
How a model learns

Key takeaways

1. Training never changes the architecture. It chooses the weights, nudging each one so the next token the data actually contained becomes more likely.
2. The loss is minus the log probability of the correct next token, averaged over the data. No human labels anything: the label is the text's own next token.
3. Backpropagation runs the chain rule backward, reusing saved values, so one backward pass, costing about two forward passes, yields every weight's gradient.

Common misunderstandings of model training

Which claim would you defend right now? Choose before the evidence.

“Training optimizes both the architecture and the weights.”

“Training needs a human to mark the correct answer for every example.”

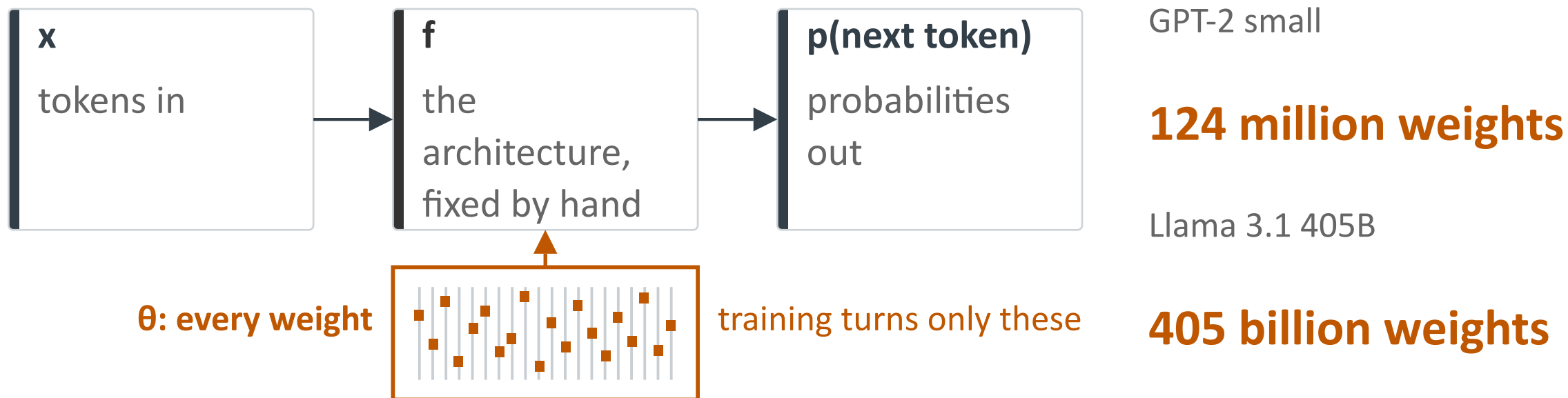
“Getting the gradient for 70 billion weights means measuring the effect of each weight one at a time.”

Keep your choice. Look for the mechanism or counterexample that would make it fail.

The model is a function of its weights

REPRISE

$f(x; \theta)$ maps tokens to next-token probabilities. The architecture fixes the shape of f by hand, before training; training chooses θ , every weight.



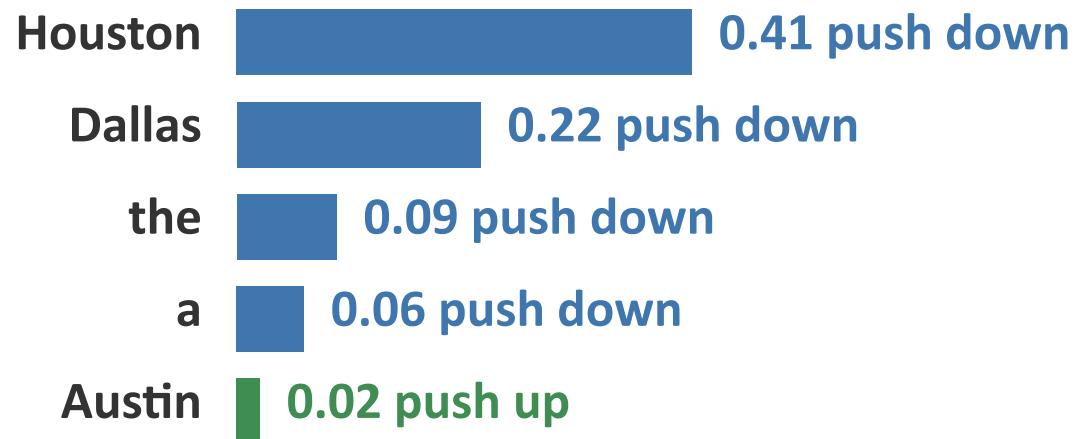
Nothing in what follows changes the box; only the weights move.

The loss: how wrong was the guess?

Cross-entropy loss is $-\log p(\text{correct next token})$. The label is free: it is the token that actually came next in the text.

“The capital of Texas is ...”

an early model’s guesses (illustrative)



One token’s loss

```
loss = -math.log(probs[target])
```

This guess: $p(\text{Austin}) = 0.02$

loss = $-\ln 0.02 = 3.91$

After training: $p(\text{Austin}) = 0.90$

loss = $-\ln 0.90 = 0.11$

Pretraining data labels itself: every token is the answer to the prefix before it.

A dataset's loss is one number

Draw a context x and the token y that followed it from the corpus, and average the per-token loss. Training minimizes L over θ .

$$L(\theta) = \text{average over } (x, y) \sim D \text{ of } \ell(f(x; \theta), y)$$

x a context: the tokens so far

y the next token that actually followed it

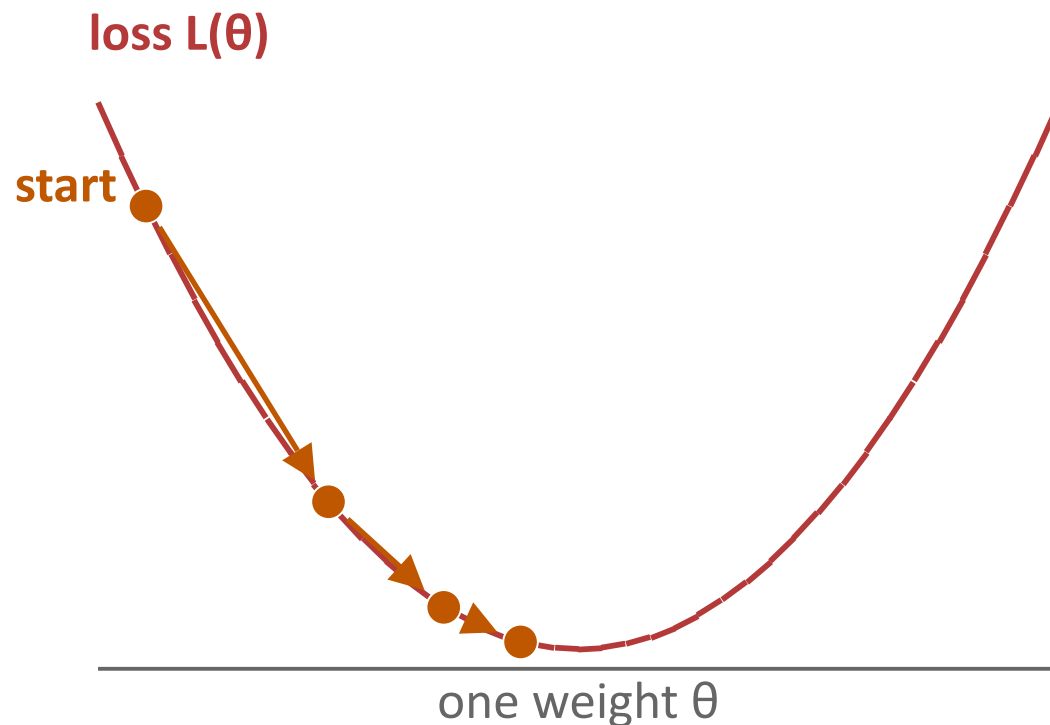
D the corpus, treated as a distribution over (x, y) pairs

ℓ one token's cross-entropy, $-\log p(y)$

L the average: one number for billions of weights against trillions of tokens

Gradient descent: follow the slope

For each weight, the derivative of L says which way is downhill. The gradient ∇L is that direction for all weights at once; the learning rate η is the step size.



$$\theta \leftarrow \theta - \eta \nabla L(\theta)$$

That is the whole algorithm

```
for step in range(T):  
    # downhill, for every weight  
    g = grad(L, theta)  
    # one small step against it  
    theta = theta - lr * g
```

In 405 billion dimensions, the rule is the same.

Stochastic gradient descent: sample a batch

The exact gradient needs a pass over trillions of tokens per step. A random mini-batch gives a noisy but unbiased estimate, so training takes many cheap steps.

Gradient descent

```
for step in range(T):  
    # exact: all of D, every step  
    g = grad(L, theta)  
    theta = theta - lr * g
```

Stochastic gradient descent

```
for step in range(T):  
    batch = sample(D, B)  
    g = grad(L_batch, theta)  
    theta = theta - lr * g
```

Llama 3.1 405B: 4M- to 16M-token batches over 15.6T tokens, about a million steps.

Variants such as momentum and Adam/AdamW rescale each weight's step; the idea is unchanged.

Instapoll

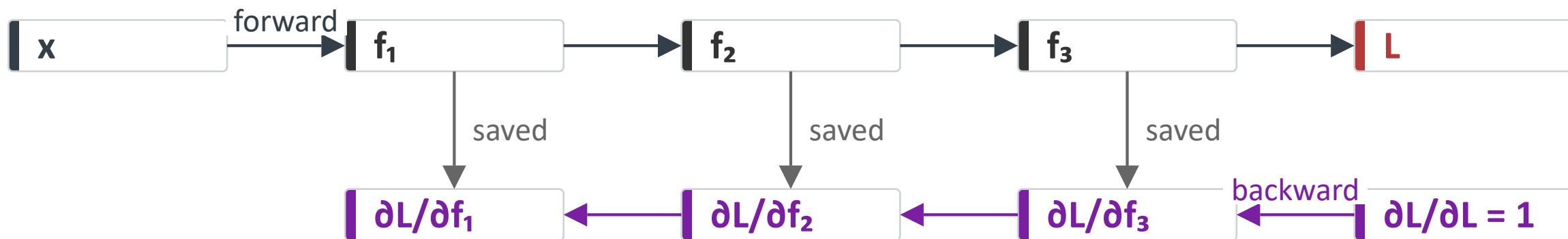
Instapoll · Review · Nov 10 B

A network has 80 layers and 70 billion weights. For one batch of training text, the loss is a single number, and training needs the derivative of that number with respect to every one of the 70 billion weights. Roughly how many times must the network be run to get them all?

- A. About 70 billion times: nudge one weight, re-run, and measure the change in the loss
- B. About 80 times: one run per layer, since the weights in a layer share a derivative
- C. About twice: one forward run to get the loss, one backward run that yields every derivative
- D. Once per training token in the batch, since each token contributes its own loss

Backpropagation: the chain rule, organized

A network is nested functions, $f_3(f_2(f_1(x)))$. The chain rule multiplies local derivatives; going from the output backward reuses every saved forward value.



```
y = w * x + b; loss = (y - t) ** 2
loss.backward()    # one backward pass
w.grad, b.grad     # each weight's share of the loss, in PyTorch
```

One backward pass, about the cost of two forward passes, yields every gradient.

Putting it together: the training loop

Sample a batch, run forward, compute the loss, run backward, step. Given pairs (x, y) drawn from D , it learns an f that maps x to y .

- 1 Sample a batch**
contexts and their next tokens
- 2 Run forward**
a probability for every token
- 3 Compute the loss**
how wrong, averaged
- 4 Run backward**
every weight's gradient
- 5 Step**
nudge every weight, reset

The same loop in PyTorch

```
for x, y in batches(D):  
    logits = model(x)           # 2. forward  
    # 3. how wrong  
    loss = F.cross_entropy(logits, y)  
    loss.backward()             # 4. all gradients  
    opt.step(); opt.zero_grad() # 5. nudge
```

No step writes a rule, changes the architecture, or has a human mark an answer.

What one training run costs

Training costs about 6 floating-point operations per parameter per token: about 2 for the forward pass and about 4 more for the backward pass.

$$6 \times 405 \times 10^9 \text{ parameters} \times 15.6 \times 10^{12} \text{ tokens} \approx 3.8 \times 10^{25} \text{ FLOPs}$$

Forward pass: 2

one multiply and one add
per weight, per token

Backward pass: about 4

derivatives for the activations
and for the weights

Llama 3.1 405B

trained on up to 16,384
H100 GPUs, for months

That is the GPU week's matrix multiplies, run for months. It is why the training data is chosen so carefully (Nov 10): every token costs a forward pass and a backward pass worth two more.

Key Takeaways

1. Training chooses the weights of a function whose shape was fixed by hand. The loss is minus the log probability of the next token the data contained, averaged over the corpus; the text labels itself, so no human marks answers.
2. Gradient descent moves every weight a small step against its derivative of the loss. Stochastic gradient descent estimates that derivative from a random batch, so many cheap steps replace a few exact ones; Adam only rescales each step.
3. Backpropagation runs the chain rule backward, reusing each layer's saved values, so one backward pass worth about two forward passes yields every weight's gradient. A training run therefore costs about 6 FLOPs per parameter per token: 3.8×10^{25} for Llama 3.1 405B.