

Train a Tiny Language Model

Gavin · AI-assisted course design

Contents

1	Goal	3
2	Part 1 — Predict before running	3
3	Part 2 — Construct the entire original corpus	3
3.1	Audit what the split actually holds out	4
4	Part 3 — Define the model precisely	5
4.1	Implementation scaffold	5
4.2	Run the completed implementation after your predictions	6
5	Part 4 — Check the mathematics before a training run	7
5.1	Uniform model	7
5.2	One numerical gradient and update	7
5.3	Label and boundary checks	7
6	Part 5 — Establish the baselines	7
7	Part 6 — Run two controlled training configurations	8
8	Part 7 — Preserve and reload the experiment	9
9	Part 8 — Inspect predictions and samples	10
10	Analytical checks and interpretation	11
11	Submit an explanation	11
12	More Learning	12

1 Goal

Train a language model whose complete learned state fits in a small table. Measure its loss, save and reload checkpoints, and explain both a successful prediction and a failure caused by the model's limited context.

The model is a **bigram softmax model**: it predicts the next token using only the current token. Its 196 trainable logits make the training mechanism visible. It has no attention, hidden layers, learned position representation, or multi-token context. This is an experiment in learning next-token probabilities, not a miniature recipe for training a capable assistant.

Prerequisites: Lesson 3.1, particularly shifted labels, cross-entropy, gradients, and held-out evaluation. Lesson 1.3 supplies additional chain-rule background.

Time: about 90–150 minutes, including predictions, implementation, checks, and interpretation. This is a learning-time estimate, not a measured program runtime.

Requirements: an existing Python 3 installation, ordinary CPU execution, and Python's standard library. No account, API, GPU, package installation, model download, external corpus, or paid compute is required. Keep the data, parameters, and reports in your local working directory. Do not replace this original corpus with private messages or downloaded text.

Expected artifact: a small program plus separate corpus, training, checkpoint, evaluation, and interpretation files. The scaffold below specifies what to implement; the numerical checks are analytical expectations, not recorded training outcomes. A completed submission must contain actual output from your program.

2 Part 1 — Predict before running

Write your answers in `submission.md` before coding or asking an assistant for the solution:

1. All 14 output logits initially equal zero. What probability does the model assign to each token? What are its mean loss and perplexity?
2. A full training record ends in the same color with which it begins. Can a model that sees only the immediately preceding action token reliably reproduce that earlier color?
3. Would you expect training to improve loss relative to a uniform distribution? Must every minibatch update improve loss on every record?
4. If full records differ between splits but their normalized bigram counts match exactly, should a bigram model's train and validation losses differ?
5. What must a checkpoint preserve besides the numeric table to give the same token predictions after reloading?

Preserve these predictions. Add corrections later without rewriting the original attempt.

3 Part 2 — Construct the entire original corpus

Use this fixed vocabulary order. The index in the list is the token ID, starting at zero:

```
<bos>, <eos>, red, blue, green, gold,  
orb, cube, cone, ring,  
glows, spins, rests, rolls
```

These are deliberately specified symbols in a toy language. They are not a vocabulary learned from evaluation text. No unknown-token behavior is needed because all allowed symbols are known in advance.

The generator has three choices:

- Colors, indexed $c = 0..3$: red, blue, green, gold
- Shapes, indexed $s = 0..3$: orb, cube, cone, ring
- Actions, indexed $a = 0..3$: glows, spins, rests, rolls

For every triple (c, s, a) , generate exactly one six-token record:

```
<bos> COLOR[c] SHAPE[s] ACTION[a] COLOR[c] <eos>
```

The repeated color is a deliberately defined consistency rule. These are synthetic records, not claims about the natural world or ordinary English grammar. There are $4^3 = 64$ unique records. Generate them in nested index order: color, then shape, then action.

Assign whole records before extracting token pairs. Compute $r = (c + s + a) \% 4$ and assign:

- $r = 0$: test
- $r = 1$: validation
- $r = 2$ or $r = 3$: training

Write the three partitions as separate JSON Lines files. Each record should include its index triple, stable ID such as $c0-s0-a2$, and ordered token IDs. Also write the vocabulary, generator version, assignment rule, record counts, and SHA-256 fingerprints of the exact saved files to a manifest.

3.1 Audit what the split actually holds out

Your audit must establish these facts:

- There are 32 training, 16 validation, and 16 test records.
- Every record has six tokens, starts with `<bos>`, and ends with `<eos>`.
- Every record's final color matches its first color.
- No full token sequence or record ID occurs in more than one split.
- All 12 ordinary vocabulary entries occur in the training records.
- Each record contributes five adjacent input-target pairs, giving 160, 80, and 80 target positions respectively.
- For every ordered token pair, its training count is twice its validation count and twice its test count.

The last condition is intentional. Full sequences are genuinely held out, but every split has the same normalized adjacent-token statistics. The splits share the generator and all its structural rules. Describe this as **held-out combinations from a shared generator**. Do not call it a test of unfamiliar language, independent real-world data, or unseen grammar.

Auditing split structure may read all three files. Parameter fitting and baseline estimation may read only training data. Validation may diagnose implementation and optimization. Do not inspect test performance to select a learning rate, epoch count, or checkpoint. Keep the audit and the performance evaluation as separate operations.

4 Part 3 — Define the model precisely

Store a 14×14 real-valued table W . Row i contains the next-token logits after current token i :

$$p(j \mid i) = \frac{\exp(W_{ij})}{\sum_{k=0}^{13} \exp(W_{ik})}.$$

Initialize every entry to zero. The model contains 196 stored trainable scalars, although additive shifts of a whole row do not change that row's probabilities. The `<eos>` row receives no training inputs because we stop each record at its end marker. It remains unchanged under the specified updates.

This table can be written as a single linear transformation of a one-hot current-token vector. With the column-vector convention of Lesson 1.3, $z = W^T e_i$. There are no extra hidden features or biases. A direct row lookup produces the same logits more simply.

For input–target pair (i, y) , calculate

$$\ell(i, y) = \log \sum_k \exp(W_{ik}) - W_{iy}.$$

Use a stable computation: let $m = \max_k W_{ik}$, then evaluate $m + \log \sum_k \exp(W_{ik} - m) - W_{iy}$. Compute softmax with the same subtraction. Do not repair overflow by silently clamping probabilities or changing the objective.

For a batch of M pairs, the gradient is

$$G_{ij} = \frac{1}{M} \sum_{(x,y) \text{ in batch}} \mathbf{1}[x = i] (p(j \mid i) - \mathbf{1}[j = y]).$$

Subtract ηG from W . Compute every contribution using the same pre-update table, then update once. Changing a row immediately after each pair would implement sequential single-example updates rather than the specified minibatch update.

4.1 Implementation scaffold

Build a local `tiny_lm.py` program with these logically separate operations:

```
prepare(output_directory)
    create the fixed vocabulary and all 64 records
    assign complete records to the three partitions
    write split files and a reproducible manifest

audit(data_directory)
    verify identities, boundaries, coverage, counts, and bigram balance
    write structural findings without training a model
```

```

train(training_file, validation_file, config, output_directory)
    initialize W to zero or load an authorized training checkpoint
    extract pairs within individual records only
    compute training-only baselines
    record epoch-zero train and validation metrics
    for each epoch:
        shuffle a fresh copy of the canonical training-pair list
        for each minibatch:
            compute stable loss and probabilities
            accumulate the gradient and divide by actual batch size
            apply one SGD update
        evaluate the current snapshot on train and validation
        save metrics and the scheduled checkpoints
    never read the test file

evaluate(checkpoint, evaluation_file, output_directory)
    load frozen W, vocabulary, and baseline probabilities
    compute total negative log-likelihood and included-token count
    write metrics without updating parameters or baselines

sample(checkpoint, sampling_config, output_directory)
    generate from frozen W with an independent sampling seed
    save raw token IDs and displayed tokens

```

Use ordinary readable functions for stable softmax, negative log-likelihood, pair extraction, minibatch gradients, and SGD. The scaffold is a specification rather than a complete downloadable implementation. If you use an assistant to help implement it, check these operations yourself before trusting the output.

4.2 Run the completed implementation after your predictions

Download the standard-library reference program into a local working directory. Read its pair extraction, stable loss, minibatch gradient, and checkpoint code against the specification above before using it. The program runs the fixed protocol without downloading data or packages.

```

python3 tiny_lm.py selftest --output selftest
python3 tiny_lm.py experiment --output experiment

```

Use a fresh output directory each time; existing nonempty output is rejected. The experiment operation constructs and audits the corpus, runs both declared configurations, freezes both final checkpoints before test scoring, verifies both reload/resume paths, and saves all unfiltered samples and a four-curve SVG. Its generated files are your actual local results, separate from the analytical expectations in this Lab. Record your Python version, commands, program SHA-256, and elapsed time.

The individual operations are also available through `python3 tiny_lm.py --help` and each subcommand's `--help`. For example, to inspect a frozen checkpoint without changing it:

```
python3 tiny_lm.py evaluate --checkpoint experiment/lr-1.0/checkpoints/
epoch-300.json --data experiment/data/test.jsonl --output evaluation
python3 tiny_lm.py sample --checkpoint experiment/lr-1.0/checkpoints/
epoch-300.json --output samples
```

5 Part 4 — Check the mathematics before a training run

Implement a self-test operation that does not fit the corpus.

5.1 Uniform model

With zero logits, every vocabulary item has probability $1/14$. Every included target therefore has loss

$$\ln 14 \approx 2.639057330,$$

and perplexity 14. These checks include `<eos>` as a target and retain all 14 possible outputs, including `<bos>`. Do not mask “invalid” outputs during evaluation; doing so changes the model.

5.2 One numerical gradient and update

Separately use a three-entry row $z = [0, 0, 0]$ with target index 1. Check the analytical gradient $[1/3, -2/3, 1/3]$.

For each coordinate, compare it with the centered finite difference

$$\frac{\ell(z + \epsilon e_j) - \ell(z - \epsilon e_j)}{2\epsilon}, \quad \epsilon = 10^{-5}.$$

An absolute-error tolerance of 10^{-6} is a reasonable check for this small double-precision calculation. Report the observed errors. Then update all three logits with $\eta = 0.3$ and verify $[-0.1, 0.2, -0.1]$, target probability approximately 0.402959911, and loss approximately 0.908918198.

Also check a batch containing repeated input IDs. Its minibatch gradient should equal the arithmetic mean of the individually computed pre-update gradients. This catches overwrite errors when several examples contribute to one table row.

5.3 Label and boundary checks

For `<bos> red orb glows red <eos>`, pair extraction must return exactly five pairs. In particular, there must be no `<eos>` to `<bos>` training pair connecting different records. Confirm that evaluation leaves a serialized copy or fingerprint of W unchanged.

Only begin the experiment after these checks pass. Record failures and fixes; do not replace failed outputs with expected values.

6 Part 5 — Establish the baselines

Use two baselines on exactly the same target positions as the model:

1. **Uniform:** every output has probability $1/14$.
2. **Training unigram:** ignore the input token and predict from training target counts, with add-one smoothing:

$$p_{\text{uni}}(j) = \frac{n_j + 1}{160 + 14}.$$

Count targets, not all tokens in saved records. The training targets contain 32 <eos> tokens, 16 of each color, eight of each shape, eight of each action, and zero <bos> tokens. The smoothing gives <bos> a nonzero probability and keeps every logarithm finite.

Freeze and save these baseline probabilities before evaluating validation or test data. The expected unigram loss on any partition here is

$$-0.4 \ln(17/174) - 0.4 \ln(9/174) - 0.2 \ln(33/174) \approx 2.447578618.$$

This number is an analytical check on corpus accounting, not a trained-model result. A context-sensitive model should improve over the unigram baseline, but a small gain alone does not establish the value of conditioning. Add-one smoothing itself raises the baseline loss slightly: the optimal context-free unigram distribution for these counts has loss approximately 2.441214529 nats per target. Compare against that analytical value too before attributing a gain to previous-token context.

7 Part 6 — Run two controlled training configurations

Before inspecting results, fix these settings for the main run:

- Initialization: all-zero logits
- Optimizer: SGD, with no momentum, weight decay, or gradient clipping
- Learning rate: 1.0
- Epochs: 300
- Minibatch size: 20 adjacent pairs
- Training-order seed: 17
- Numerical representation: ordinary Python floating-point numbers
- Checkpoint epochs: 0, 1, 5, 20, 100, 300

At the beginning of epoch e , numbered from 1, copy the canonical training-pair list and shuffle it using a fresh `random.Random(17 + e)` instance. This gives a reproducible per-epoch order without storing a mutable random-generator state. Never shuffle the already shuffled list from the preceding epoch.

There are eight full minibatches per epoch and 2,400 updates in 300 epochs. The program should still divide by the actual minibatch size if the implementation later encounters a shorter last batch. There are 48,000 training target presentations, although only 160 target positions exist in the saved training corpus.

Run a second configuration with learning rate 0.1, changing nothing else. Write a prediction about which run will reduce loss more quickly before executing either. Give the runs different output directories. The purpose is to inspect update scale under a controlled comparison, not to declare a universally good learning rate.

After every epoch, record full-corpus training and validation measurements at the same frozen parameter snapshot. Save total negative log-likelihood, target count, mean nats per token, bits per token, perplexity, epoch, update count, elapsed time, and learning rate. A mean over successive minibatch

losses during the epoch describes several different parameter states, so label it separately if you also record it.

Plot training and validation loss against updates. A small locally generated SVG is sufficient and can be written using the standard library; a dependency-free text plot plus the complete metric file is also acceptable. Show each learning-rate run clearly. Do not hide one curve because it overlaps another.

Complete both predefined runs, freeze their final epoch-300 checkpoints, and then evaluate their test losses. If a bug invalidates a run, document it and rerun the declared protocol. Do not adjust hyperparameters in response to the test scores and continue calling the same data an untouched test set.

8 Part 7 — Preserve and reload the experiment

Each checkpoint should contain JSON-compatible data for:

- the full W table and vocabulary order;
- model type and format version;
- epoch and optimizer-step count;
- training configuration and the per-epoch shuffle rule;
- training data and vocabulary fingerprints;
- frozen baseline distributions.

This SGD setup has no momentum buffers or adaptive moments. Because checkpoints occur at epoch boundaries and the next shuffle is determined by epoch number, resuming needs no hidden partially consumed minibatch. More elaborate optimizers and checkpoint positions would require additional state.

Save actual learned parameters, not just loss curves or generated text. For each learning-rate run, load the epoch-100 checkpoint and verify its probabilities and evaluation metrics against those obtained immediately before saving. Resume a copy from epoch 100 through epoch 300 and compare its final table with the uninterrupted run using the same implementation and configuration. Report equality or the maximum absolute difference, plus any implementation-dependent explanation.

Use safe, explicit JSON loading for your own files rather than executing arbitrary serialized objects. Keep all output local. The original synthetic records are designed for this exercise, but your submission may still contain private notes; no uploading or public sharing is required.

A useful directory layout is:

```
my-tiny-lm/  
  tiny_lm.py  
  submission.md  
  data/  
    vocabulary.json  
    manifest.json  
    train.jsonl  
    validation.jsonl  
    test.jsonl  
  audit/
```

```
split-audit.json
self-tests.json
runs/
  lr-1.0/
    config.json
    metrics.jsonl
    checkpoints/
    samples.jsonl
  lr-0.1/
    config.json
    metrics.jsonl
    checkpoints/
    samples.jsonl
evaluation/
  final-test-metrics.json
  reload-check.json
  resume-check.json
  loss-curves.svg
```

Record the exact commands, Python version, program fingerprint, and actual wall-clock time in the submission. Determinism is a property to check under the recorded environment, not a promise that every platform produces identical floating-point bytes.

9 Part 8 — Inspect predictions and samples

At each saved checkpoint, inspect the next-token probabilities after <bos>, red, orb, and glows. Preserve the values rather than only reporting the highest-probability token.

Then generate eight samples per checkpoint. Start each at <bos>, use temperature 1 with no top-k or other filtering, and sample from all 14 output IDs using categorical probabilities. Stop at the first generated <eos> or after 20 generated tokens. Keep a flag for samples truncated at the limit.

For sample index $s = 0..7$, create a fresh sampling generator with seed $23 + s$, separately for each checkpoint. Sampling must not consume the training-order generator. Save all samples, including malformed records and repeated markers. Do not retry until a sample looks good.

Check whether each sample has the required six-token structure and whether its two colors match. These checks describe generated behavior; they are distinct from next-token loss under observed prefixes.

Finally, compare the distributions after these two supplied prefixes:

```
<bos> red orb glows
<bos> blue orb glows
```

The full prefixes differ, but both end in glows. Your bigram model must return identical distributions. A program returning different ones has introduced additional context or has a bug. Explain why neither more epochs nor a different learning rate can remove this architectural constraint.

10 Analytical checks and interpretation

i Open after making your predictions

Why the loss curves coincide within a run: each split's normalized pair counts are identical. For any fixed W , averaging $-\ln p(y \mid x)$ with those same pair frequencies gives the same result. Training, validation, and test loss therefore match up to floating-point summation differences. Check train versus validation each epoch; test need only be scored at the end. This is an accounting invariant, not evidence of broad generalization or an effective overfitting detector.

The best bigram probabilities on this corpus: after `<bos>`, the four colors each have probability $1/4$. After a color, `<eos>` has probability $1/2$, and each shape has probability $1/8$. After a shape, each action has probability $1/4$. After an action, each color has probability $1/4$. Other outputs have zero probability at the ideal distribution.

The color row mixes two roles because the model cannot tell the first color position from the last. It can therefore end a record prematurely or continue when it should finish.

For each valid six-token record, the ideal bigram negative log-likelihood is

$$\ln 4 + \ln 8 + \ln 4 + \ln 4 + \ln 2 = 10 \ln 2.$$

Dividing by five targets gives $2 \ln 2 = \ln 4 \approx 1.386294361$ nats per token, or perplexity four. This is the infimum for the unregularized softmax-table model: finite logits cannot assign exactly zero probability to the other outputs. A finite training run need not reach the limit.

A measured full-corpus loss materially below this bound is a reason to inspect the implementation and objective. Possible causes include target leakage, extra context, excluded target positions, a changed vocabulary mask, or an incorrect denominator. It is not a breakthrough in optimization.

The optimal action row cannot copy an earlier color: it assigns $1/4$ to each color regardless of the prefix. Lower next-token loss and this structural failure can coexist. A context-sensitive architecture could represent additional dependencies, but its capabilities would still require separate evaluation.

11 Submit an explanation

Include the code and generated artifacts, plus a concise written interpretation containing:

1. Your original predictions and any revisions
2. The actual split audit and mathematical self-test outputs
3. Both run configurations, metric curves, and final test measurements
4. Baseline comparisons with consistent units and target counts
5. Checkpoint reload and resume evidence
6. Unfiltered sample outputs and the two-prefix comparison
7. An explanation of where learning occurred and which data never entered an update
8. The strongest conclusion supported by the experiment and one tempting conclusion it does not support

A strong submission does not require attractive samples or a particular final trained loss. It requires a correct, reproducible experiment and an honest explanation of its evidence. Diagnose unexpectedly weak optimization using the training and validation artifacts before making larger claims.

12 More Learning

- Lesson 3.1 connects the visible training loop to larger language models.
- PyTorch CrossEntropyLoss documentation gives a framework reference for logits and loss reductions; no framework is required here.
- Lee et al., Deduplicating Training Data Makes Language Models Better explores why removing overlap matters in real datasets, where the split structure is less transparent than this generator.