

The transformer block: residuals, layer norm, and an MLP

Attention is not a complete transformer. You already have a way for token positions to exchange information through multiple heads. Now put that operation inside a repeatable unit that preserves the current representation, prepares useful inputs for each computation, and transforms each position's features. This chapter builds that unit and tests what it can and cannot guarantee. The reading-plus-lab time above is a rough study allowance, not a measured completion time.

Why this matters

A model definition can look deceptively simple: a list of blocks, one after another. The difficult part is hidden inside the word block. If you replace the input every time attention runs, use normalization over the wrong dimensions, or feed the MLP an outdated representation, the code can still produce a tensor of the right shape. Shape alone does not prove that you built the intended model.

This is useful knowledge when reading an open model implementation, diagnosing a training failure, or comparing an optimized kernel with a reference. You want to know which operations communicate between positions and which only transform a position already containing context. That distinction tells you where a future-token leak can occur. It also tells you what a meaningful regression test should change and what it should leave unchanged.

For agent engineering, keep the levels separate. The block is part of the language model's internal computation. It does not execute a tool, enforce an operating-system permission, or decide whether a retrieved document is trusted. Understanding its structure helps you avoid treating an attention map as a complete explanation of an agent's behavior. Permission boundaries still belong in the surrounding software.

The Academy baseline introduces residuals, normalization, a feature-expanding MLP, and shape preservation. We retain those ideas, then add a numerical derivation, parity with the existing course GPT, a real backward pass, a causal stack test, and a deliberately wrong implementation that demonstrates why the tests matter. This is an offline CPU experiment with random weights, not a trained language model or a model-quality benchmark.

The idea in plain words

Imagine that every token position carries a notebook of numbers. Instead of erasing that notebook and replacing it after every computation, ask each computation for a correction. Add the correction to what was already there. In the lab, the state is $[1.0, 3.0]$, the correction is $[0.5, -0.5]$, and their sum is $[1.5, 2.5]$. This is ordinary element-by-element addition, not concatenation and not a probability mixture.

That operation is a residual connection. The current notebook, repeatedly updated as it passes through the model, is often called the residual stream. A correction may reinforce an existing feature, oppose it, or introduce another combination of features. The addition does not force it to be small. It merely makes leaving the state alone easy: the correction can be zero.

A block makes successive corrections. First, attention reads a normalized view of the state and produces a contextual update. Add that update to the original state. Next, a small neural network reads a normalized view of the updated state and produces another update. Add that too. The neural network is called a multilayer perceptron, abbreviated MLP.

Normalization prepares a branch's input; it does not replace the untouched path around that branch. Attention mixes information between permitted positions. The MLP mixes features within each position, applying the same learned function at every position. Its input can already contain information gathered from earlier tokens, so local computation does not mean context-free computation.

Hold this single sentence: preserve the state, let attention add context, then let a shared nonlinear feature function add a further correction. The rest of the chapter turns each phrase into an equation and a test.

How it actually works

Part 1: residual addition and its derivative

Write one residual operation as $y = x + F(x)$. Here x is the incoming vector, F is the learned branch, and y is the outgoing vector. Both terms being added must have the same shape. In a pre-normalized attention branch, F includes both normalization and attention. The symbol does not mean an arbitrary scalar added to every coordinate.

For a scalar, the derivative is $dy/dx = 1 + F'(x)$. The prime means derivative: how much the branch output changes when its input changes a little. The extra one comes from the direct x -to- y path. For a vector, replace one with an identity matrix I and F' with the Jacobian J_F , the matrix of output sensitivities to input coordinates. The relation becomes $J_y = I + J_F$. Backpropagation still combines all paths; it does not ignore the learned branch.

Our controlled scalar experiment repeats a branch that multiplies its input by 0.5 across 8 layers. Without the skip path, the derivative is the product of those repeated factors. With the skip path, each layer contributes a factor of $1 + 0.5$. The lab reports gradients 0.00390625 and 25.62890625 respectively. This isolates the algebra, not the statistics of a real transformer.

Notice that the residual result grows rather than staying close to one. That is intentional: a skip connection is not a mathematical guarantee against exploding gradients. The lab also uses a branch multiplier of -1. Its update cancels the direct path and the resulting gradient is 0.0. Residuals supply a useful direct route, but initialization, normalization, optimizer settings and learned branch derivatives still matter. Do not turn a helpful design into a universal stability theorem.

Part 2: LayerNorm from a single row

Take the lab row $x = [1.0, 3.0, 5.0, 7.0]$. Let C be the number of features in that row. The mean, written μ , is the sum of the features divided by C . Subtract it from each feature to get the centered values. The lab computes $\mu = 4.000000$ and the centered row $[-3.0, -1.0, 1.0, 3.0]$.

The population variance, written v , is the average of the squared centered values. For this row, $v = 5.000000$. We divide by C , not C minus one. This is not an estimate of an unknown population variance from a sample; it is the statistic defining this normalization operation. PyTorch's LayerNorm uses the

same convention [1]. A default sample-variance function can silently give a different result, so the lab checks with `unbiased=False` when printing variance.

For feature j , the normalized value is $z_j = (x_j - \mu) / \sqrt{v + \epsilon}$. The subscript j selects a feature, `sqrt` means square root, and `epsilon` is a small positive stabilizer inside that root. In this lab `epsilon` is 0.00001. Working through the row by hand gives the expression $[-3, -1, 1, 3] / \sqrt{5 + 0.00001}$. The executed result is $[-1.341639, -0.447213, 0.447213, 1.341639]$.

There is no mystery layer left in that calculation. Find the center, measure the spread, subtract the center, and divide by a stabilized measure of the spread. Keeping the reduced dimension in code lets the row's mean and variance broadcast back across its features without accidentally crossing a batch or token axis.

The mean of z is 0.000000, but its measured variance is 0.999998, not exactly one. `Epsilon` explains the difference. If the input row is constant, its centered values and variance are zero; the stabilizer prevents division by zero and the normalized row is $[0.0, 0.0, 0.0, 0.0]$. Demanding unit variance from a constant row would be an incorrect test.

`LayerNorm` then usually applies a learned affine transformation: $out_j = \gamma_j * z_j + \beta_j$. `Gamma` is a learned scale per feature and `beta` is a learned offset per feature. They start at one and zero in `PyTorch`, but need not remain there [1]. The lab's illustrative choice of `gamma = 2` and `beta = 3` at every feature produces $[0.316721, 2.105574, 3.894426, 5.683279]$, with mean 3.000000 and variance 3.999992. These deliberately chosen values are arithmetic examples, not trained parameters.

That is why "LayerNorm always outputs mean zero and variance one" is an unsafe statement. It describes the core standardization approximately for a nonconstant row before learned affine parameters, not every output of the module.

Normalize over features, never accidentally over time

Our tensor has shape (B, T, C) . B is the batch axis, T is the sequence-position axis, and C is the feature axis. The lab uses $(2, 4, 8)$: separate sequences, positions within each sequence, and features at each position. `nn.LayerNorm(C)` reduces only the last axis. Every token position computes its own statistics without reading another position or another sequence.

`LayerNorm`'s `gamma` and `beta` are shared across positions in this setup, but its means and variances are computed separately for each row. Shared parameters do not imply shared statistics. It also uses input-derived statistics in evaluation mode, rather than switching to stored running averages [1]. This distinction matters when you compare it with `BatchNorm`.

Changing the normalized shape to (T, C) still returns the same outer tensor shape, but now earlier rows depend on the values at future positions. A causal attention mask cannot repair a leak that happened before attention. Our negative control changes only a feature of the final position. Correctly normalized causal blocks leave earlier positions unchanged. Wrong normalization changes earlier values by as much as 1.571958 in this run. Passing a shape assertion would not catch that mistake.

Part 3: assemble the pre-norm block

The complete equations are $a = \text{Attention}(\text{LN}_1(x))$, $y = x + a$, $m = \text{MLP}(\text{LN}_2(y))$, and $out = y + m$. Here a and m are updates, y is the state after attention, and `LN_1` and `LN_2` are separately learned

normalizations. The order is essential. The second normalization consumes y , not the original x , because the MLP should process the representation that already includes the attention update.

It is called pre-norm because each normalization comes before its learned branch. A post-norm arrangement instead normalizes after a residual addition, for example $\text{LN}(x + \text{Attention}(x))$. Moving normalization changes the function and the backward paths. It is not a cosmetic rewrite. Xiong and colleagues analyze the consequences for initialization and warm-up in their Pre-LN versus Post-LN study [4]. Their results do not imply that every pre-norm configuration is automatically stable.

We reuse the existing course attention implementation. It projects the full row into queries, keys and values, splits those projected vectors into heads, applies a causal mask, combines value vectors, rejoins the heads, and projects back to width C . The lab uses 2 heads, each with width 4. Heads do not read disjoint raw input slices before projection. That detail carries forward from the previous chapter.

All exposed block intermediates have shape $(2, 4, 8)$. Shape preservation makes the block composable: another block can accept its output. Stacking creates fresh modules with their own parameters; it need not mean repeatedly applying the same module instance. The lab deliberately constructs a new second block. It does not confuse reused weights with repeated architecture.

Part 4: what the MLP contributes

Let u be one normalized token row. The MLP computes $h = u W_1 + b_1$, then $g = \text{GELU}(h)$, then $m = g W_2 + b_2$. In this row-vector notation, W_1 maps C features to a wider hidden representation and W_2 maps back to C . The b terms are learned biases. PyTorch stores a Linear weight in output-by-input order and internally uses its transpose, so the code and row-vector notation express the same mapping.

In our GPT-style MLP, the intermediate width is four times C . The executed shapes are $(2, 4, 8)$, then $(2, 4, 32)$, then $(2, 4, 8)$. Expansion provides intermediate features for the nonlinear operation, while contraction restores the width required by residual addition. Fourfold expansion is a design choice in this implementation, not part of the definition of all transformers [3].

GELU is the Gaussian Error Linear Unit [2]. Its definition is $\text{GELU}(s) = s * \Phi(s)$, where s is a scalar hidden activation and Φ is the standard normal cumulative distribution function. $\Phi(s)$ is a smooth number between zero and one. The deterministic activation scales s by that smooth value; it does not draw a random sample. The lab evaluates GELU at $[-1, 0, 1]$ and obtains $[-0.158655, 0.0, 0.841345]$. Unlike ReLU, it does not set every negative input exactly to zero.

Why put a nonlinearity between the linear maps? Without it, multiplying by W_1 and then W_2 , including the corresponding biases, collapses into one affine mapping. Expansion alone does not prevent that collapse. GELU lets the effective feature transformation depend on the input rather than remain one fixed affine map. Attention is already input-dependent, so this is not a claim that attention itself is linear in x . It is a statement about the pair of maps inside the MLP.

The MLP acts separately at every position and shares its parameters across them. If two rows entering it are identical, their outputs match when stochastic effects are absent. Different rows can produce different outputs under those same weights. Changing a final input row leaves all earlier MLP output rows unchanged in the lab, with maximum difference 0.000000. The surrounding attention branches are what let these local transformations operate on contextualized information.

The parameter count makes another misconception visible. With biases included, the attention projections contribute 288 parameters, the MLP 552, and each normalization 16, giving 872 in the block.

These counts come from the actual modules. The general counts here are $4CC + 4C$ for *attention*, $8CC + 5C$ for the MLP, and $2 \cdot C$ for each norm. The MLP is not merely decorative glue around attention; in this example it owns more learned parameters than attention does. Parameter count is not the same thing as runtime or importance, however.

Build it

The executable is `lab.py` in this chapter folder. Its reusable helpers live in the new `labs/tutorlab/block_demo.py`; its tests are in `labs/tests/test_block_demo.py`. No existing course library file is changed. The inspectable block imports `CausalSelfAttention` and `MLP` from `tutorlab.gpt`, so the new work focuses on the surrounding wiring rather than duplicating attention.

From the project root, run:

```
env -u PYTHONPATH .venv/bin/python chapters/llm-fundamentals/06-the-transformer-block/lab.py
```

The file locates the project relative to its own resolved path and adds the existing labs directory to Python's import path. It does not need an API key, network connection, downloaded model, or GPU. The CPU seed is fixed and dropout is disabled. Double precision makes the numerical comparisons easier to inspect. Random weights remain random, reproducible weights; a seed is not training.

Parts 1 and 2: arithmetic before modules

The first part builds the scalar gradient experiment with autograd. A leaf tensor has `requires_grad=True`; repeated operations create a computation graph; `backward()` propagates the sensitivity of the final scalar back to that leaf. The experiment changes only whether the direct addition exists. The cancellation case keeps the conclusion honest.

The next part implements normalization without calling `LayerNorm`:

```
mean = x.mean(dim=-1, keepdim=True)
variance = (x - mean).square().mean(dim=-1, keepdim=True)
normalized = (x - mean) / torch.sqrt(variance + eps)
```

`dim=-1` chooses features, and `keepdim=True` leaves an axis of length one so broadcasting stays aligned. The tests compare both values and gradients with PyTorch's implementation, including a constant row. Agreement on a single printed vector would be weaker than agreement on the backward calculation too.

Parts 3 and 4: keep the state updates visible

The trace method names every intermediate rather than hiding both updates in compact reassignment:

```
norm1 = self.ln1(x)
attention_update = self.attn(norm1)
after_attention = x + attention_update
norm2 = self.ln2(after_attention)
mlp_update = self.mlp(norm2)
output = after_attention + mlp_update
```

Read the additions carefully: neither adds onto the normalized view. They add onto the persistent state. The trace returns a dictionary of tensors, while `forward()` returns only its final output. That keeps the module usable in a standard sequential stack and makes the educational path inspectable.

A reference `tutorlab.gpt.Block` receives exactly the same state dictionary before comparison. Its maximum output error is 0.000000 in this run. Comparing separately initialized modules would be meaningless because they would contain different random parameters. Copying the weights turns the comparison into a test of the implementation, not a test of whether random draws happen to match.

The lab also reports root mean square, abbreviated RMS, of several tensors. RMS means square the values, average those squares, and take the square root. It is a scale diagnostic, not a probability or an accuracy score. The normalized branch inputs have RMS close to one in this initialization, while the final residual stream has RMS 1.132338. Normalizing a branch does not pin the residual stream to unit scale.

Parts 5 and 6: tests that can fail meaningfully

The future-change test runs the same stack on an input and a copy whose last position has one altered feature. Earlier outputs must match because no operation may use future information. The final position should change, which proves that the perturbation was not accidentally ineffective. Beside the correct path, the wrong normalization intentionally includes the token axis. Its nonzero earlier-row difference is evidence that the test can expose that class of leak.

Then the lab copies the block and zeros the output projection weights and biases of both branches. The internal query, key and first MLP projections remain intact, but both updates become exactly zero. The whole pre-norm block must now be the identity function. Summing its output and backpropagating must give a gradient of one at every input coordinate. Forgetting an output bias when zeroing a branch would invalidate this experiment.

Finally, a normal block computes a mean-square diagnostic scalar and backpropagates through it. Every parameter has a present, finite gradient, and both the attention and MLP input projections receive nonzero gradients. This proves connectivity and finite computation for the tested case. It does not prove the model learned language, that an optimizer converges, or that a deep stack will be stable under every setting.

Actual output, excerpted from the saved run

These are verbatim lines from `build/lab-output.txt`, not predictions:

```
normalized mean=0.000000, variance=0.999998
gamma=2, beta=3: [0.316721, 2.105574, 3.894426, 5.683279]
affine mean=3.000000, variance=3.999992
parameters: {'attention': 288, 'MLP': 552, 'norm1': 16, 'norm2': 16} total= 872
existing tutorlab.gpt.Block max error=0.000000
MLP shapes: (2, 4, 8) → (2, 4, 32) → (2, 4, 8)
GELU([-1, 0, 1]): [-0.158655, 0.0, 0.841345]
correct block stack earlier rows max difference=0.000000
changed final row max difference=10.453190
WRONG norm over (tokens, features): earlier rows max difference=1.571958
zero branch outputs: identity error=0.000000, gradient-to-ones error=0.000000
```

```
diagnostic mean-square loss=1.282190, input gradient norm=0.307260
every parameter gradient present and finite=True
```

The complete successful run ends with `CHECK PASSED`. Small backend-dependent rounding differences can occur; use the mathematical assertions and tolerance-based comparisons, not a text snapshot, for correctness. This run's exact transcript remains the source for the numbers printed above.

Predict before editing

Use a copy for experiments. Change gamma and beta, predicting the standardized mean separately from the affine output mean. Move the perturbation from the final row to an earlier row, predicting which later outputs are allowed to change. Remove one residual addition and predict what the zero-branch identity check will report. Each modification targets one mechanism. Do not change the implementation and the expected result together merely to keep the output green.

Common mistakes and misconceptions

"Residual means the branch always makes a small change." It can learn a small change, but addition does not constrain its magnitude. The gradient-chain counterexample also shows why a skip path does not guarantee bounded derivatives. A precise statement is that the architecture provides a direct identity path and lets the branch learn an additive correction.

"LayerNorm normalizes the whole sequence." That depends on the configured normalized shape. Our decoder normalizes each position over its features only. Normalizing across time would let an earlier position depend on a future value before the attention mask is even applied. Shape checks alone cannot reveal this.

"The normalized output always has mean zero." The standardization core does, up to numerical precision, but learned affine parameters can change the output mean and scale. Epsilon also means the standardized variance is not exactly one. Constant inputs are a particularly useful counterexample to overly broad claims.

"The MLP gives each token its own private weights." It gives every position the same function, evaluated on that position's current vector. Independence refers to the current operation's position axis, not to parameter ownership and not to a lack of context in the vector.

"The second branch can normalize the original input because the shapes match." The shapes do match, but the intended sequential computation has changed. In the pre-norm block here, the MLP must receive the post-attention state. An equation-by-equation parity test catches this mistake more directly than a generic output-shape assertion.

"A backward pass means the model is trained." Backward computes derivatives. Training also needs an objective tied to a task, data, optimizer updates, and evaluation. The diagnostic scalar in this lab is deliberately not a language-model loss. Treat its gradient check as implementation evidence, not a capability result.

In the real world

As of 2026-10-10, the inspected nanoGPT `model.py` uses pre-norm residual attention followed by pre-norm residual MLP, a GELU between the MLP projections, and a final normalization after the block stack

[3]. Its implementation also supports an optimized scaled-dot-product attention path. Those implementation choices are observable source code, not guesses about a proprietary model.

The original Transformer paper presents residual connections around sublayers followed by normalization, and a position-wise feed-forward network [5]. That historical architecture is not identical to the pre-norm decoder block built here. “Transformer block” describes a family of closely related patterns rather than a promise that normalization order, activation, biases, or head layout is always identical.

The PyTorch LayerNorm documentation, checked on the same date, states that normalization axes follow `normalized_shape`, variance uses the population convention, and statistics come from the input in both training and evaluation modes [1]. Use those API facts when reviewing a port. Do not rely on a slogan such as “keeps numbers sane,” which hides the axes, affine parameters and epsilon that determine the actual result.

Our local `labs/tutorlab/gpt.py` already contains the production path for this course's small model: token and position embeddings, blocks, final normalization and an output head. This chapter verifies block parity without changing that implementation. The next chapter can assemble those pieces while retaining the local tests. A real deployment would still need workload-specific performance checks, numerical comparisons under the intended dtype, and task evaluation. None of those are claimed by this small CPU demonstration.

Go deeper

The lecture links below were verified on 2026-10-10 using the creator's published chapter markers. YouTube blocked transcript retrieval from this host. The anchors in `lectures.json` are marker titles, not invented transcript quotations; the suggested intervals identify topics rather than promising a particular sentence at an exact second.

- Andrej Karpathy, “Let's build GPT: from scratch, in code, spelled out.” Start at 84:25, “feedforward layers of transformer block”: <https://www.youtube.com/watch?v=kCc8FmEb1nY&t=5065>. Watch how a position-wise network fits after communication through attention. Compare the architectural role with this lab, while checking activation choices directly rather than assuming all demonstrations are identical.
- In the same lecture, start at 86:48, “residual connections”: <https://www.youtube.com/watch?v=kCc8FmEb1nY&t=5208>. Follow the direct path around the learned operation and relate it to the identity test, not to a claim that all gradients must remain small.
- Continue at 92:51, “layernorm (and its relationship to our previous batchnorm)”: <https://www.youtube.com/watch?v=kCc8FmEb1nY&t=5571>. Track which dimensions determine the statistics and where normalization sits relative to addition.

For the underlying normalization idea, read Ba, Kiros and Hinton's Layer Normalization paper [6]. For the effect of placement, read Xiong and colleagues [4], distinguishing their theory and experiments from universal claims about all architectures. Then open nanoGPT's compact `BLOCK` and `MLP` classes [3] beside the equations in this chapter. Reading a real implementation is most useful when you can predict the shapes and identify the invariants before running it.

Explain it back

Explain one token position's journey through this pre-norm block to a colleague, without reading the code: what is preserved, what attention adds, what gets normalized, and what the shared MLP transforms. Use the lab's zero-branch identity experiment to explain why the residual path matters. Your study partner will identify one gap and ask one follow-up at a time; an accurate explanation must then be followed by a correct application to a new case before mastery is recorded.

Check yourself

The answers and grading rubrics live in [quiz.json](#). Attempt the questions before opening the answers. The application cases distinguish knowing the vocabulary from being able to diagnose a block.

1. Which state is the input to the second normalization in a pre-norm block?
2. Which axes does `LayerNorm(C)` normalize for a (B, T, C) tensor?
3. Why can a LayerNorm module produce a nonzero output mean?
4. What changes if the MLP activation is removed between its affine maps?
5. What output and input gradient should the zero-branch identity experiment produce?
6. What should happen to earlier outputs when only the final position changes, and how does the wrong normalization differ?
7. Explain why the scalar residual experiment is evidence for a direct derivative path but not a proof of stable deep training.
8. Explain how the MLP can use context while operating independently at every position, and apply that to moving the perturbation to an earlier position.

Sources

1. PyTorch, LayerNorm API: axes, population variance, epsilon, learned affine parameters, and train/eval statistics. Documentation checked 2026-10-10: <https://docs.pytorch.org/docs/2.14/generated/torch.nn.LayerNorm.html>
2. PyTorch, GELU API: definition and approximation options, checked 2026-10-10: <https://docs.pytorch.org/docs/2.14/generated/torch.nn.GELU.html>
3. Andrej Karpathy, nanoGPT, [model.py](#), inspected 2026-10-10. Public source supporting the implementation comparison: <https://github.com/karpathy/nanoGPT/blob/master/model.py>
4. Xiong et al., "On Layer Normalization in the Transformer Architecture," 2020. Placement, initialization analysis and warm-up experiments: <https://arxiv.org/abs/2002.04745>
5. Vaswani et al., "Attention Is All You Need," 2017, architecture and position-wise feed-forward sections: <https://arxiv.org/html/1706.03762v7>
6. Ba, Kiros and Hinton, "Layer Normalization," 2016: <https://arxiv.org/abs/1607.06450>
7. Andrej Karpathy, "Let's build GPT: from scratch, in code, spelled out," creator chapter markers verified 2026-10-10: <https://www.youtube.com/watch?v=kCc8FmEb1nY>
8. KrypteiaSec Academy, chapter baseline and preceding multi-head attention chapter, inspected 2026-10-10: <https://krypteiasec.com/academy/llm-fundamentals/ch6> and <https://krypteiasec.com/academy/llm-fundamentals/ch5>
9. Local execution evidence: [build/lab-output.txt](#); implementation definitions: [labs/tutorlab/gpt.py](#) and [labs/tutorlab/block_demo.py](#); regression checks: [labs/tests/test_block_demo.py](#). All numerical experiment results in this chapter come from the saved lab run. No learner mastery or language-model performance is inferred from those results.