

Generation: sampling text from your model

Why this matters

Training gave you a machine that scores possible next tokens. It did not give you a finished paragraph. Something still has to choose a token, attach it to the prompt, ask the model again, and decide when to stop. That outer loop is generation. This chapter builds it with the real character GPT already stored in the course, without changing its learned weights.

The distinction matters when debugging an AI application. Two runs can use identical weights and still produce different answers because their decoding policies or random draws differ. Conversely, a run that repeats the same bad answer may be perfectly deterministic. Repeatability is useful engineering evidence, but it is not evidence of correctness. You need to inspect both the predictor and the policy that converts its scores into choices.

The previous chapter introduced the training loop: forward pass, loss, backward pass, parameter update. This chapter removes the loss and update. We run the forward pass to obtain scores, choose an output, and feed the extended sequence back in. The transformer architecture from chapter seven stays intact. We do not retrain a bigram and call that a transformer, and we do not pretend that a Shakespeare continuation model is an instruction-following assistant.

For agent engineering, this separation is especially useful. A model may generate a plausible tool request, but its sampling settings do not validate arguments or authorize execution. Even a deterministic request needs ordinary program checks. Our lab is about the generation mechanism, not permission enforcement or offensive workflows. The practical lesson is to keep the choice of text separate from the application's decision to act on it.

The idea in plain words

Imagine a machine offers four possible next tokens, A, B, C and D. It gives them weights of 4, 2, 1 and 1. The total is 8, so the corresponding chances are 0.5, 0.25, 0.125 and 0.125. You can always take A, or you can draw a token with those chances. The second method sometimes chooses a less likely option, but it still favors A. It is weighted selection, not choosing uniformly from the vocabulary.

After choosing, attach the token to the prompt. The model reads the new context and supplies new scores. Choosing B can change the next distribution because the model is now continuing a different sequence. That feedback is the heart of autoregressive generation: the sequence being produced becomes the input used to produce its continuation.

Temperature reshapes the chances before a draw. For the lab's distribution, temperature 0.5 raises A's chance to 0.727273; temperature 2 makes it 0.369398. A remains the favorite. The probabilities change without any training. Top-k goes further by restricting which candidates may be chosen. With k equal to 2, only A and B remain, with probabilities 0.666667 and 0.333333 after normalization.

Do not translate those controls into claims like "cold means true" or "hot means creative." They describe a probability distribution. They do not measure truth, usefulness, or originality. We will inspect one actual sample from each policy, while keeping the evidence narrower than those tempting labels.

How it actually works

Start with one row of scores

The GPT returns a tensor shaped batch by time by vocabulary. Every time position has a score for every candidate next token. During training, many positions contribute predictions for their known next-token targets. During ordinary uncached generation, we need the prediction after the last input token, so we select `logits[:, -1, :]`.

The lab prompt is `ROME0:\n`. With this character tokenizer it contains 7 tokens. The GPT returns shape `(1, 7, 65)`, and selecting the final position gives `(1, 65)`. The first dimension is the number of sequences, the second in the full tensor is the input length, and the final dimension is the vocabulary size. Selecting every position and drawing from every row would not produce the same continuation loop. Earlier rows predict after earlier prefixes, not after the full current prompt.

The scores are called logits. They can be negative, positive, or zero; they are not probabilities and need not sum to anything special. For the hand calculation we deliberately choose logits that are natural logarithms of familiar weights: `log(4)`, `log(2)`, `log(1)`, `log(1)`. The lab prints them as 1.386294, 0.693147, 0 and 0. Taking their exponentials recovers the original weights.

Turn logits into probabilities

For candidate i , write:

$$p_i = \exp(z_i / T) / \sum_j \exp(z_j / T), \text{ for } T > 0$$

Here z_i is the logit for token i , T is the positive temperature, and $\exp$ is the natural exponential. The index j ranges over all vocabulary candidates still allowed. The denominator adds their positive weights. Dividing by that sum ensures the resulting probabilities add to one. This normalization is softmax.

At T equal to 1, the hand example simply divides each weight by 8. At T equal to 0.5, dividing a log weight by one half doubles it. Exponentiating a doubled logarithm squares the original weight. We therefore get weights 16, 4, 1 and 1, whose sum is 22. Dividing each by 22 produces 0.727273, 0.181818, 0.045455 and 0.045455, rounded as in the output.

At T equal to 2, the weights become square roots: 2, 1.414214, 1 and 1. Their sum is 5.414214. The probabilities become 0.369398, 0.261204, 0.184699 and 0.184699. Notice the shape rather than memorizing the decimals. Lower positive temperature amplifies relative score differences. Higher temperature reduces their effect. It does not reverse which finite logit is largest.

A more compact way to see that is through odds. The probability ratio for candidates i and j is $\exp((z_i - z_j) / T)$. The shared denominator cancels. A positive score gap becomes a more extreme odds ratio when divided by a smaller positive temperature. Equal scores stay tied. This explains why lowering temperature does not magically resolve a model's uncertainty when its largest scores are identical.

The code subtracts each row's maximum score before dividing by temperature. Adding or subtracting the same constant from every logit leaves softmax unchanged mathematically because it contributes the same multiplicative factor to every exponential. Subtracting the maximum keeps the largest exponent at zero and avoids overflowing it. The helper uses float64 for an inspectable CPU

demonstration, not as a serving-speed recommendation. The tests check a large common offset and extreme scores.

Measure spread, not intelligence

The lab also prints Shannon entropy in bits:

$$H(p) = -\sum_i p_i * \log_2(p_i)$$

H is a measure of distributional spread. The logarithm uses base two, hence bits. Each candidate contributes its probability times its surprise, with a minus sign making the total nonnegative. A candidate with probability zero contributes zero by continuity; the code avoids taking its logarithm directly.

The example's entropy is 1.186704 bits at temperature 0.5, 1.750000 at temperature 1, and 1.936752 at temperature 2. Those measurements support the claim that this distribution flattens as temperature rises. Entropy is not a creativity score. A distribution can spread probability across many useless tokens, or concentrate on a wrong token. Also, these are next-token entropies for one fixed logit row, not a measurement of whole generated paragraphs after their contexts diverge.

Greedy decoding is an explicit branch

Greedy decoding chooses the largest logit rather than drawing. We implement `temperature=0` as an explicit request for greedy decoding. We never divide by zero. This is a convention of our helper, not a promise that every API accepts that parameter value or interprets it identically.

For tied largest logits, our rule chooses the lower vocabulary ID. A positive temperature approaching zero still distributes probability among exact maximum ties in exact arithmetic; it is not necessarily the same as that deterministic tie-breaking rule. Explicit greedy behavior is easier to reason about than silently clamping an invalid temperature to a tiny number. Negative or nonfinite temperatures are rejected instead of being repaired silently.

Greedy also does not generally maximize the probability of an entire sequence. The lab contains a constructed tree, clearly separate from the trained GPT. The first choice has probabilities A equal to 0.600 and B equal to 0.400. The best continuation after A has probability 0.510; the best after B has probability 0.990. Multiplying the conditional probabilities gives a greedy path probability of 0.306 and an alternative of 0.396. Taking the best local branch missed the better complete path.

Sequence probability is a product of conditional probabilities, each conditioned on the preceding choices. This is why a locally attractive next token need not lead to the best complete answer. Beam search keeps multiple partial paths, but that is a different algorithm and not implemented here. Even the most probable complete sequence need not be the most useful human answer. Holtzman and colleagues investigate the mismatch between likelihood-oriented decoding and good open-ended text.

Top-k restricts the candidate count

Top-k sorts the scores and keeps only the k highest-ranked candidates. We mask all other logits with negative infinity before softmax. Their exponential becomes zero. The remaining tokens are renormalized, so probabilities still sum to one.

In our four-token example, k equal to 2 removes C and D. A and B retain weights 4 and 2, giving probabilities 0.666667 and 0.333333. A is favored, but B can still be drawn. Top- k does not mean selecting the k th token, averaging candidates, or drawing k output tokens. You still draw one token for the next position.

Tie handling is an implementation choice worth making visible. A cutoff implementation that retains every score at least as large as the k th score can retain more than k candidates when scores tie. Our helper instead uses a stable descending sort and keeps exactly k indices, capped at vocabulary size. Equal scores retain their original lower-ID ordering. The existing `GPT.generate` method uses a cutoff, so we add a separate helper instead of quietly changing shared behavior from earlier chapters.

With k equal to 1, only the selected maximum can be drawn. Under our tie rule, it matches greedy decoding, which the lab verifies using the actual model. Raising temperature cannot restore a candidate that has already been removed. With k larger than the vocabulary, this implementation keeps all available candidates. Invalid k values are rejected, and `None` means no top- k restriction.

Top-p adapts the candidate count

A fixed candidate count ignores whether the model's distribution is sharply peaked or broad. Nucleus sampling, also called top- p , instead keeps the smallest highest-probability prefix whose cumulative mass reaches a chosen threshold. Our top- p threshold must be greater than zero and at most one.

For probabilities 0.5, 0.25, 0.125 and 0.125, the cumulative masses are 0.5, 0.75, 0.875 and 1. At top- p equal to 0.7, A alone is insufficient, but A plus B reaches the threshold. Keep both, including B, the token that crosses the threshold. Then renormalize the retained mass to obtain 0.666667 and 0.333333.

An easy bug is to discard every token whose cumulative mass exceeds the threshold. That discards the crossing token and can even leave an empty candidate set. Our code checks the cumulative mass before each candidate. If that earlier mass is still below the threshold, the candidate is kept. This both includes the crossing token and stops immediately after an exactly reached boundary, subject to floating-point arithmetic.

For a flat distribution over the same four tokens, top- p equal to 0.7 keeps three candidates rather than two. The lab prints the surviving probabilities as 0.333333 each. That is the adaptive part: the threshold is fixed, while the candidate count changes with the distribution. Setting top- p equal to 1 disables this filter in our code.

Filter order is part of the policy

Our policy is positive temperature, then exact top- k , then top- p on the renormalized remaining distribution. This order is documented and tested. Do not assume a collection of knob values fully specifies behavior without knowing how they are composed.

The lab gives a second constructed distribution: 0.600000, 0.250000 and 0.150000. Top- p equal to 0.7 alone retains the first two candidates. But top- k equal to 2 first removes the last candidate and renormalizes the others. The first candidate now has probability 0.705882, already above the nucleus threshold. Applying top- p afterwards retains only it. This is a concrete example of combined filters being stricter than either label might suggest.

Draw, append, repeat, stop

`torch.multinomial(probs, 1, generator=rng)` returns the chosen vocabulary index for each row. Despite the function's name, our single draw is categorical selection. Its inputs must be nonnegative finite weights with a nonzero row sum. They do not strictly have to sum to one, but our helper normalizes them to make inspection straightforward.

The lab makes 20,000 independent draws from one unchanged distribution, with replacement, and counts the results. The observed frequencies are 0.504650, 0.245850, 0.125100 and 0.124400. They approximate the specified probabilities rather than matching them exactly. That experiment checks the sampler, not a language model's long-run token frequencies. In real autoregressive generation, the distribution changes after each appended token.

The loop retains the complete output but crops the context sent to the GPT to its last 128 tokens. A 130-token prompt therefore yields context lengths `[128, 128]` when two more tokens are requested, while the returned sequence has 132 tokens. Cropping the model's view is not the same as deleting the stored history. It is also not unlimited memory: omitted tokens cannot influence this uncached forward pass directly.

This simple GPT rebuilds learned position indices from zero inside each cropped window. Our demonstration is a bounded-context policy, not an exact extension of the model's original context capacity. Production position handling and cache eviction require their own design. The loop is intentionally uncached so every operation remains inspectable.

Stopping is separate from sampling. A maximum-new-token budget prevents an unbounded loop. An end-of-sequence token can additionally stop generation if the model and tokenizer have a meaningful EOS convention. The Shakespeare character checkpoint has no special EOS token, so its actual continuations stop at the length budget. A separate labelled fixture always emits ID 2 and proves the stop branch: a request for 10 new tokens returns one new token and the reason `eos`. That is a control-flow test, not evidence that the Shakespeare model learned when to end a conversation.

Build it

Run the complete lab

From the project root:

```
env -u PYTHONPATH .venv/bin/python chapters/llm-fundamentals/09-generation/lab.py
```

The lab resolves `labs/` relative to its own file, runs offline on CPU, and loads the existing trusted course checkpoint from `labs/models/shakespeare-char.pt`. No training, downloads, or hosted API calls occur. The reading-and-experiment allocation in the front matter is a study estimate, not the program runtime.

Reusable logic lives in `labs/tutorlab/sampling_demo.py`; regression checks live in `labs/tests/test_sampling_demo.py`. Existing GPT and training modules are reused unchanged. The six numbered lab parts correspond to the hand distribution, filtering and draws, sequence counterexample, checkpoint inspection, real generation, and stopping fixture.

Read the loop before the wrappers

The central operations, excerpted from the helper, are:

```
ctx = ids[:, -model.cfg.block_size:]
logits, _ = model(ctx)
probs = probabilities(logits[:, -1, :], temperature, top_k, top_p)
if temperature == 0:
    nxt = probs.argmax(dim=-1, keepdim=True)
else:
    nxt = torch.multinomial(probs, 1, generator=rng)
ids = torch.cat([ids, nxt], dim=1)
```

The slice keeps the most recent allowed context. The model returns logits plus an unused loss slot. Selecting the last position preserves a matrix with one vocabulary row per sequence. `keepdim=True` keeps the chosen ID shaped as a new time column. Concatenating along dimension one appends that column, rather than adding another batch row. On the next iteration, this chosen token is part of the input.

Our generation helper deliberately accepts only one nonempty, unpadded CPU prompt. The probability helper can process a matrix of independent rows, but that does not make the outer generation loop a batched serving engine. Padding masks, independently finished sequences, and cache management would all need explicit treatment before broadening that interface.

The model must be in evaluation mode. That disables training-time dropout behavior. The helper also uses inference mode, which avoids recording autograd work. These are different controls: PyTorch explicitly warns that inference mode does not call `model.eval()` for you. Neither one updates the weights, and the lab separately checks that parameters remain exactly unchanged and gradients remain absent.

Inspect real output

The following excerpts are copied from `build/lab-output.txt`, not hand-written target output:

```
parameters=1816896 vocabulary=65 block_size=128
prompt='ROME0:\n' ids=[30, 27, 25, 17, 27, 10, 0]
logits shape=(1, 7, 65) last shape=(1, 65)
next character='I' probability=0.144857
next character='T' probability=0.124463
next character='W' probability=0.105116
next character='A' probability=0.104540
next character='0' probability=0.068980
```

The most likely next character is not overwhelmingly likely. Greedy nevertheless always picks it for this fixed context. A sampled run may choose another plausible starting character. The tokenizer must match the trained checkpoint, because vocabulary IDs determine both embedding lookup rows and the meanings of output columns. Swapping tokenizers would change those meanings even if tensor shapes still fit.

Here are the actual continuations. Escaped newlines make the complete bounded strings visible without hiding line breaks:

```
greedy: 'ROMEO:\nI will be so the sea of the season of the sea\nOf the season of the season of
the season\nOf the s'
cool: 'ROMEO:\nThe last cause we have not been thine eyes to the world.\n\nKING RICHARD
III:\nNow, my lord, I will'
warm: "ROMEO:\nCarllant; add 'aEd in root as ease anon.\n\nROMEO:\nAy, with it with this mazesns
himself,\nWill not"
nucleus: 'ROMEO:\nThe land cause we have not been well,\nOr father, and with him so true, but in
her second all.\n\nR'
```

The settings are greedy; temperature 0.5 with no filters; temperature 1.2 with no filters; and temperature 0.8 with top-k 20 followed by top-p 0.9. Each requests 96 new character tokens using seed 1337. Each returns 103 total tokens including the prompt. The last policy changes multiple controls, so the comparison does not isolate the effect of top-p alone.

The greedy sample visibly repeats “season.” The warm sample includes malformed-looking words. Those observations describe these samples, not a benchmark proving that one policy always wins. A more useful evaluation would fix a prompt set, run multiple controlled samples, and measure the application's actual success criteria. A successful sampler can generate poor text when the predictor is limited, and a good-looking sample cannot certify a correct sampler.

Understand reproducibility and the checks

A local `torch.Generator` is seeded inside each generation call, rather than repeatedly resetting the application's global random stream. Repeating the same nucleus call produces identical IDs in the tested environment. Greedy and top-k equal to 1 also match. These are different checks: one tests random-stream replay; the other tests a degenerate distribution with only one available candidate.

As of 2026-10-10, the verified PyTorch reproducibility documentation explicitly does not guarantee identical results across releases, platforms, or CPU and GPU executions. The lab prints its actual library version and device. Treat those as part of the experiment, along with checkpoint, tokenizer, prompt, policy, seed and code. A seed by itself is not a complete reproducibility specification.

The assertions check real behavior: hand probabilities, temperature ordering of entropy, shift invariance, removed-token exclusion, draw frequencies within a declared tolerance, output length, vocabulary membership, prompt preservation, same-seed equality, context cropping, EOS stopping and unchanged parameters. The tests additionally exercise malformed inputs, exact nucleus boundaries, score ties, matrix rows, and a fixture whose position-specific logits detect accidentally sampling from the wrong time position.

Try the lab's predict-first modifications one at a time. Set top-k to 1 in both sampled temperature settings and predict what remains variable. Change the nucleus threshold in the hand example and determine the surviving IDs before running it. Change the seed while holding all model and policy settings fixed, then distinguish what must remain unchanged from what is allowed to differ. Do not assume that different seeds must always produce different text; a policy with only one surviving token has no choice to vary.

Common mistakes and misconceptions

"Temperature changes what the model knows." It changes the distribution used for token selection, not the trained parameters. Our lab compares frozen weights before and after generation. Better factual reliability requires evidence and evaluation, not a probability-sharpening label.

"Top-k removes nonsense." It removes low-ranked candidates. A rare correct token can be removed, and a high-ranked incorrect token can survive. Candidate ranking comes from the predictor; a cutoff does not introduce a truth checker. This is especially important when a generated string will later be parsed as a structured request.

"Softmax is applied to token IDs." IDs are labels for vocabulary entries. Softmax is applied to the model's real-valued scores for those labels. Taking softmax of the integer IDs would favor large ID numbers for no linguistic reason.

"Generation learns from the tokens it writes." Its context changes, so its future predictions change, but this loop does not update the model weights. Training and in-context conditioning are different mechanisms. The checkpoint stays the same after every sample.

"Same seed guarantees identical results everywhere." It specifies a random stream under particular implementation conditions. Hardware, kernels, software releases, prompt formatting and other settings can change results. Our replay assertion supports repeatability here, not a universal guarantee.

"An output token budget is the final sequence length." It counts newly generated tokens in this helper. The returned sequence includes the original prompt. Early EOS may shorten the generated portion. Cropping the model context affects neither the stored prompt nor this accounting rule.

In the real world

As of 2026-10-10, the Hugging Face Transformers GenerationConfig documentation exposes separate controls for `do_sample`, `temperature`, `top_k`, `top_p`, `max_new_tokens`, stop strings, EOS handling and cache use. This is the same separation we built: model scores, candidate policy, choice method, and stopping. Consult the model's actual generation configuration rather than assuming our helper's defaults or zero-temperature convention transfer directly.

The official PyTorch `torch.multinomial` documentation specifies the finite nonnegative input-weight requirements and an optional generator. Our lab demonstrates those primitives without importing a large generation framework. The real framework adds many important features, but its existence does not remove the need to understand tensor shapes and probability normalization underneath it.

Efficient transformer inference commonly reuses attention keys and values from earlier tokens. Hugging Face's cache explanation describes this per-layer KV cache. The initial prompt pass is often called prefill; subsequent decoding steps can process the new token while attending to cached history. Our uncached loop instead recomputes its full visible context every step. Adding a cache would optimize computation, not turn sampling into training or make future tokens available in advance.

Caches have validity conditions. Stored values correspond to a particular prefix, model and position convention. Sliding or rebasing a window cannot be treated as arbitrary deletion with no consequences, especially with the learned absolute positions in our small GPT. We keep caching conceptual here rather than presenting an untested optimization as equivalent behavior.

For a deployed application, make decoding parameters observable and evaluate their impact on the task. Keep model output validation, permissions and stop handling separate. A reproducible but invalid response should fail validation just as a sampled invalid response would. No sampler parameter replaces that boundary.

Go deeper

The lecture timestamps below were checked on 2026-10-10 using the course finder. YouTube transcripts were IP-blocked, so verification uses creator-published chapter markers rather than claims about exact spoken sentences. The anchor text in `lectures.json` is the marker title, not a quotation attributed to the speaker.

- Andrej Karpathy, "Let's reproduce GPT-2 (124M)," at 33:31: <https://www.youtube.com/watch?v=I8pRSuU81PU&t=2011> . The marker is "sampling init, prefix tokens, tokenization." Watch for how the prompt becomes token IDs and how model construction connects to generation.
- The same lecture at 37:02: <https://www.youtube.com/watch?v=I8pRSuU81PU&t=2222> . The marker is "sampling loop." Compare its final-position selection, normalization and sampling with our small helper. Pause and identify tensor dimensions before continuing.
- Holtzman et al., "The Curious Case of Neural Text Degeneration": <https://arxiv.org/abs/1904.09751> . Read the motivation for nucleus sampling and distinguish the paper's empirical findings from universal statements about all models or tasks.
- PyTorch reproducibility notes: <https://docs.pytorch.org/docs/2.14/notes/randomness.html> . Use the limitations to design a repeatable evaluation rather than promising more determinism than the system provides.

Explain it back

Without looking at the code, explain how the frozen GPT turns `ROME0:\n` into a continuation: where the next-token scores come from, how temperature and candidate filtering affect a draw, why the chosen token changes the next prediction, and why low temperature is not a truth guarantee. Use the four-token example to make one step concrete. Your study partner will check one gap at a time, then ask for a new application before recording mastery. Reading or running this chapter alone does not establish mastery.

Check yourself

`quiz.json` contains the answer key and explanation rubrics. Try these without the key first:

1. Which slice of a batch-by-time-by-vocabulary tensor supplies the next continuation scores?
2. What does lowering a positive temperature change, and what stays fixed?
3. What does top-k equal to 1 do under the lab's tie policy?
4. Which token must top-p retain when cumulative mass crosses the threshold?
5. For log weights of 4, 2, 1 and 1, predict probabilities at temperature 0.5.
6. For a 130-token prompt and two new tokens, predict model context lengths and returned length.
7. Explain why the greedy first choice in the constructed tree misses the more probable complete path.
8. Explain how to compare decoding policies without confusing sampling changes with training or claiming success from one attractive sample.

9. Why are `eval()` and inference mode both used?
10. Predict the difference between top-p alone and top-k followed by top-p in the filter-order example.

Sources

- Local executed evidence: `build/lab-output.txt`; implementation: `lab.py` and `labs/tutorlab/sampling_demo.py`; regression tests: `labs/tests/test_sampling_demo.py`. All displayed sample text and measured numerical results come from this run, not external performance claims.
- Academy chapter used as the topic baseline, read 2026-10-10: <https://krypteiasec.com/academy/llm-fundamentals/ch9> . This chapter expands its loop, temperature and top-k topics with a real GPT checkpoint, nucleus boundaries, reproducibility and stopping tests.
- Academy preceding chapter used for continuity while the local training chapter was being built: <https://krypteiasec.com/academy/llm-fundamentals/ch8> .
- Andrej Karpathy, "Let's reproduce GPT-2 (124M)": <https://www.youtube.com/watch?v=l8pRSuU81PU> . Timestamp verification is creator-marker based as disclosed above.
- PyTorch, `torch.multinomial`, checked 2026-10-10: <https://docs.pytorch.org/docs/stable/generated/torch.multinomial.html> . Source for weighted-draw requirements and the generator argument.
- PyTorch, inference mode, checked 2026-10-10: https://docs.pytorch.org/docs/2.14/generated/torch.autograd.grad_mode.inference_mode.html . Source for its autograd behavior and the need to call evaluation mode separately.
- PyTorch, reproducibility, checked 2026-10-10: <https://docs.pytorch.org/docs/2.14/notes/randomness.html> . Source for limits across releases, devices and platforms.
- Hugging Face, `GenerationConfig`, main documentation checked 2026-10-10: https://huggingface.co/docs/transformers/main/en/main_classes/text_generation . This is a moving documentation branch, not a pinned dependency used by the lab.
- Hugging Face, caching: https://huggingface.co/docs/transformers/en/cache_explanation . Source for per-layer KV reuse during autoregressive inference.
- Holtzman, Buys, Du, Forbes and Choi, "The Curious Case of Neural Text Degeneration," ICLR 2020: <https://arxiv.org/abs/1904.09751> . Source for nucleus sampling and the empirical discussion of repetitive likelihood-oriented decoding.