

Building an LLM from Zero

Chapter 13: The Training Loop

Module 3: Training and Generation

Truong (Jack) Luu · jackluu.io/book

Where we are

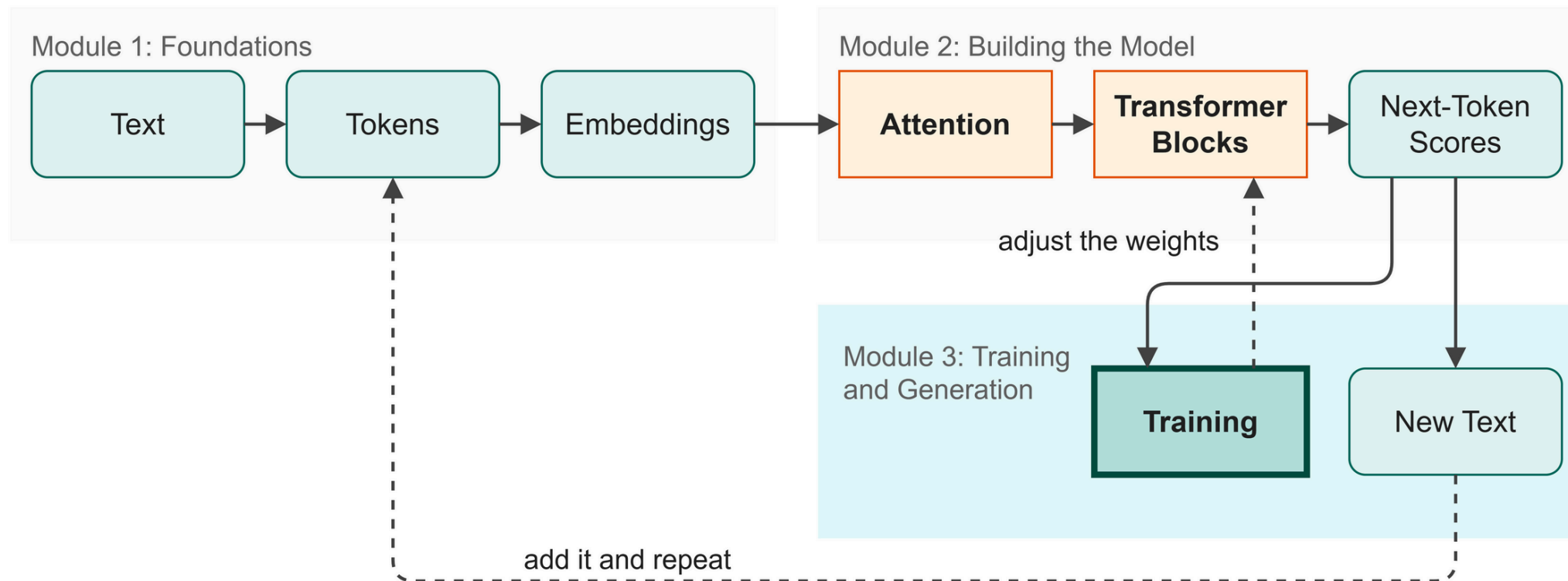


Figure 13.1: The model makes predictions, measures its error, and adjusts its weights to improve.

Everything built so far comes down to this chapter. The model guesses no better than chance. Today we write the loop that teaches it.

What this builds on (60-second recap)

- **Chapter 12** built the data pipeline: a **sliding window** over the text, packed into **batches** of 32 sequences of 128 characters
- The **DataLoader** shuffles and delivers them: 31,367 batches per pass over the text
- The model (Chapter 10) scores every position; the **loss** (Chapter 11) measures its surprise

Next: how does a wrong guess become a better weight?

In this chapter you will

1. Understand **gradient descent** as walking down a hill
2. See what the **learning rate** does, by measuring three of them
3. Build the **four-step training loop**
4. Watch the model learn in real time, and read what its loss actually means

Words to know (each is glossed again where it first appears):

- **Optimizer**: the algorithm that updates the model's weights; we use Adam, a popular and steady choice
- **Gradient**: the direction that would increase the error; we move the opposite way to decrease it
- **Backpropagation**: calculating the gradient for every single weight in the model
- **Learning rate**: how far the optimizer moves the weights on each step

The run is already going

Started at the top of class. This is what the first lines look like:

```
$ python src/ch12_train.py
Chapter 13: The Training Loop

Model parameters: 824,832
Training on      : cpu
Steps           : 3,000
Batch size      : 32
Block size      : 128

Starting training... (eval every 300 steps)
```

- 3,000 steps; every 300 steps it reports the training loss and the validation loss
- It finished in under half an hour on the author's 2020 laptop with no GPU

Walking down the hill

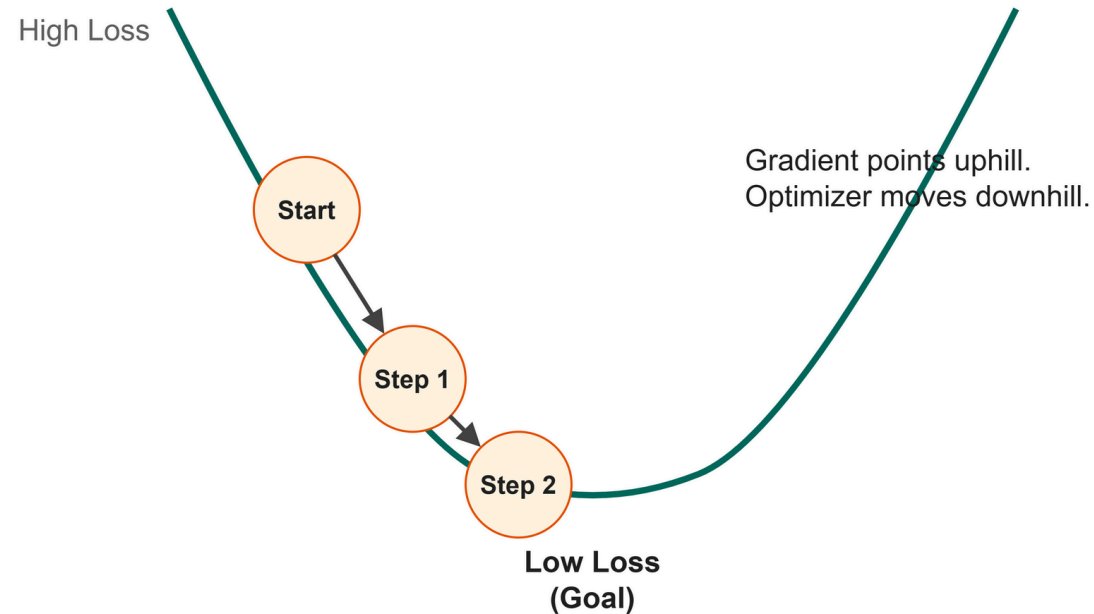


Figure 13.2: The optimizer takes small steps down the loss landscape to find the best weights.

Blindfolded on a hillside, you want the lowest point (the lowest loss). You cannot see the hill, but you feel the slope under your feet: if it rises to your right, step left.

Gradient descent: the slope is the gradient (found by backpropagation). A small step downhill is the optimizer updating the weights.

The training cycle

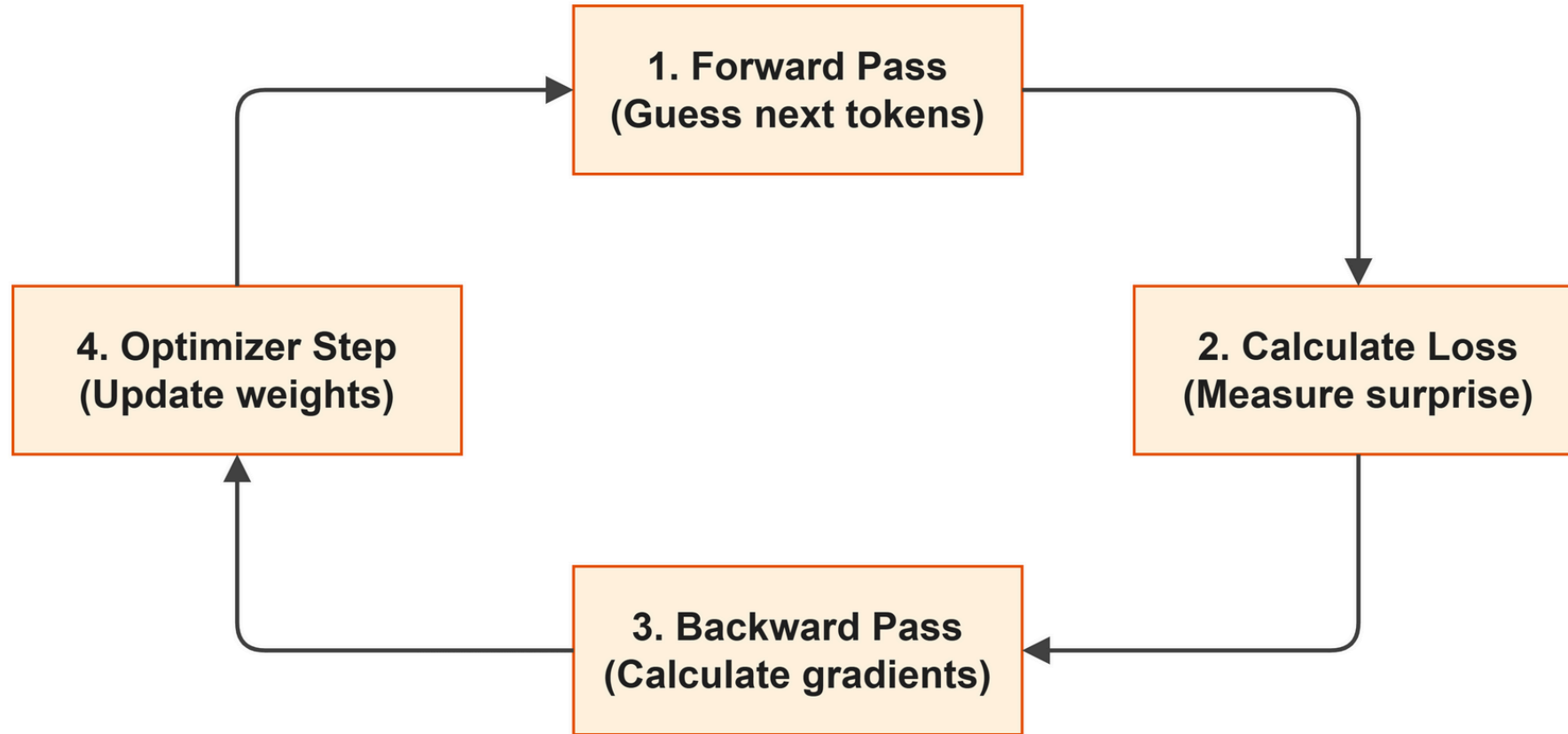


Figure 13.3: The four steps of the training loop.

Round and round: predict, measure, trace the blame, adjust.

The four-step loop

1. **Forward pass:** a batch goes through the model to get predictions
2. **Calculate loss:** compare predictions with the targets using cross-entropy
3. **Backward pass:** PyTorch's autograd engine computes the gradient for every weight (`loss.backward()`)
4. **Optimizer step:** adjust the weights slightly in the right direction (`optimizer.step()`)

Four steps, repeated 3,000 times. That is training.

How big should the step be?

Gradient descent says which way is downhill, not how far to walk. That distance is the **learning rate**. Same model, same starting weights, three rates, 150 steps each:

```
$ python src/examples/ch13_learning_rate.py
lr=0.003      start 4.33  after 150 steps 2.35
lr=0.0003     start 4.33  after 150 steps 2.60
lr=3e-05      start 4.33  after 150 steps 3.31
```

- **Bottom row:** at 0.00003 the model has barely moved (3.31, where random guessing is 4.17). It is learning, far too slowly to be useful
- **Top row:** ten times the book's rate learned *faster* here. So why not use it?

150 steps is not 3,000. A large step can overshoot or blow up. The book's 0.0003 is the cautious choice: it reaches 1.74 over the full run without drama.

It learned

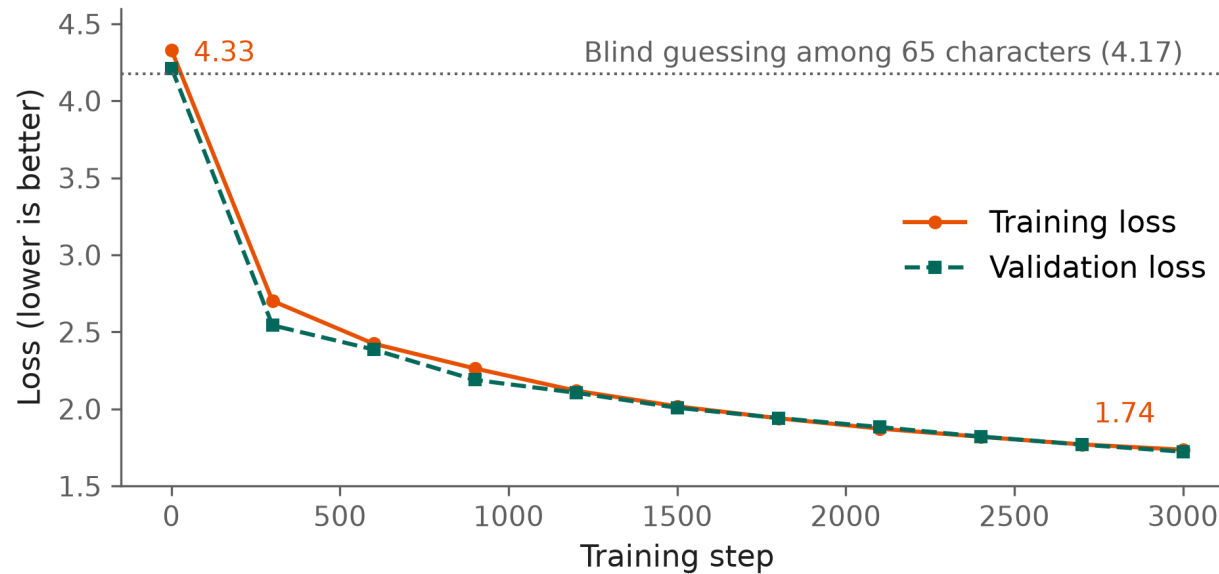


Figure 13.4: Over 3,000 steps, the model goes from blind guessing to predicting text with high confidence.

The loss starts around **4.3**, worse than random guessing (4.17). By step **300** it is at **2.7**; by step **3,000** it reaches **1.74**.

The loss plummets early and keeps falling. This chart is plotted from the real training log, not drawn by hand.

What the loss number actually means

Cross-entropy is the negative logarithm of the probability the model gave to the right character. Undo the logarithm and the probability comes back:

```
$ python src/examples/ch13_loss_meaning.py
random guessing  loss 4.17 ->  1.5% on the right character
step 1           loss 4.33 ->  1.3% on the right character
step 300         loss 2.70 ->  6.7% on the right character
step 3000        loss 1.74 -> 17.6% on the right character
```

- The trained model is **twelve times better than chance**
- It is still wrong about the next character roughly **four times in five**

It has learned which characters follow which, how long words run, where lines break. It has not learned English, and it does not mean anything.

Code: the loop

File: `src/ch12_train.py` (source files are numbered one lower than chapters)

```
optimizer = torch.optim.Adam(model.parameters(), lr=train_cfg.learning_rate)
for step in range(1, train_cfg.max_iters + 1):
    x, y, train_iter = get_batch(train_loader, train_iter, device)
    # Forward pass
    logits = model(x)
    loss = F.cross_entropy(logits.view(-1, gpt_cfg.vocab_size), y.view(-1))
    # Backward pass
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

- Line 1 creates the **Adam** optimizer (the algorithm that updates the weights); lines 5 and 6 are steps 1 and 2
- Lines 8 to 10 are steps 3 and 4: clear old gradients, compute new ones, update the weights

The whole training loop is these twelve lines.

Run it: read the log

Back to the running window. The chapter's own run printed:

```
step      1/3000 | train loss: 4.3280 | val loss: 4.2114 | elapsed: 3s | ETA: 7595s
step     300/3000 | train loss: 2.7016 | val loss: 2.5433 | elapsed: 55s | ETA: 496s
...
step     900/3000 | train loss: 2.2633 | val loss: 2.1889 | elapsed: 160s | ETA: 372s
step    1200/3000 | train loss: 2.1185 | val loss: 2.1053 | elapsed: 210s | ETA: 314s
step    1500/3000 | train loss: 2.0174 | val loss: 2.0078 | elapsed: 248s | ETA: 248s
```

What just happened

- The training loss fell from **4.32** toward **1.74**; the validation loss (on text the model never trains on) follows it down
- Every 300 steps the script pauses to measure both; at the end it saves the learned weights for Chapter 14

Shape check

Variable	Shape	Meaning
<code>model.parameters()</code>	824,832	The total number of weights the optimizer is adjusting

Every one of those 824,832 numbers gets its own gradient and its own nudge, 3,000 times.

Try it: autograd on one weight

A one-weight "model": $\text{output} = \text{weight} \times 3$, and we want the output to be 12. Run `src/examples/ch13_autograd_demo.py`:

```
$ python src/examples/ch13_autograd_demo.py
Initial weight: 2.00
Step 1: Output=6.00, Loss=36.00, Gradient=-36.00
Step 2: Output=11.40, Loss=0.36, Gradient=-3.60
Step 3: Output=11.94, Loss=0.00, Gradient=-0.36
Final weight: 4.00
```

- `loss.backward()` computes the gradient; the update moves the weight against it
- Then change the starting `weight` to `10.0`: the chapter says the gradient now pulls it down instead of pushing it up, still aiming for 12

The same mechanics, one weight instead of 824,832.

In business: the loss curve is the dashboard

Training is where the real investment happens.

- The company pays for the compute time to run this loop billions of times across thousands of documents
- The **loss curve** is the main dashboard metric for the house-style assistant
- As long as it is going down, the assistant is getting better at mimicking the corporate voice

One number, falling, tells you the money is working.

Watch out: zero the gradients

Never forget `optimizer.zero_grad()` before `loss.backward()`.

- PyTorch **accumulates** gradients by default: it adds them up
- Without zeroing, each step mixes the current batch with all previous batches, and the model wanders off in the wrong direction

```
optimizer.zero_grad()  
loss.backward()  
optimizer.step()
```

Clear, compute, step. Always in that order.

Discussion

The chapter leaves one experiment to you: 0.003 for all 3,000 steps.

In pairs, 3 minutes:

1. Predict: does it beat 1.74, or does it blow up? What is your reasoning?
2. What would you watch in the log to know which is happening?

Take two answers from the room.

Recap: today in 5 points

1. Training is a loop of four steps: **forward**, **loss**, **backward**, **step**.
2. **Gradient descent** finds a low loss by taking small steps downhill.
3. The **learning rate** sets the step size: too small crawls, too large can overshoot. The book measures three.
4. The **Adam** optimizer handles the update math for all 824,832 weights.
5. The loss fell from **4.32** to **1.74**: from 1.3% to 17.6% on the right character, twelve times better than chance and still wrong four times in five.

The model has learned the patterns of its text. Next we save what it learned.

Check your understanding

1. What are the four main steps inside the training loop?
2. What does `loss.backward()` actually do?
3. Why is a dropping loss curve a good sign?

Bonus: the log prints a train loss and a val loss. What would it mean if train kept falling while val started to rise?

Next up: Chapter 14, Checkpointing

- Training took real time. Next you **save** the learned weights to a file and **load** them back without retraining
- You will also switch the model from training mode to evaluation mode

Before next class: let `python src/ch12_train.py` run to the end on your own machine, so `checkpoints/model.pt` exists.

References

Book and figures

- Luu, T. (2026). *Building an LLM from Zero*, Chapter 13. <https://jackluu.io/book/>
- Figures 13.1 to 13.3 are the book's own diagrams (draw.io sources in the book repository, `diagrams/`); Figure 13.4 is plotted by the book's chart script from the training log.
- Terminal output: `src/ch12_train.py`, `src/examples/ch13_learning_rate.py`, `src/examples/ch13_loss_meaning.py` and `src/examples/ch13_autograd_demo.py`.

Further reading (from the chapter)

- Kingma, D. P., & Ba, J. (2014). *Adam: A method for stochastic optimization* (arXiv:1412.6980).
- Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323, 533–536.

Dataset

- Tiny Shakespeare, from Andrej Karpathy's char-rnn project: <https://github.com/karpathy/char-rnn>