Australian National University

My First Language Model

Instructor Notes

Ben Swift

Contents

Overview

These instructor notes accompany the *My First Language Model* teaching project for building language models from scratch using both manual (pen-and-paper) and automated approaches. They are designed to give students hands-on understanding of how large language models (LLMs) work by having them create working models themselves — either in 20 minutes with grid paper and dice, or through automated tools that generate dice-powered text generation booklets. This material is made available under a Creative Commons BY-NC-SA 4.0 License.

CYBERNETIC STUDIO

The primary teaching materials are the "module cards" (numbered 00-09) in a `modules.pdf` file you should have received alongside this `instructor-notes.pdf` one. Each one is a double-sided landscape PDF designed to be used as a workshop handout. The first two modules (01 and 02) are pre-requisites for the rest and should be done in order, and then the later ones can be done in (almost) any order.

Each module card includes these sections:

- **you will need**: required materials (dice, paper, grids, etc.)
- **your goal**: what students should accomplish
- **key idea**: the core concept being demonstrated
- **algorithm**: step-by-step instructions for the hands-on activity
- **example**: worked demonstration showing the algorithm in action

This document also includes extra "instructor notes" for each module:

- **discussion questions**: prompts to help students reflect on what they've learned
- **connection to current LLMs**: how the hands-on activity relates to modern AI systems

Learning outcomes

The activities in these modules demonstrate the fundamental operations of language models. The main advances in modern AI come from doing these same operations at massive scale with learned (rather than hand-crafted) patterns.

The primary learning outcome is that students will understand that language models — whether ChatGPT or their hand-built version — work the same way: they learn by counting patterns in text, then generate new text by repeatedly making random choices weighted by what they've seen before.

In addition, students will gain the following key insights as to how modern LLMs work (in reference to the activities they'll do themselves in these modules):

1. **scale is the main difference**: your small bigram/trigram models (dozens, perhaps up to hundreds of parameters) vs billions of parameters, but the core concepts are identical
2. **randomness creates variety**: both your dice and LLMs use controlled randomness to avoid repetitive output
3. **more context improves prediction**: more context (bigram → trigram → context columns) enables better text generation
4. **embeddings capture meaning**: words used similarly get similar vectors, whether hand-calculated or learned by neural networks
5. **training is just counting**: at its core, training means observing patterns in data, exactly what you did with tally marks

There's a glossary at the end of this document. Note that while the terms AI/genAI/LLMs get thrown around and used interchangeably these days, these modules are fundamentally about *language* models. While there are some similarities to the way that other generative AI models (e.g. text-to-image) work, they're beyond the scope of this particular course.

About these resources

These resources have been developed through the delivery of many workshops and courses at the ANU School of Cybernetics over the period 2023–present. It's suitable for any audience from high-schoolers up to adults, and no particular background in LLMs/AI/Machine Learning is required.

The software tools used to produce all these materials are available at on GitHub at https://github.com/ANUcybernetics/my-first-lm. This work is part of the Cybernetic Studio's *Human-Scale AI* project. If you've got questions or if you've used this material successfully in your classroom I'd love to hear about it — drop me a line at ben.swift@anu.edu.au.

Historical context: Markov and Shannon

The N-gram language models students build in this workshop have a lineage stretching back over a century. Understanding this history helps situate the activities within broader developments in probability theory, information theory, and computation.

Andrey Markov introduced the idea of "Markov Chains" in 1913 while analysing letter sequences from Pushkin's "Eugene Onegin". His work was fundamentally mathematical — he used the statistical dependencies in text to prove properties of stochastic processes. Markov counted letter transitions and calculated probabilities, but he never used them to generate synthetic text. His interest was in the mathematics of dependent random variables, not in producing artificial language.

Claude Shannon built on this foundation three decades later. Between 1948 and 1951, Shannon applied information theory to language and made several crucial contributions. First, he used N-gram models to measure entropy and redundancy in English, connecting statistical patterns to fundamental limits on compression. Second, Shannon was the first to systematically generate synthetic text at increasing N-gram orders — 0-gram (random letters), 1-gram (letter frequencies), 2-gram (letter pairs), and so on — to demonstrate how higher-order models produce increasingly realistic output. This generative approach revealed that language structure emerges from statistical dependencies at multiple scales.

Connection to current LLMs

Modern language models descend directly from Shannon's work:

- **scaling up N-grams**: current models use vastly more context (128,000+ tokens vs Shannon's letter pairs), but the core idea remains — longer context enables better prediction
- **entropy and loss**: neural networks minimise "cross-entropy loss" during training, directly applying Shannon's entropy framework
- **compression and understanding**: recent research shows that better language models are better compressors, confirming Shannon's 1948 insight about the entropy-compression connection

- **human evaluation**: Shannon's experiments prefigure modern practices of using human raters to evaluate model quality

The key difference is scale and method. Shannon counted letter transitions by hand and generated text manually. Modern models learn patterns from trillions of words using neural networks and generate text computationally. But the fundamental insight — that language structure can be captured through statistical dependencies and revealed through synthetic generation — comes directly from Shannon's work in the mid-twentieth century.

Module notes

00. Weighted randomness

Note: this is a "pre-module"; it's usually ok to start from module 01 and just have this module card handy to refer to if students want more detailed instruction about weighted random sampling.

Discussion questions

- which method feels most "random" to you, and why?
- which is fastest for getting repeated random selections?
- how would you handle weights like 17, 23, 41?
- what happens when one option has 95% probability?
- can you invent your own weighted random selection method?

Connection to current LLMs

This module introduces weighted random sampling before students encounter language models. While not specific to LLMs, this operation is fundamental to how they work:

- **generation mechanism**: every time an LLM produces a word, it's performing weighted random sampling from a probability distribution
- **probability distributions**: neural networks output probabilities for thousands of possible next tokens; these probabilities become the "weights" for sampling
- **physical intuition**: dice and tokens make the mathematics tangible — when students later learn about "sampling from a distribution," they'll have hands-on experience with what that means

The key insight: weighted randomness is a general computational technique with applications far beyond language models (simulations, games, procedural generation). In the context of language models, this same operation happens billions of times during text generation. These physical methods (dice, tokens) implement the exact same mathematical operation that occurs inside LLMs when they choose the next word.

01. Basic training

Discussion questions

- what can you tell about the input text by looking at the filled-out bigram model grid?
- how does including punctuation as “words” help with sentence structure?
- are there any other ways you could have written down this exact same model?
- how could you use this model to generate *new* text in the style of your input/training data?

Connection to current LLMs

This counting process is exactly what happens during the “training” phase of language models:

- **training data**: your paragraph vs trillions of words from the internet
- **learning/training process**: hand counting vs automated counting by computers
- **storage**: your paper model vs billions of parameters in memory

The key insight: “training” a language model means counting patterns in text. Your hand-built model contains the same type of information that current LLMs store — at a vastly smaller scale.

02. Basic inference

Discussion questions

- how does the starting word affect your generated text?
- why does the text sometimes get stuck in loops?
- if this is a *bigram* (i.e. 2-gram) model, how would a unigram (1-gram) model work?
- how could you make generation less repetitive?
- does the generated text capture the style of your training text?

Connection to current LLMs

This generation process is identical to how current LLMs produce text:

- **sequential generation**: both generate one word at a time
- **probabilistic sampling**: both use weighted random selection (exactly like your dice or tokens)
- **probability distribution**: neural network outputs probabilities for all 50,000+ possible next tokens
- **no planning**: neither looks ahead — just picks the next word
- **variability**: same prompt can produce different outputs due to randomness

The fact: sophisticated AI responses emerge from this simple process repeated thousands of times. Your paper model demonstrates that language generation is fundamentally about sampling from learnt probability distributions. The randomness

is why LLMs give different responses to the same prompt and why language models can be creative rather than repetitive. These physical sampling methods demonstrate the exact mathematical operation happening billions of times per second inside modern language models.

03. Pre-trained model inference

Discussion questions

- can you guess what text the model was trained on from the generated output?
- how does using a pre-trained model differ from training your own?
- what vocabulary size does the booklet model have compared to your hand-built model?
- why might some word combinations feel more natural than others?
- without looking at the title: can you identify the training text's genre or style?

Connection to current LLMs

This module demonstrates the foundation of how people interact with modern AI:

- **pre-training**: companies train massive models on huge text corpora (like your booklet model, but with trillions of words)
- **inference as a service**: users generate text without seeing or modifying the underlying model (just like using the booklet)
- **model distribution**: the booklet format shows how models can be packaged and shared — current LLMs are distributed as parameter files
- **deterministic inference**: given the same starting word and dice rolls, you get the same output (though randomness creates variety between runs)

The key insight: training and inference are separate processes. Most AI users never train models — they use pre-trained ones through APIs or interfaces. Your hand-built model from *Basic Training* gives you insight into what's inside the booklet, but you don't need that knowledge to generate text. This separation is why companies like OpenAI can provide AI services: they do the expensive training once, then millions of users perform inference. The booklet captures thousands of training examples in a portable format, just as neural networks compress training data into billions of parameters.

04. Trigram model

Discussion questions

- how does the trigram output compare to basic (bigram) model output?
- what happens when you encounter a word pair you've never seen before?
- how many rows would you need for a 100-word text?
- can you find word pairs that always lead to the same next word?
- what's the tradeoff between context length and data requirements?

Connection to current LLMs

The trigram model bridges the gap between simple word-pair models and modern transformers:

- **context windows**: current models use variable context up to 2 million tokens
- **sparse data problem**: with more context, you need exponentially more training data

Your trigram model shows why longer context helps — `see` + `spot` predicts `run` perfectly, while just `spot` could be followed by `run` or `,`. This is why modern LLMs can maintain coherent conversations over many exchanges — they consider much more context than just the last word or two.

05. Context columns

Discussion questions

- which context columns are most useful for your text?
- can you think of other helpful context patterns?
- how do context columns reduce repetition in generated text?
- what happens when multiple contexts apply at once?
- are grammatical contexts (verb→object, pronoun→verb) more reliable than word-specific ones (`word_a`→`word_b`)?

Connection to current LLMs

Your hand-crafted context columns are what the “attention mechanism” in transformers learns automatically:

- **manual vs learnt**: you chose 3 grammatical contexts; transformers learn hundreds of attention patterns
- **fixed vs dynamic**: your contexts are the same for all words; transformers adapt attention per word
- **the innovation**: instead of pre-defining important contexts, transformers learn which previous words to “attend to” for each prediction

This is why it’s called “attention” — the model learns to pay attention to relevant context. When a model predicts the next word after “The capital of France is”, it automatically learns to attend strongly to “capital” and “France” while ignoring less relevant words. Your grammatical context columns (verb→object, pronoun→verb) do this manually, while modern AI discovers these patterns — and many more — through learning.

06. Word embeddings

Discussion questions

- which words cluster together? why?
- do grammatically similar words have similar embeddings?

- can you predict which words will be close before calculating?
- how do context columns affect word similarity?
- what information is captured in these vectors?

Connection to current LLMs

Word embeddings revolutionised NLP by turning words into numbers that computers can process:

- **dimensions**: your (e.g.) 8-dimensional vectors → modern models use hundreds or thousands of dimensions
- **learning**: you used occurrence patterns → modern models learn from billions of contexts
- **semantic capture**: state-of-the-art embeddings encode meaning so well that "`king` - `man` + `woman` ≈ `queen`" actually works
- **foundation**: every modern language model starts by converting words to embeddings

The insight: words with similar meanings appear in similar contexts, so their usage patterns (and thus embeddings) are similar. Your hand-calculated vectors demonstrate this principle: `cat` and `dog` would have similar embeddings because they both follow `the` and precede `ran` or `sat`. This discovery enabled computers to "understand" that words have relationships and meanings beyond just their spelling.

Note on the activity: while the module focuses on calculating distances between embeddings (the similarity matrix), this is pedagogically deliberate. Embeddings themselves are just rows of numbers, but distances reveal the relationships between words — which is what makes embeddings useful in practice. The activity emphasises the practical application of embeddings rather than just their construction.

07. Sampling strategies

Discussion questions

- which strategy produces the most "human-like" text?
- when would you want predictable vs surprising output?
- how do constraints (haiku, no-repeat) spark creativity?
- can you invent your own sampling strategy?

Connection to current LLMs

Current LLMs use these same mechanisms, though the specific strategies differ:

Temperature control:

- **temperature parameter**: divides probabilities just like you divide tallies; higher temperature means more random output
- the module uses manual temperature adjustment, while LLMs do this computationally before every token

Truncation techniques in modern LLMs:

- **top-k sampling**: only consider k most likely tokens (truncates rest to zero)
- **top-p (nucleus) sampling**: consider tokens until cumulative probability reaches p (dynamic truncation)
- **repetition penalty**: discourage repeating recent tokens
- **frequency penalty**: discourage common tokens
- **presence penalty**: discourage any repetition

Truncation techniques in this module:

The module presents different strategies (greedy, haiku, non-sequitur, no-repeat, alliteration) that are designed to be performed manually with dice rather than computationally. While these aren't the exact techniques used in modern LLMs, they demonstrate the same core principle: you can dramatically change output behaviour by changing the sampling rules, not the model itself.

Your paper model demonstrates that "creativity" in AI comes from two controls: adjusting temperature (probability distribution shape) and applying truncation strategies (which tokens to exclude). The same trained model can produce scholarly essays (low temperature, strict truncation) or wild poetry (high temperature, constraint-based truncation) just by changing these parameters!

The key insight: generation control is as important as training data. Your paper model proves that creative output comes not from the model itself, but from how you control temperature and which tokens you truncate from consideration.

08. LoRA

Discussion questions

- how much training data do you need for the LoRA layer compared to training from scratch?
- what happens if you scale the LoRA values by 2 or 0.5 before adding them?
- can you create multiple LoRA layers for different domains?
- which words change most between base and adapted models?
- when would you want a separate LoRA layer vs retraining the whole model?

Connection to current LLMs

Low-Rank Adaptation revolutionised how modern LLMs are customised:

- **efficiency**: training a LoRA layer requires 100-1000x less computation than full fine-tuning
- **modularity**: you can have one base model plus many LoRA layers for different tasks (medical, legal, creative writing)
- **preservation**: the base model stays unchanged, so it retains its general capabilities
- **combination**: multiple LoRA layers can be combined or switched on-the-fly
- **distribution**: LoRA layers are small (megabytes vs gigabytes), making them easy to share

The key insight: most model adaptation happens in a small subspace of all possible changes. Instead of adjusting billions of parameters, LoRA identifies and modifies only the dimensions that matter for the new domain. Your paper implementation makes this concrete: rather than recreating the entire grid, you only track the changes needed for the new text style. When you add the base and LoRA counts together, you're doing exactly what neural networks do when they apply LoRA layers during inference. This is why organisations can maintain one large foundation model and create thousands of specialised versions through lightweight LoRA layers.

09. Synthetic data

Discussion questions

- what patterns from the original survived to generation 2?
- what new patterns emerged that weren't in the original?
- how does vocabulary shrink or change across generations?
- can you identify when loops or repetitions started?
- what would happen if you continued to generation 3, 4, 5?
- (for joker mode) can a completely random model produce anything coherent? why or why not?
- (for joker mode) does randomness compound across generations, or does some structure emerge?

Connection to current LLMs

Model collapse from synthetic data is a major concern in modern AI:

- **training data contamination**: as LLMs generate more web content, future models risk training on AI-generated text rather than human text
- **mode collapse**: models trained on synthetic data lose diversity and converge toward common patterns (like your `run run` example)
- **error amplification**: small errors in generation 1 become large errors in generation 2
- **recursive training**: some research deliberately uses synthetic data to improve models, but this requires careful curation
- **data provenance**: companies now track whether training data is human-written or AI-generated

The key insight: models trained on their own outputs (or outputs from similar models) degrade over generations. Your hand-built demonstration shows why: each generation is a lossy sample from probability distributions. Rare patterns get lost, common patterns get amplified, and statistical noise becomes signal. This is exactly what researchers observe when training neural networks on synthetic data — vocabularies shrink, creativity decreases, and outputs become more repetitive and stereotyped. Your generation 2 model demonstrates that "training data quality" isn't just about correctness — it's about maintaining the diversity and richness of patterns that make language interesting. This hands-on experience shows why AI companies are concerned about the increasing volume of AI-generated text on the internet: if future models train on today's AI outputs, we risk a cascade of model collapse.

Glossary: the language of language models

This glossary connects the physical activities you've been doing with the technical terms used in modern language models.

- **token**: a single unit of text — in our activities, each word or punctuation mark (`.`, `,`) is a token (note: modern LLMs use subword tokens that can be parts of words)
- **matrix**: a grid or table showing relationships between tokens — your hand-drawn grids are matrices tracking which words follow other words
- **training**: the process of counting patterns in text to build your grid — when you tallied word transitions, you were "training" your model
- **inference**: using your trained model to generate new text — rolling dice to select the next word is inference
- **vocabulary**: all unique tokens your model knows — the words across the top and side of your grid form your vocabulary
- **parameters**: the numbers stored in the model — each tally mark in your grid is a parameter. Large models have hundreds of billions of parameters.
- **fine-tuning**: additional training on specific text, e.g. adding more tallies to your grid from a new text source
- **prompt**: the starting text you give the model; your initial "starting work" word when generating text in the inference phase
- **tokenisation**: breaking text into tokens — when you separated "See Spot run." into `see` `spot` `run` `.`, you were tokenising
- **bigram model**: a model that predicts the next word based on *one* previous word
- **trigram model**: a model using *two* previous words for prediction
- **context window**: how many previous tokens the model considers (bigrams have a context window of 1, trigrams have 2, modern models have 128,000+)
- **embeddings**: numerical representations of words — each row in your grid is that word's embedding vector
- **weighted random sampling**: choosing the next token with probability proportional to its frequency. Your dice rolls implement this
- **temperature**: a parameter controlling randomness. Dividing tallies by temperature (module 06) makes generation more (high temp) or less (low temp) random