

The training loop: how a model actually learns

A model learns when an update changes its adjustable numbers so that its predictions better match the examples it is trained on. There is no separate moment when the computer understands a sentence. There is a repeated computation: make predictions, score them against targets, calculate which changes would locally reduce that score, and apply an update. Today you will inspect each operation, then run exactly the same training function on a table and on the GPT assembled earlier.

Allow roughly 70 minutes for reading, the lab, and an explain-back. This is a study estimate, not a measured lab duration. The lab itself is a small offline CPU experiment.

Why this matters

The previous chapter assembled the forward pass. Tokens entered a sequence of embeddings and transformer blocks, and vocabulary logits came out. Shape checks and causal tests established that the computation was wired correctly. They did not make its predictions useful. This chapter supplies the mechanism that changes the weights using examples.

If you work on AI systems, this distinction helps diagnose failures. A model can be architecturally correct but poorly trained. A training program can run without an exception while predicting the wrong target, accumulating stale gradients, or accidentally evaluating on training data. Printing a decreasing number does not distinguish these possibilities. You need to know what produced the number and what actually changed.

The same distinction matters in agent engineering. Adding text to a prompt changes the current computation's input. Taking a training step changes model parameters used by future computations. Those are different mechanisms. A successful interaction with a model does not, by itself, show that its weights were updated. Application memory is another separate mechanism, not a synonym for gradient descent.

We return first to the bigram from chapter 3 because its entire decision rule fits in a table. That lets us verify the gradient and compare learned probabilities with counts. Then we replace that table with our existing GPT without changing the training interface. The point is not to train a useful chatbot here. It is to remove the mystery from the loop that a larger training system repeats.

The idea in plain words

Suppose a model must choose among three possible next tokens. At first it assigns each equal probability. The real next token is the middle candidate. We penalize the model for assigning that candidate too little probability. A calculation tells us how the penalty would change if each score moved slightly. We push the correct score up and its competitors down, then predict again.

The lab makes this example literal. Its starting logits are `[0, 0, 0]`, and its starting loss is `1.098612`. The gradient is approximately `[0.333333, -0.666667, 0.333333]`. With learning rate `0.3`, plain gradient descent changes the logits to `[-0.1, 0.2, -0.1]`. The correct candidate's probability becomes `0.402960`, and the loss falls to `0.908918`.

These numbers are not hand-written pretend output. Part 1 computes them, and checks the gradient using two independent routes. The example is deliberately small enough to work through on paper as well.

Now name the operations. Forward means computing the predictions with the current weights. Loss means reducing their mismatch with the targets to a score. Backward means calculating derivatives of that score through the computation. The optimizer uses those derivatives to change the weights. Resetting the stored gradients prevents unrelated updates from being combined accidentally. Repeating this sequence is training.

There is an important limit to the story. Lower training loss shows better fit under that training objective on those examples. It does not prove reliable answers on different inputs, factual correctness, instruction following, or safe tool use. Later in the lab, a deliberately mismatched evaluation set gets worse while training loss gets better.

How it actually works

Make the target unambiguous

For next-token prediction, each target is the token immediately after the corresponding input token. Our toy training string is `a cat sat. a cat ran. a rat ran. a rat sat..` A fixed character vocabulary gives it numeric IDs. The lab creates input IDs from all but the final character and targets from all but the first character.

If `tokens` has shape $(B, T+1)$, we form:

```
x = tokens[:, :-1].contiguous()
y = tokens[:, 1:].contiguous()
```

`B` is the number of sequences in the batch. `T` is the number of scored positions per sequence. The colon selects every batch row. The input slice omits the final token; the target slice omits the first. Both resulting arrays have shape (B, T) . Contiguous storage matters because our existing GPT flattens targets using `view` internally.

The target is not the input itself. If you set `y = x`, you reward the model for reproducing a token it can already see. A causal mask does not fix that objective: causal attention permits seeing the current token. Likewise, shifting targets twice would ask for a different future token than intended. Check the model interface before assuming that the library shifts anything for you. Our shared GPT does not.

During training, the correct earlier tokens are supplied as context. We do not sample a guessed token at every position and feed that guess back into the loss computation. This use of known preceding tokens is commonly called teacher forcing. Causal masking prevents later tokens in the supplied sequence from leaking into an earlier prediction. Sampling belongs to the next chapter.

Turn scores into a penalty

For one position, let `zj` be the logit for candidate token `j`. A logit is a real-valued score, not yet a probability. Let `pj` be the probability after softmax, and let `k` be the correct target ID:

```

p_j = exp(z_j) / sum_i exp(z_i)
ell = -ln(p_k)
L = (ell_1 + ... + ell_N) / N

```

`exp` is the exponential function. The sum in the denominator includes every vocabulary candidate `i`. `ln` is the natural logarithm. `ell` is one position's negative log likelihood. `L` is the mean over `N` scored positions. In our unpadded examples, `N` is `B*T`. With padding or masked targets, the averaging rule must instead account for which positions really contribute.

The penalty grows when the probability assigned to the observed target shrinks. It is sensitive to confidence, unlike a simple correct-or-incorrect count. Improving the probability of a correct answer can lower the loss even if the highest-scoring token does not change. That is why loss can reveal useful learning before a displayed sample becomes visibly different.

PyTorch's `cross_entropy` expects raw logits for this case, together with integer target IDs. It combines logarithmic normalization with the target penalty. Do not apply softmax first and pass those probabilities as if they were logits. For this lab, there are no class weights, ignored positions, or smoothed targets. Those options change the precise formulas and normalization, so we keep them out of the first derivation.

Derive the gradient rather than trusting a slogan

Rewrite the one-position loss as:

```

ell = ln(sum_i exp(z_i)) - z_k
d ell / d z_j = p_j - indicator(j == k)

```

The indicator is one when candidate `j` is the target and zero otherwise. The derivative of the logarithmic sum gives `p_j`; differentiating the subtracted target score gives the indicator. Thus an incorrect candidate receives a positive derivative, while the target receives a nonpositive derivative. Increasing a competing score locally increases loss. Increasing the target score locally decreases it.

For the mean loss over a batch, the derivative for each logit row is divided by `N`. Saying only "probabilities minus the target" omits that reduction factor. It also identifies the gradient with respect to the logits, not every weight in a transformer. Backpropagation must continue through the head, blocks and embeddings to obtain parameter gradients. Those involve the chain rule and contributions from every use of a shared parameter.

Work through the small example. Equal logits make all three probabilities equal to one third. The target vector is `[0, 1, 0]`. Subtracting it gives `[1/3, -2/3, 1/3]`. For plain stochastic gradient descent without momentum or weight decay:

```

theta_new = theta_old - eta * gradient

```

`theta` names the trainable parameter. `eta` is the learning rate. In this special example the trainable parameters are the logits themselves. For the middle score, subtracting `0.3 * (-2/3)` adds `0.2`. For either other score, subtracting `0.3 * (1/3)` gives `-0.1`. Substitution into softmax produces the measured target probability and lower loss reported above.

The minus sign is essential. It moves against the local uphill direction. However, a gradient is local information, not a guarantee about an arbitrarily large step. A learning rate that is too large can move to a region with higher loss. Changing minibatches can also make successive logged losses fluctuate even when the overall run is improving.

Backward and update are different operations

Calling `loss.backward()` populates gradient buffers such as `parameter.grad`. It does not itself apply the optimizer update. Part 2 explicitly checks that weights remain unchanged after backward. The optimizer's `step()` is the operation that changes these trainable parameters in our loop.

PyTorch accumulates gradients into existing buffers. If we recompute exactly the same loss at unchanged weights and call backward again without clearing, we add the same derivative again. The lab observes a gradient ratio of `[2.0, 2.0, 2.0]`. It uses a fresh forward computation for each backward, so this demonstration does not require retaining a previously used computation graph.

The reset `optimizer.zero_grad(set_to_none=True)` sets gradient attributes to `None`. That is not a weight reset. It prepares the buffers for the next computation. A parameter that receives no gradient after this reset can remain `None`, which is observably different from having a tensor of zeros. The official optimizer documentation describes that distinction.

Accumulation can be intentional. To emulate a larger batch, accumulate appropriately weighted microbatch losses before a single optimizer step. Equal-sized microbatches can use equal scaling. Unequal-sized microbatches require weighting by contributing target counts to reproduce the full mean. Our companion test compares a full batch with unequally sized pieces and checks their parameter gradients match. Stepping after each piece would be a different procedure because subsequent gradients would be evaluated at changed weights.

Learn the same probabilities that counting estimates

The neural bigram is `nn.Embedding(V, V)`, where `V` is vocabulary size. Here an embedding lookup returns a row of vocabulary logits rather than an intermediate semantic feature vector. Input character `x` selects row `W[x]`. The model cannot distinguish two occurrences of that character using earlier context because it never reads that context.

The toy vocabulary has 8 entries and the training string supplies 42 next-character targets. We initialize every logit to zero, so the initial probability distribution is exactly uniform. Its mean loss is $\ln(8)$, measured as 2.079442. This is a deliberate initialization property, not a claim that every randomly initialized neural model is perfectly uniform.

Counting gives us a useful independent reference. For each current character, divide each successor count by the total outgoing count. These empirical conditional frequencies minimize the unrestricted bigram training objective. Some successors never occur; exact zero probabilities would require infinitely negative relative logits in a finite softmax table. Therefore we call the count-based optimum an infimum for the softmax model, rather than promising finite weights can attain it exactly.

The lab computes this floor as 0.731409. After 200 AdamW updates, with weight decay explicitly disabled, the neural table reaches training loss 0.735081. Its largest row-probability discrepancy from the empirical table is 0.005222. This comparison is more informative than merely checking that loss decreased: it checks the learned object against an independently calculated statistical target.

Notice that the floor is not zero. The same current character can have different successors. After `a`, for example, the empirical probability of `t` is `0.5000`, while space receives `0.3333` and `n` receives `0.1667`. The bigram lacks information to choose a different distribution for each occurrence. A longer-context model can use more information, but that alone does not guarantee perfect prediction either.

Separate fitting, validation and stress testing

Training examples drive gradients and optimizer steps. Validation examples help judge choices such as training duration and learning rate without being used directly for those updates. If you repeatedly choose settings based on validation, it is no longer a pristine final test. Reserve genuinely independent final evaluation data when making broader performance claims.

Our separate validation string is `a rat sat. a cat ran..` Its loss falls from `2.079442` to `0.739163`. However, it shares phrases and the same tiny patterns with training. It is an interface demonstration for evaluating a separate batch, not convincing evidence of generalization to unseen natural language. In a real corpus, split at meaningful document boundaries and examine duplication before treating held-out numbers as independent evidence.

Part 5 makes the limitation impossible to miss. A different two-token model trains on alternating IDs while a stress string repeatedly uses zero. Training loss falls from `0.693147` to `0.025389`, but stress loss rises from `0.693147` to `3.686117`. The distribution was deliberately changed. This is not a measured overfitting curve on naturally sampled validation data; it is a controlled counterexample to the assertion that better training fit guarantees better performance everywhere.

Build it

Run from the project root:

```
env -u PYTHONPATH .venv/bin/python chapters/llm-fundamentals/08-the-training-loop/lab.py
```

The lab finds `labs/` relative to its own path. It runs on CPU, requests deterministic algorithms, and sets a fixed random seed. It downloads nothing, loads no pretrained checkpoint, and does not write learning progress. Exact floating-point details can differ across environments, so the checks use numerical tolerances where appropriate.

The reusable additions are in `labs/tutorlab/training_loop_demo.py`. Existing GPT and training-library files are reused unchanged. The new tests are in `labs/tests/test_training_loop_demo.py`.

Parts 1 and 2: audit one update

Part 1 compares autograd with the analytic mean-loss gradient. It also perturbs the target logit slightly in each direction and measures the slope using a central finite difference. That numerical slope is `-0.666667` when printed to six decimal places, matching the analytic derivative. Finite differences are useful for a small diagnostic, but would be expensive and numerically delicate as a general way to train a large network.

Part 2 stores copies of weights and gradients. It proves that backward leaves the weights alone, that repeated backward accumulates, and that one clean SGD step matches the hand calculation. Copies matter: holding another reference to a mutable tensor would not preserve its earlier value.

`detach().clone()` stores a separate snapshot without keeping it attached to autograd history.

Parts 3 and 4: inspect and train the bigram

Part 3 builds aligned pairs, counts transitions, and computes the empirical loss floor. We include only observed transitions in the logarithmic sum, avoiding an invalid numerical calculation of zero times the logarithm of zero. Rows are normalized by outgoing counts, not by total corpus length; the final average loss is weighted by each transition's occurrence count.

Part 4 calls this training function:

```
def train_step(model, optimizer, x, y):
    model.train()
    optimizer.zero_grad(set_to_none=True)
    _, loss = model(x, y)
    loss.backward()
    optimizer.step()
    return loss.detach().item()
```

`model.train()` selects training behavior for layers such as dropout; it does not take a training step by itself. The model returns logits and loss. The underscore ignores logits because this function only needs loss. Backward traverses that loss's computation graph. The optimizer was constructed from the model's registered parameters, so it knows which tensors it may update.

The returned scalar is the loss computed before the update. It does not magically become the post-update loss when weights change. That is why our printed trajectory calls `evaluate` after the stated number of completed updates. The row labeled zero is genuinely before any optimizer step. Matching metrics to update counts avoids a common off-by-one storytelling error.

Here is a verbatim excerpt from `build/lab-output.txt`:

```
Part 4: learn a probability table instead of counting it
after 0 updates: train=2.079442 validation=2.079442
after 1 updates: train=1.945874 validation=1.945888
after 10 updates: train=1.125601 validation=1.128769
after 50 updates: train=0.748651 validation=0.752532
after 200 updates: train=0.735081 validation=0.739163
successors of a (empirical / learned): ' ':0.3333/0.3324 '.' :0.0000/0.0006 'a':0.0000/0.0006
'c':0.0000/0.0006 'n':0.1667/0.1665 'r':0.0000/0.0006 's':0.0000/0.0006 't':0.5000/0.4980
max row probability error=0.005222
```

AdamW supplies an adaptive update rather than the exact plain-SGD formula used in Part 1. It tracks running averages of gradients and squared gradients, and keeps its weight-decay mechanism separate from those averages. Its optimizer state matters: we construct the optimizer once, outside the loop, rather than discarding that state on every iteration. Do not use the hand SGD update as an exact numerical prediction for AdamW. We set `weight_decay=0.0` explicitly for this experiment so the objective comparison with empirical counts is not mixed with a weight-decay effect.

Part 5: challenge the conclusion

Read the mismatch experiment before admiring the training curve. It teaches an evidence boundary: the number is meaningful only for the measured examples and loss definition. A worse stress score is not an implementation failure here. It is the expected consequence of making predictions more confident in one pattern while evaluating on another. The assertions require both the improved training fit and worsened stress fit.

Part 6: swap in the assembled GPT

The shared GPT accepts the same inputs and targets and returns the same pair of logits and optional loss. We train it on short repeating character-ID sequences using the exact `train_step` function above. It has 3488 unique parameters. After 80 updates, its fixed training-batch loss falls from 1.156632 to 0.002131.

```
Part 6: replace the table with the shared GPT
shapes: inputs=(2, 8) logits=(2, 8, 3) targets=(2, 8)
params=3488; updates=80; training loss 1.156632 → 0.002131
attention weight max change=0.178390
evaluation preserved state=True; restored train mode=True
tiny GPT result is a memorization smoke test, not a held-out score
CHECK PASSED
```

The attention projection's weights also change, so learning is not confined to a standalone output table. Nevertheless, this task is simple enough that a bigram could learn its transitions. The result proves that the shared GPT can be optimized through its complete computation. It does not demonstrate an advantage from attention or meaningful language competence.

Evaluation uses both `model.eval()` and `torch.no_grad()`. The former controls layer behavior such as dropout. The latter prevents recording the differentiation graph for this evaluation. Neither is an optimizer step. Our helper restores the caller's root train/eval mode, even on an error. Its intended models use a uniform mode across their submodules; it is not a general restoration utility for deliberately mixed per-layer modes.

The tests check that evaluation preserves weights and existing gradient buffers. They also cover malformed input batches, analytic versus numerical gradients, stale-gradient clearing, unequal-microbatch weighting, a zero learning rate, and the GPT integration. A passing run establishes these specific contracts, not arbitrary training correctness.

Predict before editing. Set the bigram learning rate to zero and identify which operations still happen. Remove the Part 2 reset and predict how its SGD update changes. Change the stress pattern and predict which transition gets penalized. Keep the baseline output and change one variable at a time. Deliberate changes can make the original assertions fail. Explain which original assumption no longer applies, then restore the baseline rather than weakening checks simply to force a pass.

Common mistakes and misconceptions

"Backward trains the model." Backward computes and accumulates derivatives. In this loop, `optimizer.step()` changes the parameters. A nonzero gradient shows a learning signal exists, not that an update happened.

"Cross entropy needs probabilities as input." The PyTorch function used here needs raw logits. Feeding softmax output into it applies the wrong normalization to the wrong input representation. The target IDs remain integer labels, not scores.

"The gradient of every weight is probability minus target." That simple expression applies to the logits for the specified loss, with the mean-reduction factor included. Parameter gradients require chaining derivatives through all the preceding operations and summing shared contributions.

"Resetting gradients erases learning." It clears temporary derivative buffers. It does not reset weights or AdamW's optimizer state. Forgetting to reset changes the update rule; resetting between intentionally accumulated microbatches also changes it.

"A decreasing training curve proves generalization." It proves better fit on the scored training objective. Label leakage, duplicate evaluation data, memorization, or distribution differences can still make deployment behavior poor. Part 5 supplies a concrete counterexample.

"Evaluation mode switches autograd off." Mode and gradient recording are separate controls. Evaluation code typically uses both eval mode and a no-grad context, while diagnostic backward computations can validly be run in eval mode.

"An optimizer step is an epoch." A step is one parameter update. An epoch is a pass through a defined dataset. Our full-batch bigram repeatedly uses the entire toy training set; sampled batches in a larger run need different bookkeeping. Log update counts and processed targets explicitly rather than using those words interchangeably.

In the real world

The PyTorch autograd notes, CrossEntropyLoss contract, gradient-zeroing recipe and AdamW documentation were checked on 2026-10-10. They separate forward prediction, differentiation and parameter update exactly as this lab does. These are library-interface observations, not claims about the private training recipe of a hosted model.

Karpathy's public nanoGPT [train.py](#), inspected on the same date, preserves this structure while adding operational machinery: gradient accumulation, learning-rate warmup and decay, gradient clipping, mixed-precision handling, distributed execution, evaluation and checkpoints. Its equal-sized microbatches divide the loss by the accumulation count before backward. Its optimizer steps after accumulation, then clears gradient buffers.

Those additions have distinct jobs. Accumulation changes the effective batch assembled for an update. Clipping limits the gradient norm before the optimizer uses it. Learning-rate scheduling changes update scale over the run. Mixed precision changes numerical representation and may require scaling safeguards. None makes an incorrect target shift or contaminated evaluation set correct. Treat basic data and objective checks as prerequisites, not optional polish after a large run fails.

The local [labs/tutorlab/train.py](#) already demonstrates several of these ideas for a larger character GPT. We read it as a reference but do not run that larger training job or change it here. Its batch sampler constructs shifted targets and its loop performs reset, backward, clipping and an AdamW step. Our chapter isolates those mechanisms so each can be understood before the additional machinery is introduced.

Checkpointing also needs precise claims. Weights and model configuration can restore a forward computation. Resuming an optimizer trajectory additionally depends on optimizer state, schedule position, data position and relevant random-number state. nanoGPT's inspected checkpoint includes model and optimizer states, configuration and iteration information; that does not mean every possible exact-replay requirement is automatically satisfied. This chapter does not claim to have tested full training resumption.

Go deeper

The following moments are in `lectures.json`. Transcript requests were IP-blocked during preparation. The timestamps and anchor phrases were verified against creator-published chapter markers instead, so the anchors are titles, not claimed quotations from speech.

- Andrej Karpathy, "Let's build GPT: from scratch, in code, spelled out," at 34:53: <https://www.youtube.com/watch?v=kCc8FmEb1nY&t=2093>. The marker is "training the bigram model." Watch how the prediction table, loss, gradient reset and optimizer step fit together. Compare where the displayed loss is measured with our explicit post-update evaluation rows.
- Andrej Karpathy, "Let's reproduce GPT-2 (124M)," at 154:09: <https://www.youtube.com/watch?v=l8pRSuU81PU&t=9249>. The marker is "gradient accumulation." Connect smaller microbatches with the need to normalize their gradient contributions before one optimizer update. Compare equal-sized microbatches with the unequal-token-count test in our helper suite.

For reading, compare the official `CrossEntropyLoss` equations with `mean_ce_logit_gradient`, then inspect nanoGPT's accumulation loop. Identify exactly which averaging assumptions make its loss scaling appropriate. Reading code with those questions is more useful than memorizing an optimizer name.

Explain it back

Explain how a model moves from a wrong next-token prediction to changed weights. Use the three-candidate example to describe loss, the sign of the target gradient, gradient reset and the optimizer step. Finish by explaining why falling training loss is not sufficient evidence of useful behavior on new inputs.

Your study partner should identify one gap and ask one focused follow-up. Then apply the explanation to a new case: an engineer splits a batch into unequal pieces, averages each piece's mean loss equally, and expects the same update as the original batch. Diagnose the mismatch and describe the correct weighting. An accurate explanation plus a correct application is the mastery criterion. Running this file or receiving the chapter does not record mastery.

Check yourself

The questions and answer keys live in `quiz.json`. Attempt each question before revealing its answer. The check asks:

1. Which operation changes the weights, and which computes gradients?
2. What input should PyTorch cross entropy receive for integer next-token targets?
3. What changes in the logit gradient when loss uses a batch mean?
4. Why must gradients normally be reset between optimizer updates?
5. What can falling training loss establish about a different evaluation distribution?

6. Predict the hand example's SGD update and the direction of the target-probability change.
7. Predict the result of a zero learning rate in the zero-initialized bigram.
8. Explain the complete next-token training loop and its data alignment.
9. Diagnose unequal-microbatch weighting and distinguish it from taking multiple optimizer steps.

Recall cards revisit gradients versus weights, raw logits, target shifts, mode versus no-grad, and the limits of training loss. They are material for later retrieval practice, not evidence that the learner has already understood them.

Sources

- KrypteiaSec Academy, chapter 8, inspected 2026-10-10 as a coverage baseline, not copied: <https://krypteiasec.com/academy/llm-fundamentals/ch8>
- PyTorch, CrossEntropyLoss, raw-logit inputs, integer targets and reduction formulas, checked 2026-10-10: <https://docs.pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html>
- PyTorch, Autograd mechanics, chain-rule execution and evaluation versus gradient modes, checked 2026-10-10: <https://docs.pytorch.org/docs/2.14/notes/autograd.html>
- PyTorch, Zeroing out gradients, accumulation behavior, checked 2026-10-10: https://docs.pytorch.org/tutorials/recipes/recipes/zeroing_out_gradients.html
- PyTorch, AdamW, adaptive state, decoupled weight decay and reset semantics, checked 2026-10-10: <https://docs.pytorch.org/docs/2.14/generated/torch.optim.AdamW.html>
- Andrej Karpathy, nanoGPT training implementation, accumulation, scheduling, clipping and checkpoint fields inspected 2026-10-10: <https://github.com/karpathy/nanoGPT/blob/master/train.py>
- Andrej Karpathy, "Let's build GPT: from scratch, in code, spelled out": <https://www.youtube.com/watch?v=kCc8FmEb1nY>
- Andrej Karpathy, "Let's reproduce GPT-2 (124M)": <https://www.youtube.com/watch?v=I8pRSuU81PU>
- Local executable evidence: `lab.py`, `build/lab-output.txt`, `labs/tutorlab/training_loop_demo.py`, and `labs/tests/test_training_loop_demo.py`. GPT implementation and training reference: unchanged `labs/tutorlab/gpt.py` and `labs/tutorlab/train.py`. All miniature-model measurements in this chapter come from the recorded lab run.