MicroGPT

Visual Walkthrough

A complete GPT built from scalar autograd in ~200 lines of Python
No tensors. No PyTorch. No CUDA. Just math.

AI Engineering Cohort | TensorTonic
Teaching reference for 200 engineers

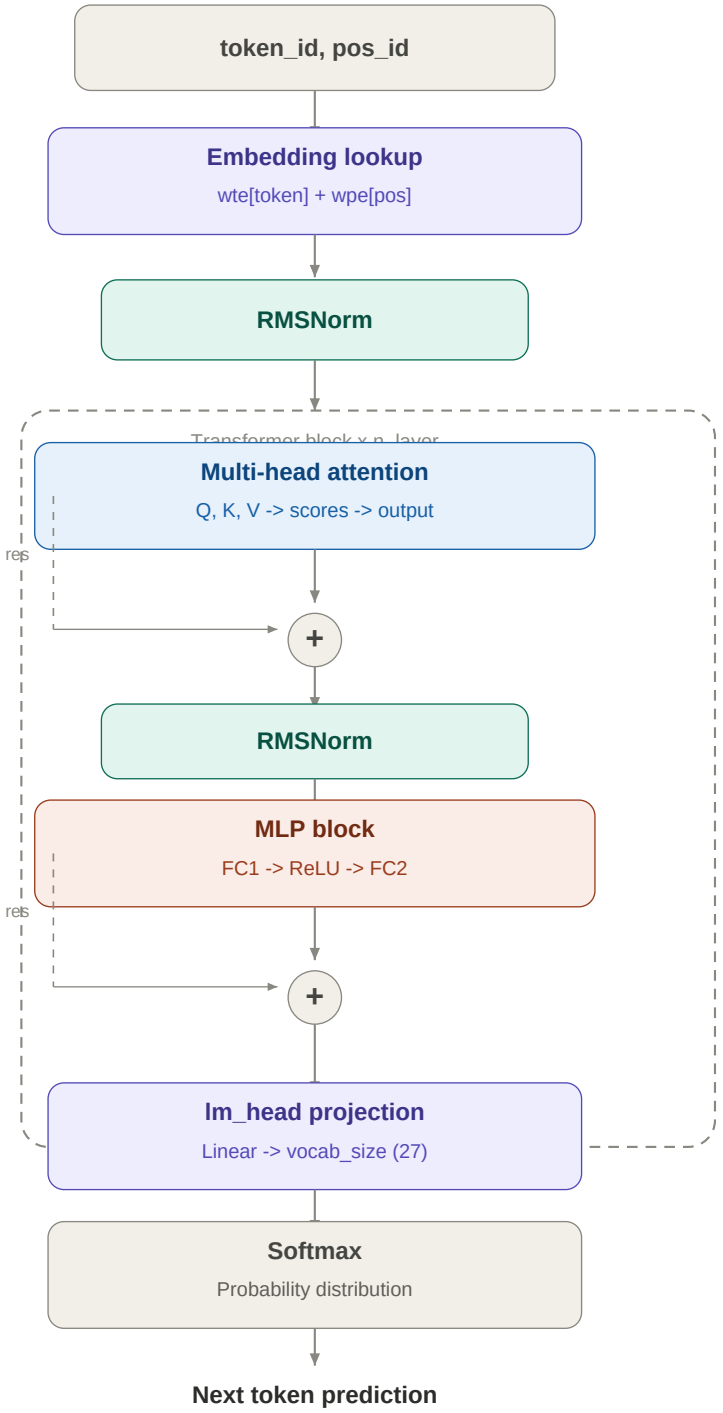
Model configuration

Concept	Value	Code
Vocabulary	27 chars (a-z + BOS)	`chars = sorted(set(''.join(docs)))`
Embedding dim	16	`n_embed = 16`
Attention heads	4 heads x 4-dim each	`n_head = 4, head_dim = n_embed // n_head`
Layers	1 transformer block	`n_layer = 1`
Context window	16 tokens max	`block_size = 16`
Total params	~5K Value objects	`params = [p for matrix in state_dict.values() ...]`
Batch size	8 names per step	`batch_size = 8`
Training steps	20	`num_steps = 20`
Optimizer	Adam (B1=0.85, B2=0.99)	`LR decays linearly 0.01 -> 0`
Init	Gaussian(0, 0.08)	`random.gauss(0, 0.08)`

What's in this document

1. GPT Architecture Overview

The gpt() function — decoder-only transformer, one token at a time




```
gpt(token_id, pos_id, keys, values)
```

```
state_dict['wte'][token_id]
state_dict['wpe'][pos_id]
```

```
rmsnorm(x)
```

```
q = linear(x, wq)
attention_scores -> softmax -> weighted V
```

```
x = [a + b for a, b in zip(...)]
```

```
linear(x, mlp_fc1)
[xi.relu() for xi in x]
linear(x, mlp_fc2)
```

```
linear(x, state_dict['lm_head'])
```

```
softmax(logits)
```

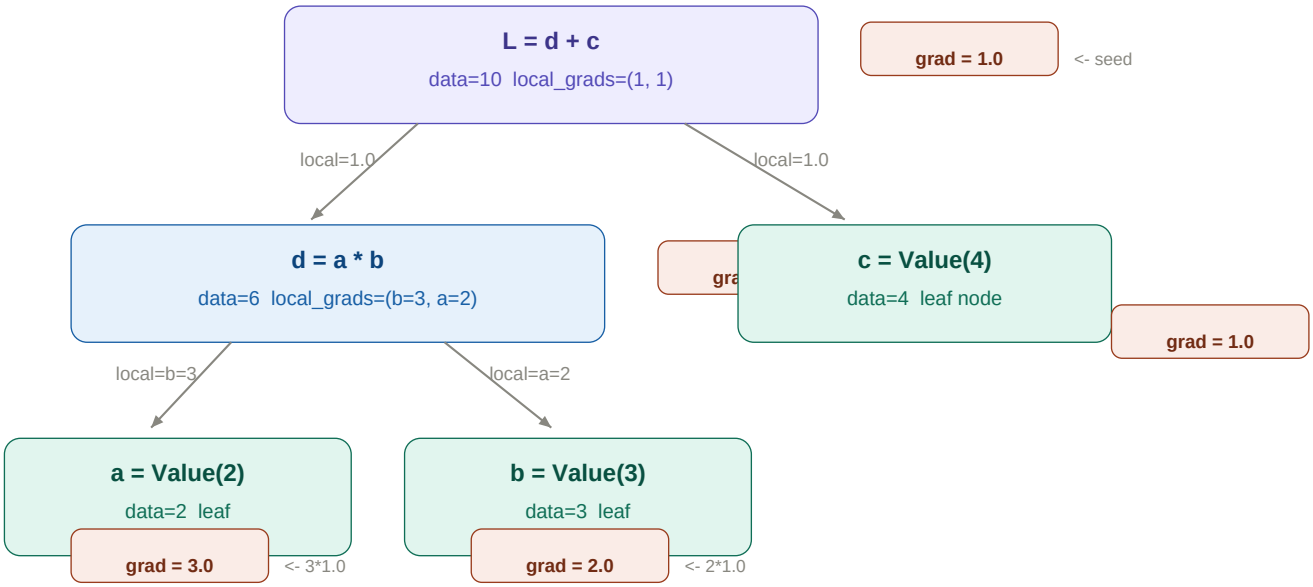
Key insight: gpt() processes one token at a time. The KV cache (keys, values lists) stores previous tokens' representations so attention can look backward without recomputing. This is how the model 'remembers'.

2. Autograd — The Value Class

Scalar computation graph with automatic differentiation via backward()

Every operation builds a DAG node: Value(data, children, local_grads)

Example computation: L = (a * b) + c where a=2, b=3, c=4



backward() — chain rule in 3 lines:

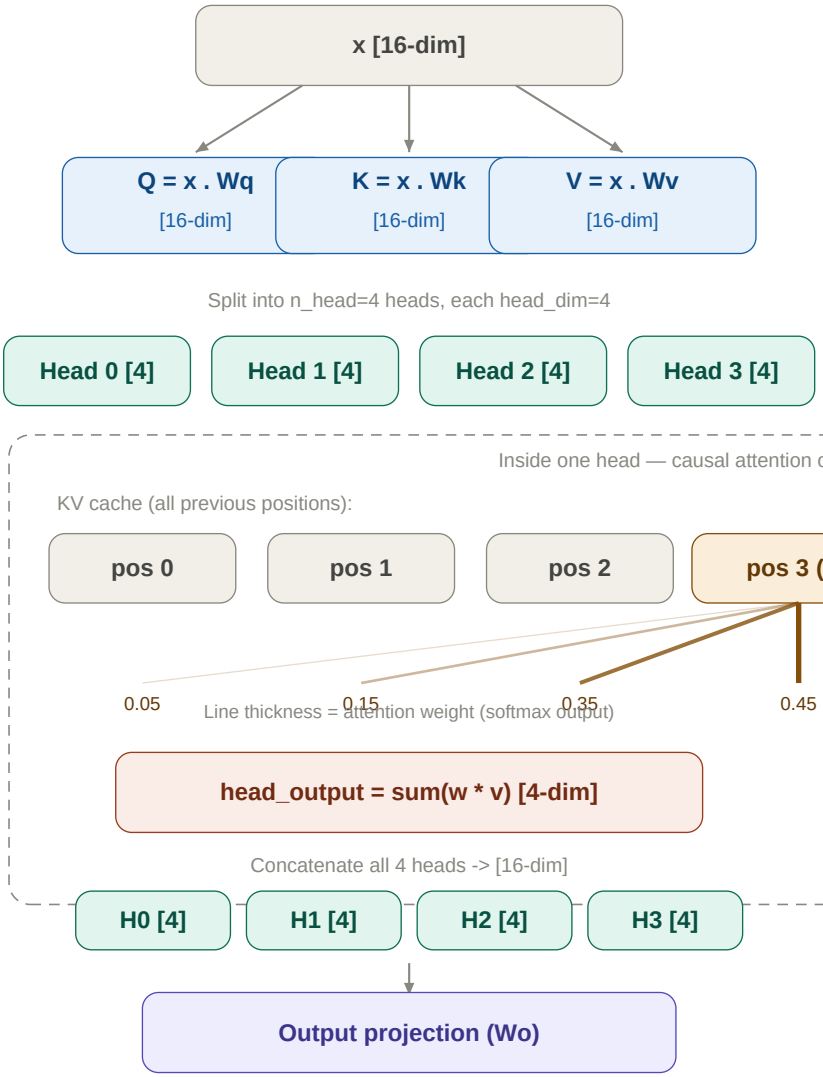
```
for v in reversed(topological_graph):
    for child, local_grad in zip(v._children, v.local_grads):
        child.grad += local_grad * v.grad
```

Operation	Forward	local_grads	Chain rule intuition
`a + b`	`a.data + b.data`	`(1.0, 1.0)`	Addition passes gradient through unchanged
`a * b`	`a.data * b.data`	`(b.data, a.data)`	Multiply swaps: dL/da = b * upstream
`a ** n`	`a.data ** n`	`(n * a^(n-1),)`	Power rule from calculus
`a.log()`	`log(a.data)`	`(1/a.data,)`	d/da log(a) = 1/a
`a.exp()`	`exp(a.data)`	`(exp(a.data),)`	Exponential is its own derivative
`a.relu()`	`max(0, a.data)`	`(float(a > 0),)`	Gradient is 1 if positive, 0 otherwise

Key insight: This is the exact same algorithm PyTorch uses internally — topological sort + chain rule. The only difference is PyTorch operates on tensors (batched), while microgpt.py operates on individual floats (scalar).

3. Multi-Head Attention

Q, K, V projections, head splitting, scaled dot-product attention with KV cache

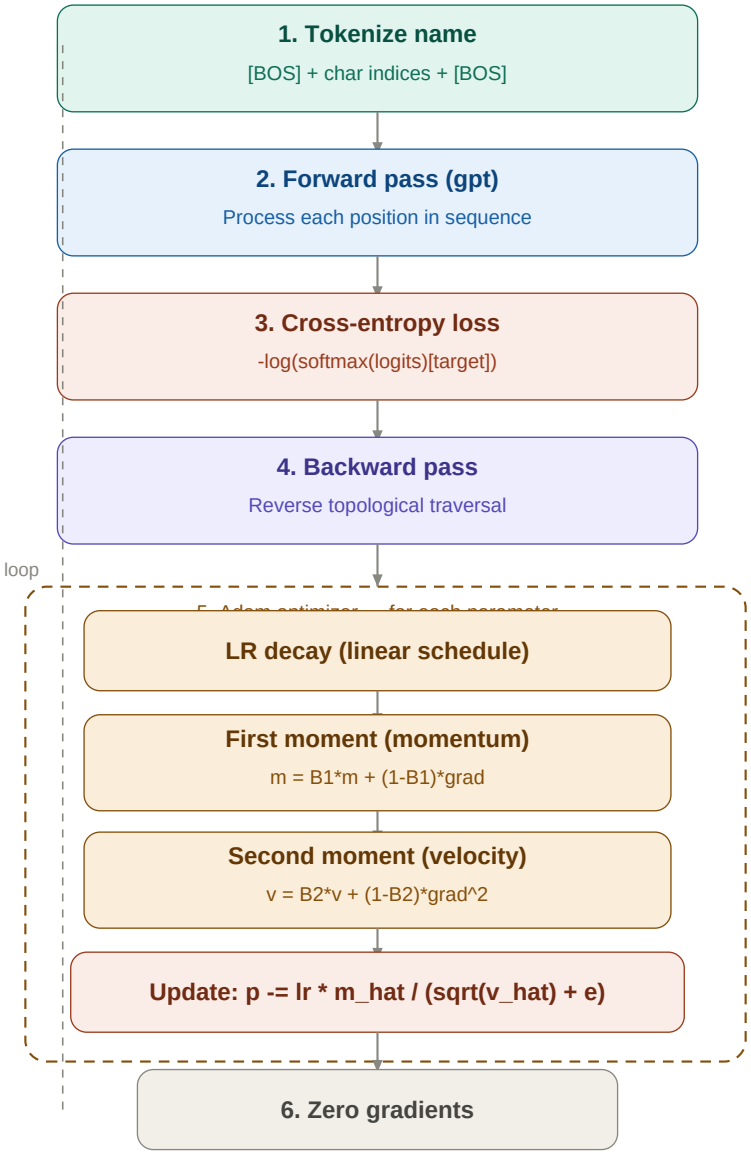


KV cache: keys[i].append(k) and values[i].append(v) store every token's K and V. At inference, the model doesn't recompute past tokens — it just extends the cache. This is exactly how production LLMs work.

4. Training Loop + Adam Optimizer

Forward pass, cross-entropy loss, backward(), parameter update — one complete step

One training step (repeated 20 times, batch_size=8):




```
tokens = [BOS] + [chars.index(c)
         for c in doc] + [BOS]
```

```
logits = gpt(token_id, pos_id,
             keys, values)
```

```
probs = softmax(logits)
losses.append(-probs[target_id].log())
```

```
loss.backward()
```

```
lr_target = lr * (1 - step/num_steps)
```

```
m[i] = beta1*m[i] + (1-beta1)*grad
m_hat = m[i] / (1 - beta1**(step+1))
```

```
v[i] = beta2*v[i] + (1-beta2)*grad**2
v_hat = v[i] / (1 - beta2**(step+1))
```

```
param.grad = 0
```

Why Adam? First moment (m) = exponential moving average of gradients (direction). Second moment (v) = EMA of squared gradients (magnitude). Bias correction compensates for initialization at zero in early steps.

20 steps x 8 names per batch = 160 training examples total. Loss typically drops from ~3.3 to ~2.8 in this tiny training run.

5. Embedding + Positional Encoding

Learnable lookup tables: what the token IS + where the token SITS

Example: processing "emma" — input tokens:

BOS	e	m	m	a	BOS
26	4	12	12	0	26

token_id = 4 ('e'), pos_id = 1

```
[BOS] + [chars.index(c) for c in 'emma'] + [BOS]
```

wte [27 x 16] — Token embedding table

Learnable. Each row = one character's 16-dim vector.

- row 0 (a)
- row 1 (b)
- ...
- **row 4 (e) -> [0.02, -0.05, 0.11, ...]**
- ...
- row 25 (z)
- row 26 (BOS)

```
state_dict['wte']  # [vocab_size x n_embed]
state_dict['wte'][token_id]  # row lookup
# vocab_size = 27 (a-z + BOS)
```

wpe [16 x 16] — Position embedding table

Learnable. Each row = one position's 16-dim vector.

- row 0 (position 0)
- **row 1 (position 1) -> [0.08, 0.01, ...]**
- ...
- row 15 (position 15)

[16-dim] [16-dim]

```
state_dict['wpe'][pos_id]  # position lookup
```

+

Element-wise addition

```
[te + pe for te, pe in zip(...)]
```

x = token + position [16-dim]

Input to RMSNorm -> transformer

Both tables are learnable: wte and wpe start as random Gaussian noise (mean=0, std=0.08) and are trained via backprop like every other parameter. No sin/cos positional encoding — fully learned, which works for block_size=16.

Bonus: Inference (generation)

1. Start with token_id = BOS
2. Run gpt() to get logits
3. Apply temperature scaling (T=0.5) and softmax
4. Sample next token from probability distribution
5. If sampled BOS -> stop, else append char and repeat
6. Output: generated name (e.g. 'aalon', 'zerah', etc.)

```
probs = softmax([l / temperature for l in logits])
token_id = random.choices(range(vocab_size), weights=[p.data for p in probs])[0]
```

Temperature < 1 makes the distribution sharper (more confident). Temperature > 1 makes it flatter (more creative/random).

Quick Reference Cheat Sheet

The complete microgpt.py architecture on one page

Architecture flow

Input	token_id (0-26), pos_id (0-15)
Embed	wte[token] + wpe[pos] -> [16-dim]
Norm	RMSNorm(x)
Attn	Q,K,V -> 4 heads -> concat -> Wo
Residual	x = attn_out + x_residual
Norm	RMSNorm(x)
MLP	FC1(64) -> ReLU -> FC2(16)
Residual	x = mlp_out + x_residual
Project	lm_head: [16] -> [27] logits
Output	softmax -> next token prob

Parameter count breakdown

`wte`	27 x 16 = 432
`wpe`	16 x 16 = 256
`Wq, Wk, Wv, Wo`	4 x (16x16) = 1024
`mlp_fc1`	64 x 16 = 1024
`mlp_fc2`	16 x 64 = 1024
`lm_head`	27 x 16 = 432
TOTAL	**~4,192 scalars**

Key functions

`linear(x, w)`	Matrix-vector multiply (list comprehension)
`softmax(logits)`	exp(x-max)/sum with numerical stability
`rmsnorm(x)`	x / sqrt(mean(x^2) + eps)
`gpt(...)`	Full decoder pass, returns logits
`Value.backward()`	Reverse-topo chain rule

Training recipe

Dataset: 32K names from Karpathy's makemore

Tokenizer: character-level (a-z + BOS = 27 tokens)

Loss: cross-entropy (next character prediction)

Optimizer: Adam (beta1=0.85, beta2=0.99)

Learning rate: 0.01 with linear decay to 0

Batch size: 8 names per step

Steps: 20 (very small — for demo purposes)

Init: Gaussian(mean=0, std=0.08) for all weights

Attention math (per head)

1. Q = linear(x, Wq) -> split by head
2. K = linear(x, Wk) -> append to cache
3. V = linear(x, Wv) -> append to cache
4. scores = Q . K^T / sqrt(head_dim)
5. weights = softmax(scores) # causal: sees past only
6. output = weights . V # weighted sum
7. Concat all heads -> output projection Wo

Adam update (per parameter)

```
m = 0.85*m + 0.15*grad          # momentum
v = 0.99*v + 0.01*grad^2        # velocity
m_hat = m / (1 - 0.85^t)        # bias correct
v_hat = v / (1 - 0.99^t)        # bias correct
param -= lr * m_hat/(sqrt(v_hat)+1e-8)
param.grad = 0                   # zero grad
```

Why this file matters

Everything is scalar — no tensor abstraction hiding anything

Same architecture as GPT-2 / GPT-3 (just tiny)

Same autograd as PyTorch (topo sort + chain rule)

Same Adam optimizer (moment estimates + bias correction)

Same KV cache trick used in production LLMs

~200 lines = the whole story from scratch