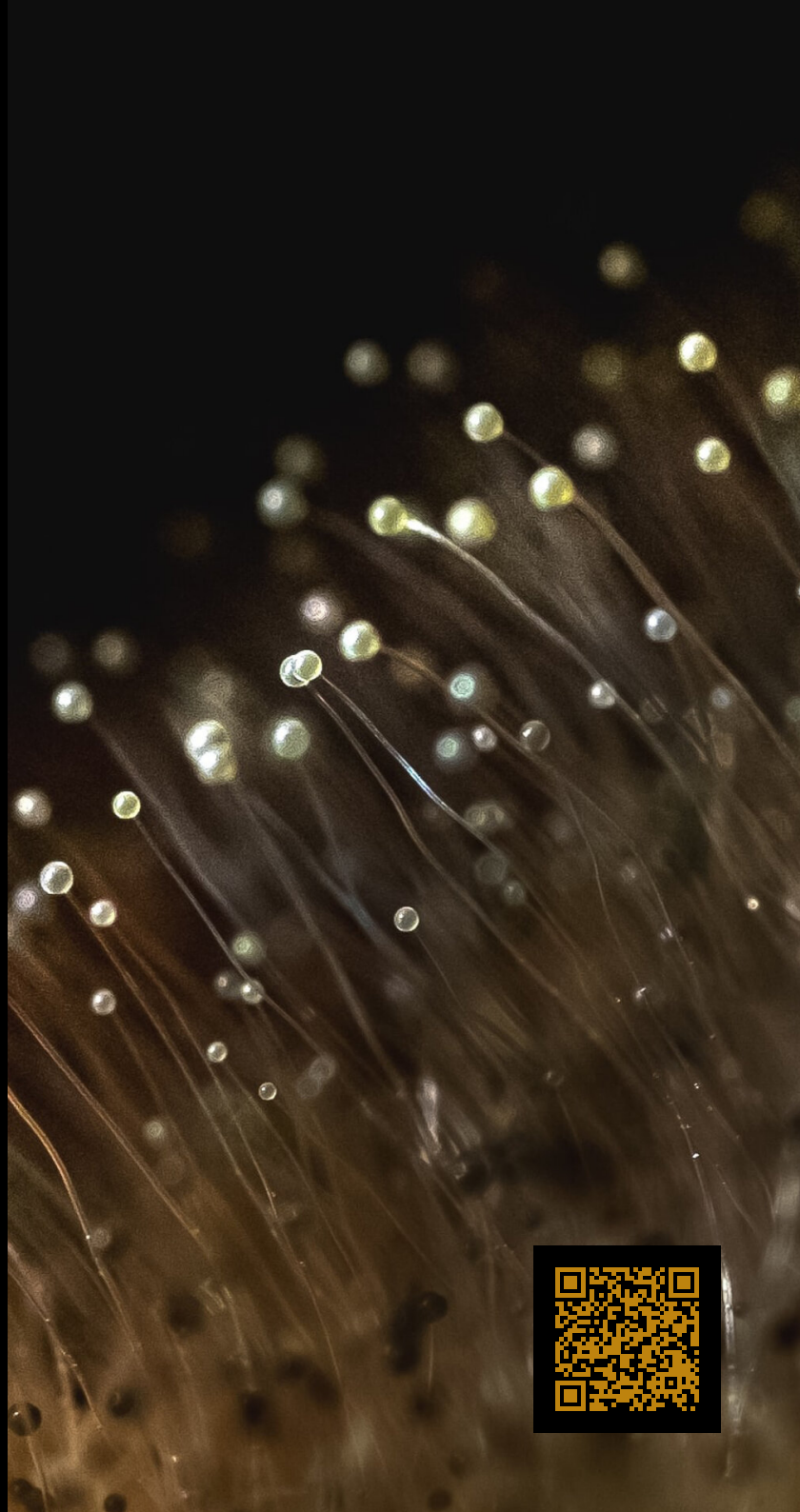
- a completed model from an earlier lesson
- pen, paper & dice for text generation
- grid paper for a new model

Your goal

To generate synthetic text using your model, then train a new “generation 2” model on that synthetic output. Compare the two models to observe what patterns are preserved or lost. **Stretch goal**: train a generation 3 model on generation 2 output. Or go “full Joker”.

Key idea

Models trained on synthetic data (output from other models) can drift from the original patterns. This demonstrates model collapse and the importance of real training data.



Algorithm

1. **generate synthetic text**:
   - use your existing model to generate text (as in Grid Generation)
   - generate enough text for meaningful training (at least 50-100 words)
   - this is your *synthetic training corpus*
2. **train generation 2 model**:
   - create a new grid following the Grid Training algorithm
   - use your synthetic text as the input corpus
   - this new model learns from AI-generated text, not human-written text
3. **compare the models**:
   - look for words that appear in the original but not in generation 2
   - compare the counts for cells that appear in both
   - generate text from both models and compare the outputs

Example

Original training text: *"See Spot run. See Spot jump."*

Generation 1 model's synthetic output: *"See run. Run spot. Spot run run."*

Notice how the synthetic text:
- uses all the same words as the original
- has different patterns (more `run` `run`, no `spot` `jump`)
- might lose some variety from the original

Generation 2 model trained on the synthetic output will amplify these changes:
- `run` `run` becomes more common
- `spot` `jump` disappears entirely
- new unlikely patterns may emerge

Joker mode

Instead of generating synthetic text from an existing model, create a completely random model:

1. draw a grid with any words you choose in the rows and columns
2. add tally marks in any cells you want, with any frequencies
3. this creates a model with no connection to real text patterns
4. generate text from this random grid using dice as normal
5. train a generation 2 model on the output from your random grid

Example

A completely random Joker grid might look like this:

	`pizza`	`robot`	`moon`	`dance`
`pizza`	3		1	2
`robot`	1	4		1
`moon`		2	1	3
`dance`	2	1	2	