

Assemble the GPT: stacking blocks into a model

A complete GPT is a chain of familiar operations, not a new mysterious ingredient. Turn token IDs into vectors, add position vectors, pass the result through independent transformer blocks, normalize the result, and score the vocabulary. Today you will inspect that entire chain and prove several things about it. You will not train it yet. A correctly assembled predictor and a useful predictor are different achievements.

Allow roughly 65 minutes for reading, running the lab and explaining the result. That is a study estimate, not a measured completion time. The actual computation is much shorter.

Why this matters

In the previous chapter, you built a transformer block that preserves its input shape. Attention mixes information between permitted token positions. The feedforward network transforms each position separately. Residual additions keep an existing representation available while adding updates, and layer normalization controls the representation supplied to each sublayer. Here we treat that block as a component with a known contract, then connect the input and output machinery around a stack of blocks.

This is the point where the pieces become a language model. It is also a useful boundary in engineering. If a model produces poor text, the fault might be an incorrect computation, an incorrectly prepared dataset, an optimization problem, or simply insufficient learning. Without a tested forward pass, those explanations become tangled. With one, you can test the wiring before blaming the data or the optimizer.

The same distinction matters when inspecting an AI system at work. The neural network produces scores for tokens. A surrounding application decides how to select tokens, maintain conversation state, retrieve documents and execute tools. Nothing in the forward pass we build opens a file or runs a command. Understanding that boundary helps you ask precise questions about what belongs to model behavior and what belongs to application permissions.

Our aim is therefore stronger than printing something that resembles text. We will show where every tensor goes, count the unique learned numbers, check that future inputs cannot change past predictions, and verify that a saved set of weights restores the same computation. These checks are useful even before the model has learned anything.

The idea in plain words

Imagine a row of workspaces, one workspace per token position. Each starts with information about which token occupies it and where it sits. A block lets a workspace read allowed earlier workspaces, including itself, then perform its own local computation. Another block receives the updated workspaces and repeats the process with different learned weights. Finally, a shared scoring rule reads each workspace and gives a score to every possible next token.

Here is the concrete lab configuration. The vocabulary is `abcdefghijklmnopqrstuvwxyz.`, with 27 entries. We use 2 sequences at once, each containing 6 input tokens. Each position has a vector with 8 coordinates. The model has 3 blocks, each containing 2 attention heads. After every block, the representation still has shape `(2, 6, 8)`. Only the output head changes the final dimension, producing `(2, 6, 9)`.

Those dimensions mean separate things. The first counts independent examples. The second counts positions within each example. The third counts either internal features or vocabulary scores, depending on which stage you are inspecting. Confusing a position with a feature is an easy way to write code that runs but computes the wrong thing.

The final distribution for the first sequence assigns `0.121568` to `a` and `0.118569` to the period in our recorded run. These are not discoveries about language. The weights are untrained. They are merely a demonstration that the full computation produces a valid distribution. Valid probabilities are a structural property; accurate probabilities require evidence from training and evaluation.

How it actually works

The shape contract

We use these symbols throughout:

- `B` is batch size, the number of independent sequences processed together.
- `T` is the number of input positions in each sequence for this call.
- `C` is the width of the internal token representation.
- `V` is vocabulary size, the number of possible token IDs.
- `S` is the configured maximum context length, called `block_size` in the code.
- `L` is the number of transformer blocks. `H` is the number of attention heads inside each block.

The input is an integer tensor of shape `(B, T)`. An entry is an index into a table, not a continuous feature. Our first row represents `bad ca`. The corresponding targets represent `ad ca.`. At each input position, the target is the next character from the original string `bad ca.`. The final period is a target, not an extra input position in this call.

This alignment is essential. If you accidentally use the input itself as the target, the model can learn to repeat the token it already sees. That would be a different task, not next-token prediction. Our existing GPT implementation does not shift the targets for you. The lab explicitly makes both slices from the original sequence.

Token identity plus position

Let `E` be the token embedding table, with shape `(V, C)`. Let `P` be the learned position table, with shape `(S, C)`. If `i[b,t]` is the token ID at batch row `b` and position `t`, the initial representation is:

$$x[b,t,:] = E[i[b,t],:] + P[t,:]$$

The colon means all feature coordinates. The square brackets mean indexing. The plus sign is coordinate-by-coordinate addition, not concatenation. The position lookup has shape `(T, C)` and broadcasts across the batch because every sequence uses the same position numbers for this call. The result keeps shape `(B, T, C)`.

In the first input, `a` appears at positions 1 and 5, using zero-based indexing. Looking up the same token twice gives identical vectors, so the lab measures a maximum coordinate difference of `0.0000000000` before positions are added. After adding position vectors, the maximum difference is `0.061671633`. The token identity is the same, but the initial representation now contains a different location signal.

Position vectors are learned parameters, just like token vectors. They are not necessarily increasing ramps or human-readable labels after training. They give the model an explicit source of location information that it can use. Adding them does not guarantee that every difference in order will produce a meaningful difference in output.

There is an important qualification to the usual phrase “attention is order blind.” Unmasked content-only self-attention has a permutation symmetry when you reorder its inputs and outputs together. A causal mask already makes positions different by changing which earlier positions are visible. Therefore, removing learned position vectors from a causal model does not prove that all order information disappears. Our narrower, correct claim is that this GPT design supplies an explicit learned absolute-position signal at its input.

Stack independently parameterized blocks

Write one block using two intermediate equations:

```
u = x + Attention(LayerNorm1(x))
y = u + MLP(LayerNorm2(u))
```

`x` is the block input. `u` is the representation after the attention update. `y` is the block output. Both residual additions combine tensors of the same shape. The first normalization and the second normalization have their own learned scale and shift. Attention combines allowed positions; the MLP uses the same learned function separately at each position.

All these input and output representations have width `C`. The MLP expands internally to `4*C`, applies GELU, and projects back to `C`. Multi-head attention splits its projected features among `H` heads, each of width `C/H`, then recombines them into width `C`. The division must be exact, which the implementation checks.

Stacking means passing one block's output into the next block's input. It does not mean concatenating the outputs. The sequence does not become longer and the representation does not become wider just because another block is added. What increases is the number of transformations and, in this implementation, the number of independent parameters.

Our blocks are created using `nn.ModuleList(Block(cfg) for _ in range(cfg.n_layer))`. Each constructor call creates a distinct block. Repeating a reference to the same block would share weights across depth instead, a different architecture. `ModuleList` also registers the modules with PyTorch, so their parameters appear in the model's parameter iterator and state dictionary. A plain Python list alone does not provide that registration. See the official `ModuleList` documentation in Sources.

Normalize, then read out the vocabulary

After the stack, `ln_f` applies a final layer normalization over the feature coordinates at each position. It does not mix sequence positions. Layer normalization subtracts that row's feature mean, divides by the square root of its feature variance plus a small stabilizing epsilon, then applies learned coordinatewise

scale and shift. Because of epsilon and those learned affine parameters, “the output always has exactly unit variance” is not a universal guarantee.

Let $h[b,t,:]$ be that normalized final representation. The output head uses a matrix with one row per vocabulary entry. In this implementation, that matrix is the very same parameter as the token embedding table E :

```
z[b,t,v] = sum over c of h[b,t,c] * E[v,c]
p[b,t,v] = exp(z[b,t,v]) / sum over j of exp(z[b,t,j])
```

z is a logit, an unrestricted real score. v selects the candidate vocabulary token. c ranges over feature coordinates. j ranges over all vocabulary entries in the denominator. p is a probability after softmax. The denominator normalizes over candidates for one position, not over positions or examples.

Sharing the input and output table is called weight tying. It saves a separately learned output matrix and lets the table participate in both reading and scoring. It does not invert the earlier computation or recover the original input token. The hidden representation has been transformed by positions and blocks, and the output task asks for a successor, not a reconstruction of the current ID.

Work one head calculation by hand

For a deliberately hand-sized example, use a representation $h = [1, 2]$ and these candidate rows:

```
candidate 0: [1, 0]
candidate 1: [0, 1]
candidate 2: [1, -1]
```

The first score is $1*1 + 2*0$, which gives 1. The second is $1*0 + 2*1$, which gives 2. The third is $1*1 + 2*(-1)$, which gives -1. This is a small illustration of the head operation, not a claim that these are hidden states from our randomly initialized model. Part 3 executes this exact example using double-precision tensors.

Apply softmax to those scores. For the second candidate, its probability is $\exp(2)/(\exp(1)+\exp(2)+\exp(-1))$. The measured probabilities are $[0.259496, 0.705385, 0.035119]$. If candidate index 1 is the target, the negative logarithm of its probability is 0.349012. A good prediction receives a smaller penalty than a bad prediction under this objective.

The full model computes this kind of scoring operation at every batch row and position. During training, cross entropy compares each vocabulary-score row with its shifted target ID. PyTorch expects the raw logits and performs the logarithmic normalization internally. Applying softmax before `cross_entropy` would feed the wrong kind of input into that operation.

Count parameters without counting storage twice

For this exact implementation, each block contains attention projection matrices, MLP matrices, their biases and two affine layer normalizations. The attention weights contribute $4*C*C$: the combined query, key and value matrix contributes $3*C*C$, and the output projection contributes $C*C$. The two MLP matrices contribute $8*C*C$ together. Linear biases contribute $9*C$; the two normalization layers add $4*C$. Thus the per-block total is $12*C*C + 13*C$.

At our configured width, the lab reports 872 parameters per block and 2616 across the stack. The shared token-table/head parameter contains 72 numbers. The position table contains 48. The final normalization contains 16. The complete count is 2752, matching `sum(p.numel() for p in model.parameters())` exactly.

The masks also occupy storage, but they are buffers, not learned parameters. This lab reports 108 mask buffer elements. Likewise, the state dictionary contains names for both uses of the tied table, but those names do not imply independent learned weights. Count unique parameters when describing model capacity, and distinguish parameter count from total runtime memory. Activations, masks and optimizer state are separate concerns.

Causality survives composition

At a position `t`, the embedding uses only the token at `t` and its position. Each causal attention sublayer reads only positions up to `t`. The normalizations and MLP act within individual positions. Residual addition does not bring in another position. Therefore, if a block receives representations that contain no future information, it cannot introduce future information itself. Applying that argument repeatedly covers the entire stack and the final head.

This reasoning motivates an executable test. Edit only a suffix of the input, keeping the earlier prefix unchanged. Earlier logits must remain unchanged within numerical tolerance. Do not merely check that probabilities sum to one: a model that leaks future tokens can still produce perfectly normalized probabilities.

The lab changes positions from index 3 onward. Its earlier logits have maximum difference 0.000000000; the changed suffix has maximum gap 0.151288509. The positive suffix difference matters because it confirms we actually changed a computation that affects outputs, rather than accidentally comparing identical inputs.

Build it

Run from the project root:

```
env -u PYTHONPATH .venv/bin/python chapters/llm-fundamentals/07-assemble-the-gpt/lab.py
```

The lab finds `labs/` relative to its own location, so an absolute invocation also works from another working directory. It uses the existing `tutorlab.gpt` components rather than introducing an incompatible second GPT. The new `tutorlab.gpt_assembly` module exposes the intermediate tensors and an exact parameter budget for that implementation. The companion tests live in `labs/tests/test_gpt_assembly.py`.

Everything runs on CPU without downloads, credentials or a pretrained checkpoint. The random seed fixes initialization for this demonstration, dropout is disabled, and deterministic algorithms are requested. Tiny floating-point differences can still depend on software or hardware, so behavioral assertions use tolerances where appropriate. The recorded output below is evidence from this environment, not a promise of bitwise identity on every machine.

Part 1: prepare aligned inputs and targets

The lab encodes the strings `bad ca.` and `cab ad.` using the fixed toy vocabulary. It makes inputs from all but the final character and targets from all but the first. Calling `.contiguous()` materializes slices in a

layout that the existing GPT loss path can flatten with `.view()`. This is a storage-layout detail, not a change to the data.

We deliberately use a batch rather than a single sequence. It makes the three-dimensional shape contract visible, and later lets us test that running sequences together does not cause them to exchange information. No padding is needed because the examples have equal lengths.

Part 2: expose the complete forward pass

The essential assembly in `trace_forward` is short:

```
pos = torch.arange(idx.shape[1], device=idx.device)
trace = {'tokens': model.tok_emb(idx), 'positions': model.pos_emb(pos)}
x = model.drop(trace['tokens'] + trace['positions'])
trace['residual_in'] = x
for i, block in enumerate(model.blocks):
    x = block(x)
    trace[f'block_{i}'] = x
trace['normalized'] = model.ln_f(x)
trace['logits'] = model.head(trace['normalized'])
```

The dictionary retains the tensors so you can inspect every stage. `arange` creates the position IDs on the same device as the inputs. Broadcasting supplies the position table to each batch row. The loop updates `x`, making the blocks a chain rather than independent parallel branches.

We compare this explicitly assembled result with `GPT.forward` on the same model. That is an integration check: both routes must compute the same logits. It is not an independent proof that every attention equation is correct because the routes reuse the same underlying components. The causal, shape, gradient and serialization checks provide additional evidence about distinct contracts.

Part 3: inspect scores without inventing competence

Softmax runs over the final dimension of the logits. The lab verifies finite, nonnegative probabilities and a normalized row for every position. It prints the last row of the first sequence because that is the distribution you would use to choose a token after the complete input prefix. Earlier rows predict successors to shorter prefixes.

The random model's next-token loss is `2.213514`. A strictly uniform distribution over this vocabulary would have loss $\ln(9) = 2.197225$. Being near that baseline is unsurprising here, but we do not assert that every random initialization must be uniform or achieve a particular loss. Tied weights and residual paths can produce preferences even before training.

Part 4: preserve the computation

The checkpoint test writes `model.state_dict()` into an in-memory byte stream and loads it into a fresh GPT with the same configuration. This is a real serialization round trip with `torch.save` and `torch.load`, not a direct assignment pretending to be a save. It intentionally creates no file outside the chapter.

Loading matching weights is not enough if the architecture differs. The receiving model must use the same configuration and preserve the tied input/output parameter relationship. We check both the

identical logits and the shared-parameter identity after restoration. For a reusable checkpoint on disk, also preserve the configuration and vocabulary ordering. The state dictionary alone does not tell you which character each numeric ID means.

Part 5: test behavior across the model boundary

The suffix intervention checks no future leakage. Running only the prefix checks consistency with the prefix rows from a longer call. Running one batch row alone checks that another example cannot influence it. These comparisons use evaluation mode. With active dropout, separate calls could differ from stochastic masks rather than from a causal bug.

The lab then computes loss and runs backward without taking an optimizer step. Every parameter must have a present, finite gradient, and the attention projection in every block must receive a nonzero gradient norm. This shows that the learning signal can reach those parts of the model. It does not establish that training will converge or generalize.

Finally, a sequence longer than the configured context must fail rather than silently use an invalid position lookup. Cropping context, extending position representations and generating more text are deliberate policies, not interchangeable behaviors. Later chapters cover training and generation; here we test the forward contract and stop.

Recorded output

The following is copied from `build/lab-output.txt`; it is the actual lab output, not manually predicted output.

```
Part 1: inputs and next-token targets
  vocabulary='abcdefg .'; B=2 T=6 C=8 V=9 layers=3 heads=2
  inputs=[[1, 0, 3, 7, 2, 0], [2, 0, 1, 7, 0, 3]]
  targets=[[0, 3, 7, 2, 0, 8], [0, 1, 7, 0, 3, 8]]

Part 2: shapes through the assembled GPT
  tokens      (2, 6, 8)
  positions   (6, 8)
  residual_in (2, 6, 8)
  block_0     (2, 6, 8)
  block_1     (2, 6, 8)
  block_2     (2, 6, 8)
  normalized  (2, 6, 8)
  logits      (2, 6, 9)
  explicit assembly vs GPT.forward max error=0.000000000
  repeated a: token-vector difference=0.000000000
  repeated a: after positions difference=0.061671633

Part 3: output distributions (untrained)
  last-row probabilities: a:0.121568 b:0.109120 c:0.106267 d:0.105326 e:0.111535 f:0.118241
  g:0.097069 _:0.112305 .:0.118569
  row sums min=1.000000 max=1.000000
  next-token loss=2.213514; uniform baseline ln(9)=2.197225
```

```
hand example: h=[1.0, 2.0] table=[[1.0, 0.0], [0.0, 1.0], [1.0, -1.0]]
hand logits=[1.0, 2.0, -1.0]
hand probabilities=[0.259496, 0.705385, 0.035119]
hand target index=1 loss=0.349012
```

Part 4: unique learned numbers

```
token_table_and_tied_head=72
position_table=48
blocks=2616
final_norm=16
per block=872
total=2752; formula=2752
tied head is same Parameter=True
mask buffer elements (not parameters)=108
checkpoint round-trip max error=0.000000000
```

Part 5: overlong input rejected: sequence 7 longer than block_size 6

```
suffix edit: past max error=0.000000000; changed suffix max gap=0.151288509
prefix-only max error=0.000000000; separate batch-row max error=0.000000000
attention gradient norms=[0.049288, 0.042088, 0.041107]
all parameter gradients present and finite=True
optimizer steps=0; no language competence claimed
```

CHECK PASSED

Predict before trying an edit. Reducing the number of blocks changes the parameter count, but should it change the final tensor shape? Increasing the head count while holding the width fixed changes how attention divides features, but does it create additional projection parameters here? Changing only the final input ID should affect which output rows? The [TRY THIS](#) comments identify these experiments. Keep the original output as a baseline and change one thing at a time.

Common mistakes and misconceptions

"Three blocks generate three new tokens." Blocks add depth to one forward computation. Every block processes the existing sequence and preserves its length. Choosing and appending a new token is a separate operation around the model.

"A probability distribution proves the model works." It proves only a limited structural fact. Normalized scores can be random, inaccurate or based on leaked future information. Causal tests, learning tests and held-out evaluation answer different questions and should not be conflated.

"Position embeddings make the feature vector longer." We add equal-width vectors rather than concatenate them. The feature dimension stays c . Concatenation would require changing downstream shapes and parameters; it is not what this GPT implements.

"The output head returns a token." It returns a vector of logits for every input position. Softmax converts a score vector to probabilities. Sampling or an argmax policy makes a token choice later. Returning all positions is convenient for training and inspection; generation normally needs the final position's distribution.

"Weight tying means all blocks share weights." Only the named token embedding and output head share a parameter here. Each transformer block has its own parameters. Shared shapes are not shared storage, and shared storage must be checked explicitly.

"Evaluation mode means no gradients." Evaluation mode changes behavior such as dropout. Disabling gradient recording is a separate choice, such as entering `torch.no_grad()`. The lab intentionally runs backward while the model is in evaluation mode. That is valid and useful for a deterministic connectivity check.

"Layer normalization uses statistics from other examples." Our `LayerNorm(C)` uses feature statistics within each position, not statistics across the batch. That is one reason individual examples can be evaluated independently and match their rows from a batched computation.

In the real world

The implementation comparison was checked on 2026-10-10 against Karpathy's public nanoGPT `model.py`. It uses learned token and position embeddings, a registered block stack, pre-normalization residual blocks, a final normalization and a tied output head. These are direct code observations, not inferred details of a proprietary hosted model. The repository is linked in Sources so you can inspect the definitions yourself.

There are also differences worth noticing. nanoGPT can use PyTorch's scaled dot-product attention operation, whereas our shared teaching implementation writes the attention matrix operations explicitly. nanoGPT applies a depth-dependent initialization adjustment to residual projection weights; our shared implementation uses its simpler initialization. nanoGPT's no-target forward path projects only the final position to vocabulary scores, while our teaching GPT returns every position regardless of whether targets are supplied. Those are deliberate scope differences, so do not treat this lab as a bit-for-bit reproduction or a speed benchmark.

The GPT-2 paper's architecture section describes moving layer normalization to the input of sub-blocks and adding a final layer normalization. That historical source explains why this pre-normalization arrangement appears in a GPT-style model. It does not imply that every later language model keeps every detail. Position representations, normalization choices, attention arrangements and feedforward designs can change while the broad "contextual vectors to vocabulary scores" idea remains useful.

PyTorch's `ModuleList`, `LayerNorm` and `CrossEntropyLoss` documentation establishes the library contracts used here. We inspected those documents on 2026-10-10. In particular, module registration matters for optimizers and checkpoints, normalization dimensions matter for independence, and raw-logit loss inputs matter for correct learning. A surprising training failure can come from violating one of these ordinary programming contracts rather than from a subtle mathematical issue.

For your own systems, keep the scope of the claim narrow. This lab demonstrates a small causal language-model computation. It does not reproduce a chat service, establish instruction-following behavior, or measure whether an agent is safe to grant tools. Those are additional systems and evaluation questions. Separating them makes both engineering and safety reviews clearer.

Go deeper

The moments below are recorded in `lectures.json`. YouTube transcript retrieval was IP-blocked during preparation, so their anchors were verified against the creator-published chapter markers, not against

spoken transcript excerpts. The titles locate relevant sections; they are not presented as quotations from speech.

- Andrej Karpathy, "Let's reproduce GPT-2 (124M)," at 13:47: <https://www.youtube.com/watch?v=l8pRSuU81PU&t=827>. Watch the implementation of the GPT-2 module and identify where components are registered. Pause when a block is constructed and distinguish creating a new instance from reusing an existing one.
- The same lecture at 31:00: <https://www.youtube.com/watch?v=l8pRSuU81PU&t=1860>. Follow the forward pass to logits. Annotate every tensor with its meaning rather than merely copying its variable name.
- The same lecture at 66:14: <https://www.youtube.com/watch?v=l8pRSuU81PU&t=3974>. Study parameter sharing between the input table and language-model head. Compare the storage identity in that explanation with the lab's explicit identity assertion.
- Karpathy, "Let's build GPT," at 60:18: <https://www.youtube.com/watch?v=kCc8FmEb1nY&t=3618>. Revisit positional encoding if it is still unclear why a token lookup alone gives the same initial vector every time the token occurs.

For reading, compare the GPT-2 paper's architecture section with the nanoGPT source, then check the PyTorch documentation for whichever line you cannot explain. The useful habit is moving between equations, executable code and measured behavior until all three describe the same thing.

Explain it back

Without looking at the code, explain how `bad ca` becomes a distribution over the next character. Include what position addition changes, why the blocks can be stacked, what the final head computes, and why passing the probability checks does not show that the model has learned language. Use the lab's shapes as anchors, not as a substitute for explaining their meaning.

Your study partner should identify one gap in your explanation and ask one follow-up, rather than treating a long answer as proof of understanding. Then apply the idea to a new case: an engineer reports that editing only a later suffix changes logits for an unchanged earlier prefix. Describe which contract is broken and which components you would inspect first. Mastery requires both an accurate explanation and a correct application; running the lab alone does not establish it.

Check yourself

Use `quiz.json` for the questions, answer keys and explanation rubrics. Keep answers hidden until you have made a prediction or attempted an explanation. The check covers:

1. What changes, and what stays fixed, when position embeddings are added?
2. Which dimension changes at the final output head?
3. Why is `ModuleList` useful, and why must each block be constructed independently?
4. What do valid softmax probabilities establish about an untrained model?
5. Which parameter is shared by weight tying in this implementation?
6. If only the final input token changes, which output rows are allowed to change?
7. If the head count changes while total width stays fixed, what happens to the parameter count here?
8. Explain the complete path from IDs to next-token scores, including why stacking preserves shape.

9. Diagnose future leakage and distinguish it from dropout-induced differences between calls.
10. Explain why using inputs as targets is not the next-token task and how this lab aligns the data.

The recall cards focus on shape contracts, raw logits versus probabilities, weight tying and whole-model causality. They are material for later review, not evidence that you already know the answers.

Sources

- KrypteiaSec Academy, chapter 7, coverage baseline inspected 2026-10-10:
<https://krypteiasec.com/academy/llm-fundamentals/ch7>
- KrypteiaSec Academy, chapter 6, prerequisite block coverage:
<https://krypteiasec.com/academy/llm-fundamentals/ch6>
- Andrej Karpathy, nanoGPT model implementation, assembly and initialization comparison inspected 2026-10-10: <https://github.com/karpathy/nanoGPT/blob/master/model.py>
- Radford and colleagues, "Language Models are Unsupervised Multitask Learners," section 2.3, GPT-2 architecture: https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf
- PyTorch, ModuleList registration contract, inspected 2026-10-10:
<https://docs.pytorch.org/docs/2.14/generated/torch.nn.ModuleList.html>
- PyTorch, LayerNorm dimensions, epsilon and learned affine transform, inspected 2026-10-10:
<https://docs.pytorch.org/docs/2.14/generated/torch.nn.LayerNorm.html>
- PyTorch, CrossEntropyLoss raw-logit input contract, inspected 2026-10-10:
<https://docs.pytorch.org/docs/2.14/generated/torch.nn.CrossEntropyLoss.html>
- Andrej Karpathy, "Let's reproduce GPT-2 (124M)," module construction, forward pass and parameter sharing: <https://www.youtube.com/watch?v=l8pRSuU81PU>
- Andrej Karpathy, "Let's build GPT: from scratch, in code, spelled out," positional encoding and block construction: <https://www.youtube.com/watch?v=kCc8FmEb1nY>
- Local executable evidence: `lab.py`, `build/lab-output.txt`, `labs/tutorlab/gpt.py`, `labs/tutorlab/gpt_assembly.py`, and `labs/tests/test_gpt_assembly.py`. All reported miniature-model measurements come from the recorded lab run; no held-out language-quality result is claimed.