

Scaling, and what comes next

A larger language model is not a different species of machine. But making the matrices larger is not the same thing as making the whole system better. This final chapter turns the GPT you assembled, trained and sampled into something you can budget, inspect and question. The reading-plus-lab time above is a rough study allowance, not a measured completion time.

Why this matters

In the previous chapter, generation repeatedly turned a context into logits, sampled a token and appended it. Nothing about that loop tells you whether a larger model will fit in memory, whether its training data was adequate, or whether it will answer your application's questions correctly. Those are different questions with different evidence.

Suppose an internal documentation assistant gives incomplete answers. Someone proposes a bigger model. Before accepting, ask whether the relevant passage reached the context, whether the task needs different training, and whether the current model fails even when given the right evidence. More parameters might help. They cannot be assumed to repair a missing document, an incorrect retrieval result or a poorly defined success criterion. Scaling is a possible intervention, not a diagnosis.

This chapter makes a narrower, testable promise. You will count the learned numbers in an actual runnable GPT, predict what happens when each size knob changes, and separate weight storage from training and inference memory. Then you will examine why model size and training data must be considered together. Finally, you will place post-training, retrieval, evaluation and agent systems around the model without confusing those layers with the model itself.

The Academy baseline introduces a leading parameter estimate and a comparison with GPT-2. We retain those useful ideas but replace the tolerance-based toy check with exact accounting against the real shared GPT. We also qualify its broad claim that architecture never changes. The residual-stream and next-token-prediction ideas transfer; exact projection shapes, position methods, normalization and execution strategies do not always stay the same.

For AI security work, this distinction matters defensively. A model's size is neither an authorization boundary nor evidence that its output is trustworthy. Evaluate factual support, permissions and tool behavior separately. This lab does not test security behavior and does not confer a safety rating on any model.

The idea in plain words

A parameter is a learned number. A matrix contains many such numbers. Making the matrix wider usually makes it larger in both directions, whereas adding another block adds another copy of that block's learned tensors. This is why width and depth have different costs.

Our runnable toy has 2,752 unique learned numbers. Doubling only its layers gives 5,368. Doubling only its width gives 10,112. Doubling only its attention heads leaves 2,752, because those heads divide the

same total width into smaller pieces. These are actual counts from constructed PyTorch models, not rounded marketing labels.

Now add an important distinction: learned numbers are not all the numbers that exist during a run. An attention score is computed for a particular input. An optimizer keeps additional arrays while training. A serving engine may retain keys and values from earlier tokens. Those all occupy memory, but they are not additional learned parameters. A parameter counter cannot be a complete memory profiler.

Think of the model as a fixed set of reusable calculation rules. The parameters specify the rules. A request generates temporary working values as those rules are applied. Training additionally keeps information needed to change the rules. Growing the request can grow the working space without changing a single rule. That is the intuition behind the context-memory experiment below.

How it actually works

Part 1: count the exact architecture, not a slogan

Use these symbols consistently: V is vocabulary size; C is embedding width; L is the number of transformer blocks; S is the learned position-table capacity. Later, T will mean the actual number of tokens processed in a request, B the batch size and H the attention head count. Separating S from T prevents a common mistake: using more of an existing context window does not create parameters.

Our token embedding has V rows and C columns, so it contains VC numbers. The learned position table has S rows and C columns, so it contributes SC. These are independent tables: one identifies tokens, the other supplies their positions.

Inside each block, the query, key and value projections each map width C to width C. Packing them into one linear layer changes storage layout, not the total matrix count. Together they contribute 3C squared. The attention output projection contributes another C squared. Attention therefore contributes 4C squared matrix weights.

The MLP expands width C to 4C and contracts it back to C. Each matrix contains 4C squared weights, making 8C squared together. Attention plus MLP gives the familiar leading term 12C squared per block. The multiplier comes from specified matrix shapes. It is not a law about every transformer ever built.

But our linear layers also have biases. The packed query/key/value bias has 3C entries; the attention output bias has C. The MLP biases have 4C and C entries. That is 9C bias entries per block. Each affine LayerNorm has a learned scale and offset of length C. There are two norms per block, contributing another 4C. Thus biases and norms contribute 13C per block. A final affine LayerNorm outside the blocks contributes 2C.

The output head is bias-free and shares the token embedding's Parameter object. We count that matrix once even though it is used in two places. The exact total for this implementation is:

$$P = VC + SC + L(12C^2 + 13C) + 2C$$

P means unique learned scalar entries. This formula includes learned positions, linear biases and affine LayerNorms. It excludes dropout, which has no learned tensor, and causal masks, which are registered buffers. Untying the output head adds VC, provided the separate head remains bias-free. It does not remove the vocabulary-scoring computation when the head is tied.

Work the toy example by hand using $V = 9$, $C = 8$, $S = 6$ and $L = 3$. Token lookup contributes 9 times 8, or 72. Positions contribute 6 times 8, or 48. All blocks together contribute 3 times the quantity 12 times 8 squared plus 13 times 8, or 2,616. The final norm contributes 16. Add the printed components: $72 + 48 + 2,616 + 16 = 2,752$. Dropping biases and norms gives 2,424 instead, missing 328 actual learned numbers.

This is why the lab checks exact equality rather than an arbitrary tolerance. At small width, supposedly minor linear terms need not be negligible. A broad tolerance could hide a missing layer, an accidentally untied head or a wrong bias setting. The real model also executes a forward pass and backward pass, so we are not just counting an inert collection of matrices.

Part 2: change one knob at a time

Depth multiplies the block contribution while leaving embeddings and the final norm alone. Hence doubling layers does not exactly double the total. Width appears squared in the large block matrices but only linearly in embeddings, biases and norms. Hence doubling width does not exactly quadruple the total either. The measured toy ratios are 1.951 for depth and 3.674 for width.

Vocabulary growth adds embedding rows and makes vocabulary scoring larger. With tied weights, it adds one matrix's worth of new rows, not two. Increasing learned position capacity adds SC storage according to the new S . These are different interventions even when both involve token-related dimensions.

Head count is the counterexample to the slogan that every larger configuration number increases parameters. With total width fixed, our full multi-head attention simply repartitions the same projections. The count stays fixed as long as C divides evenly into H heads. This does not mean head count has no effect on behavior or temporary memory. It means the weight shapes in this implementation do not change.

Part 3: reach GPT-2's scale without pretending to train it

The public GPT-2 small configuration specifies 12 layers, width 768, 12 heads, vocabulary 50,257 and position capacity 1,024. These dimensions are verified in the linked checkpoint configuration and OpenAI's released model code. The lab uses those dimensions in our shared implementation.

Counting gives 124,439,808 unique parameters. The leading-only estimate gives 124,318,464. The token embedding alone contains 38,597,376 parameters, or 31.02 percent of the exact total. If the output head were a separate matrix, it would add another 38,597,376. That is why weight tying matters even though the two uses of the table perform different operations.

We verify the large count with PyTorch's meta device. A meta tensor records its shape and type but holds no numeric data. Constructing the module this way checks what parameters the module would contain without allocating all the weight values. It is a real module-construction check, not a trained GPT-2 instance, downloaded checkpoint, text-quality evaluation or bit-for-bit reproduction of GPT-2. Our shared model's activation and initialization choices are not a checkpoint-compatibility guarantee.

Part 4: separate persistent weights from runtime memory

For logical weight storage, multiply P by the bytes used per value. Our GPT-2-shaped count needs 497,759,232 bytes for float32 weight values, or 248,879,616 bytes if all weight values occupy two bytes

each, as in float16 or bfloat16. These are arithmetic storage totals, not measured process memory or checkpoint file sizes. Object metadata, buffers and runtime allocations are absent.

For one explicit training layout, suppose weights, gradients and both Adam moment arrays are all float32. Each array contributes 4P bytes, so the four arrays contribute 16P. Our count gives 1,991,036,928 bytes for those arrays alone. The associated test actually takes an AdamW step on the toy and counts those tensors. This checks the accounting convention; it does not measure a large training job.

Peak training memory also includes activations retained for backward, attention intermediates, masks and temporary storage. Mixed-precision and sharded optimizers can use different layouts. Some systems keep extra higher-precision weight copies; some distribute optimizer arrays across devices. Do not apply 16P as a universal runtime law. Name the layout before using the estimate, then profile the implementation you intend to run.

An explicit dense attention score tensor has shape B by H by T by T. Its element count is $BH T^2$. A causal mask blocks future information but our implementation still constructs the full square score tensor. At batch one and the GPT-2 head count, increasing actual context from 512 to 1,024 grows the one-layer score count from 3,145,728 to 12,582,912. That is four times as many elements, with no change to the learned weights.

An inference key/value cache is different. In full multi-head attention, each token contributes a total width C of keys and C of values per layer. Across all layers, the value count is $2LBTC$. Multiply by bytes per value to get logical cache bytes. With two-byte values, our example grows from 18,874,368 to 37,748,736 bytes when context doubles. This storage grows linearly with T, not quadratically. It excludes allocator overhead and unused reserved capacity.

The distinction is crucial: our `GPT.generate` recomputes its cropped context and does not implement this cache. The printed cache numbers describe a hypothetical cached implementation of the same full multi-head architecture. Grouped-query attention requires a different formula because its key/value width can be smaller than its query width. A cache estimate is not proof that caching has been implemented.

FlashAttention changes the memory story again. Its tiled exact-attention algorithm avoids materializing the entire attention matrix in high-bandwidth memory. It does not mean the mathematical attention operation no longer compares token pairs. The naive score-storage estimate explains our educational implementation, not every optimized backend. See Dao and colleagues for the actual algorithm and measured results.

Part 5: scaling the vocabulary trades one cost for another

Character tokenization makes the lookup table small but sequences long. GPT-2's byte-level BPE spends more vocabulary entries to encode common pieces compactly. In the lab, the same sentence has 42 character tokens and 13 GPT-2 BPE tokens. The character vocabulary built from local Shakespeare contains 65 entries; the GPT-2 vocabulary contains 50,257.

At width 768, the respective input tables contain 49,920 and 38,597,376 numbers. The sequence gets shorter while the table gets much larger. That is a tradeoff, not a free compression win. Shorter sequences can reduce attention work for a fixed amount of text, but vocabulary scoring and embedding storage get larger. Actual speed depends on the full implementation and workload.

The local tiktoken cache makes this comparison offline. Both encodings must decode back to the exact original sentence. We do not swap tokenizers into a trained checkpoint: its embedding rows are tied to its existing token IDs. Changing that mapping without retraining or a deliberate adaptation would change the meaning of the inputs.

Part 6: model size, data and compute belong together

A parameter count describes capacity and part of the computational cost. It does not describe what training accomplished. To discuss training budgets, let N be a model-size count and D the number of processed training tokens, including repeat visits. The common dense-training approximation is F approximately $6ND$, where F is floating-point operations. A multiply-add is conventionally counted as two operations; forward and backward through the dominant dense matrices motivate the multiplier. Hoffmann and colleagues use this approximation in their compute-allocation analysis.

This is deliberately rough. Definitions of N and treatment of embeddings matter; attention's context-dependent work and other operations are omitted. Our helper accepts a count rather than pretending to be a profiler. It uses hypothetical N and D values for budget arithmetic, not the toy's measured execution time. Embedding-heavy tiny models and long contexts especially demand more careful accounting.

The lab gives one million parameters and one hundred million processed tokens a rough budget of 600,000,000,000,000 FLOPs. Doubling the model and halving the tokens preserves that product. Doubling both instead gives 2,400,000,000,000,000 FLOPs. More data and a larger model together require more compute; they cannot both be increased at a fixed product budget.

How should you divide a fixed budget? Empirical scaling laws address that question by fitting measured loss across controlled training runs. Kaplan and colleagues found predictable power-law relationships over their tested ranges. Hoffmann and colleagues revisited the allocation and found that their compute-optimal frontier called for scaling training data alongside model size. Chinchilla is historical experimental evidence for that allocation under the paper's objective, not a promise that any larger dataset improves every task.

We use a visibly constructed function to make the tradeoff inspectable:

```
illustrative_loss(n, d) = 1.5 + 1/sqrt(n) + 1/sqrt(d)
subject to n*d = 16
```

Here n and d are dimensionless model and data units. They are not counts of parameters and tokens. The floor 1.5 and both inverse-square-root terms are chosen for teaching, not fitted from our run or copied from a paper. At $n = 4$ and $d = 4$, both square roots are two, so the loss is $1.5 + 0.5 + 0.5 = 2.5$. The tested extremes, $n = 1$ with $d = 16$ or the reverse, give 2.75. Improving one resource while starving the other can be worse than balancing them.

The lower bound follows without a training run. For a fixed product nd , the product of the two positive inverse-square-root terms is fixed. Their sum is smallest when the terms are equal. Because this particular function uses equal coefficients and exponents, its optimum is balanced. If the coefficients or exponents differ, that balance shifts. Real fits must estimate those terms from data and then be checked against runs not used to fit them.

Do not call this table an empirical scaling result. It is an executable illustration of constrained allocation. It cannot predict GPT-2 loss, modern benchmark scores or a useful training duration. Repeated low-quality tokens still consume D without necessarily providing the benefit of new useful information. An actual experiment needs stable tokenization, a clean validation split, controlled optimizer schedules, adequate training and task-level evaluations.

Build it

The runnable entry is this folder's `lab.py`. New reusable accounting functions are in `labs/tutorlab/scaling_demo.py`, reusing chapter 7's exact `parameter_budget`. Existing GPT and tokenizer files are unchanged. From the project root, run:

```
env -u PYTHONPATH .venv/bin/python chapters/llm-fundamentals/10-scaling-and-what-comes-next/lab.py
```

The lab finds `labs/` relative to its own resolved file path rather than the working directory. It seeds PyTorch and limits CPU threads. It does no network access, downloads no model and needs no accelerator. The large configuration uses metadata only; actual tensor arithmetic is confined to the tiny model. The dependency set is the project's existing PyTorch and tiktoken environment.

Part 1 constructs the actual GPT and asks `model.parameters()` for unique Parameter objects. `numel()` counts scalar entries in each tensor. Do not substitute a blind sum over checkpoint dictionary entries: multiple names can refer to tied storage, and persistent buffers are not learned weights. The lab also runs next-token loss backward and verifies finite gradients exist on all parameters. This demonstrates connectivity, not useful training or converged language quality.

The untied comparison clones the head into a new `torch.nn.Parameter`. Merely referring to the same object through another name would not untie anything. Equal initial values do not imply shared identity. The extra parameter count must equal VC, making this an especially direct test of whether the formula matches the actual implementation.

Part 2 uses `dataclasses.replace` to change a single field without mutating the base configuration. Each variant is constructed and checked against the exact formula. This controlled sweep isolates causes: a combined jump in layers, width and vocabulary would not tell you which change caused the increase.

Part 3 uses `with torch.device('meta')` only around the large construction. The scope matters: the earlier toy still contains real values. We assert that every large parameter is meta and that its unique count matches both the formula and the documented GPT-2-shaped result. Part 4 computes explicitly labeled storage counts; it never allocates the large score tensors or claims measured GPU memory.

Part 5 encodes a shared sentence with the existing character tokenizer and cached GPT-2 BPE. Part 6 prints both the dense-compute arithmetic and the constructed objective. The final `check()` combines these invariants and prints `CHECK PASSED` only after they hold. Unit tests independently cover invalid dimensions, incompatible head counts, context overflow, tensor byte counts and an actual optimizer-state allocation.

Output from the executed lab

The following is the recorded run, copied from `build/lab-output.txt`. In the final section, “illustrative loss” means evaluation of the declared teaching function, not a training measurement.

Part 1: exact unique parameters, not checkpoint dictionary entries

token_table_and_tied_head: 72
position_table: 48
blocks: 2,616
final_norm: 16
untied_head: 0
leading estimate=2,424 exact=2,752 torch=2,752
omitted biases and norms=328
real forward logits=(1, 6, 9) finite gradients on every parameter=True
tied=2,752 untied=2,824 extra=72
causal mask buffer elements=108 (not parameters)

Part 2: one change at a time (same exact architecture)

base	params= 2,752	ratio=1.000
layers x2	params= 5,368	ratio=1.951
width x2	params=10,112	ratio=3.674
vocab x2	params= 2,824	ratio=1.026
context x2	params= 2,800	ratio=1.017
heads x2	params= 2,752	ratio=1.000

Part 3: GPT-2 small shape, metadata only

layers=12 width=768 heads=12 vocab=50,257 context=1,024
leading=124,318,464 exact=124,439,808 meta module=124,439,808
token table=38,597,376 (31.02% of total)
untied head would add=38,597,376
weights only: fp32=497,759,232 bytes fp16/bf16=248,879,616 bytes
fp32 Adam arrays only=1,991,036,928 bytes (not peak training memory)

Part 4: context costs (GPT-2 shape, batch=1)

tokens=512: one-layer score elements=3,145,728; hypothetical all-layer 2-byte KV cache=18,874,368 bytes

tokens=1024: one-layer score elements=12,582,912; hypothetical all-layer 2-byte KV cache=37,748,736 bytes

score ratio=4.0; cache ratio=2.0; learned parameter count unchanged
tutorlab.generate recomputes context; it does NOT implement this KV cache

Part 5: same text, different sequence lengths

text='To be, or not to be, that is the question.'
character vocab=65, tokens=42
GPT-2 BPE vocab=50,257, tokens=13
at width=768: character table=49,920; BPE table=38,597,376

Part 6: budget accounting, not a trained scaling-law experiment

N=1,000,000 D=100,000,000: rough 6ND=600,000,000,000,000 FLOPs
N=2,000,000 D=50,000,000: rough 6ND=600,000,000,000,000 FLOPs
N=2,000,000 D=200,000,000: rough 6ND=2,400,000,000,000,000 FLOPs
constructed objective: $1.5 + 1/\sqrt{n} + 1/\sqrt{d}$; $n*d=16$

```
model units= 1 data units=16.0 illustrative loss=2.7500
model units= 2 data units= 8.0 illustrative loss=2.5607
model units= 4 data units= 4.0 illustrative loss=2.5000
model units= 8 data units= 2.0 illustrative loss=2.5607
model units=16 data units= 1.0 illustrative loss=2.7500
best tested allocation: n=4, d=4.0; this is NOT a fitted empirical law
```

CHECK PASSED

Predict before editing

Start with the head-count experiment in the [TRY THIS](#) comments. State whether the unique count will change before trying each valid head count. Then explain why the score tensor's head dimension can change despite that fixed count. If the explanation only repeats the formula, trace the packed projection's shape in [gpt.py](#).

For a second application, compare tied and untied accounting across the variants. Predict which term changes, then confirm by constructing an independent head. Finally, change the coefficient of the constructed model term. Predict the direction of the allocation shift before rerunning the illustrative table. None of these exercises requires training a large model.

Common mistakes and misconceptions

"Everything larger is exactly the same architecture." The high-level transformer story transfers, but the exact counter is conditional on particular biases, norms, projections and embeddings. LLaMA's published architecture replaces learned absolute positions with rotary positions and uses RMSNorm and a gated feedforward design. You must derive the count for the implementation you have, not reuse 12C squared without checking.

"More heads means more parameters." Not when full-attention heads partition fixed total width. If instead you hold head width fixed and increase the number of heads, total width grows, which is a different experiment. State what is fixed before drawing the conclusion.

"Weight memory tells me whether training fits." It omits optimizer state, gradients and activations. Even a correct weight-only byte count can badly understate peak memory. Our Adam-array formula has explicit assumptions and exclusions; the large-model memory numbers are not measured peaks.

"A smaller loss means the assistant is truthful and follows instructions." Next-token loss measures predictive performance on a specified distribution. It is not a direct measure of factual correctness, instruction following, permission discipline or success on an application workflow. Those require separate evaluation and, often, different training or system design.

"The balanced toy table proves the scaling laws." Its assumptions were chosen by us. It shows why a tradeoff can exist. Historical scaling papers earn their claims from trained models and measured losses. A curve drawn from a convenient formula is not evidence about a real model's optimum.

"More tokens always means more useful knowledge." Repetition, contamination, distribution mismatch and poor quality can weaken that inference. Processed-token count is useful for compute accounting but does not measure independent useful information. Keep a validation set separate and check the intended task, not just a training-loss curve.

In the real world

Historical examples and official references in this section were checked as of 2026-10-10. They are examples of design choices, not a current model leaderboard, pricing survey or forecast of future capability.

GPT-2 supplies a concrete bridge from the tiny machine to a recognizable model. OpenAI's released implementation uses token and position embeddings, pre-normalized residual blocks and a tied vocabulary projection. The linked small-checkpoint configuration supplies the dimensions used in the lab. Matching its parameter shape is useful evidence that you understand the accounting, but is not evidence that random weights behave like the trained checkpoint.

LLaMA's original paper supplies two useful corrections to "just make it larger." Architecturally, it uses RMSNorm, rotary position information and SwiGLU. Economically, it explicitly distinguishes a training-compute optimum from an inference-budget objective. A smaller model trained for longer can be worthwhile when serving it many times dominates total use. That is a design objective, not a blanket rule that smaller models are better.

Grouped-query attention changes how query heads share key/value heads. Its paper investigates an intermediate number of key/value heads between multi-query and full multi-head attention. The runtime-storage implication is direct: a cache count based on full width C must be revised when the key/value width changes. This is a reason to inspect architecture metadata before making a memory promise.

Sparse mixture-of-experts models distinguish total parameters from parameters active for a particular token. The Mixtral paper describes a router selecting a subset of feedforward experts. Total weight storage and per-token computation therefore cannot be inferred from one undifferentiated parameter headline. Our dense helper does not implement expert routing and must not be used as an exact compute estimator for such a system.

At the systems layer, Karpathy's GPT-2 reproduction lecture develops gradient accumulation and distributed training from the same optimization loop. Accumulation combines gradients from microbatches before an optimizer step; it is useful when an effective batch is too large for one forward/backward pass. Distributed data parallelism processes different batches on model replicas and synchronizes gradients. It increases available throughput but does not automatically divide each replica's weight storage. These are execution choices, not new definitions of language modeling.

What comes next is not one giant leap. The course queue continues into prompt engineering, where you control instructions, examples and supplied context. Training and fine-tuning address changes to the weights. The InstructGPT paper illustrates how supervised demonstrations and preference-based training can change assistant behavior without making model size the only intervention. Retrieval supplies relevant external evidence at request time rather than assuming everything must be stored in parameters. Evaluation checks whether those interventions actually help.

Agent engineering then puts a controlled software loop around a model: obtain context, request a prediction, validate proposed actions, execute permitted tools and inspect results. Treat that as application design with explicit permissions and tests, not as a capability implied by parameter count. This chapter intentionally gives no tool-execution workflow. The immediate engineering habit is smaller: identify the bottleneck, choose an intervention and measure the result against a defined task.

Go deeper

Lecture anchors below were verified on 2026-10-10 with `tools/find_lecture_moment.py`. YouTube blocked transcript retrieval, so verification uses the creator-published chapter markers. The anchor phrases are marker titles, not claimed verbatim speech. `lectures.json` records the exact start and end seconds for each selected segment.

- Andrej Karpathy, "Let's reproduce GPT-2 (124M)," 1:06:14 to 1:13:47, "parameter sharing wte and lm_head": <https://www.youtube.com/watch?v=l8pRSuU81PU&t=3974> . Watch how object sharing connects the input table and output projection. Compare that implementation with the toy's exact tied-versus-untied check.
- The same lecture, 2:34:09 to 2:46:52, "gradient accumulation": <https://www.youtube.com/watch?v=l8pRSuU81PU&t=9249> . Track the difference between a microbatch and an optimizer step. Do not confuse a larger effective batch with a larger model.
- Andrej Karpathy, "[1hr Talk] Intro to Large Language Models," 25:43 to 27:43, "LLM Scaling Laws": https://www.youtube.com/watch?v=zjkBMFhNj_g&t=1543 . Use the overview as an entry point, then read Kaplan and Hoffmann for the measured objectives and limitations. The historical talk is not a statement about today's best model.

For paper reading, begin with the abstracts of Kaplan and Hoffmann and write down what each study optimized. Then inspect Hoffmann's parametric-loss and compute-frontier section. Ask which coefficients came from fitting and which constraints came from the experimental setup. Finally read LLaMA's introduction to see how adding deployment considerations changes what "optimal" means.

Explain it back

Explain why doubling a model's width, doubling its head count at fixed width, and doubling the tokens in a request are three different interventions. Use the toy's measured counts, distinguish parameters from working memory, and finish by explaining why a bigger model is not automatically the right fix for missing evidence in an assistant's answer. Dictation is fine. Your study partner should find one gap in the explanation, address it, and then use a new case from the quiz to check application. Receiving this chapter is not evidence of mastery.

Check yourself

The answer keys and rubrics stay in `quiz.json`. Use these questions without looking at the recorded output first:

1. Which expression counts every unique learned parameter in this tied-head GPT?
2. At fixed embedding width, what does doubling the number of full attention heads do to parameter count?
3. What has the GPT-2-shaped meta-device check actually demonstrated?
4. Which interpretation of the lab's 16P-byte estimate is correct?
5. What conclusion is justified by the constructed loss table?
6. If you untie the toy output head by cloning its weights, what are the new and added parameter counts?
7. When actual context grows from 512 to 1,024 in the fixed GPT-2-shaped model, how do dense scores, the hypothetical cache and parameters change?
8. At the same rough 6ND budget, how many tokens fit after the hypothetical model size doubles?

9. Why can an identical checkpoint run out of memory on a longer request?
10. When an internal assistant lacks the relevant source documents, how would you diagnose and evaluate an improvement before choosing a bigger model?

The final question is a transfer exercise, not another counting exercise. An accurate answer must connect the mechanism to a system-design decision. Your study partner records mastery only after an accurate independent explanation and correct application, never from the existence of these files or a successful lab run.

Sources

Primary sources and official documentation checked as of 2026-10-10. All reported local counts and printed results come from this chapter's executed `lab.py` and `build/lab-output.txt`. The architecture equations are derived from `labs/tutorlab/gpt.py`; new tests are in `labs/tests/test_scaling_demo.py`. The constructed objective is explicitly our teaching example, not a published empirical result.

- Academy chapter baseline and scope: <https://krypteiassec.com/academy/llm-fundamentals/ch10>
- OpenAI GPT-2 model implementation, architecture and shared output projection: <https://github.com/openai/gpt-2/blob/master/src/model.py>
- GPT-2 small checkpoint configuration, dimensions: <https://huggingface.co/openai-community/gpt2/blob/main/config.json>
- OpenAI tiktoken, byte-level BPE implementation and encodings: <https://github.com/openai/tiktoken>
- PyTorch meta device, shape-only tensors: <https://docs.pytorch.org/docs/stable/meta.html>
- PyTorch AdamW, moment-state algorithm and implementation-dependent memory caveats: <https://docs.pytorch.org/docs/stable/generated/torch.optim.AdamW.html>
- Kaplan et al., "Scaling Laws for Neural Language Models" (2020), empirical loss relationships and compute allocation: <https://arxiv.org/abs/2001.08361>
- Hoffmann et al., "Training Compute-Optimal Large Language Models" (2022), especially section 3.3 for the parametric loss and approximate 6ND constraint: <https://arxiv.org/html/2203.15556v1>
- Touvron et al., "LLaMA: Open and Efficient Foundation Language Models" (2023), introduction and section 2.2 for inference-budget objectives and architecture changes: <https://arxiv.org/html/2302.13971v1>
- Dao et al., "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness" (2022), tiled exact attention: <https://arxiv.org/abs/2205.14135>
- Ainslie et al., "GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints" (2023), shared key/value heads: <https://arxiv.org/abs/2305.13245>
- Jiang et al., "Mixtral of Experts" (2024), sparse expert routing and active-versus-total parameters: <https://arxiv.org/abs/2401.04088>
- Ouyang et al., "Training language models to follow instructions with human feedback" (2022), post-training and the limits of size alone: <https://arxiv.org/abs/2203.02155>
- Karpathy, "Let's reproduce GPT-2 (124M)," implementation and systems lecture: <https://www.youtube.com/watch?v=I8pRSuU81PU>
- Karpathy, "[1hr Talk] Intro to Large Language Models," historical overview: https://www.youtube.com/watch?v=zjkBMFhNj_g