

LLM Systems & Interpretability

Gavin · AI-assisted course design

Contents

1	Curriculum	11
2	LLM Systems & Interpretability	11
2.1	Curriculum v6 — Public Curriculum	11
2.1.1	Course overview	11
2.1.2	Intended outcomes	11
3	Course Roadmap	11
4	How This Course Works	12
5	Module 1 — Foundations: From AI to LLMs	12
5.1	1.1 — The AI Zoo: What Counts as AI?	13
5.1.1	Core questions	13
5.1.2	Topics	13
5.1.3	Math to know / refresh	13
5.1.4	Early labs	13
5.2	1.2 — Text Becomes Numbers	13
5.2.1	Topics	13
5.2.2	Core question	14
5.2.3	Math to know / refresh	14
5.2.4	Lab ideas	14
5.3	1.3 — Neural Networks Without the Mysticism	14
5.3.1	Topics	14
5.3.2	Math to know / refresh	14
5.3.3	Lab ideas	14
6	Module 2 — The Transformer	14
6.0.1	Recommended Lab — Reading <i>Attention Is All You Need</i>	15
6.1	2.1 — The Residual Stream	15
6.1.1	Topics	15
6.1.2	Math to know / refresh	15
6.1.3	Lab ideas	15
6.2	2.2 — Attention	15
6.2.1	Topics	15
6.2.2	Core equations	15
6.2.3	Math to know / refresh	16
6.2.4	Lab ideas	16
6.3	2.3 — Multilayer Perceptrons (MLPs), Features, and Superposition	16
6.3.1	Topics	16
6.3.2	Math to know / refresh	16
6.3.3	Lab ideas	16
6.4	2.4 — From the Original Transformer to Modern LLMs	16
6.4.1	Topics	16
6.4.2	Math to know / refresh	16
6.4.3	Lab ideas	17

6.5	2.5 — Mixture of Experts	17
6.5.1	Topics	17
6.5.2	Math to know / refresh	17
6.5.3	Lab ideas	17
7	Module 3 — Where Models Come From	17
7.1	3.1 — Pretraining	17
7.1.1	Topics	17
7.1.2	Math to know / refresh	18
7.1.3	Lab ideas	18
7.2	3.2 — Post-Training, Alignment, and Safety Learned into Weights	18
7.2.1	Topics	18
7.2.2	Core distinction	18
7.2.3	Math to know / refresh	18
7.2.4	Lab ideas	18
8	Module 4 — Running Models and Turning Them into Systems	18
8.1	4.1 — Inference: Running a Trained Model	18
8.1.1	Topics	18
8.1.2	Math to know / refresh	19
8.1.3	Lab ideas	19
8.2	4.2 — Reasoning and Inference-Time Compute	19
8.2.1	Topics	19
8.2.2	Math to know / refresh	19
8.2.3	Lab ideas	19
8.3	4.3 — From an LLM to ChatGPT/Gemini-Like Products	19
8.3.1	Topics	19
8.3.2	Recurring question	20
8.3.3	Math to know / refresh	20
8.3.4	Lab ideas	20
8.4	4.4 — Tool Calling, Agents, and External Governance	20
8.4.1	Topics	20
8.4.2	Math to know / refresh	20
8.4.3	Lab ideas	20
8.5	4.5 — Build Our Own Tiny Codex	20
8.5.1	Goal	20
8.5.2	Components	20
8.5.3	Math to know / refresh	21
8.5.4	Lab ideas	21
9	Module 5 — Mechanistic Interpretability	21
9.1	5.1 — Looking Inside a Model	21
9.1.1	Topics	21
9.1.2	Math to know / refresh	21
9.1.3	Lab ideas	21
9.2	5.2 — Sparse Autoencoders and Learned Features	21

9.2.1	Topics	21
9.2.2	Math to know / refresh	22
9.2.3	Lab ideas	22
9.3	5.3 — Steering, Ablation, and Causal Intervention	22
9.3.1	Topics	22
9.3.2	Math to know / refresh	22
9.3.3	Lab ideas	22
9.4	5.4 — Emotion, Persona, and Internal State	22
9.4.1	Topics	22
9.4.2	Math to know / refresh	23
9.4.3	Lab ideas	23
10	Module 6 — Experimental Methods and Integration	23
10.1	6.1 — Experimental Design for LLM Labs	23
10.1.1	Topics	23
10.1.2	Math to know / refresh	23
10.1.3	Lab ideas	23
10.2	6.2 — The LLM Microscope	23
10.2.1	Goal	23
10.2.2	Likely pipeline	24
10.2.3	Math to know / refresh	24
10.2.4	Lab ideas	24
11	Research Capstone — Ask Your Own Question	24
12	1.1 — The AI Zoo: What Counts as AI?	24
12.1	Learning goals	24
12.2	1. Start by separating the layers	25
12.3	2. Explicit knowledge and searching possible futures	26
12.3.1	Symbolic and rule-based systems	26
12.3.2	Search and planning	26
12.4	3. Optimization and evolution	26
12.5	4. Learning a mapping from examples	27
12.6	5. Ask where the learning signal comes from	28
12.6.1	Supervised learning	28
12.6.2	Unsupervised learning	28
12.6.3	Self-supervised learning	28
12.6.4	Reinforcement learning	28
12.7	6. Where large language models fit	29
12.8	7. Images, video, speech, and multimodal systems	29
12.8.1	Image generation	29
12.8.2	Video generation	30
12.8.3	Speech and audio	30
12.8.4	Multimodal models and multimodal products	30
12.9	8. Assembling an agent around models	30
12.10	9. Use a map, not a single family tree	32

12.11	Check your understanding	32
12.11.1	Discussion	33
12.12	More Learning	33
12.13	References	33
13	1.2 — Text Becomes Numbers	34
13.1	Prerequisites and learning goals	34
13.2	1. Establish the interfaces	35
13.3	2. Why tokens are not words	35
13.3.1	Bytes, characters, and visible marks	36
13.4	3. A conversation needs a representation too	36
13.5	4. Token IDs select embedding rows	37
13.5.1	An original six-token example	37
13.5.2	Coordinates require a reference frame	38
13.6	5. Context changes representations	38
13.7	6. Logits put scores on the vocabulary	39
13.8	7. Softmax turns relative scores into probabilities	39
13.9	8. Selection is a separate decision	40
13.10	9. Continue the sequence and repeat	41
13.11	Check your understanding	43
13.11.1	Discussion	43
13.12	More Learning	43
13.13	References	44
14	1.3 — Neural Networks Without the Mysticism	44
14.1	A network you can account for	44
14.2	Numbers with roles and shapes	45
14.3	Weights and biases define an affine layer	46
14.4	Activation functions change what composition can express	46
14.5	The complete microscopic network	47
14.5.1	Compute every intermediate value	49
14.5.2	Intervene and explain the difference	49
14.6	Parameters persist while activations change	50
14.7	Training adds an objective and an update	50
14.8	What the neuron metaphor leaves out	52
14.9	Evaluation asks a different question	52
14.10	The bridge to Transformers	52
14.11	Check your understanding	53
14.12	More Learning	53
15	2.1 — The Residual Stream	54
15.1	Following a representation through depth	54
15.2	One vector per token position	55
15.3	Where the stream starts	55
15.4	A block computes updates and adds them	56
15.5	An exact trace with three coordinates	59

15.6	What normalization changes	60
15.7	Direct paths help optimization without guaranteeing preservation	61
15.8	Accumulation is not a readable list of facts	62
15.9	Inspect the actual boundary	63
15.10	Check your understanding	63
15.11	More Learning	64
16	2.2 — Attention	64
16.1	How one position reads another	64
16.2	Three projections of the same input	65
16.3	Scores become a distribution over sources	66
16.4	A mask defines the available information	67
16.4.1	Why a position can include itself	68
16.5	A complete three-position calculation	68
16.6	Several heads write back to the residual stream	70
16.7	Contextual computation and stored weights	71
16.8	Self-attention and cross-attention	71
16.9	What an attention map can establish	72
16.10	Check your understanding	73
16.11	More Learning	73
17	2.3 — Multilayer Perceptrons, Features, and Superposition	74
17.1	From a contextual vector to a new contribution	74
17.2	A shared transformation at every position	75
17.3	Expansion gives more nonlinear responses	77
17.4	Each unit has a read vector and a write vector	77
17.4.1	An exact residual write	78
17.5	Features are not the same objects as units	79
17.6	Three features in two dimensions	79
17.7	What the construction establishes	81
17.8	From an activation pattern to an explanatory claim	81
17.9	Check your understanding	82
17.10	Return to the original paper	83
17.11	More Learning	83
18	2.4 — From the Original Transformer to Modern LLMs	83
18.1	Reading a model as a set of design choices	83
18.2	From two stacks to one causal sequence	84
18.3	Giving attention information about position	85
18.3.1	An exact two-coordinate rotation	85
18.3.2	Extending the construction to a head	86
18.4	A brief normalization checkpoint	87
18.5	Query heads and KV heads need not have equal counts	87
18.6	Reusing previous work with a KV cache	88
18.7	Context length and local attention describe different limits	88
18.8	Turn cache dimensions into bytes	89

18.9	Read configuration metadata without inventing runtime facts	90
18.10	Precision changes representation and memory	91
18.11	Check your understanding	92
18.12	More Learning	92
19	2.5 — Mixture of Experts	93
19.1	Choosing which computation to run	93
19.2	Dense and routed feed-forward branches	93
19.3	From router logits to a residual update	94
19.4	A complete four-expert calculation	97
19.5	The gpt-oss specimen	98
19.6	Count parameters before estimating memory	99
19.7	A fresh route at each position and layer	100
19.8	Learning routes and keeping experts useful	100
19.9	What makes an expert an expert	101
19.10	Check your understanding	101
19.11	More Learning	102
20	Lab 01 Map AI Systems	103
20.1	Purpose	103
20.2	The questions to keep separate	103
20.3	Two worked examples	104
20.3.1	Example 1 A polished help window	104
20.3.2	Example 2 One text model call	104
20.4	Your worksheet	104
20.5	Six system cards	105
20.5.1	A Greenhouse alarm	105
20.5.2	B Route desk	105
20.5.3	C Sorting gate	105
20.5.4	D Surface inspector	105
20.5.5	E Illustrated request	105
20.5.6	F Warehouse rover	106
20.6	Predict three interventions	106
20.7	Your artifact and review	106
20.8	Self-check answer key	106
20.8.1	Card classifications and layers	106
20.8.2	Intervention answers	107
20.8.3	Check the quality of your reasoning	107
21	Lab 02 — Inspect Tokenization	108
21.1	Goal and prerequisites	108
21.2	Local setup and commands	108
21.3	1. Identify the specimen	109
21.4	2. Predict before inspecting	110
21.4.1	Prose and whitespace	110
21.4.2	Code and identifiers	110

21.4.3	Numbers	110
21.4.4	Unicode	111
21.5	3. Collect the observations	111
21.6	4. Inspect special tokens separately	111
21.7	5. Separate encoding length from model context	112
21.8	6. Interpret and submit	112
21.8.1	Completion criteria	112
21.9	More Learning	112
21.10	References	112
22	Tiny Network, Visible Computation	113
22.1	Goal	113
22.2	The fixed network	113
22.3	Part 1 — Make a hand trace	113
22.4	Part 2 — Predict interventions	114
22.5	Part 3 — Implement or independently recalculate	114
22.5.1	Coding route	114
22.5.2	Paper route	115
22.5.3	Interpret the comparison	116
22.6	Part 4 — Optional one-parameter training step	116
22.7	Checkpoints and solutions	116
22.8	Submit an explanation	117
22.9	More Learning	117
23	Reading the Original Transformer Paper	118
23.1	Goal	118
23.2	Part 1 — Orient before Lesson 2.1	118
23.2.1	Find the problem and the proposed change	118
23.2.2	Make an orientation sketch	119
23.2.3	Stop deliberately	119
23.3	Part 2 — Return after Lesson 2.3	119
23.3.1	Trace rather than recite	119
23.3.2	Compare with one documented decoder-only model	120
23.3.3	Separate evidence from extrapolation	120
23.4	Self-check and discussion guide	120
23.5	Submit your reading note	121
23.6	More Learning	121
24	Trace a Residual Stream	121
24.1	Goal	121
24.2	Part 1 — Predict a fresh residual sequence	121
24.3	Part 2 — Compare three different changes	122
24.4	Part 3 — Optional implementation of the toy	122
24.5	Part 4 — Optional trained-model observation	122
24.5.1	Use a bounded public specimen	122
24.5.2	Local setup and commands	123

24.5.3	Implementation requirements	124
24.5.4	Observe and interpret	124
24.6	Checkpoints and solutions	125
24.7	Submit an explanation	125
24.8	More Learning	125
25	Compute and Perturb Attention	126
25.1	Goal	126
25.2	The fixed head	126
25.3	Part 1 — Compute before coding	126
25.4	Part 2 — Predict controlled changes	127
25.5	Part 3 — Implement small arrays	127
25.5.1	Learner scaffold	127
25.5.2	Run and compare	128
25.5.3	Hand-worked alternative	129
25.6	Part 4 — Explain what changed	129
25.7	Completed reference implementation	130
25.8	Optional checks with explanatory value	132
25.9	Completion criteria	132
26	Inspect MLP Responses and Feature Geometry	133
26.1	Goal	133
26.2	Part 1 — Trace a new MLP input	133
26.3	Part 2 — Test a dictionary under different feature combinations	134
26.4	Part 3 — Separate a distributed code from superposition	134
26.5	Part 4 — Optional standard-library implementation	135
26.6	Part 5 — Optional contrasting-prompt observation	135
26.6.1	Choose the documented specimen	135
26.6.2	Local runner and two observation phases	136
26.6.3	Fix prompts and a position before looking at activations	137
26.6.4	Name and verify the capture boundary	137
26.6.5	Explore, freeze a prediction, then check	138
26.6.6	Explain the first-block control	138
26.7	Analytically derived checkpoints	139
26.8	Submit evidence and an explanation	139
26.9	More Learning	139
27	Read Model Configurations and Estimate Memory	140
27.1	Goal	140
27.2	Part 1 — Establish exactly which sources you read	140
27.3	Part 2 — Translate keys into a computation	141
27.4	Part 3 — Predict how cache memory changes	142
27.4.1	A — GPT-2 full-length payload	142
27.4.2	B — Gemma full-length payload	143
27.4.3	C — Gemma with bounded local retention	143
27.5	Part 4 — Estimate weights without loading them	143

27.6	Part 5 — Optional standard-library calculator	144
27.6.1	Running the supplied tools	144
27.7	Checkpoints and analytical answers	145
27.8	Submit a bounded conclusion	146
27.9	More Learning	146
28	Trace and Perturb Expert Routing	147
28.1	Goal	147
28.2	The fixed two-layer toy	147
28.2.1	Layer 0	148
28.2.2	Layer 1	148
28.3	Part 1 — Compute a baseline trace	148
28.4	Part 2 — Specify and predict interventions	149
28.5	Part 3 — Implement and save the trace	149
28.5.1	Data and learner scaffold	149
28.5.2	Verification checks	151
28.6	Part 4 — Inspect one real configuration without loading a model	151
28.7	Reference functions	152
28.7.1	Running the supplied completed reference	154
28.8	Solutions and interpretation	155
28.8.1	Baseline	155
28.8.2	Interventions at position 0	155
28.8.3	Parameter accounting	156
28.9	Optional future extension — Trace a trained model	157
28.10	Submission	157
28.11	More Learning	157

1 Curriculum

2 LLM Systems & Interpretability

2.1 Curriculum v6 — Public Curriculum

2.1.1 Course overview

This is a hands-on course for technically curious learners who want a practical understanding of modern AI systems, from the broad AI landscape down to the internal mechanics of large language models and outward again to complete systems such as ChatGPT, Gemini, Claude, and coding agents.

The course emphasizes understanding how the pieces fit together, inspecting real open models, and using small experiments and labs to make abstract ideas concrete.

2.1.2 Intended outcomes

By the end of the course, a learner should be able to:

- distinguish major families of AI and explain how they differ;
 - distinguish an algorithm, a trained model, a neural-network architecture, an inference engine, an agent harness, and a product;
 - explain the path from text to tokens to embeddings to generated output;
 - read the major parts of a modern Transformer architecture;
 - explain attention, residual streams, multilayer perceptrons (MLPs), normalization, positional encoding, KV caches, and Mixture of Experts;
 - distinguish pretraining, post-training, fine-tuning, safety training, and inference-time behavior;
 - explain where behavior can live: weights, context, inference settings, orchestration, or external governance;
 - understand reasoning effort and inference-time computation;
 - build a small tool-using coding agent around an open model;
 - inspect activations, routing, and learned features in open models;
 - perform controlled interventions such as ablation, steering, and forced MoE routing;
 - run reproducible instructional experiments and distinguish correlation from causation.
-

3 Course Roadmap

Module 1 begins by looking broadly at what AI means, what kinds of systems exist, and how they differ. From there we narrow our focus toward large language models, introducing tokenization and the neural-network foundations needed to understand them.

Module 2 focuses on Transformers, the architecture behind most modern LLMs. We explore the residual stream, attention, multilayer perceptrons, Mixture of Experts, and the evolution from the original Transformer to modern models.

Module 3 explains how those models come to exist in the first place. We look at how pretraining turns initially untrained weights into a language model, and how post-training then shapes that model into something that can follow instructions, use tools, reason, and behave more like a useful assistant.

Module 4 moves from the model itself to what happens when we actually use one. We look at inference and then at the larger systems built around LLMs, including context, retrieval, memory, tools, agents, safety controls, and coding assistants.

Module 5 turns inward again, introducing mechanistic interpretability and techniques for inspecting activations, learned features, steering, ablation, and other ways of investigating what is happening inside a model.

Module 6, finally, brings us to experimental methods: how to design careful LLM experiments, control for misleading effects, and bring the techniques from the course together in a reusable model-inspection environment. The course ends with a small research capstone, inviting the learner to choose a question they find interesting and investigate it for themselves.

4 How This Course Works

The course has two kinds of activity: **Lessons** and **Labs**. Lessons provide the conceptual spine; Labs make ideas concrete through coding, model inspection, mathematics, paper reading, experiments, or tooling. Some Labs use local hardware, while others use AWS or rented GPU compute when the hardware meaningfully enables the exercise.

Mathematics is introduced when it becomes useful rather than front-loaded as a separate prerequisite course. Each lesson identifies **Math to know / refresh**, distinguishing what is needed immediately from material that is useful to revisit.

We recommend reading a lesson once for the overall picture, then returning to difficult sections interactively and doing Labs where they are recommended. Important ideas should sometimes be explained back, predicted, sketched, compared, or applied rather than simply reread.

Before suitable experiments, make a prediction and record why you expect it. Comparing that prediction with the result is part of the learning process.

Optional papers, documentation, videos, mathematical refreshers, and deeper material appear under **More Learning**. The core lesson should remain understandable without requiring every optional resource.

A recurring question throughout the course is: **Where does this behavior live?** The answer may involve the tokenizer, architecture, trained weights, prompt/context, inference settings, agent or orchestration layer, external controls, or product layer.

5 Module 1 — Foundations: From AI to LLMs

Module 1 begins with the wider AI landscape, then deliberately narrows toward large language models. After the AI Zoo, the remaining lessons introduce the representations and neural-network foundations needed before we study Transformers in detail.

5.1 1.1 — The AI Zoo: What Counts as AI?

5.1.1 Core questions

- What does “AI” actually refer to?
- Which systems are algorithms, which are trained models, and which are products built from many components?
- How is an LLM different from an image generator, video model, genetic algorithm, or agent?

5.1.2 Topics

- symbolic and rule-based AI;
- search and planning;
- optimization methods;
- evolutionary and genetic algorithms;
- classical machine learning;
- neural networks;
- supervised, unsupervised, and self-supervised learning;
- reinforcement learning;
- LLMs;
- image-generation models;
- video-generation models;
- speech and audio models;
- multimodal models;
- agents and orchestration systems;
- how an LLM may interpret a prompt that is then executed by a different model.

5.1.3 Math to know / refresh

Needed now: none beyond basic functions and probability intuition.

Useful refresh: optimization as searching for maxima/minima; probability distributions.

5.1.4 Early labs

- Map familiar AI systems into the categories above.
- Identify which parts of a modern AI product are models versus software around models.
- Optional AWS familiarization lab: billing dashboard, budgets, VPC/EC2 vocabulary, and launch templates without running expensive compute.

From this point onward, the course focuses primarily on large language models.

5.2 1.2 — Text Becomes Numbers

5.2.1 Topics

- vocabulary;
- tokenizer;
- tokens versus words;
- token IDs;
- special tokens;

- embedding matrices;
- embeddings;
- logits;
- softmax;
- next-token prediction;
- autoregressive generation.

5.2.2 Core question

What precisely happens between typing text and the model choosing another token?

5.2.3 Math to know / refresh

Needed now: vectors, indexing, exponentials, probability normalization.

Useful refresh: matrix lookup notation and basic probability.

5.2.4 Lab ideas

- Inspect a real open-model tokenizer.
 - Encode and decode arbitrary text.
 - Compare tokenization of ordinary prose, code, numbers, and unusual words.
-

5.3 1.3 — Neural Networks Without the Mysticism

5.3.1 Topics

- vectors and matrices;
- weights;
- biases;
- activations;
- layers;
- linear transformations;
- nonlinear activation functions;
- what a “neuron” means in modern networks;
- training versus inference.

5.3.2 Math to know / refresh

Needed now: matrix multiplication, dot products, functions, simple derivatives.

Useful refresh: coordinate transformations and linear algebra notation.

5.3.3 Lab ideas

- Build and inspect a microscopic neural network.
 - Manually compute a forward pass for a tiny example.
-

6 Module 2 — The Transformer

The Transformer architecture was introduced in 2017 in *Attention Is All You Need* by Vaswani and colleagues. Its attention-based design had an enormous impact on modern AI and became the archi-

tectural foundation for most large language models. This module works through the main pieces of the Transformer and then follows how they evolved into today's LLMs.

6.0.1 Recommended Lab — Reading *Attention Is All You Need*

Use the original paper as a recurring reference rather than trying to understand it all at once. At the start of the module, read the abstract, introduction, architecture figure, and conclusion; identify the problem the authors were trying to solve and note unfamiliar terminology. After 2.3, return to the architecture and attention sections, connect them to what you have learned, and identify which parts modern decoder-only LLMs retain, change, or omit.

6.1 2.1 — The Residual Stream

6.1.1 Topics

- embeddings as vectors;
- Transformer blocks;
- residual stream;
- residual connections;
- normalization;
- information accumulation across layers.

6.1.2 Math to know / refresh

Needed now: vector addition, norms, affine transformations.

Useful refresh: geometric interpretation of high-dimensional vectors.

6.1.3 Lab ideas

- Load a small Transformer and record the residual stream after each layer.
- Compare layer-to-layer changes for selected tokens.

6.2 2.2 — Attention

6.2.1 Topics

- queries, keys, and values;
- attention scores;
- causal masks;
- self-attention;
- multi-head attention;
- what attention does and does not tell us.

6.2.2 Core equations

$$Q = XW_q$$

$$K = XW_k$$

$$V = XW_v$$

$$\text{Attention}(Q,K,V) = \text{softmax}(QK^T / \sqrt{d})V$$

6.2.3 Math to know / refresh

Needed now: matrix multiplication, transpose, dot products, exponentials, softmax, basic probability.

Useful refresh: cosine similarity and vector projection.

6.2.4 Lab ideas

- Compute a tiny attention example by hand.
 - Visualize attention matrices for real sentences.
-

6.3 2.3 — Multilayer Perceptrons (MLPs), Features, and Superposition

6.3.1 Topics

- feed-forward / multilayer perceptron (MLP) layers;
- expansion dimensions;
- activation functions;
- features;
- polysemantic neurons;
- superposition;
- why one neuron rarely means one concept.

6.3.2 Math to know / refresh

Needed now: matrix transformations, nonlinear functions.

Useful refresh: basis vectors, linear combinations, dimensionality.

6.3.3 Lab ideas

- Compare MLP activations for contrasting prompts.
 - Inspect which units respond across different contexts.
-

6.4 2.4 — From the Original Transformer to Modern LLMs

6.4.1 Topics

- decoder-only models;
- positional information;
- RoPE;
- RMSNorm;
- grouped-query attention;
- KV caches;
- context windows;
- local/sliding attention;
- quantization;
- model configuration files.

6.4.2 Math to know / refresh

Needed now: vectors, rotations, norms.

Useful refresh: complex-number or rotation-matrix intuition for RoPE; numerical precision.

6.4.3 Lab ideas

- Read a real model configuration file and translate its parameters into plain English.
 - Estimate memory requirements at different numerical precisions.
-

6.5 2.5 — Mixture of Experts

6.5.1 Topics

- dense models versus MoE;
- routers;
- expert networks;
- top-k routing;
- routing probabilities;
- total versus active parameters;
- expert specialization;
- load balancing;
- gpt-oss as a primary specimen.

6.5.2 Math to know / refresh

Needed now: softmax, ranking/top-k selection, weighted sums.

Useful refresh: sparse computation and conditional probability.

6.5.3 Lab ideas

- Capture which experts are selected for each token and layer.
 - Force, prohibit, or boost selected experts.
 - Compare routing across prompt categories.
-

7 Module 3 — Where Models Come From

7.1 3.1 — Pretraining

7.1.1 Topics

- training corpora;
- prediction objectives;
- loss;
- gradient descent;
- backpropagation;
- batches;
- learning rates;
- checkpoints;
- compute requirements;
- what becomes encoded in weights.

7.1.2 Math to know / refresh

Needed now: derivatives, gradients, chain rule, logarithms, cross-entropy.

Useful refresh: multivariable calculus and optimization landscapes.

7.1.3 Lab ideas

- Train an intentionally tiny language model.
 - Plot loss over training and inspect checkpoints.
-

7.2 3.2 — Post-Training, Alignment, and Safety Learned into Weights

7.2.1 Topics

- base models versus assistant models;
- instruction tuning;
- supervised fine-tuning;
- preference training;
- RLHF and related methods;
- reasoning training;
- tool-use training;
- policy-following and refusal behavior;
- safety behavior learned into model weights;
- limits of trained-in safety.

7.2.2 Core distinction

A model may know how to do something while post-training changes how it responds to requests for that capability.

7.2.3 Math to know / refresh

Needed now: loss functions, probability, expected reward.

Useful refresh: reinforcement-learning terminology and optimization.

7.2.4 Lab ideas

- Compare base-model and instruction-tuned behavior where suitable open checkpoints exist.
 - Classify examples of behavior as likely coming from pretraining, post-training, or system-level controls.
-

8 Module 4 — Running Models and Turning Them into Systems

8.1 4.1 — Inference: Running a Trained Model

8.1.1 Topics

- checkpoint loading;
- inference engines;
- GPU memory;

- numerical precision;
- BF16, FP16, FP8, 4-bit formats;
- quantization;
- batching;
- sampling;
- temperature;
- top-p;
- deterministic generation;
- KV cache.

8.1.1.2 Math to know / refresh

Needed now: probability distributions, logarithms, numerical representation.

Useful refresh: floating-point precision and information loss.

8.1.1.3 Lab ideas

- Run an open model locally or on rented compute.
 - Measure speed, memory, and output changes under different sampling settings.
-

8.2 4.2 — Reasoning and Inference-Time Compute

8.2.1 Topics

- reasoning tokens;
- inference-time compute;
- low/medium/high reasoning effort;
- test-time scaling;
- trained reasoning behavior;
- same weights versus different models;
- what reasoning controls can and cannot mean.

8.2.2 Math to know / refresh

Needed now: computational complexity intuition.

Useful refresh: scaling relationships and expected-value reasoning.

8.2.3 Lab ideas

- Compare reasoning-effort settings on a model that exposes them.
 - Measure latency, token use, and task performance.
-

8.3 4.3 — From an LLM to ChatGPT/Gemini-Like Products

8.3.1 Topics

- system/developer/user messages;
- conversation formats;
- context windows;
- files and retrieval;

- memory;
- tool definitions;
- safety layers;
- context management and compaction;
- product behavior versus model behavior.

8.3.2 Recurring question

Where does each behavior live: weights, prompt, inference engine, orchestration, or UI/product layer?

8.3.3 Math to know / refresh

Minimal new mathematics.

8.3.4 Lab ideas

- Trace a hypothetical request through a full product stack.
 - Design a minimal conversation protocol around an open model.
-

8.4 4.4 — Tool Calling, Agents, and External Governance

8.4.1 Topics

- function calling;
- tool schemas;
- observations;
- agent loops;
- stopping conditions;
- permissions;
- sandboxes;
- network controls;
- approval gates;
- safety classifiers and safeguard models;
- capability versus authority.

8.4.2 Math to know / refresh

Minimal new mathematics.

8.4.3 Lab ideas

- Give an open model a calculator or file-read tool.
 - Add permissions and observe what the harness, rather than the model, prevents.
-

8.5 4.5 — Build Our Own Tiny Codex

8.5.1 Goal

Build an actual coding agent around an open model.

8.5.2 Components

- model;

- system instructions;
- repository reader;
- file editor;
- shell/test tool;
- agent loop;
- sandbox;
- permissions;
- context management;
- execution limits.

8.5.3 Math to know / refresh

No major new mathematics; this is primarily a systems lesson.

8.5.4 Lab ideas

- Repair a deliberately broken codebase.
 - Compare unrestricted versus governed tool access.
 - Record failures and retry behavior.
-

9 Module 5 — Mechanistic Interpretability

9.1 5.1 — Looking Inside a Model

9.1.1 Topics

- hooks;
- activation caches;
- residual-stream inspection;
- attention inspection;
- MLP activations;
- logit lens;
- probing;
- correlation versus intervention.

9.1.2 Math to know / refresh

Needed now: vector norms, dot products, projections, cosine similarity.

Useful refresh: linear regression/probes and statistical association.

9.1.3 Lab ideas

- Track an internal representation through successive layers.
 - Compare activations for controlled prompt pairs.
-

9.2 5.2 — Sparse Autoencoders and Learned Features

9.2.1 Topics

- superposition;

- sparse representations;
- dictionary learning;
- autoencoders;
- sparse autoencoders;
- learned features;
- feature activation;
- feature interpretation;
- Gemma Scope as a primary specimen.

9.2.2 Math to know / refresh

Needed now: linear combinations, reconstruction error, sparsity.

Useful refresh: optimization with regularization; basis/dictionary representations.

9.2.3 Lab ideas

- Inspect existing SAE features.
 - Compare feature activations across prompt families.
-

9.3 5.3 — Steering, Ablation, and Causal Intervention

9.3.1 Topics

- activation vectors;
- contrastive vectors;
- activation addition;
- ablation;
- forced routing;
- causal interventions;
- controls and confounders.

9.3.2 Math to know / refresh

Needed now: vector subtraction, projection, normalization, effect measurement.

Useful refresh: causal inference vocabulary.

9.3.3 Lab ideas

- Construct a contrastive activation direction.
 - Add/subtract it at selected layers.
 - Measure behavioral and internal changes.
-

9.4 5.4 — Emotion, Persona, and Internal State

9.4.1 Topics

- emotion-related representations;
- persona vectors;
- behavioral correlates;
- causal intervention;

- representation versus subjective experience;
- anthropomorphism;
- careful interpretation of mechanistic findings.

9.4.2 Math to know / refresh

Needed now: similarity metrics, averaging, variance.

Useful refresh: dimensionality reduction and statistical testing.

9.4.3 Lab ideas

- Compare emotional-state prompt sets.
 - Test whether a discovered direction generalizes to held-out prompts.
 - Intervene and measure whether behavior changes systematically.
-

10 Module 6 — Experimental Methods and Integration

10.1 6.1 — Experimental Design for LLM Labs

10.1.1 Topics

- hypotheses;
- independent and dependent variables;
- controls;
- confounders;
- train/test separation;
- sample sizes;
- reproducibility;
- falsification;
- exploratory versus confirmatory analysis.

10.1.2 Math to know / refresh

Needed now: mean, variance, distributions, confidence intervals, effect sizes.

Useful refresh: hypothesis testing, multiple comparisons, regression.

10.1.3 Lab ideas

- Turn one vague claim about model behavior into a falsifiable lab plan.
 - Record expected measurements before running the experiment, then compare prediction with result.
-

10.2 6.2 — The LLM Microscope

10.2.1 Goal

Integrate the course into a reusable learning environment for instrumenting open models, capturing activations and MoE routing, applying interventions, and producing reproducible lab results.

10.2.2 Likely pipeline

open model → instrumented inference → activation/routing capture → intervention → repeated experiment → analysis → visualization → write-up

10.2.3 Math to know / refresh

Depends on the chosen lab or advanced topic.

10.2.4 Lab ideas

- Provision appropriate AWS compute only when needed.
 - Run a complete instructional experiment with saved configuration, data, plots, and conclusions.
-

11 Research Capstone — Ask Your Own Question

Choose a question about LLMs or AI systems that genuinely interests you and use the concepts, tools, and experimental habits from this course to investigate it. Define the question, choose an appropriate method, and document what you learn.

12 1.1 — The AI Zoo: What Counts as AI?

Imagine asking an assistant: “Make a poster for a community astronomy night, then explain the design.” The answer appears in one conversation. Behind that interface, a language model might interpret your request, a separate image model might generate the artwork, and ordinary software might save the file. Another system could combine those capabilities differently.

Calling the whole experience “AI” tells us very little about how it works. This lesson builds a map for asking better questions: What computation is happening? What was learned? Which component produced the behavior?

AI is a broad field concerned with systems that perform tasks involving abilities such as perception, learning, reasoning, planning, and action. Its boundaries overlap with statistics, optimization, and computer science. A useful engineering description identifies mechanisms rather than trying to settle whether every individual program deserves the label.

12.1 Learning goals

By the end of this lesson, you should be able to:

- Explain how explicit rules, search, optimization, and learned models solve different kinds of problems.
- Distinguish an algorithm, an architecture, trained weights, an inference runtime, orchestration, and a product.
- Identify the learning signal in supervised, unsupervised, self-supervised, and reinforcement learning.
- Explain how language, image, video, speech, and multimodal models can participate in one system.
- Trace an agent’s proposed action through the software that actually executes it.

i Math to know / refresh

Needed now: Basic functions and probability intuition. A function maps inputs to outputs; a probability distribution assigns probabilities to possible outcomes. No calculus is required.

Useful refresh: Optimization means looking for inputs or parameters that make an objective smaller or larger. For a finite set of mutually exclusive and exhaustive outcomes, probabilities are nonnegative and sum to one.

Side trail: Graphs describe states and possible transitions. Gradients describe how a numerical objective changes locally. Both will become useful later; neither needs a formal treatment here.

12.2 1. Start by separating the layers

Suppose we build the astronomy-poster assistant. Six distinctions help us describe it precisely.

An **algorithm** is a procedure: follow rules to perform a computation. Searching candidate routes, updating a model during training, and choosing an output during generation all involve algorithms. “Algorithm” is therefore broader than “machine learning.”

A neural-network **architecture** specifies the organization of a model’s computation: which components connect, which operations they perform, and the shapes of their inputs and outputs. A Transformer is an architecture family. It leaves many design choices open and does not specify one set of learned capabilities.

Trained weights are numerical parameters produced by training. Think of the architecture as defining a family of possible functions and the weights as selecting a particular member. A saved checkpoint contains these parameters and may contain other training state. Two checkpoints with the same architecture can behave differently.

An **inference runtime**, or inference engine, loads the model and executes it. Hardware, numerical precision, generation settings, and implementation choices affect speed, memory use, and sometimes outputs. Inference means using a model; training means adjusting its learned parameters. The two involve computation for different purposes.

Orchestration connects model calls with surrounding software: assembling context, retrieving documents, calling tools, checking results, and deciding whether to continue. An agent harness is one kind of orchestration. A **product** packages the system into an experience with an interface, accounts, storage, and operational controls.

These are analytical layers, not necessarily six separate programs. “Model” itself is used loosely, so ask whether a speaker means an architecture, a checkpoint, or a hosted service.

Consider four changes to our assistant. Replacing its checkpoint changes learned parameters. Adding an event brief to its context changes the input. Adjusting a sampling setting changes generation behavior. Giving it a file-saving tool changes its available actions. Similar improvements at the interface can have very different causes.

Our recurring question is: **Where does this behavior live?**

12.3 2. Explicit knowledge and searching possible futures

12.3.1 Symbolic and rule-based systems

A symbolic system represents information using explicit objects, facts, relationships, or rules. A rule engine can match current facts against rule conditions and apply the matching rules. CLIPS is a concrete, maintained example of software for building rule-based expert systems. CLIPS documentation

Here is a small illustrative system:

- Fact: the event is outdoors.
- Fact: the weather forecast says rain.
- Rule: if an event is outdoors and rain is forecast, recommend the indoor backup venue.

The conclusion follows because the conditions match. No training dataset is necessary for this rule. The system can show which facts and rule led to its recommendation.

That transparency has limits. If nobody supplied a rule for strong wind, the system may miss that problem. If the forecast is wrong, a correctly executed rule can still recommend the wrong action. Larger knowledge bases also require careful handling of conflicting rules and missing information.

An ordinary conditional statement does not, by itself, make a program an interesting AI system. The connection to symbolic AI becomes useful when explicit representations support reasoning over a domain. Rule-based components also remain useful inside systems containing learned models, especially where a precise condition must be enforced.

12.3.2 Search and planning

Search explores possible alternatives. Planning asks which sequence of actions can move a system from its current state to a goal. A planner needs some representation of available actions and their consequences; those representations may be supplied by people or learned.

Suppose a delivery robot can reach the venue by two routes. Route A has legs costing 3 and 6 minutes, for a total of 9. Route B has three legs costing 2 minutes each, for a total of 6. Choosing the route with fewer legs gives the wrong answer. The objective concerns total travel time.

On a large map, enumerating every complete route is expensive. Heuristic search uses estimates to prioritize promising partial routes. A* combines cost already incurred with an estimate of remaining cost; its guarantees depend on the heuristic and search conditions. Hart, Nilsson, and Raphael's original paper

Search can work without learning. A learned estimate can also guide search. These categories already overlap.

Retrieval check: In the venue example, which information came from explicit facts, which came from a rule, and which would require exploring possible action sequences?

12.4 3. Optimization and evolution

Optimization begins with a choice and an objective. A schedule might minimize lateness while respecting room capacities. A delivery plan might minimize distance while meeting time windows. Tools such as OR-Tools supply several solver families for these constrained problems. OR-Tools introduction

A tiny numerical example makes the idea concrete. Let the cost of choosing a number x be:

$$L(x) = (x - 3)^2.$$

Choosing 0 costs 9; choosing 2 costs 1; choosing 3 costs 0. We can solve this one by inspection. More complicated objectives require procedures that propose, evaluate, and improve candidates. An optimizer’s answer depends on the objective, constraints, search method, and available computation. “Optimized” does not automatically mean “globally best.”

Evolutionary algorithms search using populations of candidates. A typical genetic algorithm evaluates fitness, selects candidates, creates variants through mutation and often crossover, and repeats. Crossover combines parts of parent candidates; mutation changes a candidate. The DEAP library makes these operators explicit. [DEAP operator tutorial](#)

For an illustrative four-bit candidate, define fitness as the number of ones. The candidate 1010 has fitness 2. Flipping its second bit produces 1110 with fitness 3. That is a worked calculation, not an experimental result. In a real scheduling problem, the candidate might encode assignments, and an evaluation might be costly. Some changes would make a candidate worse or violate a constraint.

Evolutionary search is useful when candidates can be evaluated even if a convenient gradient is unavailable. It can optimize a design directly or help choose parts of a learning system. A neural network can itself be optimized using evolutionary methods. Conversely, many neural networks are trained with gradient-based optimization. “Evolutionary” describes a search strategy; “neural network” describes a model family.

Prediction check: If a timetable optimizer rewards only short walking distances, what undesirable schedule could score well? Write down a missing constraint before continuing.

12.5 4. Learning a mapping from examples

Machine learning builds useful behavior from data rather than specifying every decision rule manually. A learning procedure fits a model; the fitted model is then evaluated on cases beyond those used for fitting. Classical model families include linear models, decision trees, nearest-neighbor methods, and ensembles such as random forests. [scikit-learn’s supervised-learning guide](#)

Imagine predicting attendance from the number of advance registrations. An illustrative fitted model is $f(x) = 4 + 2x$. If $x = 3$, it predicts 10 attendees. The formula’s form and the numerical values play different roles: $a + bx$ describes a model family, while $a = 4$ and $b = 2$ specify this member.

In an actual project, data would determine whether that relationship was useful. An excellent fit to old events could fail on a new kind of event. Evaluation on held-out examples helps expose that problem; preprocessing must also avoid leaking information from evaluation data into training. [scikit-learn’s fit, predict, and evaluation workflow](#)

A **neural network** composes parameterized transformations, commonly including weighted combinations and nonlinear functions. Training adjusts parameters so these transformations become useful for the objective. A multilayer perceptron is one example; convolutional, recurrent, and Transformer architectures organize computation differently. A neural “unit” is a mathematical component, not a miniature biological brain. [Neural-network documentation](#)

Neural networks can learn intermediate representations as well as final outputs. That helps with rich inputs such as images, but it does not eliminate choices about data, architecture, objectives, or evaluation. “Classical” and “neural” are useful descriptions, not a ranking that determines which approach will work best for every task.

12.6 5. Ask where the learning signal comes from

A model family and a learning setup answer different questions. The same architecture can participate in several setups, sometimes at different stages of its training.

12.6.1 Supervised learning

Supervised learning uses example inputs paired with desired targets. An input might be an event description and its target a topic category, or registration features and a measured attendance count. Targets need not be manually written by a human; they can come from records or other measurement processes. Classification predicts categories, while regression predicts numerical quantities. scikit-learn’s supervised-learning guide

For example, a spam detector can learn from messages paired with spam/not-spam labels. The learning procedure receives information about what the answer should be for each training example.

12.6.2 Unsupervised learning

Unsupervised learning seeks structure without supplied target labels for the task. We might cluster event descriptions by similarity, then inspect whether the groups are useful. K-means, for example, fits cluster centers using a distance-based objective. “Unsupervised” does not mean the algorithm has no objective or that its groups are guaranteed to match meaningful human categories. Clustering documentation

If a cluster contains astronomy and photography events, perhaps their shared vocabulary dominates the chosen representation. Calling the cluster “science” is a human interpretation that still needs checking.

12.6.3 Self-supervised learning

Self-supervised learning constructs targets from the data itself. Hide part of a sentence and predict the hidden content, or use the beginning of a passage to predict what follows. The original text supplies the target without a separate person labeling each training example. BERT’s masked-language-model objective is a well-known example. BERT paper

For “The telescope is on the roof,” we can hide “roof” and use the rest as input. We have created a supervised-style prediction problem from an existing sentence. Terminology varies: self-supervised methods are sometimes grouped under unsupervised learning. The important distinction here is how the target was obtained.

12.6.4 Reinforcement learning

Reinforcement learning concerns an agent choosing actions in an environment and learning from rewards to improve expected return. Feedback may arrive after several actions, so the learner must address which choices contributed to the outcome. A reward evaluates what happened; it need not specify the correct action at every step. OpenAI’s introduction to RL

A robot might receive a reward for completing a delivery and penalties for collisions. A reward based only on speed could encourage behavior the designer did not intend. Defining a measurable reward and specifying the real goal are separate tasks.

RL need not use neural networks. Nor does every deployed agent learn while it runs. AlphaZero is a useful hybrid example: its training combines self-play reinforcement learning with a neural network, while tree search helps select moves. AlphaZero paper

Retrieval check: An image model predicts noise that software added to training images. Is the target supplied by a human labeler, derived from the data and corruption process, or delivered as an environmental reward? Explain the distinction rather than relying on the word “image.”

12.7 6. Where large language models fit

A language model learns statistical relationships in language sequences. Generative LLMs commonly produce text by repeatedly predicting a distribution over the next small piece of text and choosing from that distribution. Lesson 1.2 explains those pieces, called tokens, and how text becomes numbers. The GPT-3 paper provides a concrete autoregressive language-model example. GPT-3 paper

“Large” has no timeless parameter threshold. More importantly, the label does not describe a complete assistant product. Many LLMs use Transformer-based architectures, but architecture and training objective remain separate concepts. The original Transformer was introduced for sequence-to-sequence tasks including machine translation; later systems adapted its components in different ways. Original Transformer paper

Training a model to predict language can produce abilities useful across many tasks. Additional training can shape instruction following and other assistant behavior. One published recipe combines supervised demonstrations with reinforcement learning from human feedback. That recipe is an example, not a claim that every assistant follows the same process. InstructGPT paper

A prompt can change behavior without changing weights. In the poster assistant, “Use a calm, scientific tone” becomes part of the current input. Providing examples in context can also change performance without a training update, as demonstrated in GPT-3’s few-shot evaluations. GPT-3 paper

Distinguish three things: stored parameters, information in the current context, and records saved by the surrounding product. Remembering something during a conversation is not sufficient evidence that model weights changed. Similarly, fluent prose is not evidence that a claim was checked against an external source.

12.8 7. Images, video, speech, and multimodal systems

The input and output medium provides another axis of classification. It does not uniquely identify an architecture or learning algorithm.

12.8.1 Image generation

An image classifier assigns a label to an image; an image generator produces image content. One influential generative approach is diffusion. During training, images are corrupted with noise and a network learns a denoising-related prediction. During generation, a sampling procedure repeatedly uses the trained network to transform noise into a sample. Those inference steps normally reuse fixed weights rather than retraining the network for each picture. Denoising diffusion paper

Latent diffusion performs this process in a learned compressed representation, then decodes the result into pixels. Text can condition the generation process. “Latent” here means an internal representation, not a hidden original photograph waiting to be uncovered. Latent diffusion paper

Image generation also includes adversarial methods, which train a generator against a discriminator, and flow-matching methods, which learn a field that guides transformations between distributions. These are examples of different generative approaches, not a complete catalog. GAN paper, flow-matching paper

Architecture cuts across these labels: a diffusion model can use a Transformer as its neural backbone. “Transformer” therefore does not imply “language model.” Diffusion Transformer paper

12.8.2 Video generation

Video adds time. Independently generating plausible frames does not ensure that an object keeps its identity or moves coherently. Video models must represent temporal relationships as well as visual appearance. Make-A-Video illustrates one published approach that extends text-to-image generation with temporal components and learning from video. It is a historical architectural example, not a description of every current video product. Make-A-Video paper

12.8.3 Speech and audio

Speech recognition maps recorded speech toward text. Text-to-speech goes in the other direction. Other audio tasks include generating music, separating sources, or classifying sounds. Their outputs, data, and evaluation criteria differ.

Whisper is a concrete speech-recognition example trained with audio and corresponding transcripts. WaveNet is a generative audio example that predicts waveform samples conditioned on earlier samples. Autoregressive generation therefore also applies beyond written language. Whisper paper, WaveNet paper

12.8.4 Multimodal models and multimodal products

A multimodal model works across more than one modality. A vision-language model might accept an image and a question and produce text. That capability does not automatically imply it can generate images.

For a documented specimen, Gemma 3’s 4B, 12B, and 27B variants combine a vision encoder with a language-model architecture; the 1B variant lacks that vision encoder. This is a specific published model-family distinction. Gemma 3 technical report, Sections 2–2.1

A multimodal product can also connect separate models: speech recognition, a text model, and speech synthesis, for example. Hearing a spoken answer alone cannot tell us whether one model or several produced it. We need documentation, configuration, or execution traces to distinguish these designs.

12.9 8. Assembling an agent around models

In the broad AI sense, an agent receives observations and selects actions. In LLM engineering, “agent” often describes a system that repeatedly uses a model to choose actions, calls tools, observes results, and continues toward a goal. Usage varies. A useful distinction is between a predefined workflow and a loop in which the model helps choose the next step. Anthropic’s workflow and agent distinction

Return to the astronomy poster. Here is one possible design, created for this lesson. It makes no claim about the hidden internals of a particular commercial assistant.

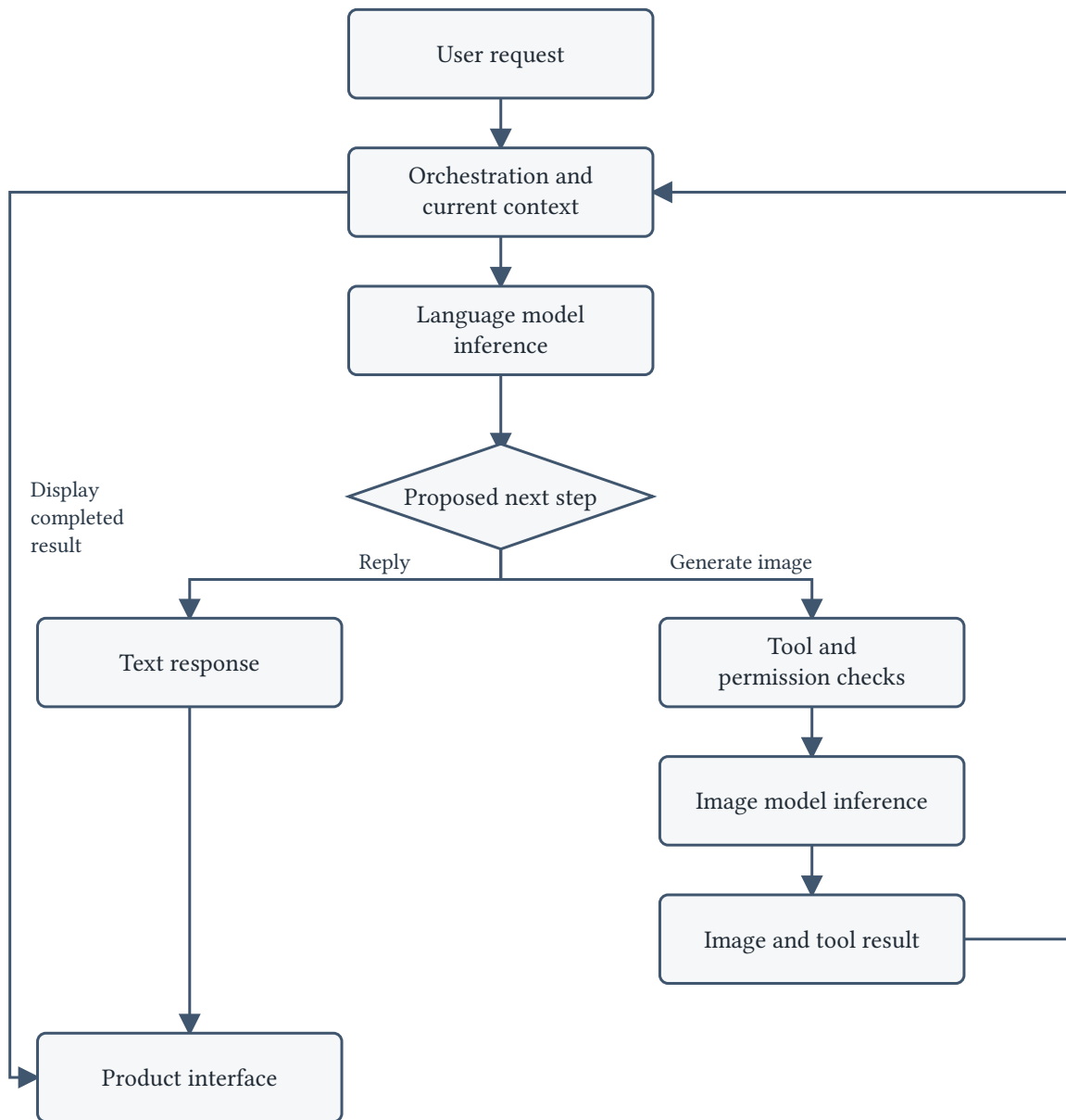


Figure 1. An illustrative language-model-plus-image-tool system. Each model call runs an architecture with trained weights in an inference runtime. The loop connecting calls belongs to orchestration.

Prose alternative: The product receives the request and supplies it to an orchestrator. The orchestrator prepares context for the language model. The model proposes either a response or an image-generation action. Software checks the action and, if permitted, calls the image model. The resulting image and tool status become an observation available for another step. The interface displays the completed output.

The LLM might turn “astronomy night” into a more detailed image prompt specifying a telescope, a dark sky, and space for event information. In this design, the LLM produces instructions; another model executes the image-generation task. The orchestration layer must actually dispatch the call. Text that looks like a tool request has no external effect by itself.

If image generation fails, the model could receive the failure, revise the request, or explain the problem. ReAct is an early published example of interleaving language-model reasoning with actions and observations. The surrounding execution environment still determines which actions are available. ReAct paper

An operational agent also needs stopping conditions and limits. “Stop when the requested artifact exists,” “stop after the budget is exhausted,” and “ask before publishing” govern different risks. A model’s ability to propose publication does not grant permission to publish. Tools, permissions, and runtime checks help determine what can actually happen.

💡 Lab recommended here

Work through Lab 01 — Map AI Systems. Classify systems along several axes and separate models from surrounding software. Record your initial predictions before checking the supplied evidence. Pay special attention to claims that a user interface alone cannot establish.

12.10 9. Use a map, not a single family tree

A useful system description answers several questions at once:

- **Task and modality:** What goes in, and what comes out?
- **Mechanism:** Rules, search, learned prediction, generation, or a combination?
- **Learning signal:** Labels, derived targets, structure in data, or rewards?
- **Model family:** A tree, linear model, neural architecture, or no trained model?
- **System boundary:** One model call, a workflow, an agent loop, or a complete product?
- **Evidence:** Which details are documented, observed, inferred, or unknown?

An image-capable coding agent could be neural, Transformer-based, multimodal, trained through several learning stages, and embedded in tool-using orchestration. All of those descriptions can be true simultaneously.

This map is enough to narrow our focus. Next, we examine the representations that let a language model turn text into computation.

12.11 Check your understanding

Answer before reading the discussion below.

1. A route finder uses a hand-written map and A*. Must it have trained weights?
2. Two applications use the same model checkpoint. One retrieves documents and one does not. Must their answers be equally informed?
3. An image generator takes 30 sampling steps. Does that establish that it performed 30 training updates?

4. A text model was trained using RL, then deployed with fixed weights and no tools. Is it necessarily a tool-using agent?
5. A product returns both text and pictures. What evidence would distinguish one multimodal generator from a language model connected to an image tool?

12.11.1 Discussion

1. No. Search can operate on explicit representations without a learned model.
2. No. Retrieved information can change the supplied context even when weights stay fixed.
3. No. Sampling can repeatedly evaluate fixed weights. Training updates must be established separately.
4. No. Its training method does not determine its deployed orchestration or available actions.
5. Model documentation, system documentation, and execution traces can help. The visible output alone does not settle the architecture.

For the earlier checks: the optimizer could schedule back-to-back events without enough preparation time; denoising targets can be constructed from the training data and known added noise. Both answers concern the design of the learning or optimization problem, not the marketing label attached to the system.

12.12 More Learning

Choose the resource that addresses your biggest remaining question; these are optional.

- CLIPS documentation — Read the opening of the User’s Guide to see explicit facts and rules in a real expert-system language. No neural-network background is needed.
- OR-Tools introduction — Compare routing, scheduling, and constraint solving. Identify the decisions, objective, and constraints in one example.
- scikit-learn Getting Started — Follow the separation between fitting a model and using it to predict. Notice where evaluation enters the workflow.
- AlphaZero paper — Read the algorithm description first. Locate the neural network, tree search, self-play data, and training objective as separate components.
- Latent diffusion paper — Read the abstract and introduction for an example of separating image representation from generative computation. The derivations are a later side trail.
- ReAct paper — Follow one action-and-observation example. Ask what the model writes and what the environment actually executes.

12.13 References

Primary papers and official documentation support the claims linked above. The numerical examples and system diagram are original instructional examples, not reported experiments. Sources checked 7 October 2026.

- CLIPS. Official documentation.
- Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths.
- Google OR-Tools. Introduction.
- DEAP. Operators and Algorithms.
- scikit-learn. Supervised learning, Getting Started, neural networks, and clustering.

- Devlin, J., et al. (2018). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.
- OpenAI Spinning Up. Key Concepts in RL.
- Silver, D., et al. (2017). Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm.
- Brown, T. B., et al. (2020). Language Models are Few-Shot Learners.
- Vaswani, A., et al. (2017). Attention Is All You Need.
- Ouyang, L., et al. (2022). Training language models to follow instructions with human feedback.
- Ho, J., Jain, A., and Abbeel, P. (2020). Denoising Diffusion Probabilistic Models.
- Rombach, R., et al. (2022). High-Resolution Image Synthesis with Latent Diffusion Models.
- Goodfellow, I. J., et al. (2014). Generative Adversarial Networks.
- Lipman, Y., et al. (2023). Flow Matching for Generative Modeling.
- Peebles, W., and Xie, S. (2023). Scalable Diffusion Models with Transformers.
- Singer, U., et al. (2022). Make-A-Video: Text-to-Video Generation without Text-Video Data.
- Radford, A., et al. (2022). Robust Speech Recognition via Large-Scale Weak Supervision.
- van den Oord, A., et al. (2016). WaveNet: A Generative Model for Raw Audio.
- Gemma Team (2025). Gemma 3 Technical Report.
- Anthropic (2024). Building effective agents.
- Yao, S., et al. (2023). ReAct: Synergizing Reasoning and Acting in Language Models.

13 1.2 — Text Becomes Numbers

Type “Paint it” into a text-generation system. Before another word appears, software must represent those characters as tokens, identify the tokens by number, retrieve numerical vectors, run a trained model, and choose a continuation. Each operation has a different job. Confusing them makes it difficult to explain even a small change in output.

This lesson follows that path through a conventional autoregressive text language model. We will specify what enters and leaves its central neural computation, while leaving attention and other Transformer internals for later lessons. The aim is to make the boundary between text processing, learned computation, and generation decisions precise.

13.1 Prerequisites and learning goals

Start with Lesson 1.1 — The AI Zoo, particularly its distinctions among architecture, trained weights, inference runtime, and orchestration. No programming or calculus is required here.

By the end, you should be able to:

- Distinguish a vocabulary, tokenizer, token ID, and embedding vector.
- Trace text through embedding lookup to contextual representations, logits, probabilities, and token selection.
- Explain why whitespace, Unicode, and conversation formatting affect model input.
- Compute a small lookup and softmax example.
- Separate next-token probability from factual confidence and repeated generation from training.

Needed now: A vector is an ordered list of numbers. A matrix is a rectangular array. Indexing selects an entry or row. Exponentiation produces positive numbers, and normalization divides positive weights by their total to obtain probabilities.

Useful refresh: Zero-based indexing, matrix dimensions, and conditional probability. We write $P(a \mid b)$ for the probability of a given b . The vertical bar means “given,” not division.

Side trail: Logarithms undo exponentiation: $e^{\ln 3} = 3$. This identity makes our worked example easy to check. Dot products, gradients, and information theory become useful later; they are not prerequisites for following the main argument.

13.2 1. Establish the interfaces

A **vocabulary** is a finite inventory of token types, with an integer ID assigned to each. A **tokenizer** supplies the rules and data used to turn text into a sequence drawn from that inventory, and usually a way to decode IDs back into text. A **trained language model** uses numerical representations of the sequence to calculate scores for possible continuations.

The tokenizer’s vocabulary may itself have been learned from a corpus. That training is distinct from training the neural model’s weights. During ordinary use, a fixed tokenizer configuration applies its established procedure; it does not ask the language model which next word would make sense. SentencePiece is one concrete toolkit that makes tokenizer training and tokenization separate operations. Kudo and Richardson, 2018

The ID-to-token mapping is part of the interface. If ID 500 refers to one token in tokenizer A and another in tokenizer B, sending A’s IDs to a model trained with B changes the input’s interpretation. Matching vocabulary sizes would not fix that mismatch.

Think of a laboratory sample with a numbered label. The label lets you retrieve the right sample; the number itself does not describe its chemistry. Similarly, adjacent token IDs need not have related meanings. ID 501 is not “slightly more” of whatever ID 500 represents.

Retrieval check: Which component could you inspect from tokenizer files alone: the tokenizer, the input embedding matrix, or the model’s next-token logits?

13.3 2. Why tokens are not words

A word-level vocabulary faces a practical problem: new names, compounds, spellings, and code identifiers keep appearing. A character-level system can represent a wider variety of strings with a smaller inventory, but often needs longer sequences. Subword tokenization offers a compromise, representing frequent pieces together while permitting less common strings to use several pieces. Sennrich and colleagues demonstrated this approach for neural machine translation using byte-pair encoding, or BPE. Sennrich, Haddow, and Birch, 2016

For intuition, imagine a tokenizer whose learned inventory includes paint, ing, and re. It could represent repainting using several pieces. This is an invented illustration, not the output of a particular

tokenizer. Actual boundaries follow the tokenizer’s procedure and vocabulary; they need not match syllables, grammatical components, or a person’s preferred spelling explanation.

In a basic BPE construction, frequently occurring adjacent units are repeatedly merged to form larger units. Encoding subsequently uses the learned merge information. Implementations also make choices about preprocessing, which boundaries merges may cross, and special-token handling. Knowing that two tokenizers use BPE is therefore insufficient to predict identical results.

13.3.1 Bytes, characters, and visible marks

GPT-2 provides a useful concrete example. Its tokenizer uses a byte-level BPE scheme built on the 256 possible byte values, with additional merged tokens and rules restricting some merges. This lets it represent text through UTF-8 bytes without needing a vocabulary entry for every possible word. The original report explains this representation in Section 2.2. Radford and colleagues, 2019

These units should stay separate in your thinking:

- A displayed symbol is what you see.
- Unicode code points specify text elements; a displayed symbol may combine several.
- UTF-8 encodes code points into bytes.
- The tokenizer groups its underlying units into tokens.

Consequently, one visible symbol need not be one token. Nor does a token need to decode into a complete, independently displayable Unicode character. Token-inspection tools may show readable stand-ins for bytes. Those stand-ins help expose the representation, but should not be mistaken for literal extra characters inserted into the user’s sentence.

Whitespace matters too. GPT-2’s documented tokenizer treats spaces as parts of tokens, so the same written word can receive a different encoding with a preceding space. Preserve spaces and line breaks when investigating an example. Hugging Face GPT-2 documentation

The practical consequence is straightforward: count tokens using the actual tokenizer and settings. Character counts and word counts are proxies whose accuracy depends on the text. Being able to encode a script or language also establishes nothing by itself about the model’s competence in that language.

Lab recommended here

Use Lab 02 — Inspect Tokenization to examine an actual GPT-2 tokenizer on your CPU. Predict boundaries for prose, code, numbers, whitespace, and Unicode, then inspect the resulting IDs and decoded text. The Lab requires tokenizer files, not model weights or a paid inference API. Return here when the distinction between the original text, displayed token pieces, and token IDs feels concrete.

13.4 3. A conversation needs a representation too

In a chat application, the visible user message is only one part of the input. A **chat template** serializes role-labeled messages into the model’s expected sequence. It may insert speaker markers, separators,

end-of-message tokens, and an assistant-response prefix. Different models can expect different formats even when they share an underlying model family.

Special tokens are entries given particular roles by the tokenizer, model training, or runtime. Examples include beginning-of-sequence, end-of-sequence, and padding markers. Their spellings and behavior are model-specific. A role label may also be ordinary text beside a special delimiter; every visible label is not necessarily one special token. Use the supplied template and check its output, including whether a later tokenization step would add duplicate markers. [Hugging Face chat-template guide](#)

Consider an invented serialization: a user-start marker, the message `Paint it`, a message-end marker, and an assistant-start marker. The model continues after that final marker. Changing the markers changes its input even though the interface still displays the same user sentence. The template is part of the system's behavior.

A **context window** limits the sequence a particular model/runtime configuration can process. Conversation text, inserted instructions, retrieved passages, and formatting consume positions; generated tokens also extend the sequence in the usual growing-prefix setup. For a hypothetical 100-position budget, 70 input tokens leave at most 30 positions for extension under that simple accounting rule. Actual services may impose additional limits or manage context differently.

An interface can show an old message that is absent from the current model input. To establish what the model received, inspect the assembled sequence or request trace. A tokenizer's ability to encode a long string does not establish that the neural model can process the resulting length.

13.5 4. Token IDs select embedding rows

The input embedding matrix holds one learned vector for each vocabulary entry. Let V be vocabulary size and d embedding dimension:

$$E \in \mathbb{R}^{V \times d}.$$

For token ID i , lookup retrieves row $E_{i,:}$. Given n token IDs, the lookup produces an $n \times d$ array. The sequence length, vocabulary size, and vector dimension answer three different questions: how many positions, how many possible token types, and how many coordinates per vector. PyTorch's embedding layer exposes precisely this lookup-table structure. [PyTorch Embedding reference](#)

13.5.1 An original six-token example

We will use a deliberately tiny vocabulary throughout the numerical example. Quotes show token spellings; the initial space inside several spellings is significant.

ID	Token spelling	Embedding row
0	<end>	(0, 0, 0)
1	Paint	(1, 0, 1)
2	" it"	(0, 2, -1)
3	" red"	(-1, 1, 0)

ID	Token spelling	Embedding row
4	" blue"	(1, 1, 0)
5	.	(0, 0, 1)

This is an instructional construction, not a real model’s vocabulary or measured output. For the strings used here, define the toy tokenizer to match these pieces exactly. Its end marker is reserved for stopping; we do not insert it into this example’s initial input.

Paint it becomes IDs [1, 2]. Its lookup is:

$$X = E[[1, 2], :] = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & -1 \end{bmatrix}.$$

Here $V = 6$, $d = 3$, and $n = 2$. We selected two rows, preserving order. We did not multiply the second row by the numeric value 2. Changing the prefix to Paint it red adds ID 3 and therefore adds $(-1, 1, 0)$ as another row.

Prediction check: If the input IDs were [2, 1, 2], what would the lookup array contain? Would the two occurrences of ID 2 retrieve different vectors?

The same ID retrieves the same token-embedding row in this fixed table. Later computation can produce different representations for its different occurrences. That distinction will matter throughout interpretability work.

13.5.2 Coordinates require a reference frame

An embedding is useful because the rest of the trained system knows how to use its coordinates. The first coordinate is not automatically “redness,” “importance,” or any other human-readable property.

Even a simple coordinate swap demonstrates the danger. In our toy table, exchanging columns one and two would turn the Paint vector into $(0, 1, 1)$. A downstream calculation that consistently exchanges its corresponding input weights could produce the same result. The coordinate labels alone cannot settle meaning.

Therefore, record the model, checkpoint, and representation location when comparing vectors. Two models with equal vector dimensions do not thereby share a coordinate system. A token embedding, a contextual hidden state, and a document-search embedding also come from different operations; the shared word “embedding” does not make them interchangeable.

13.6 5. Context changes representations

Lookup has supplied a starting vector at each position. The model then combines information through its learned computation, using a representation of order. In a causal text model, the prediction at a position depends on the prefix available there, rather than on future tokens. The Transformer paper describes how its decoder preserves that causal structure; the detailed mechanism belongs to Module 2. Vaswani and colleagues, 2017

Compare the prefixes Paint it and Describe it. If the same tokenizer assigns the same ID to the same piece " it", its initial token embedding is identical. Its contextual representation can differ because

the available context differs. “The vector for this token” is ambiguous unless we say whether we mean the stored lookup row or an activation at a particular position and layer.

For this lesson, call the final representation at the last input position h . It contains the result of the model’s contextual computation. The embedding table alone cannot tell us h : we also need the intervening architecture, weights, and input arrangement. This is an important limit on our numerical example. We will calculate lookup and output normalization exactly, while explicitly supplying the intervening computation’s output scores.

13.7 6. Logits put scores on the vocabulary

The output head maps the final representation to a **logit** for each candidate next token. Logits are real-valued scores before normalization. They may be negative, positive, or zero. They do not need to sum to anything meaningful. GPT-2’s documented outputs distinguish these vocabulary scores from its hidden states. Hugging Face GPT-2 reference

A common output head can be written:

$$z = hW_{\text{out}} + b.$$

With a row-vector convention, h has d entries, W_{out} has shape $d \times V$, and z has V entries. A bias b is optional. We will study the matrix multiplication itself in Lesson 1.3; here, track the change from hidden-coordinate space to vocabulary-score space.

Some architectures **tie** input and output weights, sharing the embedding matrix with the output projection, typically through a transpose in this notation. Others use separately learned parameters. Weight tying is a design choice, not a consequence of using tokens or embeddings. Press and Wolf, 2017

For our Paint it example, suppose the full model computation returns these logits, in ID order:

$$z = (0, 0, 0, \ln 3, \ln 2, \ln 2).$$

These stipulated scores are enough to calculate the next-token distribution. They are not derivable from the displayed embedding table alone. Nor are they claims about how a real trained model would continue this phrase.

13.8 7. Softmax turns relative scores into probabilities

Softmax exponentiates the scores and divides each result by the sum:

$$p_i = \frac{e^{z_i}}{\sum_{j=0}^{V-1} e^{z_j}}.$$

For finite logits in exact arithmetic, the probabilities are positive and sum to one. The denominator makes each probability depend on all candidate scores. PyTorch softmax reference

In our example, exponentiation gives:

$$e^z = (1, 1, 1, 3, 2, 2), \quad \sum_j e^{z_j} = 10.$$

Therefore:

$$p = (0.10, 0.10, 0.10, 0.30, 0.20, 0.20).$$

The token " red" has probability 0.30, and " blue" has probability 0.20. The ratio is 3/2. A zero logit produced probability 0.10 here, so a zero score clearly does not mean an impossible token.

Adding the same constant to every logit leaves the distribution unchanged: the common exponential factor cancels between numerator and denominator. Implementations can exploit this by subtracting the maximum score before exponentiation. You can verify the algebra without knowing anything about language.

What does 0.30 mean? It is probability assigned to one next token under this context and distribution. It is not a 30% probability that the whole answer will be factually correct. In this example, " red" is merely one possible continuation. Evaluating truth requires a task and evidence beyond a score attached to its first token.

A **temperature** $T > 0$ modifies the distribution by applying softmax to z/T . For the toy logits, $T = 0.5$ gives unnormalized weights (1, 1, 1, 9, 4, 4), totaling 20. The probability of " red" becomes $9/20 = 0.45$. Sharpening that distribution has changed selection preferences without adding evidence about the world. Temperature is one of the controls exposed by the Hugging Face generation API.

Prediction check: If all six logits were equal, what would each probability be? If only the " blue" logit increased, would the other probabilities stay fixed?

13.9 8. Selection is a separate decision

Greedy selection chooses a highest-scoring token. For our original distribution it selects " red", despite that token receiving only 30% of the probability mass. **Sampling** draws according to a distribution, allowing less-probable outcomes to be selected. Generation software may first modify scores or restrict eligible candidates, as with temperature, top-k, or top-p settings. The final sampling distribution can therefore differ from the model's unmodified softmax. Hugging Face generation strategies

For a concrete sampling construction, lay out the six probabilities as consecutive intervals along $[0, 1)$ in ID order. They end at 0.10, 0.20, 0.30, 0.60, 0.80, and 1.00. A uniform draw of 0.65 lands in ID 4's interval, selecting " blue". We now have two valid procedures giving different continuations from identical logits.

For fixed scores and a fixed tie-breaking rule, greedy selection has a fixed answer. This does not establish that every deployment will reproduce identical scores across hardware, software versions, or numerical settings. Reproducibility has additional engineering requirements. PyTorch reproducibility guidance

Greedy selection also does not search all possible completed answers. Imagine one first token has probability 0.6 but its best continuation has probability 0.2, while another has probability 0.4 and its best continuation has probability 0.9. The corresponding two-token paths have probabilities 0.12 and 0.36. Taking the best immediate choice did not find the highest-probability path in this invented example.

13.10 9. Continue the sequence and repeat

After choosing " blue", append ID 4 to the toy prefix. The sequence becomes [1, 2, 4], which decodes to Paint it blue. Run the next prediction conditioned on that extended sequence. Its distribution can differ from the preceding one because the context changed.

This is **autoregressive generation**: previous outputs become inputs for later predictions. Conceptually, the loop is predict, select, append, and repeat. Efficient implementations reuse cached computation rather than recalculating every earlier operation, but the conditioning relationship remains the same. The GPT-2 interface documents such cached state separately from the model's weights. Hugging Face GPT-2 reference

Generation stops when the runtime's conditions are met, such as a designated end token or a maximum output length. Turning IDs into displayable text is called decoding too, so distinguish **tokenizer decoding** from the **generation decoding strategy** that chooses IDs.

The current sequence and cached activations can change while the model's weights stay fixed. A long reply is therefore not evidence that the model is training as it writes. Likewise, a transcript stored by a product becomes useful to a later request only through whatever input assembly or other mechanisms that system actually provides.

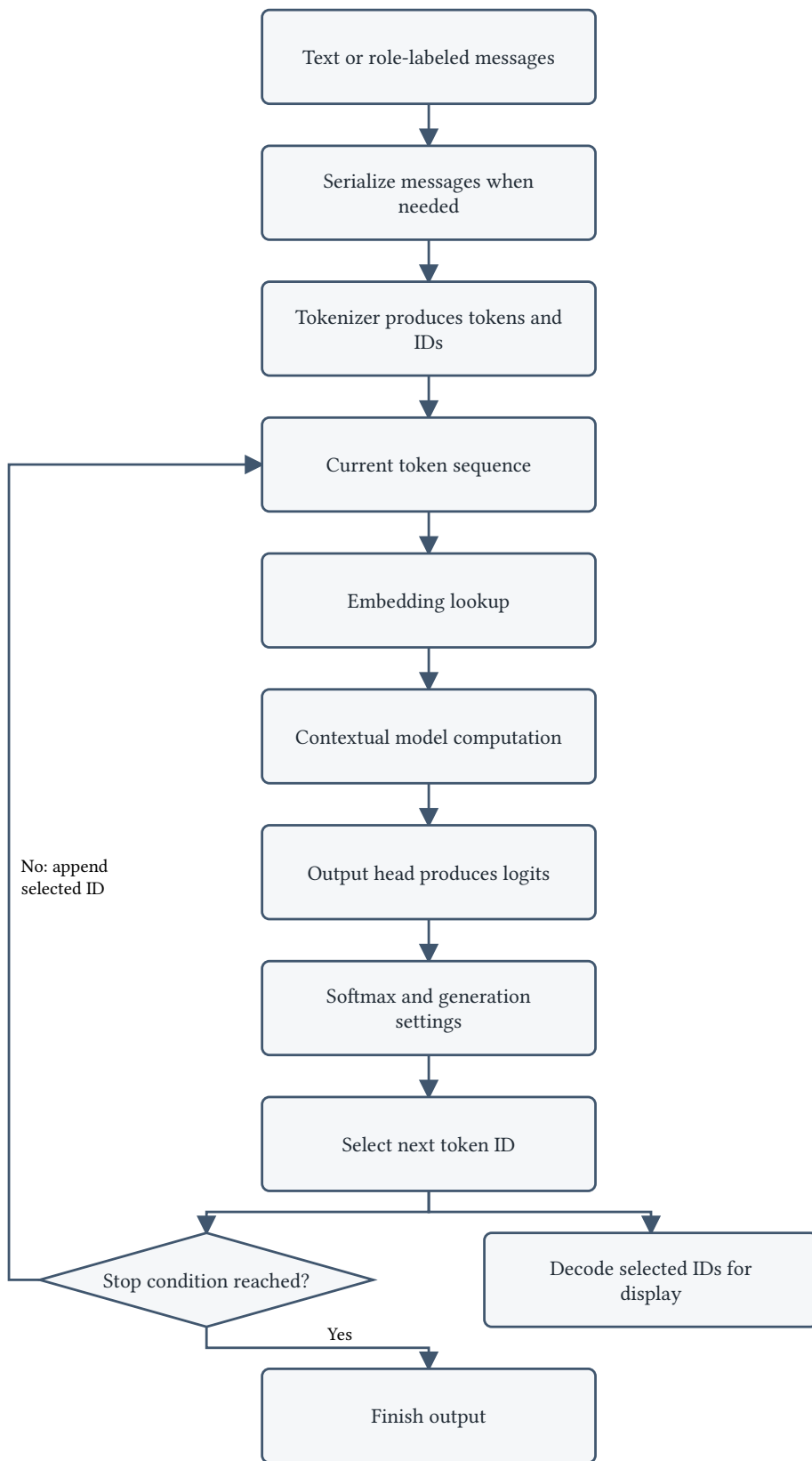


Figure 1. Original course diagram of the conceptual generation loop. The probability stage summarizes the configured selection procedure; greedy selection can choose an argmax directly without computing softmax. A runtime can stream text and reuse cached computation.

Prose alternative: Serialize messages where necessary, tokenize the resulting input, and look up vectors for its IDs. Run contextual computation and score the vocabulary. Apply the configured selection procedure. Decode selected IDs for display and, while continuation is allowed, append them to the prefix for the next prediction.

13.11 Check your understanding

Answer from memory before reading the discussion.

1. A tokenizer encodes an unfamiliar word. What has this established about model understanding?
2. Why does replacing the tokenizer require more care than matching vocabulary size?
3. Write the lookup rows for $[2, 1, 2]$ in the toy system.
4. Which missing component prevents computing the example logits from its embedding table?
5. Can greedy selection choose a token whose probability is below 50%?
6. Does choosing " blue" mean the next step uses the same distribution again?
7. Where would you look if a chat interface displayed an earlier instruction that the model appeared to ignore?

13.11.1 Discussion

1. Encoding establishes representability under that tokenizer, not comprehension or reliable performance.
2. The mapping from each ID to its learned input and output roles must agree.
3. The rows are $(0, 2, -1)$, $(1, 0, 1)$, and $(0, 2, -1)$, in that order.
4. The contextual computation and output head, including their weights, are missing.
5. Yes. Our highest probability is 30%.
6. No. The selected token extends the conditioning prefix; new scores must be calculated.
7. Inspect the actual assembled input and relevant formatting first. Visibility in the interface alone does not establish inclusion or explain the failure.

For the earlier checks: the tokenizer can be inspected from its own files; repeated IDs retrieve repeated rows; equal logits yield $1/6$ each. Raising only one logit lowers the other softmax probabilities because their shared denominator increases.

The central question remains: **Where does this behavior live?** Token boundaries come from the tokenizer; initial vectors and contextual transformations involve learned parameters; selection depends on inference settings; conversation inclusion depends on the surrounding system. We can now study the neural computations between those boundaries without conflating their responsibilities.

13.12 More Learning

- GPT-2 technical report — Read Section 2.2 for the byte-level representation decisions. Treat it as a documented model example, not a specification for every LLM.
- Neural Machine Translation of Rare Words with Subword Units — Read the motivation and BPE discussion. Ask which problems subword units address and which they leave to the model.

- PyTorch Embedding reference — Inspect input and output shapes. You can understand the lookup contract without running PyTorch.
- Chat templates — Compare two serialized conversations and identify the boundaries and response prefix.
- Generation strategies — Start with greedy selection and sampling; save beam search and implementation details for the inference lesson.
- Using the Output Embedding to Improve Language Models — Optional deeper reading on why sharing input and output parameters is an architectural choice.

13.13 References

The six-token system, calculations, path-probability counterexample, and diagram are original instructional constructions. No real-model outputs are claimed. Primary research and official documentation were checked on 7 October 2026; API defaults and model-specific formats should be verified against the version actually used.

- Kudo, T., and Richardson, J. (2018). SentencePiece.
- Sennrich, R., Haddow, B., and Birch, A. (2016). Neural Machine Translation of Rare Words with Subword Units.
- Radford, A., et al. (2019). Language Models are Unsupervised Multitask Learners.
- Vaswani, A., et al. (2017). Attention Is All You Need.
- Press, O., and Wolf, L. (2017). Using the Output Embedding to Improve Language Models.
- Hugging Face. GPT-2 reference, chat templates, generation strategies, and generation API.
- PyTorch. Embedding, softmax, and reproducibility.

14 1.3 — Neural Networks Without the Mysticism

14.1 A network you can account for

A neural network takes numbers, transforms them through a sequence of operations, and produces more numbers. Its learned parameters determine many details of those transformations. That description sounds ordinary because the individual operations usually are. The difficulty comes from how many operations interact, how training selects their parameters, and how we establish what the resulting system does.

In Lesson 1.2, we followed text into numerical representations and model outputs back toward text. Here we open the computation between those endpoints. We will use two input numbers, three hidden units, and one numerical output. Every intermediate value will fit on a page. The example is deliberately hand-constructed; nobody trained it, and it has no claimed real-world predictive ability.

By the end, you should be able to:

- read a small network equation and check its dimensions;
- distinguish weights, biases, activations, and architecture choices;
- compute a forward pass and predict a controlled intervention;
- explain why nonlinear operations matter;
- separate inference, gradient calculation, and parameter updates;
- describe what a successful calculation does, and does not, establish.

The same habits will help when a Transformer replaces our three hidden units with much larger computations.

i Math to know / refresh

Needed now: multiplying and adding signed numbers; vectors; dot products; matrix multiplication; functions; a derivative as local sensitivity. We develop each operation below.

Useful refresh: rows versus columns, transposes, coordinates, and the difference between a scalar and a vector.

Side trail: Jacobians, rank, formal approximation theorems, and optimization theory. They offer deeper explanations but are unnecessary for the forward pass. The Lab includes an optional one-parameter derivative check.

14.2 Numbers with roles and shapes

A **scalar** is one number. A **vector** is an ordered collection of numbers. A **matrix** is a rectangular arrangement of numbers. For this lesson, individual vectors are columns:

$$x = \begin{bmatrix} 2 \\ -1 \end{bmatrix}.$$

The notation $x \in \mathbb{R}^2$ says that x has two real-valued coordinates. Its column-array shape is 2×1 . An implementation may instead store a one-dimensional array with shape $(2,)$; we must specify the convention before multiplying it.

The coordinates could be two measured features, two intermediate values, or two deliberately invented inputs. The arithmetic does not tell us their real-world meaning. If they represented temperature and distance, their units and scaling would also matter. For this example they are simply dimensionless numbers.

A **dot product** multiplies corresponding coordinates and adds the results. With weights $w = [1, -1]$, the dot product with our input is

$$w \cdot x = (1)(2) + (-1)(-1) = 3.$$

Notice the negative input and negative weight: their contribution is positive. A weight's sign is not a label saying that a feature is good or bad. It specifies a numerical relationship in a particular computation.

A matrix lets us compute several weighted sums at once. Put one set of weights in each row:

$$W = \begin{bmatrix} 1 & -1 \\ 0.5 & 1 \\ -1 & 2 \end{bmatrix}.$$

There are three rows and two columns, so W has shape 3×2 . Multiplying Wx takes a dot product between each row and the same input. Three rows produce three outputs. The inner dimensions must agree:

$$(3 \times 2)(2 \times 1) = (3 \times 1).$$

This is the first useful debugging habit: write shapes beside the equation before calculating values. A 3×2 matrix cannot multiply a 4×1 column in this way. An extra or missing dimension often signals that we have mixed up examples, features, or sequence positions.

14.3 Weights and biases define an affine layer

For one unit, a weighted sum plus a **bias** is

$$z_j = \sum_{i=1}^2 W_{ji} x_i + b_j.$$

The index i selects an input coordinate; j selects an output unit. The bias contributes even when every input is zero. Collecting all three units gives

$$z = Wx + b, \quad b = \begin{bmatrix} 0 \\ 1 \\ 0.5 \end{bmatrix}.$$

Here z is the **preactivation**: the value before the activation function we will apply next. The vectors z and b both have three coordinates. Addition happens coordinate by coordinate.

Strictly, Wx is a **linear** transformation. Adding a nonzero bias makes $Wx + b$ **affine**. A linear map sends zero to zero; an affine map may shift it elsewhere. Machine-learning libraries commonly use the name “linear layer” for this affine operation. PyTorch’s `Linear`, for example, includes an optional learned bias and documents its weight shape as output features by input features. Its row-oriented input convention is equivalent to ours after transposing the vectors. PyTorch `Linear` documentation

Each row forms a weighted combination of the input coordinates. The bias shifts the resulting value. For our first unit, $z_1 = x_1 - x_2$. Inputs with $x_1 = x_2$ give zero. Changing its bias to 1 would move the zero boundary to $x_1 - x_2 = -1$.

For a dense affine layer with d inputs and m outputs, there are md weights and, if enabled, m biases. Our first layer therefore contains $3 \cdot 2 + 3 = 9$ parameters. That count describes stored adjustable numbers, not nine facts or nine concepts.

14.4 Activation functions change what composition can express

We now apply the **rectified linear unit**, or **ReLU**, separately to each coordinate:

$$h_j = \text{ReLU}(z_j) = \max(0, z_j).$$

Positive values pass through unchanged; negative values become zero. ReLU preserves the vector’s shape. We call h the hidden activation vector. “Activation” can refer either to the function, such as ReLU, or to a computed value, such as $h_2 = 1$. Context should make the meaning clear. PyTorch ReLU documentation

Why add this operation? Consider two affine layers with nothing nonlinear between them:

$$u = Ax + a, \quad y = Bu + b.$$

Substitution gives

$$y = B(Ax + a) + b = (BA)x + (Ba + b).$$

The whole composition is another affine transformation. Its factorization can matter for parameterization and optimization, but stacking these layers has not introduced a nonlinear input-output function.

ReLU breaks that general simplification. For an immediate counterexample, $\text{ReLU}(1) + \text{ReLU}(-1) = 1$, while $\text{ReLU}(1 + (-1)) = 0$. It does not preserve addition, one requirement of linearity.

Within a region where the same ReLU units remain positive, our network will still have an affine formula. Crossing a zero boundary can change that formula. The resulting function is **piecewise affine**, often called piecewise linear in neural-network discussions. Nonlinearity gives the architecture additional possibilities; particular weights can still produce a constant or affine function.

Other activation functions make different trade-offs. A nonlinear function need not clip negative values, and ReLU is not a requirement for every network. We use it because its behavior is easy to inspect by hand. Later MLP lessons will revisit the choices used inside language models.

 Predict before continuing

If a hidden unit's preactivation is negative, will increasing its output weight change this example's prediction? Write down your answer and the condition on which it depends. We will test exactly this question.

14.5 The complete microscopic network

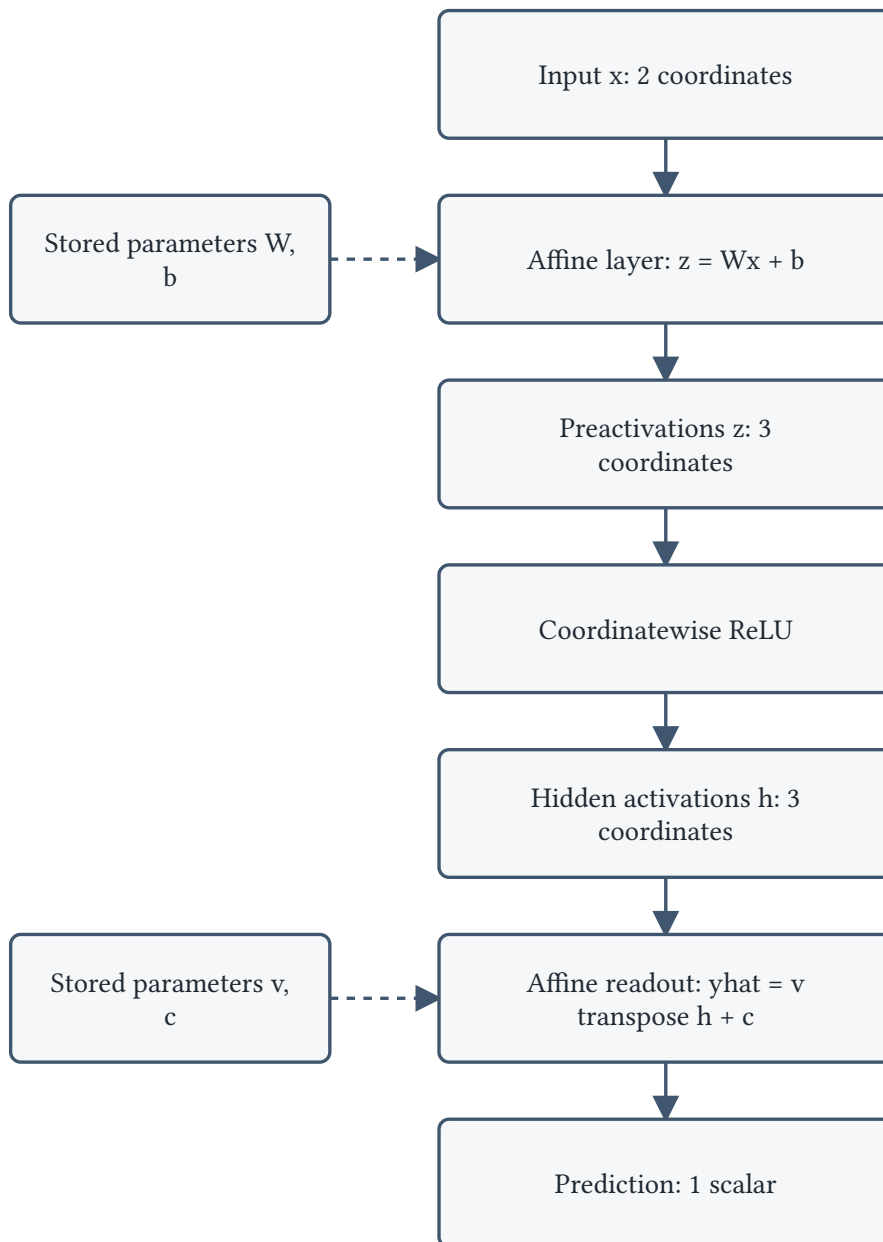
Our output is one unconstrained real number, suitable as the form of a regression prediction. We use a final affine readout without another ReLU:

$$h = \text{ReLU}(Wx + b), \quad \hat{y} = v^T h + c,$$

where

$$v = \begin{bmatrix} 2 \\ -1 \\ 1 \end{bmatrix}, \quad c = 0.5.$$

The hat in $\hat{y}$ marks a prediction. It does not mean that the prediction is accurate. The final row v^T has shape 1×3 , so its product with h is a scalar. A classification model could instead have several output scores, or logits, followed by the interpretation discussed in Lesson 1.2. We keep one output to concentrate on the network itself.



Original course diagram. Read downward along the solid arrows: two inputs become three preactivations, then three ReLU activations, then one prediction. The dotted arrows show which stored parameters each affine operation reads. Applying this graph does not itself change those parameters.

A network of this form is a small **feedforward network** or **multilayer perceptron**: intermediate computations depend on earlier ones in the forward graph. Its hidden layer is “hidden” because the task supplies inputs and desired outputs rather than a target value for every intermediate unit. The values are still observable when we instrument the computation. Goodfellow, Bengio, and Courville, Chapter 6

People count layers differently. This example has one hidden layer and one output layer, both parameterized. ReLU may be a separate software module. “Two-layer network” without a counting convention is less informative than the equations.

14.5.1 Compute every intermediate value

Start with $x = [2, -1]^T$. Before reading the answers, calculate all three entries of z .

$$\begin{aligned} z_1 &= (1)(2) + (-1)(-1) + 0 = 3, \\ z_2 &= (0.5)(2) + (1)(-1) + 1 = 1, \\ z_3 &= (-1)(2) + (2)(-1) + 0.5 = -3.5. \end{aligned}$$

Thus

$$z = \begin{bmatrix} 3 \\ 1 \\ -3.5 \end{bmatrix}, \quad h = \begin{bmatrix} 3 \\ 1 \\ 0 \end{bmatrix}.$$

The readout is

$$\hat{y} = (2)(3) + (-1)(1) + (1)(0) + 0.5 = 5.5.$$

Three details matter. First, the negative preactivation was removed before the readout. Second, the positive activation $h_2 = 1$ contributes negatively because its readout weight is -1 . Third, the output bias is added once. It is not added separately to every contribution.

The full parameter count is $6 + 3 + 3 + 1 = 13$: six entries in W , three in b , three in v , and one in c . The intermediate vectors are additional values computed for this input, not additional learned parameters.

14.5.2 Intervene and explain the difference

Hold the input and parameters fixed, but replace h_2 with zero immediately before the readout. This **ablation** produces

$$\hat{y}_{\text{ablated}} = (2)(3) + (-1)(0) + (1)(0) + 0.5 = 6.5.$$

Removing a unit’s contribution increased the output. Its original contribution was -1 . “More activation means more output” is therefore an unsafe general rule even in this tiny network.

Now restore the original activations and change only v_3 from 1 to 101. The output remains 5.5, since $h_3 = 0$. This answers the prediction question for this input. On another input, the same parameter can matter substantially. With $x = [0, 1]^T$, calculate $z = [-1, 2, 2.5]^T$ and $h = [0, 2, 2.5]^T$. The original output is 1; changing v_3 to 101 makes it 251.

A zero intervention effect on one example therefore does not establish global irrelevance. Conversely, a nonzero effect establishes a dependency in the tested computation; by itself it does not assign the unit a human concept.

! Lab recommended here

Do Tiny Network, Visible Computation, Parts 1–3. Compute a fresh forward pass by hand before implementing it, then predict an intervention before running it. The core Lab needs only paper and, optionally, ordinary Python on a CPU. No account, API, model download, or paid compute is needed.

14.6 Parameters persist while activations change

Write the network as $f_{\theta}(x)$, where θ collects W, b, v, c . This notation separates the input from the stored numbers defining the particular function.

In ordinary **inference**, we apply a chosen parameter set to an input. Changing x generally changes z, h , and $\hat{y}$. It need not change θ . In our deterministic example, the same input and same parameters give the same output. Larger systems may add sampling or other sources of variation.

In **training**, a procedure updates selected parameters to improve an objective on training examples. Parameters can be **trainable** in that procedure or **frozen**, meaning that procedure leaves them unchanged. A frozen weight still participates in the forward calculation and affects its output. If we freeze W and b , our model still stores 13 parameters, but only the four numbers in v and c remain candidates for updates.

Architecture choices sit at another level: this network’s hidden width is three, its hidden function is ReLU, and its output is scalar. Ordinary gradient updates to the 13 numbers do not decide to add another hidden unit. Likewise, a learning rate controls an update procedure rather than appearing as a learned coefficient in this forward equation.

A new conversation message can change a language model’s input and activations without an update to its stored weights. Whether a particular service separately uses conversations for later training is a product and data-policy question. The forward computation alone does not answer it.

i Shape discipline when moving to code

If B examples are stored as rows in X with shape $B \times 2$, the equivalent first-layer calculation is $Z = XW^T + \mathbf{1}_B b^T$, where $\mathbf{1}_B$ is a column of B ones. This adds the same bias row to every example. In code, a one-dimensional length-three bias array can be broadcast across those rows. Then Z and H have shape $B \times 3$, and $Hv + c$ produces one output per example. The transpose accommodates storage orientation; it does not change the network. PyTorch’s Linear specification is a useful reference when checking this convention.

Dimension agreement is necessary but insufficient. Swapping two feature columns preserves the shape while changing their meaning. Track both the size and the role of each axis.

14.7 Training adds an objective and an update

Suppose one training example asks for target $t = 4$ at our original input. Choose a loss

$$L = \frac{1}{2}(\hat{y} - t)^2.$$

At $\hat{y} = 5.5$, the error is 1.5 and the loss is 1.125. The factor $\frac{1}{2}$ makes the derivative convenient. This is a chosen instructional objective, not a claim that squared error is appropriate for every task.

A basic training step performs a forward pass, computes loss, calculates gradients, and uses an optimizer to update selected parameters. **Backpropagation calculates the gradients; an optimizer uses them to update parameters.** Merely calculating a gradient does not change a weight. PyTorch exposes this distinction through separate backward and optimizer operations. PyTorch optimization tutorial

A derivative answers a local sensitivity question. If we change one parameter slightly while holding the others fixed, how quickly does the loss change? For v_1 , the path to the loss is short:

$$\frac{\partial L}{\partial \hat{y}} = \hat{y} - t = 1.5, \quad \frac{\partial \hat{y}}{\partial v_1} = h_1 = 3.$$

The **chain rule** multiplies sensitivities along this path:

$$\frac{\partial L}{\partial v_1} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial v_1} = 1.5 \cdot 3 = 4.5.$$

The positive sign says that a sufficiently small increase in v_1 raises this loss locally. A gradient-descent update with learning rate $\eta = 0.1$, changing only v_1 , gives

$$v_1^{\text{new}} = 2 - 0.1(4.5) = 1.55.$$

The new prediction is $1.55(3) - 1 + 0.5 = 4.15$, giving loss 0.01125. These are exact hand-derived values for this chosen example. We have fitted one parameter toward one target; we have not demonstrated a useful trained model.

For an earlier weight, the sensitivity passes through more operations. Since $z_1 = 3$ is positive, ReLU has slope one there:

$$\frac{\partial L}{\partial W_{11}} = (\hat{y} - t) v_1 (1) x_1 = 1.5 \cdot 2 \cdot 1 \cdot 2 = 6.$$

This derivative uses the original parameters. ReLU's slope is zero on a strictly negative input, and its ordinary derivative is undefined exactly at zero; implementations choose a convention there. Automatic differentiation applies such local rules through a recorded computation graph. It saves us from hand-expanding a large network. PyTorch automatic differentiation tutorial

A gradient is local information. A large step can overshoot, an update that helps one example can hurt another, and a training objective may only imperfectly represent the behavior we ultimately want. Module 3 will develop training across datasets and repeated updates.

14.8 What the neuron metaphor leaves out

In this MLP, a “neuron” usually means one hidden coordinate together with its incoming weighted sum and activation function. The term does not establish that it models the detailed biology of a nerve cell. We can specify every operation here without claims about thoughts, intentions, or experience.

Nor should we assume that each coordinate has a stable, human-readable meaning. A single activation may participate in several patterns, and an interpretable property may involve several coordinates. Research on **superposition** explores how networks can represent more features than their representation has dimensions. *Toy Models of Superposition* demonstrates mechanisms in deliberately simplified models; its results motivate questions about larger networks rather than automatically settling them. We will return to this in Lesson 2.3. Elhage and colleagues, 2022

Even our example discourages quick labels. Unit 3 is inactive for one input and important for another. Unit 2’s positive activation lowers the scalar readout. Naming either unit from one observation would lose crucial conditions. A useful interpretation must say which inputs it covers, which computation it describes, and what evidence could contradict it.

14.9 Evaluation asks a different question

Getting 5.5 proves that a particular forward calculation matches the specified arithmetic. Passing the Lab’s checks can establish that an implementation matches that specification. Neither result establishes predictive accuracy on a real task.

For a trained model, examine performance on examples not used to fit its parameters. Use validation data for development decisions and protect a separate test set for final evaluation. Data leakage can happen through preprocessing and model selection as well as direct training. Scikit-learn’s guidance on data leakage

For this network, useful evaluation would first require a defined task: what do the two inputs mean, what target should the output approximate, and which mistakes matter? Only then could we choose appropriate data and measurements. A falling training loss answers one optimization question. It does not establish robustness, calibration, fairness, or suitability for a particular use.

There is also a boundary between tracing and explaining. We can trace every multiplication and still need an explanation of why this function fits a task. At larger scales, recording activations is a starting point for investigation. It is not automatically a readable account of what the model represents.

14.10 The bridge to Transformers

The original Transformer contains position-wise feedforward networks with two affine transformations and a ReLU between them. That component is a close relative of the computation we have just inspected. The architecture also includes attention and residual connections, which give the larger system additional structure. Vaswani and colleagues, Section 3.3

Carry forward three questions: What shape enters this operation? Which stored parameters does it use? What new activation does it produce? They remain useful when the numbers no longer fit on a page. The next module will add the residual stream and show how multiple components read and update token representations.

14.11 Check your understanding

Answer without looking back, then compare with the solutions.

1. Why are $Wx + b$ and $\text{ReLU}(Wx + b)$ generally different kinds of functions?
2. For $x = [1, 0]^T$, compute z , h , and $\hat{y}$ using the original parameters.
3. Remove ReLU from the original network. Find its simplified input-output equation and its output at $[2, -1]^T$.
4. A unit's ablation has no effect on one input. What can you conclude? What extra claim is unjustified?
5. Which changes an activation: changing the input, updating a weight, or both? Which necessarily updates a stored parameter?
6. Why does a correct implementation still need evaluation on a task?

Solutions

1. The first is affine. ReLU can change which coordinates pass through as inputs change, allowing a piecewise-affine function. Particular parameters can still make the full function affine.
2. $z = [1, 1.5, -0.5]^T$; $h = [1, 1.5, 0]^T$; $\hat{y} = 2 - 1.5 + 0.5 = 1$.
3. $v^T W = [0.5, -1]$ and $v^T b + c = 0$. Thus $\hat{y} = 0.5x_1 - x_2$, giving 2. The original ReLU network gives 5.5 there.
4. Only that this intervention had no measured effect under the tested conditions. The unit need not be irrelevant for other inputs or other readouts.
5. Both can change activations, though neither guarantees a change at every coordinate. Updating a weight changes a stored parameter; changing the input alone does not.
6. Correctness against a specification establishes that we computed the intended function. Task evaluation tests whether that function gives useful outputs under relevant conditions.

14.12 More Learning

1. Goodfellow, Bengio, and Courville: Deep Feedforward Networks. Read the opening definitions and Section 6.5 for a more formal treatment of composition and backpropagation. The chapter is broader than this lesson; treat the later details as optional.
2. PyTorch Linear and ReLU. Short official references for the two operations. Compare their input, output, and parameter shapes with our equations before using a library implementation.
3. PyTorch: Automatic Differentiation. Connect the hand-calculated chain rule to a computational graph and stored gradients. No PyTorch installation is required to read it.
4. PyTorch: Optimizing Model Parameters. Follow the separation of loss calculation, backward calculation, and optimizer updates. Its larger training example is optional; the nearby course Lab stays microscopic.
5. Elhage and colleagues: Toy Models of Superposition. An advanced preview of why a neuron and an interpretable feature are different objects. Focus initially on the abstract and the limits of conclusions from toy models.

6. Vaswani and colleagues: Attention Is All You Need. Read Section 3.3 and recognize the affine-activation-affine pattern. Save the full architecture for Module 2.

15 2.1 — The Residual Stream

15.1 Following a representation through depth

A Transformer repeatedly updates numerical representations of a sequence. The **residual stream** is a useful name for those representations as they pass through the network's residual connections. Understanding this stream gives us a map: where a component receives its input, where it contributes an update, and what continues to the next component.

Lesson 1.2 introduced token embeddings and contextual representations. Lesson 1.3 opened a small neural network and distinguished parameters from activations. We now connect those ideas to a stack of Transformer blocks. We will leave the detailed attention calculation for Lesson 2.2 and the internal MLP computation for Lesson 2.3. Here, we need their interfaces and their places in the computation.

By the end, you should be able to:

- identify a token-position vector inside a sequence-shaped array;
- distinguish a branch's update from the complete block output;
- trace residual additions and explain what their direct paths provide;
- recognize pre-norm, post-norm, and a parallel-residual variation;
- interpret vector changes without equating their size with knowledge or importance;
- specify exactly which activation an inspection tool should record.

 Lab recommended here

Begin Reading the Original Transformer Paper, Part 1, before continuing. Read the abstract, introduction, architecture figure, and conclusion of *Attention Is All You Need* for orientation. Keep unfamiliar terms as questions. Return to the paper after Lesson 2.3; understanding every equation now is unnecessary.

 Math to know / refresh

Needed now: vector addition, matrix shapes, Euclidean norms, and affine transformations. The examples refresh these operations.

Useful refresh: a vector's coordinates depend on a chosen basis; a dot product measures alignment with another vector.

Side trail: Jacobians and detailed optimization theory. One optional equation connects the residual path to the derivatives introduced in Lesson 1.3.

15.2 One vector per token position

Let a sequence contain n tokens, and let the model's residual width be d . At a chosen boundary in the network, collect its representations in

$$X \in \mathbb{R}^{n \times d}.$$

Each row corresponds to a **position**, not to a unique vocabulary entry. Repeated token IDs still occupy separate rows. The number of coordinates in each row is the width d , sometimes called `hidden_size` or d_{model} . It is independent of vocabulary size and sequence length.

We retain Lesson 1.3's column convention when discussing one individual vector. For position i , write $x_i \in \mathbb{R}^d$ as a column, so row i of X is x_i^T . Thus the same information can be written

$$X = \begin{bmatrix} x_1^T \\ x_2^T \\ \vdots \\ x_n^T \end{bmatrix}.$$

The transpose reconciles our two conventions. It does not perform a learned transformation. Software often adds a batch dimension, giving shape (b, n, d) for b sequences; this chapter suppresses that dimension.

For an invented three-token sequence with width four, X contains twelve activation values. Adding another token adds another row. Passing through another width-preserving block produces another 3×4 array. Neither operation adds vocabulary entries or automatically changes stored weights.

A row is not a separate little model for its token. After components exchange information between positions, that row can depend on a wider context. Conversely, the whole array is not one embedding vector for the entire conversation. Keeping both the position axis and the coordinate axis visible prevents these two pictures from being confused.

There are also two different orders to track. **Position** runs along the sequence; **depth** runs through blocks. A representation can change with depth while continuing to belong to the same position. Its eventual role in predicting a next token does not make it the embedding of that future token.

15.3 Where the stream starts

Token IDs select rows from the learned embedding matrix. For an architecture with additive positional embeddings, we can illustrate the initial representation as

$$x_i^{(0)} = e_{t_i} + p_i,$$

where t_i is the token ID at position i , e_{t_i} is its token embedding, and p_i carries positional information. Scaling and dropout may also occur, depending on the architecture and whether it is training or evaluating.

The original 2017 Transformer adds positional encodings to scaled token embeddings. Its full model has separate encoder and decoder stacks. This chapter instead uses a simplified decoder-only example,

so its main diagram should not be read as a reproduction of the original architecture. Vaswani and colleagues, Sections 3.1, 3.4, and 3.5

Adding a position vector is not universal. In the Llama implementation referenced here, the stream starts from token embeddings, while rotary positional information acts on attention queries and keys. We will study that mechanism in Lesson 2.4. For now, remember that a model can use position without adding p_i to its initial stream. Transformers Llama implementation, v4.57.1

Two occurrences of the same ID retrieve the same token-embedding vector. Their later representations can differ because of position and available context. “The vector for a word” is therefore incomplete: ask which tokenizer, occurrence, checkpoint, depth, and measurement boundary.

15.4 A block computes updates and adds them

Start with the residual pattern

$$y = x + F(x).$$

The **residual branch** computes $F(x)$, and the **skip connection** carries x directly to the addition. The result y contains both numerical contributions. Addition requires matching shapes. A branch may use wider internal representations, but its output must return to the residual width before this addition.

The shape requirement creates a common interface. A branch may project into an internal space, do substantial computation there, and project back. It cannot append extra coordinates to the stream merely because it has found another useful pattern; that would change the shape required by the addition. Different contributions therefore coexist in the same finite-width representation. Their interaction can include reinforcement and cancellation.

A zero branch and an identity branch are also different. If $F(x) = 0$, the bare residual operation returns x . If $F(x) = x$, it returns $2x$. When reading a diagram, label the branch output and the result of the addition separately rather than calling both of them “the residual.”

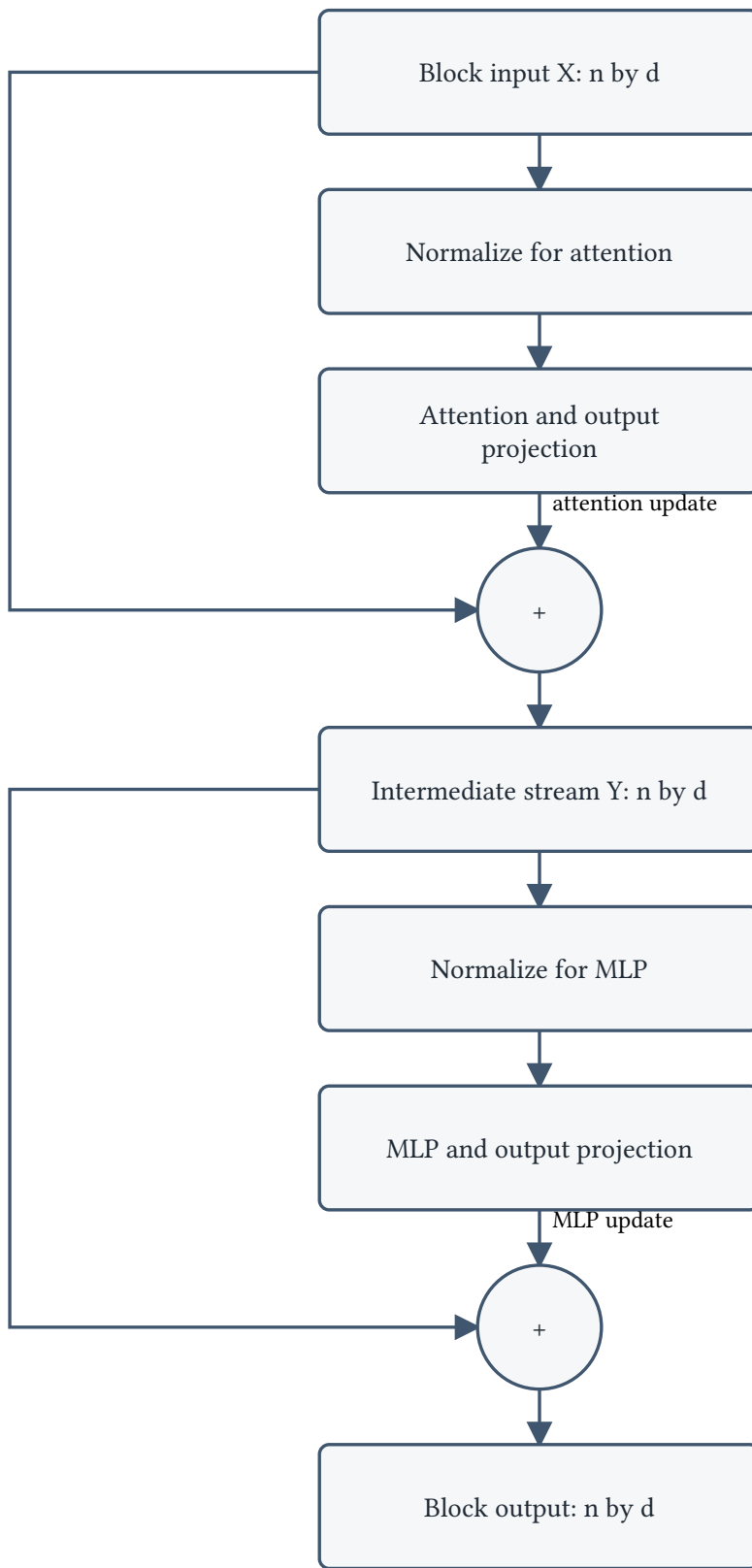
Residual learning predates the Transformer. He and colleagues used it to make very deep image-recognition networks easier to optimize. That history motivates the pattern; it does not make image models and language models identical. Deep Residual Learning for Image Recognition

For our main illustration, choose a **sequential pre-norm decoder block**, at evaluation time with dropout disabled. Let $X^{(\ell)}$ enter block ℓ . Let $N_{\ell,A}$ and $N_{\ell,M}$ be its normalization operations. Define

$$\begin{aligned}\Delta_A^{(\ell)} &= A_{\ell,A}(X^{(\ell)}), \\ Y^{(\ell)} &= X^{(\ell)} + \Delta_A^{(\ell)}, \\ \Delta_M^{(\ell)} &= M_{\ell,M}(Y^{(\ell)}), \\ X^{(\ell+1)} &= Y^{(\ell)} + \Delta_M^{(\ell)}.\end{aligned}$$

A_ℓ denotes attention, including its output projection; M_ℓ denotes the MLP, including its projection back to width d . Every displayed activation has shape $n \times d$. Subscripts identify the block’s components, whose parameters are ordinarily different from those in other blocks.

The MLP receives a normalized version of the already attention-updated stream. It does not receive only the attention update. The block output is $X^{(\ell+1)}$; calling $\Delta_A^{(\ell)}$ or $\Delta_M^{(\ell)}$ “the block output” would discard the skip-path contribution and confuse the boundary being measured. This sequential arrangement appears in the referenced Llama decoder implementation. Llama decoder block source



Original course diagram. One sequential pre-norm block; arrows describe dependencies rather than physical timing. Both direct arrows bypass a normalization-and-computation branch.

Prose alternative: The input splits into a direct path and an attention path. The attention path normalizes its input and calculates an update, which is added to the direct path. Their sum splits again. One copy goes directly to the second addition; the other is normalized and passed through the MLP. The second sum becomes the next block's input.

An arrow in this diagram establishes a possible route for influence, not proof that a particular fact traveled along it. To establish an effect on an input, we must inspect values or perform a controlled comparison. A component can receive a vector while ignoring some directions in it, and a later component can counteract an earlier update.

Attention can use representations from other allowed positions to construct each position's update. In a causal decoder, that means the current and permitted earlier positions, never later positions. A standard position-wise MLP transforms each position separately using shared parameters. Residual addition itself mixes neither positions nor coordinates: it adds matching entries. The branches supply the more complicated interactions.

15.5 An exact trace with three coordinates

We can isolate residual arithmetic without simulating attention. The following **hand-constructed residual toy** has no normalization, tokenizer, training, or claim to language understanding. Its two affine update functions are deliberately transparent:

$$F(x) = \begin{bmatrix} -x_1 \\ 2 \\ 0 \end{bmatrix}, \quad G(y) = \begin{bmatrix} y_2 \\ -y_2 \\ 2y_1 \end{bmatrix}.$$

Here the subscripts select coordinates of one vector, rather than token positions. This local use is explicit to avoid confusing the two kinds of index. Let

$$x = \begin{bmatrix} 2 \\ -1 \\ 1 \end{bmatrix}, \quad y = x + F(x), \quad z = y + G(y).$$

First calculate the branch output:

$$F(x) = \begin{bmatrix} -2 \\ 2 \\ 0 \end{bmatrix}.$$

Then add matching coordinates:

$$y = \begin{bmatrix} 2 + (-2) \\ -1 + 2 \\ 1 + 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}.$$

The next branch sees this new y :

$$G(y) = \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix}, \quad z = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}.$$

We can check the same result by summing the input and the **updates from this run**:

$$x + F(x) + G(y) = \begin{bmatrix} 2 \\ -1 \\ 1 \end{bmatrix} + \begin{bmatrix} -2 \\ 2 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}.$$

Notice cancellation. The first coordinate becomes zero after the first addition, even though the input arrived through an identity path. A residual connection does not promise to preserve each original value, or preserve every meaning we might associate with it.

The **Euclidean norm** measures vector length:

$$\|x\|_2 = \sqrt{\sum_j x_j^2}.$$

Our input has length $\sqrt{6}$, and both y and z have length $\sqrt{2}$. Equal lengths do not make y and z equal vectors. Nor must the stream's norm grow when an update is added. Negative and differently directed contributions matter.

Now predict an intervention: suppress $F(x)$, then recompute everything downstream. The second branch receives the original x , giving $G(x) = [-1, 1, 4]^T$. The new output is $[1, 0, 5]^T$. Merely subtracting the original $F(x)$ from the original z would instead give $[3, -2, 1]^T$, which is wrong for this intervention. Later updates depend on what earlier components supplied.

💡 Lab recommended here

In Trace a Residual Stream, make a fresh hand trace, compare update sizes with output effects, and explain an intervention. An optional second route records an actual small model's states. Its measurements must come from an executed implementation; the toy numbers above are analytical examples.

15.6 What normalization changes

Normalization transforms the values a component receives or returns. It preserves the array's shape while adjusting its numerical scale, and sometimes its centering. For the token-wise normalization considered here, the statistics are computed across the d coordinates of each position separately. They are not a mean across all tokens or all examples in a batch.

In a common **LayerNorm** form, first calculate a vector's coordinate mean μ and variance σ^2 . Then

$$\text{LN}(x)_j = \gamma_j \frac{x_j - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta_j.$$

$\epsilon > 0$ stabilizes division; γ_j and β_j are learned scale and offset parameters when enabled. The standardized intermediate values are not themselves the final result when those learned parameters change them. PyTorch LayerNorm reference

RMSNorm uses the root mean square of the coordinates without first subtracting their mean. A common form is

$$\text{RMSNorm}(x)_j = \gamma_j \frac{x_j}{\sqrt{\frac{1}{d} \sum_k x_k^2 + \epsilon}}.$$

The key distinction for this lesson is centering: RMSNorm does not perform LayerNorm’s mean subtraction. Details of learned parameters and numerical implementation still belong to the specific model. Zhang and Sennrich, Root Mean Square Layer Normalization; PyTorch RMSNorm reference

A small idealized comparison helps. For $x = [1, 3]^T$, set scale to one, offset to zero, and temporarily ignore ϵ . LayerNorm gives $[-1, 1]^T$ because the mean is two and the variance is one. RMSNorm gives $[1/\sqrt{5}, 3/\sqrt{5}]^T$. Neither operation says which coordinate matters to the task. Normalization is a defined numerical transformation, not a filter that recognizes irrelevant meaning.

Pre-norm places normalization on a branch’s input:

$$y = x + F(N(x)).$$

Post-norm normalizes after adding the branch output:

$$y = N(x + F(x)).$$

These equations are different computations. In our pre-norm illustration, the direct stream bypasses the branch’s normalization. In the original Transformer’s post-norm sublayers, the summed result is normalized. Original Transformer, Section 3.1

Some modern designs combine additional normalization locations. Gemma 3’s report describes both pre- and post-normalization using RMSNorm. Do not force every model into one simplified diagram; inspect exactly what is normalized, and whether normalization occurs before or after the residual addition. Gemma 3 Technical Report, Section 2

15.7 Direct paths help optimization without guaranteeing preservation

Why carry the input directly? A residual branch can learn an adjustment to an existing representation rather than having to reproduce the entire representation through that branch. Setting the branch’s contribution to zero makes the bare residual operation an identity map. That offers a useful way to express small changes.

There is a corresponding derivative intuition. For the simplified equation $y = x + F(x)$,

$$\frac{\partial y}{\partial x} = I + J_F(x),$$

where I is the identity matrix and J_F collects the local sensitivities of F . The direct term provides a gradient path in addition to the transformed path. Identity-path analysis is a central theme of subsequent residual-network research. He and colleagues, Identity Mappings in Deep Residual Networks

This does not guarantee nonvanishing gradients. If $F(x) = -x$, then $y = 0$ and $I + J_F = 0$. The same counterexample defeats a claim that residual addition must preserve recoverable input information. An available route in the computation graph is different from a mathematical guarantee about the complete function.

Placement of normalization also changes gradient behavior. Research comparing pre-norm and post-norm Transformers analyzes how their gradient scales differ at initialization and why training procedures can behave differently. The practical outcome depends on the complete architecture, initialization, and optimization setup. “Residuals solve training” is too strong. Xiong and colleagues, 2020

For our sequential pre-norm stack, algebra in exact arithmetic lets us write its final raw stream as its initial stream plus all residual updates along the actual forward pass:

$$X^{(L)} = X^{(0)} + \sum_{\ell=0}^{L-1} \left(\Delta_A^{(\ell)} + \Delta_M^{(\ell)} \right).$$

This is accounting, not a claim that the updates are independent. Each was calculated using preceding states. Nor does the same expression automatically describe the normalized boundaries of a post-norm stack. Keep the equation attached to the architecture for which it was derived.

15.8 Accumulation is not a readable list of facts

The stream can carry information useful to many later operations, but its coordinates are not text fields with stable human labels. An update might reinforce a direction, counteract an earlier contribution, or rearrange what later transformations can use. “More layers accumulate more knowledge” hides these possibilities and confuses activations with learned parameters.

Consider a downstream scalar readout $s = w^T x$. An update δ changes that immediate readout by $w^T \delta$. Its norm alone cannot determine the change. A large update perpendicular to w has zero effect on this particular readout; a smaller aligned update can matter. In a full model, later nonlinear computations complicate the relationship further.

Coordinate labels also depend on the representation’s basis. Swapping two coordinates and consistently swapping the corresponding parameters can preserve a compatible computation. Orthogonal changes of coordinates preserve Euclidean lengths and angles; arbitrary invertible changes generally do not. Do not conclude that every basis change preserves a complete Transformer without adjusting or respecting its normalization and other architectural constraints.

Within one checkpoint’s residual space, comparing vectors at stated boundaries can be useful. Across separately trained models, coordinate 17 is not automatically the same feature. Even within one model, cosine similarity or distance is a numerical observation that needs a task-specific interpretation. A heatmap does not supply that interpretation by itself.

Finally, depth is not lasting memory. Ordinary inference computes fresh activations using fixed parameters. Runtimes may cache intermediate quantities for efficient generation, but a changed residual state is not evidence that the model has learned new weights or permanently stored the conversation.

15.9 Inspect the actual boundary

An inspection must name the checkpoint, input tokens, token position, block, and location relative to normalization and addition. A field called `hidden_states` is an interface choice; it does not universally mean every raw post-addition state. A final normalization can make the last returned state different from the last block’s raw output.

There are topology differences too. The optional Lab uses Pythia-14M, whose pinned configuration selects **parallel residuals**. In that arrangement, attention and MLP read separately normalized versions of the same block input:

$$X^{(\ell+1)} = X^{(\ell)} + A_{\ell}(N_{\ell,A}(X^{(\ell)})) + M_{\ell}(N_{\ell,M}(X^{(\ell)})).$$

The MLP does not read the attention-updated state inside that block. The checkpoint has six blocks and residual width 128. Its small scale is convenient for inspection, but its topology must not be silently relabeled as our sequential example. Pinned Pythia configuration; GPT-NeoX layer implementation

After a decoder stack, a final normalization and vocabulary projection may convert the stream into logits. In row-array notation, if $H = N_f(X^{(L)})$ and W_U has shape $d \times V$, then HW_U has shape $n \times V$. An optional vocabulary bias is broadcast across positions. Residual width and vocabulary size play different roles all the way to the output.

We now have the surrounding structure. Next, attention will explain how one position computes a context-dependent update from allowed positions. The residual addition tells us where that update goes; attention tells us how it is calculated.

15.10 Check your understanding

1. A batch contains two sequences of nine tokens, with residual width 128. What is its activation shape? Does six blocks imply a width of $6 \cdot 128$?
2. In the sequential pre-norm equations, what does the MLP read? What changes in the parallel arrangement?
3. If a branch update is zero, when is the operation necessarily the identity: $x + F(N(x))$, or $N(x + F(x))$?
4. Why can a later stream have a smaller norm than an earlier stream?
5. Why was subtracting the original first update insufficient to predict our toy intervention?
6. What evidence is missing from “this layer changed the vector most, so it did the most reasoning”?

i Solutions

1. Shape (2, 9, 128). Six width-preserving blocks change depth, not width; each boundary retains that shape.
2. It reads $N_{\ell,M}(Y^{(\ell)})$, the normalized attention-updated state. In the parallel arrangement it reads its own normalized version of the block input.
3. Zeroing the pre-norm branch gives x . Zeroing the post-norm branch gives $N(x)$, which need not equal x .

4. Updates can oppose existing coordinates or directions. Normalization can also change scale, so measurement boundaries matter.
5. The intervention changed the second branch's input. Its output had to be recomputed rather than held at its baseline value.
6. Vector distance measures numerical change at a boundary. A reasoning claim requires a defined behavior, appropriate comparisons, and evidence connecting that change to the behavior, ideally including controlled interventions.

15.11 More Learning

1. Attention Is All You Need. Use the staged reading Lab. For this chapter, identify residual additions and normalization in Section 3.1; save attention's equations for the next lesson.
2. Deep Residual Learning for Image Recognition and Identity Mappings in Deep Residual Networks. Historical primary sources for residual learning and direct paths. Their computer-vision experiments are not measurements of LLM behavior.
3. Layer Normalization and Root Mean Square Layer Normalization. Read the definitions first. Derive the two-coordinate example before attempting the training analyses.
4. On Layer Normalization in the Transformer Architecture. An optional mathematical treatment of normalization placement and optimization. Separate the paper's analyzed setting from universal claims.
5. Gemma 3 Technical Report. A contemporary architecture comparison. Mark details that our simplified diagram omits; Lesson 2.4 will make more of them familiar.
6. Pythia-14M model card. Read the intended use and limitations before the optional model-inspection Lab. Small and inspectable does not imply suitable for deployment.

16 2.2 — Attention

16.1 How one position reads another

The residual stream gives each token position a working vector. Attention lets a position construct an update using vectors at other permitted positions. The update depends on both the current representations and learned parameters. Following that computation will help us understand how context changes a model's behavior without changing its stored weights.

Consider the unfinished sentence “The key beside the cabinets ...”. Predicting what follows may require information from several earlier positions. A model could use attention to bring information about “key” toward the position producing the prediction. That is a possible computation to investigate, not a claim that an unspecified model contains a dedicated grammar head. Real token boundaries may also differ from the word boundaries in this illustration.

We will build one attention head small enough to calculate exactly, then connect it to multi-head attention and the residual stream. We will also establish a careful limit: an attention map exposes one intermediate computation, but cannot by itself explain the model's final answer.

By the end, you should be able to:

- distinguish query, key, and value vectors from their learned projection matrices;

- check the dimensions of an attention calculation;
- compute scaled scores, masked row-wise softmax, and a weighted value mixture;
- explain which information a causal position may access;
- describe how several heads produce a residual-stream update;
- evaluate a claim made from an attention visualization.

The prerequisite is Lesson 2.1 — The Residual Stream. Keep its distinction between a residual state and a sublayer’s contribution in view.

i Math to know / refresh

Needed now: dot products, matrix multiplication, transpose, exponentials, and normalization by a sum. We use these operations explicitly below.

Useful refresh: the softmax from Lesson 1.2, row versus column conventions from Lesson 1.3, and the fact that a dot product depends on both direction and magnitude.

Side trail: variance calculations behind the scaling factor, derivatives through softmax, and matrix rank. None is required to complete the worked example.

16.2 Three projections of the same input

For this lesson, stack the vectors supplied to attention as **rows**:

$$X \in \mathbb{R}^{n \times d_{\text{model}}}.$$

There are n token positions and d_{model} coordinates per position. Lesson 1.3 initially used individual column vectors. Here x_i is a row, so we multiply a projection matrix on its right. Transposing both representations recovers the other convention; the underlying operation is unchanged.

X means the input actually supplied to this attention sublayer. In a pre-normalized block, it is the normalized residual state. It need not be the original token embedding, and we should not silently replace it with one.

One head forms three arrays:

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V.$$

Their roles differ:

- A **query** q_i is used by destination position i to score possible sources.
- A **key** k_j is the vector against which source position j is scored.
- A **value** v_j is the vector mixed into the destination’s result.

The names suggest a lookup, but the computation usually returns a mixture of values rather than one selected record. Queries are numerical vectors, not textual questions; keys are numerical vectors, not labels identifying concepts.

If queries and keys have d_k coordinates and values have d_v coordinates, then

$$W_Q, W_K \in \mathbb{R}^{d_{\text{model}} \times d_k}, Q, K \in \mathbb{R}^{n \times d_k},$$

$$W_V \in \mathbb{R}^{d_{\text{model}} \times d_v}, \quad V \in \mathbb{R}^{n \times d_v}.$$

The query and key widths must match because we take their dot product. The value width can differ. Within this head, the same W_Q , W_K , and W_V apply at every position. Their entries are learned parameters; the entries of Q , K , and V are computed activations. We omit optional projection biases to keep the equations readable.

These equations are the standard projected attention construction introduced in the original Transformer. The distinction between the three projections is important: a source's compatibility score and the content it contributes are calculated through different matrices. Vaswani et al., section 3.2

16.3 Scores become a distribution over sources

Form the scaled score matrix

$$S = \frac{QK^T}{\sqrt{d_k}}, \quad S_{ij} = \frac{q_i \cdot k_j}{\sqrt{d_k}}.$$

The product has shape $(n \times d_k)(d_k \times n) = n \times n$. **Rows identify destinations; columns identify sources.** Entry S_{ij} is a score for destination i reading source j . Transposing the map reverses those roles.

A score is neither a probability nor a measured semantic similarity. Learned projections determine which input directions contribute to the dot product. Increasing a vector's magnitude can change the score without changing its direction, so ordinary dot-product attention is not cosine similarity.

The divisor uses the query/key width d_k , not the number of positions and not automatically the residual width. The original paper motivates it with an idealized calculation: independent, zero-mean, unit-variance query and key coordinates give a dot product with variance d_k . Dividing by $\sqrt{d_k}$ counters that growth. Trained activations need not satisfy those assumptions; the scaling is not a guarantee that all scores have a particular distribution. Vaswani et al., section 3.2.1

We then apply softmax across the **source columns of each row**. For an allowed-source set J_i ,

$$A_{ij} = \frac{\exp(S_{ij})}{\sum_{r \in J_i} \exp(S_{ir})} \quad \text{when } j \in J_i,$$

and $A_{ij} = 0$ otherwise. Here we inspect weights with attention-weight dropout disabled. Each valid row sums to one. The columns need not sum to one: several destinations can read the same source strongly. Attention does not allocate an exclusive supply of information that other positions then lose.

These weights resemble the probabilities from Lesson 1.2, but their axis and role differ. A vocabulary softmax produces weights over possible output tokens. This softmax produces mixture coefficients over permitted source positions. No token is sampled here.

For numerical stability, software can subtract the largest allowed score in a row before exponentiating. The common factor cancels between numerator and denominator. For example, $[1000, 1000]$ and $[0, 0]$ both yield $[1/2, 1/2]$, while the latter avoids computing enormous exponentials.

💡 Predict before continuing

If all allowed scores in a row are equal, what weights result? If one score remains unchanged but a competing score increases, does the first source retain its old weight? Explain using the denominator.

16.4 A mask defines the available information

For causal self-attention, position i may read positions up to and including itself. Define an additive mask

$$M_{ij} = \begin{cases} 0, & j \leq i, \\ -\infty, & j > i. \end{cases}$$

Then the whole computation is

$$A = \text{softmax}_{\text{sources}}(S + M), \quad Z = AV.$$

The mask goes **before softmax**. Since $\exp(-\infty) = 0$, forbidden positions contribute nothing to either the numerator or the normalization sum. Setting a forbidden score to zero is insufficient: $\exp(0) = 1$. Multiplying weights by a triangular mask only after softmax also gives the wrong normalization unless it is corrected.

The mask is part of the computation's conditions. Keeping Q , K , and V fixed while changing the permitted sources can change the output and every remaining weight in the affected row. An attention weight therefore means something only relative to its competitors and mask.

A **padding mask** answers a different question. Batched sequences may contain filler positions to make arrays rectangular; those positions should not serve as real sources. Causal masking limits access by sequence order. Padding masking excludes filler. Their allowed-source conditions can be combined. Masking padded keys does not itself guarantee zero outputs at padded query positions; those positions must be handled appropriately by the surrounding computation and loss. PyTorch MultiheadAttention documentation

Our mathematical convention uses zero for allowed scores and negative infinity for blocked scores. APIs can instead accept booleans, with differing meanings. For example, PyTorch's scaled-dot-product function treats True as allowed, whereas MultiheadAttention's boolean padding mask treats True as blocked. Always check the specific API. Our Lab uses an explicitly named allowed array. PyTorch scaled-dot-product attention documentation

A valid softmax row needs at least one allowed finite score. Masking every source gives an undefined all-zero normalization in the mathematics. Do not assume a library will handle that case in the way your application needs.

16.4.1 Why a position can include itself

With token inputs t_1, t_2, t_3 , the output at position 2 is commonly trained to predict t_3 . It can use t_2 , which is already supplied, but must not read t_3 . The training targets are shifted relative to the inputs. The mask may therefore include the diagonal without revealing that position's next-token target.

Training can compute many such positions together because the mask maintains the dependency restriction. Suppose every layer mixes only from earlier or equal positions, normalization operates within a position, and other sublayers do not mix across positions. An induction over layers then shows that position i depends only on input tokens through i . Computing arrays in parallel does not create an allowed path from a later token.

16.5 A complete three-position calculation

Use three invented input vectors, with two coordinates each:

$$X = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}.$$

They represent three token positions for arithmetic purposes. They are not embeddings extracted from a trained model, and their coordinates have no assigned linguistic meaning. Choose

$$W_Q = I_2, \quad W_K = \sqrt{2} \ln(2) I_2, \quad W_V = \begin{bmatrix} 2 & 0 \\ 0 & 4 \end{bmatrix}.$$

I_2 is the two-dimensional identity matrix. The logarithm is natural, so $\exp(\ln 2) = 2$. We chose the unusual-looking key scale precisely to make softmax give simple fractions.

Projection gives

$$Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}, \quad K = \sqrt{2} \ln(2) \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}, \quad V = \begin{bmatrix} 2 & 0 \\ 0 & 4 \\ 2 & 4 \end{bmatrix}.$$

Because $d_k = 2$, the $\sqrt{2}$ in K cancels the score divisor:

$$S = \ln(2) \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 2 \end{bmatrix}.$$

For example, $q_3 = [1, 1]$ dotted with $k_3 = \sqrt{2} \ln(2) [1, 1]$ gives $2\sqrt{2} \ln(2)$. Dividing by $\sqrt{2}$ leaves $2 \ln(2)$.

Add the causal mask:

$$S + M = \begin{bmatrix} \ln 2 & -\infty & -\infty \\ 0 & \ln 2 & -\infty \\ \ln 2 & \ln 2 & 2 \ln 2 \end{bmatrix}.$$

Now normalize each row separately:

- Row 1 has exponentiated scores $[2, 0, 0]$, giving $[1, 0, 0]$.
- Row 2 has $[1, 2, 0]$, giving $[1/3, 2/3, 0]$.
- Row 3 has $[2, 2, 4]$, giving $[1/4, 1/4, 1/2]$.

Thus

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 1/3 & 2/3 & 0 \\ 1/4 & 1/4 & 1/2 \end{bmatrix}.$$

The output at each destination is a weighted mixture of **value vectors**:

$$z_i = \sum_j A_{ij} v_j.$$

For the last position,

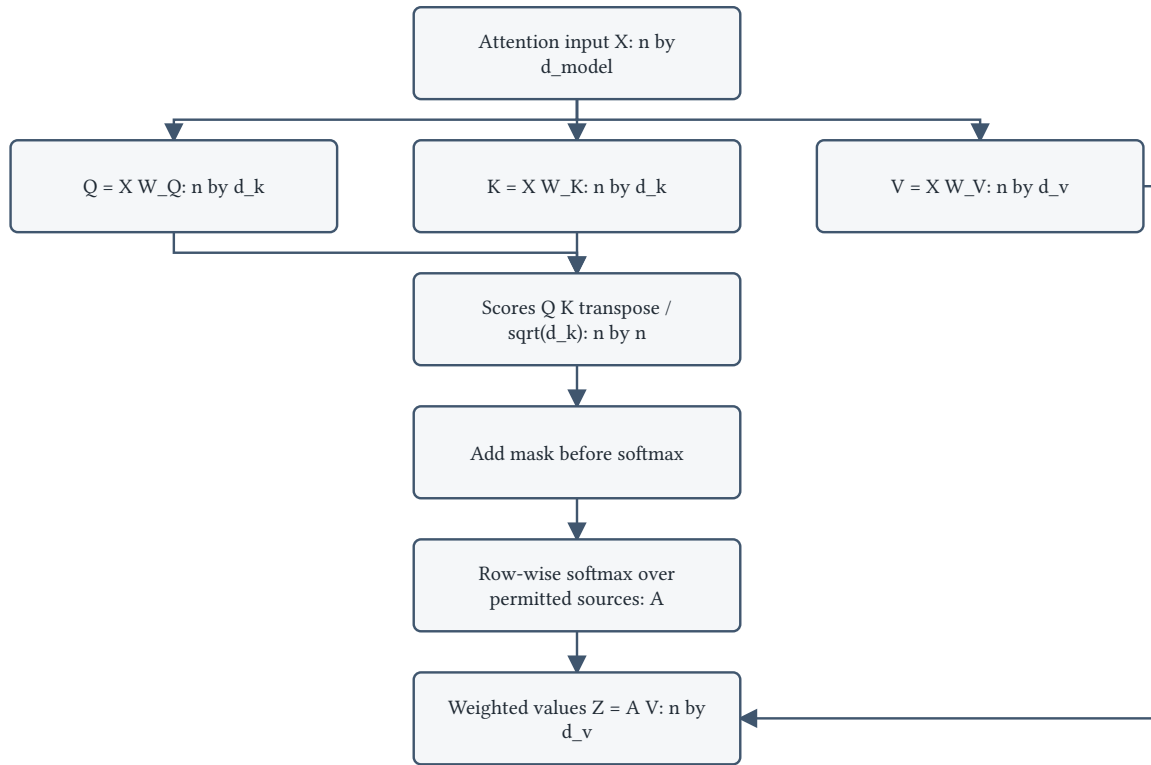
$$z_3 = \frac{1}{4}[2, 0] + \frac{1}{4}[0, 4] + \frac{1}{2}[2, 4] = \left[\frac{3}{2}, 3 \right].$$

Calculating the other rows gives

$$Z = AV = \begin{bmatrix} 2 & 0 \\ 2/3 & 8/3 \\ 3/2 & 3 \end{bmatrix}.$$

The matrix product has shape $(3 \times 3)(3 \times 2) = 3 \times 2$. It keeps one output vector per destination. The source vectors remain available to other destinations; this calculation does not consume or overwrite them.

In this no-dropout calculation, each row of Z is a convex combination of permitted values: coefficients are nonnegative and sum to one. This does not make the coordinates probabilities. Our output has a coordinate equal to 3, and valid value vectors can contain negative coordinates.



Original course diagram. Follow one destination row: its query scores the source keys, the mask removes forbidden choices, and the normalized weights mix their values. The value path reaches the output without passing through the score calculation. This separation will matter in the Lab.

! Lab recommended here

Lab 06 — Compute and Perturb Attention begins with hand calculation and masking, then implements small arrays using Python’s standard library. Record predictions before checking the answer key. The Lab separates changes to weights in the attention map from changes to the values being mixed.

16.6 Several heads write back to the residual stream

A single head produces one distribution over sources for each destination. Every coordinate of that head’s value output uses the same distribution. Several heads allow different projected representations and different distributions to coexist.

For head h , write $Z^{(h)} = A^{(h)} V^{(h)}$. With H heads of value width d_v , concatenate their output coordinates and apply a learned output projection:

$$Y = \text{Concat} \left(Z^{(1)}, \dots, Z^{(H)} \right) W_O, \quad W_O \in \mathbb{R}^{H d_v \times d_{\text{model}}}.$$

Then Y has shape $n \times d_{\text{model}}$, appropriate for the residual addition discussed in Lesson 2.1. Concatenation happens along the feature dimension, not along the sequence. Output projection can mix, amplify, suppress, or change the sign of head coordinates. Consequently the final residual update is not constrained to be a convex average of the original residual vectors. PyTorch MultiheadAttention documentation

Imagine a deliberately constructed two-head computation at position 3. One head uses $[1, 0, 0]$ and carries a feature from position 1; another uses $[0, 1, 0]$ and carries a different feature from position 2. Concatenation preserves the two outputs separately for W_O to combine. These are idealized routing patterns for explanation, not measured attention maps. With finite unmasked softmax scores, such exact one-hot patterns are approached rather than attained.

Heads do not arrive with fixed labels such as “grammar” and “facts.” Describing a trained head requires evidence across examples and a specified measurement. Multiple heads can cooperate, overlap, or be unimportant for a particular input.

For our one-head numerical example, choose $W_O = I_2$. Then $Y = Z$. If X is also the residual state in a deliberately simplified block with no normalization, the residual result is $X + Y$, whose last row is $[5/2, 4]$. In a pre-normalized block, add Y to the original residual state, not automatically to the normalized X .

16.7 Contextual computation and stored weights

Consider replacing a token while leaving the checkpoint unchanged. Its residual representation can change; so can its projected key and value. Other positions’ queries can then produce different attention weights and mixtures. Later layers propagate those changes. None of this requires updating a learned parameter.

The reverse distinction matters too. Attention is parameterized by learned matrices, including the projections that determine what is scored and what is written. Calling it “context mixing” should not suggest that attention has no learned structure. Its temporary coefficients A_{ij} and its stored parameters W_Q, W_K, W_V, W_O both matter, but on different timescales. The research distinction between query-key and output-value computations is a useful deeper view of these roles. Elhage et al., A Mathematical Framework for Transformer Circuits

At a later layer, source position j may already contain information gathered from earlier positions. Reading v_j therefore need not mean reading only token t_j in isolation. A map records the immediate source position of one mixture; tracing where that source’s contents came from requires looking at preceding computation.

The mask establishes possible routes, not a guarantee that relevant information is recovered. An allowed earlier detail may receive little weight, may be poorly represented in the values, or may be suppressed by downstream transformations. Access and effective use are separate questions.

16.8 Self-attention and cross-attention

Self-attention means queries, keys, and values are derived from the same sequence of states. It does not mean each position reads only itself. A causal decoder restricts self-attention to the prefix. An

encoder can instead let every non-padding position read the whole input, including positions to its right. BERT is a primary example of a bidirectional Transformer encoder. Devlin et al., BERT

Cross-attention uses queries from one sequence and keys and values from another. The original Transformer’s decoder queries encoder outputs. With m queries and n sources, S and A have shape $m \times n$; each row normalizes over its available sources. Vaswani et al., section 3.2.3

Choosing a mask is therefore an architectural and task decision. A translation decoder can read the supplied source sentence while remaining causal over its generated target sequence. “All attention must be triangular” would confuse one important use with the general mechanism.

Positional information also influences practical attention. Our tiny example isolates the arithmetic and includes no separate positional mechanism. Lesson 2.4 will examine positions and modern implementation choices; this lesson’s job is to make the basic computation inspectable.

16.9 What an attention map can establish

For a specified input, layer, head, and mask, a faithful plot of A shows the coefficients used to mix the value vectors. It can reveal an unexpectedly reversed axis, a broken causal mask, or a repeatable pattern worth investigating. Record whether a visualization shows one head or an average across heads; an average can hide distinct patterns.

The map omits the value vectors and output projection. Our example makes that omission concrete. Keep Q , K , and the mask fixed, but replace $v_2 = [0, 4]$ with $[0, 0]$ after projection. The map remains identical, yet

$$z_{3'} = \frac{1}{4}[2, 0] + \frac{1}{4}[0, 0] + \frac{1}{2}[2, 4] = \left[\frac{3}{2}, 2\right].$$

Identical maps can accompany different outputs. Conversely, if two sources have identical values, real-locating weight between them while preserving their combined weight leaves this mixture unchanged. Neither statement says that attention weights never matter. Each follows from the full equation $Z = AV$.

Research on Transformer attention supports examining transformed vectors as well as coefficients: Kobayashi and colleagues showed that vector norms changed conclusions drawn from weight-only analyses of BERT and a translation model. Norms add information but still omit direction, cancellation, and downstream computation. Kobayashi et al., 2020

The broader explanation debate also needs its scope preserved. Jain and Wallace found failures of weight-based explanations in the attention-equipped NLP models they tested. Wiegrefe and Pinter challenged aspects of the tests and argued for explicit definitions and stronger controls. Much of that exchange concerned recurrent models with attention, not a universal evaluation of every modern decoder Transformer. The practical lesson is to test an explanatory claim rather than declare attention always sufficient or always useless. Jain and Wallace, 2019; Wiegrefe and Pinter, 2019

For a causal claim, specify an intervention and an outcome. “Zero this head’s contribution and measure the change in the next-token logit” is testable. “This bright square explains the answer” skips the values, other heads, residual path, MLPs, and later layers. Even a measured ablation effect is conditional on

the input, intervention, and metric; it does not automatically establish a unique explanation or broad linguistic role.

16.10 Check your understanding

Try these without looking back, then compare your reasoning with the answers.

1. X has shape 5×12 , and a head has $d_k = 3$, $d_v = 4$. What are the shapes of W_Q , S , A , and Z ?
2. Why does replacing a blocked score with zero fail to implement a causal mask?
3. What happens to a source's normalized weight if its score stays fixed while a competing allowed score increases?
4. In the worked example, does $A_{11} = 1$ show that position 1 is more important to the final answer than position 3?
5. At position 3, forbid source 2 while preserving all remaining scores and values. Compute the new weights and mixture.
6. Can a token change a later position's output without changing the checkpoint? Can an unchanged attention map establish an unchanged output?

i Answers and reasoning

1. $W_Q : 12 \times 3$; $S, A : 5 \times 5$; $Z : 5 \times 4$.
2. Zero has exponential one. A forbidden source must be excluded from normalization, represented here by negative infinity before softmax.
3. It decreases because the denominator increases. Equal allowed scores give a uniform distribution over those sources.
4. No. Position 1 has only one allowed source, so its sole coefficient must be one. That says nothing about its downstream importance.
5. Remaining exponentials are $[2, 0, 4]$, giving $[1/3, 0, 2/3]$ and $[2, 8/3]$. Zeroing v_2 instead would preserve A and give $[3/2, 2]$; these are different interventions.
6. Yes: activations and mixtures change at inference with fixed parameters. No: values or output projections can change while the map stays the same.

16.11 More Learning

1. **Attention Is All You Need — Vaswani et al.** Read section 3.2 beside your hand calculation. Identify the projection, normalization, masking, and multi-head operations before returning to the whole architecture.
2. **PyTorch scaled-dot-product attention documentation** Read the reference pseudocode and mask semantics. Notice the explicit softmax axis and dropout argument; our Lab omits dropout and framework dependencies.
3. **A Mathematical Framework for Transformer Circuits — Elhage et al.** Optional deeper reading on the separate query-key and output-value computations. Pay attention to its simplified model assumptions and different vector convention.
4. **Attention is Not Only a Weight — Kobayashi et al.** Follow the move from plotting scalar coefficients to inspecting the vectors they scale. Ask which information a norm still leaves out.

5. **Attention is not Explanation — Jain and Wallace** Read the experimental setup before generalizing the conclusion. Identify the models, alternative importance measures, and counterfactual tests.
6. **Attention is not not Explanation — Wiegrefe and Pinter** Read alongside the preceding paper. Write one sentence defining the explanatory claim you would actually test and one control it would require.

The next lesson examines the position-wise MLP: what it does with the information represented at each position, and why a coordinate or neuron need not correspond neatly to one concept.

17 2.3 — Multilayer Perceptrons, Features, and Superposition

17.1 From a contextual vector to a new contribution

Attention gives a token position access to information from permitted positions. The position-wise multilayer perceptron, or **MLP**, provides another kind of computation: it transforms a vector through a wider collection of nonlinear responses and writes a vector back into the residual stream.

You already calculated an affine–nonlinear–affine network in Lesson 1.3. We will not repeat its introduction to multiplication or training. Instead, we will ask what changes when that network becomes a component inside a Transformer, and why observing its hidden units is harder to interpret than computing them.

A tempting picture gives each unit a label: one for geography, another for quotation marks, another for Python. Sometimes a unit responds selectively enough to support a useful description. But the architecture supplies numerical coordinates, not a guaranteed inventory of concepts. A feature can involve several coordinates, and one coordinate can participate in several features. We need language that separates those possibilities before interpreting an activation plot.

By the end, you should be able to:

- trace an MLP from residual-width input through hidden expansion to residual-width output;
- distinguish a unit’s incoming response from its outgoing contribution;
- separate units, feature directions, distributed representations, polysemanticity, and superposition;
- calculate when a small overcomplete representation succeeds and when information is lost;
- make a bounded claim from contrasting activations and describe stronger evidence it would require.

The prerequisites are Lesson 1.3, the residual stream, and attention.

i Math to know / refresh

Needed now: matrix shapes, dot products, affine maps, coordinatewise nonlinear functions, and vector addition. We use individual column vectors, as in Lesson 1.3.

Useful refresh: a basis gives independent coordinate directions; a linear combination adds scaled vectors; orthogonal vectors have zero dot product. A collection of directions need not be a basis.

Side trail: matrix rank, compressed sensing, and dictionary-learning objectives. The examples need only elementary arithmetic; training sparse autoencoders belongs to Lesson 5.2.

17.2 A shared transformation at every position

Let $x_i \in \mathbb{R}^d$ be the residual vector immediately before a sequential pre-norm MLP branch at position i . Its actual input is $u_i = N(x_i)$, where N is that branch's normalization. Define

$$\begin{aligned}z_i &= W_{\text{in}} u_i + b_{\text{in}}, \\h_i &= \phi(z_i), \\\Delta_i &= W_{\text{out}} h_i + b_{\text{out}}, \\y_i &= x_i + \Delta_i.\end{aligned}$$

Here z_i is the preactivation, h_i the post-activation hidden vector, and Δ_i the MLP's residual contribution. The last line is the surrounding residual addition, not another operation hidden inside the MLP. We inspect evaluation-time computation with dropout disabled.

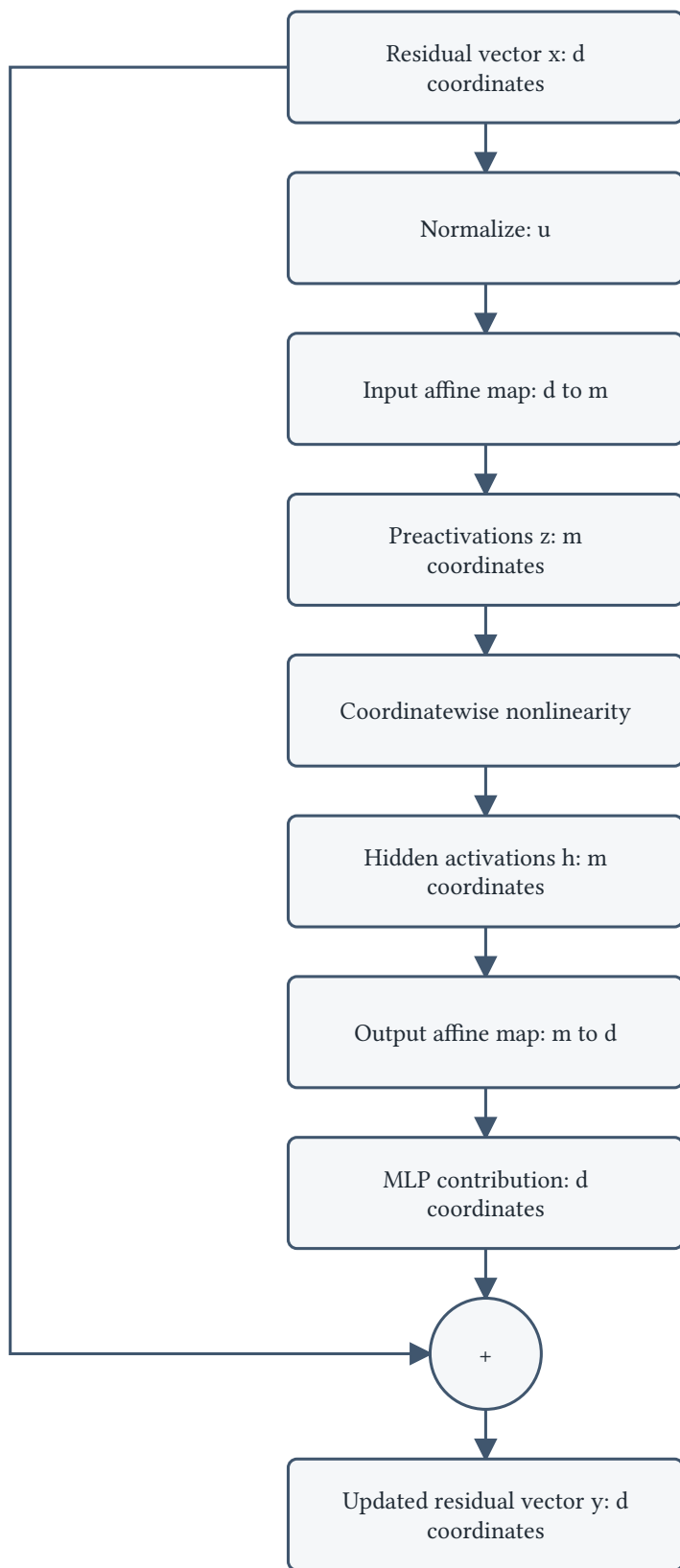
For hidden width m ,

$$W_{\text{in}} \in \mathbb{R}^{m \times d}, \quad b_{\text{in}} \in \mathbb{R}^m, \quad W_{\text{out}} \in \mathbb{R}^{d \times m}, \quad b_{\text{out}} \in \mathbb{R}^d.$$

Thus the branch changes width $d \rightarrow m \rightarrow d$. Biases may be absent in particular architectures. The output returns to d because it must be added to x_i . Hidden width is neither sequence length nor vocabulary size.

The same matrices and biases apply at every position within this MLP. Different blocks generally have different parameters. No displayed equation reads u_j for a different position j : the MLP does not itself transport information across positions. Nevertheless, u_i can already contain information gathered from the prefix. Position-wise computation can therefore respond to context without directly looking up another row.

The original Transformer's feedforward component uses this two-affine-map pattern with ReLU, residual width 512, and hidden width 2048. Its normalization is after residual addition, unlike our pre-norm illustration. The shared position-wise operation and the surrounding topology are separate choices. Vaswani and colleagues, Sections 3.1 and 3.3



Original course diagram of one position's sequential pre-norm MLP branch. Repeat the same branch at each position with shared parameters.

Prose alternative: Keep the original residual vector for the skip path. Normalize another copy, expand it to the hidden width, apply the nonlinearity to each hidden coordinate, and project back to residual width. Add that contribution to the original residual vector. The expanded vector is an intermediate activation, not a wider residual stream.

17.3 Expansion gives more nonlinear responses

A wider hidden layer provides more input-weight vectors and nonlinear response functions before the branch combines their outputs. It does not create new independent information from nowhere. Every hidden value remains a function of the supplied u_i and stored parameters.

Without a nonlinearity, the two affine maps collapse into one affine map, as you derived in Lesson 1.3. With ReLU, different inputs can select different sets of positive responses. Increasing m offers more such responses, though actual capabilities depend on the trained parameters and the rest of the network. Width alone neither counts learned concepts nor guarantees their independence.

For this two-matrix form, the weight count is $2dm$, plus $m + d$ biases when present. Doubling m doubles the matrix-weight count. That makes hidden expansion a substantial resource choice. It is not a free way to add features, and the common ratio $m/d = 4$ is a configuration choice rather than a law.

We will keep activation variants brief. **ReLU** clips negative inputs to zero. **GELU**, used in the optional Lab's checkpoint, smoothly rescales its input; in its exact mathematical form, $\text{GELU}(t) = t\Phi(t)$, with Φ the standard-normal cumulative distribution function. Negative inputs need not become exact zeros. A ReLU-style count of "active neurons" therefore does not transfer unchanged to GELU. PyTorch GELU reference

A **gated MLP** introduces an elementwise product between two input projections. A bias-free schematic is

$$\Delta = W_{\text{out}} [\phi(W_g u) \odot (W_v u)].$$

The symbol $\odot$ means multiply matching coordinates. SwiGLU uses a Swish/SiLU-type nonlinearity on one branch. This form has three weight matrices, so comparing equal hidden widths does not compare equal parameter budgets. Shazeer's experiments explicitly adjusted width for that comparison. These gates are numerical operations, not separate agents choosing what to remember. GLU Variants Improve Transformer

17.4 Each unit has a read vector and a write vector

Drop the position subscript temporarily. Let w_j^T be row j of W_{in} and c_j be column j of W_{out} . Then

$$h_j = \phi(w_j^T u + b_j), \quad \Delta = b_{\text{out}} + \sum_{j=1}^m h_j c_j.$$

This form exposes two different roles. The input weights determine the unit's response to u . The output column determines the direction it contributes, scaled by that response. Those vectors are learned separately; there is no requirement that $w_j = c_j$.

A large h_j alone does not establish a large residual effect: c_j may be small, or other contributions may cancel it. Even a large contribution need not strongly affect a particular downstream readout. For an immediate linear score $s = q^T y$, zeroing just h_j changes the score by $-h_j q^T c_j$. Direction matters as well as magnitude. Later nonlinear computation requires recomputation rather than this immediate-readout shortcut.

Geva and colleagues analyzed feedforward layers as learned key-value memories: input-side patterns activate associated output-side vectors. Their study includes a 16-layer Transformer language model trained on WikiText-103. This is a research interpretation supported by a specified experiment. It does not make each unit a literal database row holding one fact. Unlike attention's input-dependent keys and values, these particular read and write vectors are stored parameters. The coefficients h_j change with the input. Transformer Feed-Forward Layers Are Key-Value Memories

17.4.1 An exact residual write

Use the following original, hand-constructed MLP. It has no training history or assigned linguistic meaning. Omit normalization for this arithmetic example only, so $u = x$:

$$x = \begin{bmatrix} 1 \\ -1 \\ 2 \end{bmatrix}, \quad W_{\text{in}} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & -1 & 1 \\ 1 & 0 & -1 \\ -1 & 1 & 1 \end{bmatrix}, \quad b_{\text{in}} = \begin{bmatrix} 0 \\ -1 \\ 1 \\ 1 \end{bmatrix}.$$

Choose ReLU and

$$W_{\text{out}} = \begin{bmatrix} 1 & 0 & -1 & 1 \\ 0 & 1 & 0 & -2 \\ 1 & -1 & 2 & 0 \end{bmatrix}, \quad b_{\text{out}} = \begin{bmatrix} 0 \\ 1/2 \\ 0 \end{bmatrix}.$$

Check the shapes first: $3 \rightarrow 4 \rightarrow 3$. The incoming map gives

$$z = [0, 2, 0, 1]^T, \quad h = [0, 2, 0, 1]^T.$$

Only units 2 and 4 contribute through their output columns. Their contributions and the bias give

$$\Delta = 2 \begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix} + \begin{bmatrix} 1 \\ -2 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1/2 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1/2 \\ -2 \end{bmatrix}.$$

Adding to the original input produces $y = [2, -1/2, 0]^T$. Hidden unit 4 has a positive activation yet reduces the second coordinate. Units 2 and 4 cancel each other's non-bias contributions to that coordinate.

 Predict before continuing

Set only h_4 to zero after ReLU. What happens to each coordinate of y ? Does the third coordinate change? Predict the complete vector before reading on.

The new contribution is $[0, 5/2, -2]^T$ and the new residual vector is $[1, 3/2, 0]^T$. Relative to baseline, the change is $[-1, 2, 0]^T$. Unit 4 affected two coordinates but had no effect on the immediate third-coordinate readout. That is a precise causal statement about this intervention in this constructed computation. It does not identify a language feature.

17.5 Features are not the same objects as units

A **unit** here is a specified coordinate of h , with its input-side activation rule. A **direction** is a vector in a specified representation space. A **feature**, in interpretability work, is a proposed property or variable represented by some part of the computation. “Feature” is used at several levels, so a useful claim must say which level it means.

For example, an investigator might hypothesize a variable associated with being inside a quotation, following an opening bracket, or referring to a geographical location. That is a hypothesis about an input property and its internal representation. The property need not align with one hidden coordinate, and its best description may be narrower than the investigator’s first label. A scalar linear feature is one useful model of representation; not every property must be one-dimensional or linearly encoded.

A **distributed representation** uses several coordinates to represent a property. Consider two feature directions

$$p = \frac{1}{\sqrt{2}}[1, 1]^T, \quad q = \frac{1}{\sqrt{2}}[1, -1]^T.$$

The code $r = ap + bq$ distributes each feature across both coordinates. Nevertheless, $p^T q = 0$, and dot products recover $a = p^T r$ and $b = q^T r$ exactly. Two features fit in two independent directions. Distribution across coordinates alone has not established superposition.

Polysemanticity describes a unit responding to multiple distinct features or apparently unrelated input patterns. **Superposition**, in the sense used here, concerns representing more features than a space has dimensions, with overlapping directions and potential interference. Superposition can help explain polysemantic units, but the observations and the mechanism are not synonyms. A mixed-looking unit could reflect the orthogonal code above, an incomplete feature description, or correlated input properties. Elhage and colleagues, Toy Models of Superposition

There is an architectural subtlety. MLP coordinates are not freely interchangeable with arbitrary rotated coordinates: a coordinatewise nonlinearity acts along the implemented axes. Generally, rotating and then applying ReLU differs from applying ReLU and then rotating. The architecture therefore makes these axes special for computation. It still does not require training to assign one human concept to each axis. This follows from the operation itself, without assuming every unit must be interpretable or uninterpretable.

17.6 Three features in two dimensions

To see the geometric issue directly, design three unit-length feature vectors:

$$v_A = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad v_B = \begin{bmatrix} -1/2 \\ \sqrt{3}/2 \end{bmatrix}, \quad v_C = \begin{bmatrix} -1/2 \\ -\sqrt{3}/2 \end{bmatrix}.$$

Draw three arrows from the origin, pointing right, upper-left, and lower-left, separated by 120° . Every distinct pair has dot product $-1/2$. This sketch is a complete geometric picture; no trained model supplied these directions.

For nonnegative feature strengths a_A, a_B, a_C , encode them as

$$r = a_A v_A + a_B v_B + a_C v_C = D a, \quad D = [v_A v_B v_C] \in \mathbb{R}^{2 \times 3}.$$

A **dictionary** is a collection of building-block vectors such as these columns. This dictionary is **over-complete** because it contains more vectors than the representation's dimension. Its columns are not three independent basis vectors. The two-dimensional representation still has only two coordinates.

Try a simple nonlinear readout:

$$\hat{a} = \text{ReLU} (D^T r).$$

This is a stipulated decoder, not a universal solution to recovering features. If only A is present with strength one, $a = [1, 0, 0]^T$ and $r = v_A$. Then

$$D^T r = [1, -1/2, -1/2]^T, \quad \hat{a} = [1, 0, 0]^T.$$

ReLU removes the negative interference. The same reasoning recovers B or C alone, at any nonnegative strength. Our dictionary distinguishes three possible singleton features using two coordinates because the permitted inputs have additional structure: at most one feature is present. We did not invert an arbitrary three-dimensional vector through a two-dimensional linear map.

Now activate A and B together at strength one:

$$a = [1, 1, 0]^T, \quad r = \begin{bmatrix} 1/2 \\ \sqrt{3}/2 \end{bmatrix}.$$

The readout gives $D^T r = [1/2, 1/2, -1]^T$, so $\hat{a} = [1/2, 1/2, 0]^T$. Both active strengths are underestimated. Another decoder might improve this particular case, but it cannot repair all unrestricted inputs: since $v_A + v_B + v_C = 0$, the inputs $[1, 1, 1]^T$ and $[0, 0, 0]^T$ encode to exactly the same r . Information has genuinely been lost.

This gives **sparsity** a concrete role. If most feature strengths are zero on most inputs, troublesome combinations may be uncommon. A representation can work well on that restricted distribution despite failing on other combinations. That is a distribution-dependent trade-off, not a way to repeal dimensionality. Sparsity of the underlying feature strengths also does not require sparsity of the observed coordinates: B alone already uses both coordinates of r .

In a large space, many directions can be weakly overlapping without being exactly orthogonal. Their utility depends on interference, noise, feature co-occurrence, and the computations required. Our two-dimensional triangle makes the cost easy to inspect; it is not a scale model whose exact angles predict an LLM's geometry.

! Lab recommended here

Do Inspect MLP Responses and Feature Geometry. First trace a fresh MLP input and test the hand-designed dictionary. Then, optionally, compare actual post-activation MLP values for contrasting prompts in a small trained model. The synthetic calculation and trained-model observations answer different questions; preserve that distinction in your artifact.

17.7 What the construction establishes

We know A, B, and C are features because we defined them as the generating variables. We know the dictionary because we chose it. We can test reconstruction against the original strengths because those strengths are available. None of those advantages is automatic when opening a trained language model.

The published *Toy Models of Superposition* work trains small networks on controlled synthetic feature distributions and studies when superposition appears. It establishes mechanisms under those settings. Our untrained construction illustrates one mechanism without reproducing its training experiments. Neither provides a complete inventory of features in an arbitrary LLM. Elhage and colleagues

There is also no shortcut from expansion width to a superposition claim. When $m > d$, an MLP has more output columns than residual dimensions, but that algebra alone does not establish m distinct semantic features. Columns may be redundant, coefficients may be dependent, and a unit may perform useful computation without a simple semantic label. The dictionary interpretation must be tested against the representations and behavior of interest.

Later, sparse autoencoders will provide one method for learning candidate dictionaries from activations. For now, the important question is what such a method would need to establish: useful reconstruction, stable and testable feature descriptions, and a relationship to the model's computation. A readable label is evidence to investigate, not a certificate of a unique concept.

17.8 From an activation pattern to an explanatory claim

Suppose unit 137 responds more strongly on several code snippets than on several prose snippets. The immediate observation is conditional: at a named checkpoint, layer, token boundary, and prompt set, this coordinate takes different values. It does not establish a universal “coding neuron.” Code snippets may differ in punctuation, length, indentation, token identity, or position. The unit could track any combination of these.

Good comparisons preserve the raw prompts and tokenization, use controlled variations, and reserve fresh examples for checking a proposed interpretation. Choosing the most striking unit from hundreds and reporting only its discovery examples favors patterns selected by chance. A held-out check reduces that problem; a few held-out examples still do not establish broad generalization.

An intervention asks a different question. Zeroing a unit, changing a direction, or replacing an activation can test whether that change affects a specified output. The conclusion should name the replacement, position, and metric. A changed next-token logit supports causal sensitivity to that intervention under those conditions. It does not automatically show that the unit uniquely stores the

proposed concept. Ablation can remove several contributions at once or create an unusual internal state; redundancy can hide effects on other inputs.

Even successful decoding has limits. If a probe predicts a label from activations, that establishes accessible information for that probe on the evaluation set. It does not by itself establish that the original model uses that information in the same way. The synthetic example had known variables and an explicit decoder; real-model inspection must earn those interpretations with additional evidence.

The practical habit is to separate four sentences: what was measured, what pattern was observed, what explanation is proposed, and what result would weaken that explanation. This preserves the value of exploratory inspection without asking a heatmap to do more than it can.

17.9 Check your understanding

Answer before opening the solutions.

1. For residual width 128 and hidden width 512, what are the two matrix shapes under this lesson's convention? Why is the residual contribution not 512-dimensional?
2. How can a position-wise MLP respond differently to the same token in two contexts?
3. In the worked MLP, why does zeroing h_4 increase the second residual coordinate?
4. Does a distributed representation necessarily use superposition? Give a counterexample.
5. In the three-feature code, predict the reconstruction of $a = [0, 2, 0]^T$, then explain why no decoder can recover every unrestricted nonnegative input exactly.
6. A unit responds to both dates and source code. What additional work is needed before attributing the pattern to superposition?
7. A unit ablation changes one logit on one prompt. State a supported conclusion and an unsupported one.

i Solutions

1. W_{in} is 512×128 ; W_{out} is 128×512 . The output projection returns the branch to width 128 for residual addition.
2. The supplied representation may already differ because earlier attention and other computation used different prefixes. Shared MLP parameters can produce different activations from those different inputs.
3. Its output column contributes $-2h_4$ there. Removing the baseline $h_4 = 1$ removes a negative contribution, increasing that coordinate by two.
4. No. The orthogonal directions p and q above spread each of two features across two coordinates, with exact independent recovery.
5. $r = 2v_B = [-1, \sqrt{3}]^T$ and $D^T r = [-1, 2, -1]^T$, so reconstruction is exact. But all-zero and all-one strengths share the same encoding. A decoder receiving only that encoding cannot distinguish them.
6. Test the proposed features against controlled and held-out examples, rule out shared surface patterns, and obtain evidence for overlapping feature representations in the relevant space. Mixed responsiveness alone does not establish more represented features than dimensions.

7. Supported: this specified intervention affected this logit under these conditions. Unsupported: the unit uniquely represents a broad concept, or its ablation removes that concept throughout the model.

17.10 Return to the original paper

Complete Part 2 of Reading the Original Transformer Paper. Revisit Sections 3.1–3.3 and annotate the architecture using residual additions, position mixing, hidden expansion, and output projection. Explain why the paper’s feedforward network can be wider internally while its surrounding residual connection stays fixed-width.

You now have the components needed to read that diagram as a computation. Lesson 2.4 will compare it with modern architectural choices rather than treating the 2017 figure as an exact blueprint for every LLM.

17.11 More Learning

1. Vaswani and colleagues: Attention Is All You Need. Read Section 3.3 alongside the return pass of Lab 04. Translate its row-vector convention into this lesson’s column-vector equations.
2. Geva and colleagues: Transformer Feed-Forward Layers Are Key-Value Memories. Follow the input-pattern/output-vector interpretation. Separate the paper’s empirical findings from the stronger claim that every unit is a one-fact memory slot.
3. Elhage and colleagues: Toy Models of Superposition. Read the setup and synthetic-data assumptions before the geometric results. Compare learned toy representations with our explicitly chosen dictionary.
4. Shazeer: GLU Variants Improve Transformer. A brief architecture preview. Inspect the elementwise product and the parameter-matched width comparison; the benchmark details are optional.
5. EleutherAI: Pythia-14M model card and pinned final-checkpoint configuration. Use these for the optional Lab specimen and its limitations. Keep checkpoint identity separate from a model-family name.
6. Transformers GPT-NeoX implementation, v4.57.1. Locate GPTNeoXMLP and distinguish its preactivation, post-activation, and output boundaries. Source inspection prepares an observation; it is not an executed result.

18 2.4 — From the Original Transformer to Modern LLMs

18.1 Reading a model as a set of design choices

A model description might promise “RoPE, RMSNorm, grouped-query attention, a 32K context, and four-bit weights.” Each phrase answers a different question. Some describe the computation learned during training. Others describe how an inference engine stores numbers or reuses work. Understanding their boundaries lets you read a configuration file without mistaking it for a complete description of a running system.

This lesson connects the residual stream, attention, and MLPs from Lessons 2.1–2.3 to two concrete specimens: GPT-2 and Gemma 3 1B. They are useful contrasts, not a ranking or a claim that every later model follows one recipe.

By the end, you should be able to:

- identify what a decoder-only Transformer keeps and omits from the original architecture;
- calculate a small rotary position example and distinguish position from cache storage;
- translate query and KV head counts into projection and cache dimensions;
- distinguish context length, local attention, and actual cache allocation;
- estimate weight and KV payload memory with explicit assumptions;
- explain what configuration metadata cannot establish about a loaded model.

i Math to know / refresh

Needed now: vector norms, dot products, matrix dimensions, two-dimensional rotations, and multiplication of array dimensions.

Useful refresh: sine and cosine, bits versus bytes, and powers of two. One byte is eight bits; one GiB is $2^{30} = 1,073,741,824$ bytes. A decimal GB is 10^9 bytes.

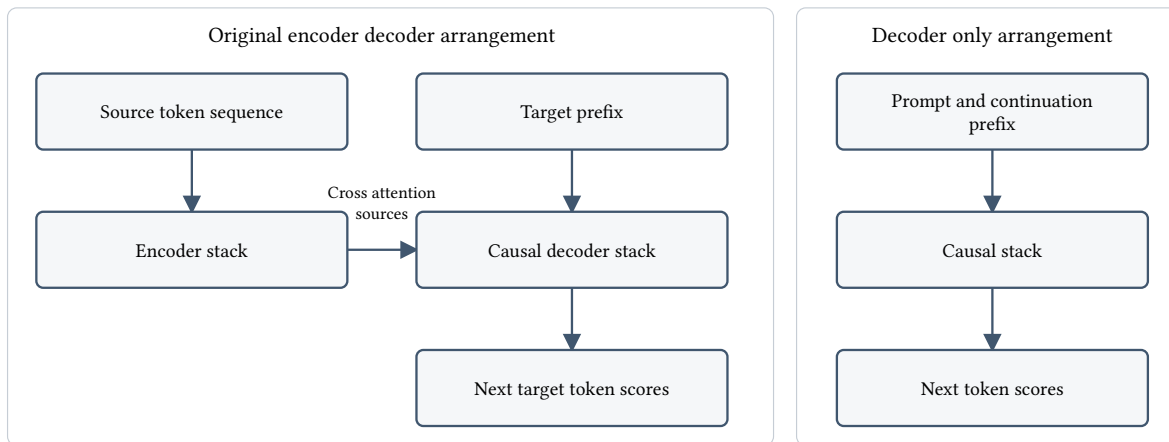
Side trail: complex-number notation for rotations, floating-point error analysis, and hardware memory bandwidth. The worked examples do not require them.

18.2 From two stacks to one causal sequence

The 2017 Transformer was presented as an encoder–decoder system for sequence transduction, particularly translation. Its encoder reads the source sequence with bidirectional self-attention. Its decoder has causal self-attention over the target prefix, cross-attention to the encoder output, and a position-wise feed-forward sublayer. Residual connections and LayerNorm surround these computations. The decoder is already autoregressive; generating one token at a time was not invented by later decoder-only models. Vaswani et al., sections 3.1–3.2

A conventional decoder-only language model removes the separate source encoder and the encoder–decoder cross-attention sublayers. It processes a single sequence containing the prompt and its continuation. Each position produces a representation used to predict the next token, with a causal mask preventing information from later positions from leaking backward.

“Decoder-only” therefore names an architectural arrangement. It does not mean that the model understands only outputs, nor that an encoder silently processes the prompt first. Prompt processing runs through the same causal stack. A model can learn translation by continuing a sequence containing a translation instruction and source text.



Original course diagram. Each stack contains multiple blocks; embeddings, normalization, and output details are omitted here.

The central components remain recognizable: learned token embeddings, residual states, attention, position-wise MLPs, and a vocabulary readout. Other choices vary independently. GPT-2 uses learned absolute position embeddings and pre-normalized blocks with LayerNorm. Its public implementation already includes past-key/value reuse. RoPE, RMSNorm, and grouped-query attention are not requirements for being a decoder-only Transformer. OpenAI GPT-2 implementation, pinned revision

Nor does the architecture alone establish conversational helpfulness, factual reliability, tool use, or a product's permissions. Those also depend on training, input formatting, decoding, and the surrounding system.

18.3 Giving attention information about position

An attention operation with no positional signals and no order-dependent mask is permutation-equivariant: reorder the input rows, and its output rows reorder correspondingly. A causal mask already makes order relevant by changing which sources are available. Nevertheless, a model also benefits from explicitly representing where tokens occur and how far apart they are. Do not confuse the mask's access restriction with a complete numerical encoding of distance.

The original paper adds fixed sinusoidal position vectors to token embeddings, and also evaluates learned position embeddings. GPT-2 adds a learned vector selected by the token's absolute position. These additions preserve the residual width. Original Transformer, section 3.5

Rotary position embedding, usually shortened to **RoPE**, instead applies position-dependent rotations to query and key coordinates. Its basic construction makes their dot product depend on relative displacement through those rotations. It is not a rotation of tokens into geographical locations or a learned table of one vector for every possible position. Su et al., RoFormer, section 3

18.3.1 An exact two-coordinate rotation

For this calculation, use individual **column vectors**, unlike the row-stacked attention arrays in Lesson 2.2. Define

$$R(\phi) = \begin{bmatrix} \cos \phi & -\sin \phi \\ \sin \phi & \cos \phi \end{bmatrix}, \quad \tilde{q}_i = R(i\omega)q, \quad \tilde{k}_j = R(j\omega)k.$$

Here R has shape 2×2 , and each vector has shape 2×1 . Choose the hand-constructed vectors

$$q = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \quad k = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \quad \omega = \pi/4.$$

This frequency makes the arithmetic exact and readable; it is not a claim about a particular checkpoint's frequency schedule. Before rotation, $q^T k = 1$.

Place the query at $i = 2$ and the key at $j = 0$. The query rotates by $\pi/2$, while the key is unchanged:

$$\tilde{q}_2 = \begin{bmatrix} -1 \\ 2 \end{bmatrix}, \quad \tilde{k}_0 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \quad \tilde{q}_2^T \tilde{k}_0 = -3.$$

The lengths remain $\sqrt{5}$ and $\sqrt{2}$. Rotation changes relative direction without changing either vector's norm. For a head of width two using the ordinary scaling, this pair contributes a score of $-3/\sqrt{2}$ before masking and softmax. One score alone is not an attention probability.

Now shift both positions forward by one while holding the unrotated vectors fixed. At $i = 3, j = 1$,

$$\tilde{q}_3 = \begin{bmatrix} -3/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}, \quad \tilde{k}_1 = \begin{bmatrix} \sqrt{2} \\ 0 \end{bmatrix}.$$

Their dot product is still -3 . The general reason is

$$\tilde{q}_i^T \tilde{k}_j = q^T R(i\omega)^T R(j\omega) k = q^T R((j-i)\omega) k.$$

The rotation term depends on $j - i$. This identity does not say that moving an actual passage always leaves the model's answer unchanged: its contextual representations and available sources can also change.

18.3.2 Extending the construction to a head

An even rotary width d_r contains $d_r/2$ coordinate pairs. Each pair rotates at its own frequency. A basic schedule uses $\omega_r = \theta^{-2r/d_r}$ for $r = 0, \dots, d_r/2 - 1$. The head's shape stays unchanged. Implementations can pair adjacent coordinates or corresponding coordinates from two halves; use the pairing expected by that implementation and checkpoint.

The rotary width need not always equal the full head width. Likewise, RoPE variants can rescale frequencies. In the Gemma specimen discussed here, the explicit head width is 256, and the reference implementation rotates query and key heads, leaving values unrotated. Different local and global frequency settings are part of its configuration. Google Gemma implementation

A formula can produce angles beyond a training sequence length. That mathematical fact does not establish reliable behavior at arbitrarily long contexts. Changing a frequency base or a context-limit field is not evidence that the model has learned to use the extended range.

18.4 A brief normalization checkpoint

Lesson 2.1 distinguished normalization type from its position in the block. LayerNorm subtracts the coordinate mean and divides by a stabilized standard deviation. RMSNorm divides by a stabilized root mean square without subtracting that mean:

$$\text{RMSNorm}(x)_a = \gamma_a \frac{x_a}{\sqrt{\frac{1}{d} \sum_b x_b^2 + \epsilon}}.$$

This common expression includes learned scaling. Specific implementations can parameterize that scaling differently. Neither normalization is a mechanism for recognizing which facts are relevant. Zhang and Sennrich, RMSNorm

“Uses RMSNorm” does not tell you where every normalization occurs. Gemma 3 also uses query/key normalization and normalization on sublayer outputs. Read the actual addition boundaries before drawing its residual block. A pre-normalized branch, a normalized branch output, and normalization of the residual sum are different operations. Gemma 3 report, section 2

18.5 Query heads and KV heads need not have equal counts

Ordinary multi-head attention gives each query head its own key and value head. **Multi-query attention**, or **MQA**, shares one key head and one value head across all query heads. **Grouped-query attention**, or **GQA**, shares a KV pair within each group of query heads. MQA is the one-group endpoint; ordinary multi-head attention is the endpoint with one group per query head. Ainslie et al., section 2.2

Let h_q be the query-head count and h_{kv} the count of key heads, also the count of value heads. With eight query heads:

Arrangement	h_q	h_{kv}	Query heads sharing each KV pair
Multi-head attention	8	8	1
Grouped-query attention	8	2	4
Multi-query attention	8	1	8

Sharing keys does not force identical attention distributions. Two queries can score the same keys differently. Sharing values also does not force identical head outputs: different coefficients produce different mixtures.

Assume equal query, key, and value head width d_h , residual width d , and evenly sized groups. In the row-vector convention,

$$\begin{aligned} W_Q &\in \mathbb{R}^{d \times (h_q d_h)}, \\ W_K, W_V &\in \mathbb{R}^{d \times (h_{kv} d_h)}, \\ W_O &\in \mathbb{R}^{(h_q d_h) \times d}. \end{aligned}$$

These are logical projections. An implementation may fuse Q, K, and V into one stored matrix or store matrix axes in the opposite order.

The KV projections become narrower when h_{kv} decreases. The query projections and number of query outputs need not shrink. Sharing changes representational capacity and storage; it is not merely a lossless file-compression setting. The original MQA work targets the memory traffic of incremental decoding. Real speed depends on the implementation and workload. Shazeer, Fast Transformer Decoding

Gemma 3 1B supplies a useful dimension check: $d = 1152$, $h_q = 4$, $h_{kv} = 1$, and $d_h = 256$. Thus its query projection outputs 1024 coordinates, its key projection 256, and its output projection maps 1024 back to 1152. Computing $1152/4 = 288$ would give the wrong head width. The explicit configuration takes precedence over a familiar shortcut. This specimen is the MQA endpoint of the GQA family. Pinned Google configuration, `get_config_for_1b`

18.6 Reusing previous work with a KV cache

Suppose a causal model has processed a prompt at positions 0 through 99. To process a new token at position 100, its attention needs earlier keys and values. With fixed weights and unchanged prefix, correctly computed earlier states do not acquire information from the new token: causality prevents that backward dependency.

A **KV cache** retains earlier keys and values, separately for each attention layer. During prompt **prefill**, the engine computes the prompt's states and populates the cache. During incremental **decoding**, it computes new states and reuses stored KV entries. It need not rerun the whole prefix through every layer for every new token. Transformers v4.51.3 caching explanation

Keep three different objects separate:

- W_K, W_V : learned parameters shared across token positions;
- K, V : activations calculated from a particular sequence;
- the cache: an inference data structure retaining those activations.

Adding a cache entry does not train the model. A different prompt generally needs different cached activations. Cache reuse also depends on the model, prefix, positions, masks, and computation being compatible. Editing an earlier token cannot safely leave all affected downstream prefix states untouched.

Past queries are usually unnecessary for ordinary next-token decoding: the current query reads past keys and values; old query results are not being recomputed. The new query still compares against permitted cached keys, so caching does not make global attention independent of context length.

Position bookkeeping matters. In RoPE implementations that store already-rotated keys, each cached key carries its original positional transformation. After processing 100 positions, the next logical position is 100, not zero. A rolling cache may overwrite a physical slot while logical positions continue increasing. Confusing slot number with logical position changes the computation. Padding and cache-position conventions are engine-specific; inspect both rather than copying a counter from a different API.

18.7 Context length and local attention describe different limits

A model's **context window** is a stated supported sequence length, subject to its implementation and training. For ordinary autoregressive use, prompt tokens and generated tokens consume the available

sequence budget together. Special tokens and other represented inputs may consume it too. A service can impose a smaller input or output cap.

A **sliding attention window** instead restricts which previous positions one layer can read directly. In a simple causal convention, a window of w includes the current position and at most $w - 1$ predecessors. Check whether a particular implementation counts the boundary the same way.

Local attention does not imply that all influence from older positions disappears after w tokens. A source representation can already contain information gathered by earlier layers. Multiple layers provide longer indirect paths, and a mixed architecture may also contain global layers. A possible path does not guarantee that a distant detail is retained or recovered accurately.

Gemma 3 interleaves five local layers and one global layer. Its 1B configuration specifies 26 layers, a 512-token local window, and 32,768 positions. Repeating that pattern gives four global layers and 22 local layers. The 1B variant is text-only; the larger family members' vision support and 128K context should not be copied onto it. Gemma 3 report; pinned 1B configuration

The architecture and storage policy are separate. An engine can discard no-longer-needed local KV entries, but it can also allocate full-length arrays and merely mask the inaccessible positions. Google's pinned reference generate implementation does the latter: it allocates the requested maximum sequence length in every layer. Therefore a local mask alone does not establish a small allocated cache. Reference allocation code

18.8 Turn cache dimensions into bytes

For conventional, uncompressed KV storage, assume:

- batch size B , with the same stored length S per sequence;
- L layers, each with h_{kv} KV heads;
- key and value width d_h ;
- b bytes per stored scalar;
- no KV sharing between sequences, layers, or devices.

Each layer stores two arrays with logical shape (B, h_{kv}, S, d_h) , regardless of physical axis order. Hence

$$M_{KV} = 2BLS h_{kv} d_h b \text{ bytes} .$$

The leading two counts keys and values. **Use KV heads, not query heads.** Kernels may temporarily expand shared KV heads for computation; that does not require the persistent cache to store those copies.

For the Gemma 1B dimensions, take $B = 1$, $L = 26$, $S = 32768$, $h_{kv} = 1$, $d_h = 256$, and $b = 2$. Full-length storage gives

$$2 \cdot 1 \cdot 26 \cdot 32768 \cdot 1 \cdot 256 \cdot 2 = 872,415,232 \text{ bytes} = 0.8125 \text{ GiB} .$$

Using four query heads in place of one KV head would overestimate this payload by a factor of four.

Now assume a different engine retains at most 512 entries in each local layer and the full prefix in each global layer. The steady-state retained payload becomes

$$\begin{aligned}
M_{KV} &= 2Bh_{kv}d_hb [L_gS + L_\ell \min(S, w)] \\
&= 1024 [4 \cdot 32768 + 22 \cdot 512] \\
&= 145,752,064 \text{ bytes} \approx 0.13574 \text{ GiB} .
\end{aligned}$$

This is an analytical estimate under the stated retention policy, not measured memory or the allocation of the reference generator. Chunked prefill, temporary tensors, alignment, reserved capacity, metadata, and device replication can change actual memory. Heterogeneous layers require summing their individual shapes. Doubling the independent batch doubles these payloads; it does not double the shared model weights.

18.9 Read configuration metadata without inventing runtime facts

Start by identifying the exact artifact and revision. This lesson uses GPT-2’s configuration snapshot and Google’s Gemma configuration factory. A factory contains defaults plus variant-specific overrides; it is not itself a trained checkpoint.

Meaning	GPT-2 configuration	Gemma 3 1B factory
Residual width	n_embd: 768	hidden_size: 1152
Block count	n_layer: 12	num_hidden_layers: 26
Query heads	n_head: 12	num_attention_heads: 4
KV heads	12, established by implementation	num_key_value_heads: 1
Head width	768/12 = 64, established by implementation	head_dim: 256
Position limit field	n_positions: 1024	max_position_embeddings: 32768

These references use different configuration schemas. The Hugging Face JSON uses vocab_size and n_positions; the historical OpenAI implementation uses n_vocab and n_ctx when constructing its embedding tables. We compare the architectural meanings, not claim that the JSON can be passed unchanged to that implementation. For this specimen the position-limit values agree at 1024.

A missing key is not permission to guess. Establish its default or interpretation from the matching model implementation. An intermediate_size describes an MLP width, not necessarily its parameter count: a gated MLP has more projections than a two-matrix MLP. Tied embedding/readout weights also affect parameter counting.

Likewise, dtype or torch_dtype metadata is not a measurement of every loaded tensor. Loading overrides, quantized modules, accumulators, and cache settings can differ. use_cache requests a behavior; it does not prove that a cache exists or occupies a particular amount of memory. A context-limit field does not certify long-context accuracy. Separate file facts, implementation facts, calculation assumptions, and runtime observations.

💡 Lab recommended here

Lab 08 — Read Model Configurations and Estimate Memory turns the two specimens into an evidence table and a memory worksheet. Its core uses public text configurations and arithmetic only: no model weights, account, API, GPU, or paid compute.

18.10 Precision changes representation and memory

FP32 stores each scalar in four bytes. FP16 and BF16 each use two, but they allocate bits differently. BF16 has a wider exponent range than FP16 and fewer fraction bits. Equal storage size does not imply equal rounding behavior. Storage, multiplication, and accumulation can use different formats; a low-precision weight file does not establish the precision of every intermediate operation. Google Cloud’s BF16 explanation

Quantization represents values with a smaller set of possible levels. A simple affine scheme stores an integer q and reconstructs an approximation $\hat{x} = s(q - z)$ using a scale s and zero point z . For example, rounding $x = 1.1$ onto multiples of 0.5 gives $q = 2$ and $\hat{x} = 1.0$. The lost 0.1 does not return merely because the result is later represented in FP32.

Real methods differ in grouping, calibration, treatment of outliers, and which tensors remain at higher precision. Post-training quantization and quantization-aware training are different procedures. Quality effects must be evaluated for the model, method, and task; “four-bit” is not a complete accuracy specification. GPTQ paper

For an original weight-payload calculation, take exactly $P = 1,000,000,000$ stored scalar parameters. This is a round hypothetical count, not a claim that the name “1B” specifies an exact checkpoint count:

Representation assumption	Payload bytes	Payload GiB
FP32, four bytes each	4,000,000,000	3.72529
FP16 or BF16, two bytes each	2,000,000,000	1.86265
Packed eight-bit values	1,000,000,000	0.93132
Packed four-bit values	500,000,000	0.46566

For four-bit storage, $P \times 4/8 = 500,000,000$ bytes before overhead. Quantization scales, zero points or other metadata, unquantized tensors, and padding must be added. Weight quantization does not automatically quantize the KV cache. Transformers quantization documentation

An inference budget must also include cache, activations, temporary workspace, and runtime overhead. Training adds other large categories, including gradients and optimizer state. A weight-payload estimate is therefore not a promise that a machine of that capacity can run the model.

Smaller representations can reduce memory traffic, but conversion work, kernel support, hardware, batch size, and context length affect speed. Cache quantization can even increase latency in some conditions. Measure an explicitly defined workload rather than translating a fourfold storage reduction into a fourfold speedup. Transformers cache strategies

18.11 Check your understanding

1. Does decoder-only imply RoPE, RMSNorm, or no processing of the prompt?
2. In the rotation example, why did shifting both positions preserve the dot product? Would shifting only the query generally preserve it?
3. A model has 16 query heads and four KV heads. How many query heads share each KV pair? Which count enters the persistent-cache formula?
4. Does a 512-token local window imply a 512-token total context limit?
5. What is missing from the claim “its weights take two GB, so it fits in two GB of memory”?
6. A configuration says BF16. What further evidence would establish that the running cache is BF16?

i Answers and reasoning

1. No. These are separate architectural choices, and the causal stack processes the prompt too.
2. The displacement $j - i$ stayed fixed while the unrotated vectors were held fixed. Moving only one position generally changes the relative rotation.
3. Four queries per KV pair; four KV heads enter the cache formula.
4. No. Local visibility is per layer; global layers or multi-layer paths can support longer dependencies. Actual supported length remains a separate question.
5. Units, exact parameter count and storage format, cache, activations, workspace, overhead, and the workload. Quantized formats also require accounting for metadata and higher-precision tensors.
6. Inspect the loaded cache tensors or the verified engine allocation path, including overrides. Configuration metadata alone is insufficient.

18.12 More Learning

1. **Attention Is All You Need — Vaswani et al.** Revisit the two stacks and identify the cross-attention path removed in a conventional decoder-only design.
2. **RoFormer — Su et al.** Read section 3.2 alongside the two-coordinate example; track the transpose and relative-position sign.
3. **GQA — Ainslie et al.** Compare the MHA, GQA, and MQA head-sharing arrangements before reading performance claims.
4. **Root Mean Square Layer Normalization — Zhang and Sennrich** Separate the normalization operation from where a model places it.
5. **Gemma 3 Technical Report** Compare the family-level description with the linked 1B configuration. Preserve variant and memory-unit distinctions.
6. **Transformers caching explanation, v4.51.3** Trace prefill, incremental updates, attention masks, and logical positions. The pinned version is a concrete reference, not a recommendation to install it.

The next lesson examines Mixture of Experts, where selecting which MLP parameters participate in a token’s computation creates another distinction between total model size and active computation.

19 2.5 — Mixture of Experts

19.1 Choosing which computation to run

A Transformer need not apply the same feed-forward network to every token. It can contain several candidate networks and use a learned **router** to select a few for each position. These networks are conventionally called **experts**. Their outputs are combined into the update written to the residual stream.

This is the central idea of a sparsely routed **mixture of experts**, or MoE. It separates two quantities that are easy to confuse: how many parameters the model contains and how many participate in processing one token. That separation can increase available model capacity without increasing expert computation in direct proportion. It also creates routing, memory, training, and scheduling problems.

Lessons 2.1–2.4 give us the necessary pieces: the residual stream, attention, position-wise MLPs, and architecture-specific configuration. Here we replace a dense MLP branch with a routed one. We will compute a tiny example completely, then inspect the documented design of gpt-oss.

By the end, you should be able to:

- trace router scores through selection, weighting, expert evaluation, and residual addition;
- distinguish full-router probabilities from the coefficients actually used;
- interpret total and active parameter counts without treating either as a memory measurement;
- explain why expert IDs do not come with human job descriptions;
- separate internal token routing from choosing a model in an application;
- design a routing log and a controlled intervention with defensible conclusions.

Math to know / refresh

Needed now: affine maps, softmax, ranking, and weighted vector sums. Remember $e^{\ln a} = a$, which will make the example exact.

Useful refresh: conditional probability, matrix dimensions, and the difference between stored parameters and temporary activations.

Side trail: gradients through conditional computation, distributed matrix multiplication, and routing alternatives. The main calculation needs no calculus.

19.2 Dense and routed feed-forward branches

For one position, let $u \in \mathbb{R}^d$ be the vector supplied to a feed-forward branch. A familiar dense MLP computes

$$f(u) = W_{\text{down}} \phi(W_{\text{up}} u + b_{\text{up}}) + b_{\text{down}}.$$

A gated MLP changes this internal formula, as discussed in Lesson 2.3. Either way, a conventional dense branch uses its same parameter arrays for each position. Some activations might be zero. That alone does not mean that the runtime skipped the corresponding matrix operations.

A routed branch instead stores E expert functions,

$$f_0(u), f_1(u), \dots, f_{E-1}(u),$$

and evaluates a selected subset. Usually each expert has the same input and output widths but its own weights. The term *sparse* here concerns which experts are executed for a token. It does not require every expert's weight matrix to contain many zero entries.

The experts are components inside a model. They need not be independently usable language models, and they do not exchange written messages. A selected expert returns a vector of width d , rather than an answer to the user. This architectural use of MoE has a long history: the sparse layer studied by Shazeer and colleagues was inserted into recurrent language models, before its widespread use in Transformers. Shazeer et al., Section 2

For an illustrative sequential pre-norm Transformer block, write

$$\begin{aligned} a &= x + \text{Attention}(\text{Norm}_1(x)), \\ u &= \text{Norm}_2(a), \quad y = a + \text{MoE}(u). \end{aligned}$$

The attention notation suppresses the other positions it can read. The router and expert equations below concern one position. Here a is the residual vector after attention. Both the router and the experts read u ; their combined update is added to a . The normalization operation and its placement belong to the architecture. Post-norm, parallel branches, shared experts, and other arrangements require different diagrams. MoE does not impose one universal normalization order.

19.3 From router logits to a residual update

For a simple affine router,

$$z = W_r u + b_r, \quad W_r \in \mathbb{R}^{E \times d}, \quad z \in \mathbb{R}^E.$$

The entry z_e is expert e 's **router logit**. These are not vocabulary logits. A softmax across experts would give

$$p_e = \frac{e^{z_e}}{\sum_{j=0}^{E-1} e^{z_j}}.$$

The numbers are positive and sum to one. They provide a distribution over expert IDs for this input at this layer. They are neither a probability that the model's answer is correct nor a measured probability that an expert understands a topic.

Choose the indices of the k highest scores:

$$S = \text{TopKIndices}(z, k).$$

Softmax preserves score order, so selecting by p gives the same set in exact arithmetic, provided ties are resolved identically. Top- k selection is usually deterministic in the forward rule considered here. The presence of probabilities does not mean that the runtime samples experts. Training noise or other routing schemes must be specified separately.

For this lesson, **renormalize over the selected experts**:

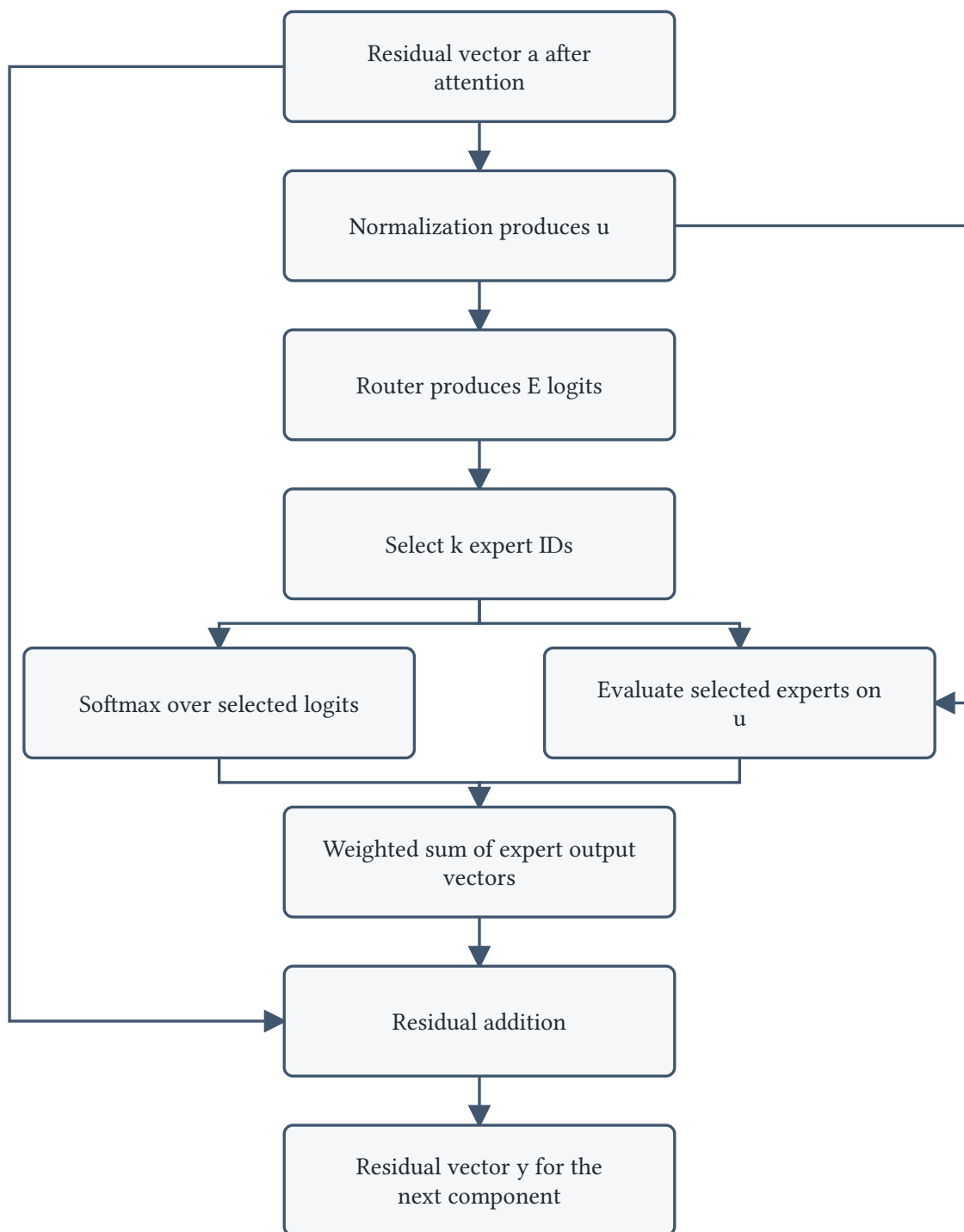
$$\alpha_e = \begin{cases} \frac{e^{z_e}}{\sum_{j \in S} e^{z_j}} = \frac{p_e}{\sum_{j \in S} p_j}, & e \in S, \\ 0, & e \notin S. \end{cases}$$

Thus p is the full softmax distribution, while α is the sparse combining vector. Computing a full softmax and renormalizing its selected entries is mathematically equivalent to applying softmax directly to the selected logits. A runtime can use the latter procedure without constructing p .

Now compute the expert outputs and combine them:

$$m = \text{MoE}(u) = \sum_{e \in S} \alpha_e f_e(u), \quad y = a + m.$$

The scalar α_e scales every coordinate of expert e 's output. The weighted sum returns to width d , so it can be added to the residual vector. Averaging expert IDs, concatenating their outputs without a projection, or summing their logits would compute something else.



Original course diagram. The router determines both which expert functions run and how their output vectors are weighted. The bypass carries the unnormalized residual vector in this pre-norm example.

Not every MoE uses this exact weighting rule. In particular, keeping selected entries of a full softmax **without** renormalization gives a different update. A top-1 selected-only softmax produces coefficient

one; Switch Transformer instead retains the winning full-softmax probability. Read the forward rule rather than inferring it from “top- k .” Fedus et al., Section 2.1

19.4 A complete four-expert calculation

Consider a hand-constructed branch with $d = 2$, $E = 4$, and $k = 2$. To isolate routing, omit normalization and attention: the branch input and residual vector are both

$$u = a = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

Use an affine router with zero bias and

$$W_r = \begin{bmatrix} \ln 4 & 0 \\ 0 & \ln 3 \\ \ln 2 & 0 \\ 0 & 0 \end{bmatrix}.$$

Define four linear expert functions:

$$f_0(u) = \begin{bmatrix} 2u_1 \\ 0 \end{bmatrix}, \quad f_1(u) = \begin{bmatrix} 0 \\ 3u_2 \end{bmatrix}, \quad f_2(u) = \begin{bmatrix} -u_1 \\ u_2 \end{bmatrix}, \quad f_3(u) = \begin{bmatrix} u_2 \\ u_1 \end{bmatrix}.$$

These deliberately simple functions replace full nonlinear MLPs. They are invented arithmetic objects, not trained experts or simplified measurements from gpt-oss. Coordinates are numbered 1 and 2 in the equations; experts are numbered 0 through 3. If logits tie, prefer the smaller expert ID. That tie rule is part of this toy’s specification, not a claim about every library.

The router calculation gives

$$z = [\ln 4, \ln 3, \ln 2, 0]^T,$$

so exponentiation yields $[4, 3, 2, 1]$ and

$$p = [4/10, 3/10, 2/10, 1/10]^T.$$

Experts 0 and 1 rank highest. Their original probability mass totals $7/10$, so the selected-only coefficients are

$$\alpha = [4/7, 3/7, 0, 0]^T.$$

Only the selected expert outputs are needed:

$$f_0(u) = [2, 0]^T, \quad f_1(u) = [0, 3]^T.$$

Therefore

$$m = \frac{4}{7} \begin{bmatrix} 2 \\ 0 \end{bmatrix} + \frac{3}{7} \begin{bmatrix} 0 \\ 3 \end{bmatrix} = \begin{bmatrix} 8/7 \\ 9/7 \end{bmatrix}, \quad y = \begin{bmatrix} 15/7 \\ 16/7 \end{bmatrix}.$$

The larger gate belongs to expert 0, yet expert 1 makes the larger contribution norm here: $9/7$ rather than $8/7$. A routing coefficient and an output contribution are different measurements. Even

contribution magnitude would not establish downstream importance, because later computations can amplify, suppress, or cancel directions.

Suppose we accidentally retain $4/10$ and $3/10$ as the combining coefficients. Then $m = [4/5, 9/10]^T$ and $y = [9/5, 19/10]^T$. Selection is unchanged, but the update has been scaled down. A log containing only the selected IDs would fail to expose this error.

💡 Predict before continuing

Keep u and every expert fixed. Predict the update if you:

1. add the same constant to all four router logits;
2. increase expert 3's logit from 0 to $\ln(3/2)$;
3. prohibit expert 0, choose the best two remaining experts, and renormalize;
4. keep the original route and weights but replace expert 0's output with zero.

The first two leave m unchanged under our rule. In the second case, the full distribution p changes, but the selected logits and their normalized coefficients do not. The third gives $m = [-2/5, 11/5]^T$; the fourth gives $m = [0, 9/7]^T$. Prohibiting a route and zeroing a contribution are different interventions.

! Lab recommended here

Begin Trace and Perturb Expert Routing. Compute the toy router by hand, then extend the trace across positions and two small layers. The core needs no trained model, account, API, GPU, or paid compute. Return to its configuration inspection after the next section.

19.5 The gpt-oss specimen

Our concrete specimen is **OpenAI's gpt-oss-20b release**, with gpt-oss-120b as a comparison. These are fixed documented designs, not recommendations about whichever model is newest.

The model card, version 1, Table 1 and Section 2.2 reports:

Model	Layers	Experts per MoE layer	Selected per token per layer	Total parameters	Active parameters
gpt-oss-20b	24	32	4	20.91 billion	3.61 billion
gpt-oss-120b	36	128	4	116.83 billion	5.13 billion

Counts in billions are the card's rounded figures. Its active-count convention includes unembedding parameters but excludes embeddings. Both models use pre-branch RMSNorm and selected-only softmax weighting. The count of experts is **per layer**, not one shared bank for the entire network.

The 20b configuration pinned to revision f81fef1ddd90d214968e951a76834f1ded130a18 records `hidden_size: 2880`, `intermediate_size: 2880`, `num_hidden_layers: 24`, `num_local_experts: 32`, and `num_experts_per_tok: 4`. It also contains `experts_per_token: 4`. Names depend on the configuration format; do not silently substitute a familiar field from another model.

Read that JSON beside OpenAI’s reference MLPBlock, pinned to commit 243a1b02767da73bd2e3975be250afa801635866. The forward method normalizes the input, projects router logits, selects experts, applies softmax to their selected logits, evaluates the corresponding expert parameters, combines their outputs, and adds the residual. Its expert nonlinearity is a particular clamped SwiGLU variant. Our linear toy reproduces the routing arithmetic, not that expert computation.

A configuration describes dimensions and settings. It does not contain learned router weights, expert weights, or the activations for your prompt. Reading `output_router_logits: false` tells you a stored default, not whether a particular service can expose routing. The reference source establishes a computation; tracing an actual trained run additionally requires compatible weights and an instrumented runtime.

19.6 Count parameters before estimating memory

Suppose an invented network contains P_s always-used parameters and L routed layers, each with E distinct equal-sized experts of P_e parameters. Put router parameters in P_s . Under a deliberately simplified accounting convention,

$$P_{\text{total}} = P_s + LE P_e, \quad P_{\text{active}} = P_s + Lk P_e.$$

For $P_s = 100$, $L = 2$, $E = 4$, $P_e = 10$, and $k = 2$, the counts are 180 total and 140 active. Multiplying the total by k/E would incorrectly give 90, because it would sparsify the always-used part too. Embedding lookups and tied weights require explicit counting conventions in real networks.

A gated expert with three weight matrices of sizes $m \times d$, $m \times d$, and $d \times m$ has $3dm$ matrix coefficients. Ignoring biases and all other components, the 20b dimensions imply

$$24 \times 32 \times 3 \times 2880^2 = 19,110,297,600$$

stored expert-matrix coefficients, versus

$$24 \times 4 \times 3 \times 2880^2 = 2,388,787,200$$

used along one token’s selected expert path. These are our dimension-based calculations, not a replacement for the whole-model counts. Attention, routers, normalization, biases, and the output projection still matter.

The other distinction is **memory residency**. Unselected experts do not disappear. A runtime must have their weights available when subsequent tokens need them. They may all reside on one accelerator, be distributed across devices, or be moved from another memory tier. Different tokens in a batch can collectively activate many more than k experts in a layer.

A raw weight-storage estimate uses the stored parameter arrays and their representations. A runtime estimate must additionally include quantization metadata, caches, activations, temporary buffers, allo-

cation overhead, and possibly duplicated or converted weights. Training adds gradients and optimizer state. Neither estimate is simply “active parameters times bytes per number.”

For example, the gpt-oss card reports quantized checkpoints of 12.8 GiB for 20b and 60.8 GiB for 120b. Checkpoint size does not guarantee runtime memory fit. Model card, Table 1 and Section 2.1

Sparse expert execution can reduce arithmetic relative to executing every expert, but arithmetic is not latency. Routing, token dispatch, memory movement, small expert batches, and cross-device communication all consume time. Actual speed needs a measured workload and runtime.

19.7 A fresh route at each position and layer

A router ordinarily reads the current hidden representation at a particular layer. Earlier attention may already have mixed information from other positions into it. Consequently, two occurrences of the same token ID can follow different routes when their contexts differ.

A token can also choose different experts at different depths. Expert 7 in layer 2 is not automatically the same function as expert 7 in layer 17. Treat the pair (**layer, expert ID**) as the component’s address. Changing one early route can alter later hidden states and therefore later routing decisions.

In a decoder model, routing occurs while processing prompt positions and while processing generated positions. It is not reserved for the visible answer. A run log should identify sequence position and whether the measurement belongs to prompt processing or a generation step. With caching, the runtime need not recompute every old position at every generation step.

Now return to the course’s recurring question: where does this behavior live? The router’s learned projection lives in model weights; the selection and combination rules live in its architecture and implementation; the vector being routed depends on the input context and previous computation. An application choosing between two whole language models is a separate, outer routing decision. A product’s “automatic model” option tells us nothing by itself about whether either chosen model contains MoE layers.

19.8 Learning routes and keeping experts useful

During training, the model’s objective supplies feedback to the parameters involved in its computation. Selected expert functions can receive gradients through their outputs. The router can receive gradients through the selected combining coefficients, within regions where the selected set remains unchanged. The discrete membership decision itself is not an ordinary smooth operation.

This creates a useful boundary case. For selected-only normalization with $k = 1$, the sole coefficient is identically one. Its local derivative with respect to the selected logit is zero. Training a router in that situation needs another learning signal or a different gate rule; one cannot assume that every top-1 scheme trains exactly like our top-2 formula. Shazeer and colleagues discuss differentiable gate values for $k > 1$ and use additional routing mechanisms in their system. Shazeer et al., Section 2.1

Ordinary inference uses the learned parameters without optimizing them. Selecting an expert is not retraining it, and a different route is not evidence that new knowledge was permanently stored. Module 3 will develop the training process more fully.

Routing also determines how much work reaches each expert. Imagine twelve token representations, each selecting two experts from a bank of four. There are 24 assignments. Counts of $[6, 6, 6, 6]$ and $[12, 12, 0, 0]$ obey the same top-2 rule, yet distribute computation and training examples very differently.

Assignment counts measure work allocations. Summed probabilities or combining coefficients measure a different quantity: a selected expert with a small coefficient still has to run. Keep those measurements separate.

Load balancing concerns the distribution of work. Some systems use auxiliary training losses to discourage concentrated routing; the exact loss, measurement unit, and trade-off vary. Switch Transformer supplies a concrete example with an auxiliary balancing loss and a fixed per-expert capacity. When its capacity overflows, a token can skip the expert computation while continuing through the residual path. “Dropped” in this setting does not mean deleting the token from the input text. Fedus et al., Section 2.2

Capacity limits and dropping are not defining requirements of MoE. MegaBlocks develops block-sparse computation that accommodates uneven assignment counts without dropping tokens. This is an implementation alternative to a particular capacity trade-off, not proof that routing imbalance has no cost. Gale et al., Sections 2–4

19.9 What makes an expert an expert

The name is suggestive, but no step in our router equation assigns a network the job “mathematician,” “poet,” or “fact checker.” Expert parameters and routing patterns are learned under the training setup. The architecture alone guarantees neither predefined roles nor a neat division into semantic domains.

Empirical specialization is a question to investigate. In its analysis of selected layers and datasets, the Mixtral paper found little clear topic-based separation and described routing patterns associated with syntax and token locality. That is evidence about the studied model and samples, not a theorem that experts can never specialize by topic. Jiang et al., Section 5

Suppose a trace selects one expert frequently in code prompts. Plausible explanations include punctuation, indentation, token identity, sequence position, or a broader feature of the hidden representation. Calling it “the coding expert” outruns the measurement. Compare matched inputs, distinguish layers, inspect output contributions, and test held-out cases before making a stronger claim.

Interventions require equal care. Prohibiting an expert, changing a logit, replacing an output with zero, and forcing an arbitrary expert set alter different computations. Forced routing can send a representation through a function that was rarely trained on such inputs. A performance drop can demonstrate sensitivity to the intervention without revealing a clean human-readable function. Use a baseline, state the exact intervention boundary, and separate immediate effects from downstream changes.

19.10 Check your understanding

Answer without looking back, then check the reasoning below.

1. What are the shapes of W_r , z , an expert output, and the combined update?
2. Why can two implementations select identical experts but produce different residual updates?
3. In the toy example, force the set $\{2, 3\}$ and renormalize their original logits. What are m and y ?

4. Does top-4 routing imply that only four expert parameter sets are stored, or that a batch uses only four?
5. What does the gpt-oss-20b configuration establish that a routing trace does not, and vice versa?
6. Why is a routing-frequency plot insufficient to name an expert's semantic role?
7. With k fixed, does doubling E necessarily double per-token expert arithmetic? What still increases or might become harder?

i Answers and reasoning

1. W_r is $E \times d$, z has E entries, and both output vectors have d entries. We use column vectors for individual positions.
2. They might use different combining coefficients, expert outputs, normalization, or residual boundaries. Selected IDs alone do not specify the computation.
3. The coefficients are $2/3$ and $1/3$. Thus $m = [-1/3, 1]^T$ and $y = [2/3, 2]^T$.
4. Neither. Selection is per token per routed layer. Other expert weights remain part of the model, and other tokens can select them.
5. The config states architectural dimensions and stored settings. A correctly instrumented trace establishes actual selected routes and values for one specified execution. Neither alone explains why a trained expert performs its function.
6. Frequency establishes an association with sampled inputs. Tokenization, position, and other features can explain it; a functional claim needs better controls and evidence.
7. Not for fixed-size experts with the same k . Stored expert parameters increase, the router has more candidates to score, and memory movement, dispatch, or load balancing can become harder. Whole-runtime cost is not fixed by the expert arithmetic alone.

19.11 More Learning

1. **gpt-oss Model Card — OpenAI, version 1.** Read Table 1 and Section 2.2. Check the active-parameter footnote before comparing the two model names.
2. **gpt-oss reference forward computation — OpenAI, pinned commit.** Match each operation in `MLPBlock.forward` to this lesson's diagram. Notice where the residual addition occurs.
3. **Outrageously Large Neural Networks — Shazeer et al.** Read Section 2 for sparse gating and Section 4 for balancing. Separate the paper's noisy training rule from our deterministic toy.
4. **Switch Transformers — Fedus et al.** Compare its top-1 gate, capacity handling, and balancing objective with our selected-only top-2 rule.
5. **Mixtral of Experts — Jiang et al.** Read Section 5 as an example of testing a specialization hypothesis. Identify the sampled layers and datasets before generalizing.
6. **MegaBlocks — Gale et al.** Optional systems reading. Trace how grouping and ungrouping token representations connects the mathematical mixture to efficient execution.

We have now assembled the main architectural pieces. Module 3 asks how training selects the weights that make these computations useful.

20 Lab 01 Map AI Systems

20.1 Purpose

A system can be neural, generative, multimodal, and part of an agent at the same time. Those labels answer different questions. This Lab turns the map from Lesson 1.1, The AI Zoo into a practical method: identify the mechanism, locate the behavior, and say what the evidence supports.

You will produce six short system analyses and three intervention predictions. The systems are fictional specifications, not descriptions of commercial products. There is no test of whether something deserves the word “AI.” The test is whether your explanation follows from its stated construction.

Time: 45–60 minutes: orientation and examples, 10; cards, 20–25; interventions, 5–10; self-check and revision, 10–15.

Requirements: paper or any text editor. No coding, installation, account, paid service, or internet connection is required once you have this Lab. Basic arithmetic is sufficient. Reading Lesson 1.1 first is helpful; the vocabulary below makes the activity standalone.

20.2 The questions to keep separate

Use several axes rather than selecting one winning category.

- **Mechanism:** explicit rules; search over candidate solutions; classical, non-neural machine learning; neural computation. A system may combine them.
- **Learning:** were parameters fitted from examples or experience? Supervised learning uses supplied targets; self-supervised learning derives targets from the data itself; reinforcement learning uses rewards associated with actions and their consequences. A reward-trained system need not be neural.
- **Task:** classification assigns a category; prediction estimates something; search selects a solution; generation constructs content. Producing a sentence from a fixed template does not establish a learned generative model.
- **Modality:** what kinds of information enter and leave? Text, images, and audio are examples. Identify the boundary: an entire product can connect modalities even when one component handles only text.
- **Action loop:** does the system choose actions, receive results, and choose again toward a goal? Call this an agent loop here. A fixed sequence of model calls is orchestration, but is not sufficient evidence of this loop. Other definitions of “agent” exist; state yours if it changes a label.

Also distinguish the six layers from the lesson:

1. **Algorithm:** a procedure, such as searching paths or updating parameters.
2. **Architecture:** the structural design of a model, such as a convolutional network or Transformer. Ordinary software also has architecture; specify which meaning you intend.
3. **Trained parameters:** fitted values, including neural weights, regression coefficients, or a learned table. Architecture alone does not specify these values.
4. **Inference runtime:** software that loads and executes a fitted model. Its implementation and numerical settings can matter even when parameters stay fixed. A card may leave these details unspecified.
5. **Orchestration:** software that passes inputs, calls components, applies controls, or repeats a loop.

6. **Product:** the complete user-facing application, including its interface.

Not every system has all six. Prompts, input data, and decoding settings are additional possible locations of behavior; do not force them into “weights.”

20.3 Two worked examples

Read each specification and make a quick classification before reading its explanation.

20.3.1 Example 1 A polished help window

A window matches a user’s typed keyword against a handwritten list, then displays a stored answer. Unmatched inputs display “Please contact support.” Nothing is fitted from data.

Predict: Is the displayed prose evidence of a language model? Where would you change the fallback?

Explanation: This is rule-based lookup with text input and output. Its algorithm matches keywords; the list and answers are manually authored data. There are no trained weights or neural architecture in this specification. The window is the product interface. Changing the fallback changes software or stored content, not model training. Fluent output alone would not reveal this implementation.

20.3.2 Example 2 One text model call

A command-line application loads a Transformer checkpoint trained on raw text to predict each next piece from preceding text in the same document. It receives a prompt, generates one completion, and exits. There are no tools or subsequent calls.

Predict: Which labels overlap, and does the application have an agent loop?

Explanation: The model is neural, generative, and text-only; its described training is self-supervised. Transformer names the architecture; the checkpoint supplies trained weights; token selection during generation is an algorithmic step. The application supplies the prompt and invokes generation. The specified one-call workflow lacks our action-result-action loop. Changing the prompt can change the completion without changing any weight.

20.4 Your worksheet

Make six copies of this compact record, one for each card. Short phrases are fine, but every important label needs evidence.

```
Card ID and system boundary:
Initial prediction:
  Mechanism(s):
  Learning method, or no learning:
  Task; input and output modalities:
  Agent loop: yes / no / not established, because...
Evidence: one or two exact details from the card:
Layers: algorithm; architecture; trained parameters; inference runtime;
        orchestration; product (use absent/unspecified when justified).
Uncertainty: one unsupported claim and evidence that would settle it:
After self-check: keep/change which claim, and why?
```

Complete all initial predictions before opening the answer key. “Not established” means evidence is missing; “no” needs evidence against the claim. Treat explicit “only” and “no” statements as constraints. Do not invent hidden models. Equally, do not fill unspecified training details with a familiar implementation.

If working with an LLM, make your first attempt independently. Then ask it to challenge the evidence for one label without replacing your entire worksheet. Record any assistance in your final artifact.

20.5 Six system cards

20.5.1 A Greenhouse alarm

A dashboard receives temperature and humidity numbers. Its entire decision procedure is: if temperature is above 28°C and humidity is below 40%, display “Water check needed”; otherwise display “No alert.” Staff wrote the thresholds and messages. There is no training, search, or learned component. The dashboard does not operate watering equipment. Its next update simply evaluates the rule on the next pair of readings.

20.5.2 B Route desk

A route-planning application receives a start and destination in a manually entered directed graph. It exhaustively enumerates the valid routes, sums their edge costs, and displays a least-cost route. There are only two routes from S to T: S–A–T costs $3 + 4$; S–B–T costs $2 + 7$. Costs are fixed numbers, and no parameters are learned. The application plans a route once; it neither drives a vehicle nor observes a journey.

20.5.3 C Sorting gate

A recycling dashboard uses logistic regression fitted to examples of two numeric sensor readings paired with human labels, “metal” or “other.” Its learned coefficients and intercept produce a metal probability. Software displays “metal” when that probability is at least 0.60; otherwise it displays “other.” For item X the probability is 0.72. Deployment loads fixed parameters, with no updates, neural components, or search. The dashboard cannot move the physical sorting gate.

20.5.4 D Surface inspector

An inspection application resizes a photograph to a fixed size, passes it through a convolutional neural network, and displays a crack probability and “crack” or “clear.” The network has four convolutional layers followed by a classification layer. Its weights were fitted to photographs with human-assigned labels. There are no other models. During use, the weights remain fixed. The application receives images only; it does not accept questions, generate new photographs, or control equipment.

20.5.5 E Illustrated request

A drawing application receives a text request. A Transformer language model generates a structured text instruction, which software passes to a separate neural image generator. The image generator was trained using paired text and images; its exact training objective is unspecified. Software then adds a fixed watermark and shows the image. These three stages always occur once in that order, with no evaluation or retry loop. The language model never receives the generated image. Its own training history is unspecified.

20.5.6 F Warehouse rover

A simulated rover observes its grid cell and battery level, chooses a movement, receives the resulting state, and repeats until it reaches its assigned destination. During training, a tabular reinforcement-learning algorithm updated state-action values using rewards for arrival and penalties for wasted moves. Deployment freezes that learned table. The rover chooses the highest-valued allowed action, using a fixed tie-break. A handwritten safety rule excludes moves into reserved cells. There is no neural network, language model, or media generator.

20.6 Predict three interventions

Before self-checking, answer each with: **changed component** → **expected effect** → **what remains unchanged** → **evidence or uncertainty**. These are deductions from specifications, not measurements from running software.

1. **A and B:** For A, readings are 29°C and 35%. Raise only the temperature threshold from 28°C to 30°C. For B, remove edge A–T, retaining the other route. Predict each new output. Did either change require training?
2. **C and D:** Raise C's display threshold from 0.60 to 0.80 for item X. Separately, change only D's resizing method while keeping its output dimensions fixed. Which outcome can you determine exactly? Can either change affect displayed predictions without changing weights? Explain why D's accuracy direction is unknown.
3. **F:** Change the reward for wasted moves in the training configuration, but do not retrain or change anything deployed. Hold observations, allowed actions, and tie-breaking fixed. Will the deployed next action change? How would actually resuming learning alter your conclusion?

20.7 Your artifact and review

Save one document named `lab-01-map-ai-systems-response` in any format you can reopen, or keep numbered paper pages. Include the six records, three intervention answers, and a final 100–150-word explanation of why “the product changed” does not necessarily mean “the model was retrained.” Preserve original predictions and append revisions instead of overwriting them.

Record the Lab title and date, any assistance, and any assumptions beyond the cards. This makes another reader able to reconstruct your reasoning. You may share the artifact with a teacher, study partner, or chosen LLM for critique. No submission to the course website is needed; sharing and publication are optional.

20.8 Self-check answer key

Stop here until your predictions are recorded. These explanations are reference reasoning, not invented learner submissions. Compare evidence and boundaries, not exact wording.

20.8.1 Card classifications and layers

A: Rule-based decision logic; no learning; numeric inputs and fixed text output; no goal-directed action loop. The threshold comparison is the algorithm, the dashboard is the product, and display/update code is orchestration. There is no neural architecture or trained parameter set. Sensor accuracy is unknown; calibration evidence would be needed. Manual threshold authorship is established.

B: Search and optimization without learning. Enumeration and minimum-cost selection are algorithms; graph edges are problem data, not learned weights. The app supplies inputs and displays a route. A planner is a component that could be placed inside an agent; this one-shot product has no execution-feedback loop. “Symbolic search” is also a defensible mechanism label.

C: Classical supervised learning for classification. Logistic regression specifies the model form; coefficients and intercept are trained parameters, even though they are not neural weights. Probability computation and the threshold rule are distinct steps. The latter lives in software. Numeric inputs and a displayed label do not imply a language model. No active gate control or agent loop is specified.

D: Neural supervised classification, with image input and probability/label output. Layer organization is architecture; fitted tensors are weights; resizing and display are surrounding software. Returning a textual label does not make the model a language generator. Its exact training optimizer is unknown. Code or training records would establish it; the visible prediction would not.

E: A neural generative pipeline and a product spanning text and images. The Transformer architecture and two trained model parameter sets are distinct from the fixed orchestration and watermark code. “Multimodal system” is justified; “the language model can see” is not. A text-conditioned image model crosses modalities, but its exact architecture and training objective remain unknown. Paired data alone does not settle the objective. There is no agent loop under this Lab’s convention.

F: Non-neural reinforcement learning combined with explicit safety rules and an agent loop. The table contains learned values; action selection and training updates are algorithms. The safety filter and repeated observation-action cycle belong to orchestration. Calling it a reactive agent is reasonable; claiming that it plans ahead is unsupported. A broader use of “weights” for learned parameters is acceptable if you explicitly distinguish the table from neural weights.

The cards do not identify particular inference runtimes or numerical settings. For C, D, and E, model execution is present but its runtime details are unspecified. F executes a learned table lookup rather than a neural inference engine; A and B have no fitted model to run. Do not infer a library or hardware platform from these descriptions.

20.8.2 Intervention answers

1. **A:** “No alert,” because 29 is not above 30. **B:** S–B–T, cost 9. Rules or problem data changed; neither system learned.
2. **C:** Item X switches to “other”; its probability remains 0.72. **D:** Changed preprocessing may change network inputs and predictions with identical weights. The effect may also be zero for some images. Accuracy could improve, worsen, or stay the same; labeled evaluation examples and both resizing procedures are needed to measure it.
3. **F:** The next deployed action stays the same. An unused training configuration cannot change a frozen deployed table. Resuming learning permits table changes, but neither an immediate action change nor improved performance is guaranteed.

20.8.3 Check the quality of your reasoning

Your artifact is ready when you can:

- support classifications with construction details rather than names or fluent output;

- identify overlapping axes and distinguish component from product boundaries;
- locate a behavior in the appropriate layer;
- distinguish absent components from missing evidence;
- predict a controlled change without claiming an unmeasured result;
- explain at least one correction, or defend an unchanged prediction with evidence.

If one criterion fails, revise that part and name the missing evidence. A well-supported “unknown” is a successful answer.

21 Lab 02 — Inspect Tokenization

21.1 Goal and prerequisites

Inspect a real tokenizer and explain what its output does, and does not, tell you about a language model. Compare ordinary prose, code, numbers, whitespace, and Unicode while changing one feature at a time.

Read Lesson 1.2 through the section on tokens and words first. You should be able to distinguish a vocabulary from a sequence of token IDs. Allow approximately 45–75 minutes, including interpretation. Basic familiarity with a Python session is helpful; an LLM study companion can assist with the inspection, but should not invent your results.

This Lab specifies the investigation and required observations. It does not require training, neural-network weights, a GPU, a paid API, or a cloud account. Use a local Python environment with the Hugging Face Tokenizers library and public tokenizer artifacts. Initial software/file downloads require network access; the inspection itself can run locally afterward. Use the supplied harmless strings rather than private messages or credentials.

21.2 Local setup and commands

The downloadable runner and pinned requirements implement the observations below. Save both files in a new local working directory. They were tested with **CPython 3.13.13** and **Tokenizers 0.20.3**, using a CPU. Record predictions before executing the runner.

On macOS or Linux with Python 3.13 installed:

```
python3.13 -m venv .venv
.venv/bin/python -m pip install --no-deps -r tokenizer-requirements.txt
.venv/bin/python inspect_tokenization.py --artifacts tokenizer-artifacts --output
run-01
```

The `--no-deps` installation is deliberate: this runner uses the local Tokenizers backend only, not the optional Hub-client integration. On Windows, create the environment with `py -3.13 -m venv .venv` and replace `.venv/bin/python` with `.venv\Scripts\python.exe` in the other commands.

The initial run retrieves only `tokenizer.json` and `tokenizer_config.json` from the exact revision below. It verifies their SHA-256 hashes before loading them. It never retrieves neural weights, executes remote custom code, or sends the test strings to a service. The repository declares an MIT license; its pinned repository page provides artifact attribution and license metadata.

Open `run-01/manifest.json`, `run-01/results.json`, and `run-01/checks.json` in your text editor. The manifest records software versions, file hashes, pipeline settings, and companion configuration. Results contain escaped inputs, code points, bytes, IDs, display pieces, decoded strings, exact-equality checks, special-token observations, per-ID Unicode decoding, and a measured over-1024-token input. These are your machine observations; compare them with your predictions rather than copying them into an answer without interpretation. If a round trip fails, both strings and their first differing position remain in the results.

After the artifact files exist, repeat without network access into a fresh output directory:

```
.venv/bin/python inspect_tokenization.py --offline --artifacts tokenizer-artifacts  
--output run-02
```

Compare `run-01/results.json` and `run-02/results.json`; the measurements should be reproducible for this pinned setup. Manifest timestamps differ. The runner rejects modified artifacts and refuses to overwrite nonempty result directories, preserving unexpected outputs for investigation. Predictions and your written interpretation remain your own work.

21.3 1. Identify the specimen

Use `openai-community/gpt2` from the Hugging Face Hub, pinned to revision `607a30d783dfa663caf39e06633721c8d4cfc7e`. Its file listing at that revision includes `tokenizer.json`, `vocab.json`, `merges.txt`, and `tokenizer_config.json`. The repository is marked with an MIT license. We use GPT-2 for inspectability, without treating its tokenizer as representative of every modern model.

Download only the tokenizer files you need. The complete `tokenizer.json` is the executable tokenizer specification for this Lab; the separate vocabulary and merge files are useful for inspection. Avoid downloading the repository's much larger model-weight files.

Use `Tokenizer.from_file` to load the local JSON through Hugging Face Tokenizers. The versioned API reference documents this operation and the encode/decode controls used below. Record your installed library version rather than assuming that unversioned documentation describes it.

Create a small run manifest recording:

- Repository identifier and full artifact revision.
- Names and SHA-256 hashes of downloaded files.
- Python and Tokenizers versions, operating system, and run date.
- The tokenizer's model, normalizer, pre-tokenizer, post-processor, decoder, and added-token configuration.
- Vocabulary size, including whether added tokens are counted.
- Padding, truncation, automatic-special-token, and decode settings.

The tokenizer's model here means its segmentation component, such as BPE. It is not a loaded neural language model. Inspecting JSON fields is enough to distinguish those objects.

i Keep the experiment controlled

Use raw strings, with no pre-splitting into words. Explicitly disable padding and truncation. For the baseline, use `add_special_tokens=False` when encoding and `skip_special_tokens=False` when decoding. These settings address different operations: inserting markers versus removing them during decoding. Preventing automatic insertion does not necessarily disable recognition of a special-token spelling already present in the input.

21.4 2. Predict before inspecting

Write a short prediction for each group below. You do not need to guess numeric IDs. Predict which pairs might have different boundaries or counts, and give your reasoning. Mark uncertainty rather than forcing a confident answer.

For every input, record the exact string in an escaped, unambiguous representation. The following quoted fixtures use JSON-style escapes: `\n` denotes an actual newline, and `\u0301` denotes the combining acute accent. The surrounding quotes are notation, not input characters.

21.4.1 Prose and whitespace

- `"Paint it blue."`
- `" Paint it blue."`
- `"Paint it blue."`
- `"paint it blue."`
- `"Paint it blue.\n"`

Does adding one visible space necessarily add exactly one token? Does lowercasing change only an ID, or can it also change segmentation? Check rather than extrapolating from word counts. GPT-2's documented handling of leading spaces gives a reason to expect formatting to matter.

21.4.2 Code and identifiers

- `"total_count = 12\n"`
- `"totalCount = 12\n"`
- `"total_count=12\n"`
- `"def total(items):\n return sum(items)\n"`

Treat these as text fixtures; do not execute them. Compare spelling, punctuation, and indentation. A readable identifier may span several tokens, and source-code formatting is part of the input.

21.4.3 Numbers

- `"1234567890"`
- `"123 456 789 0"`
- `"12.50"`
- `"12,50"`
- `"0012"`

Predict whether the tokenizer uses single digits, larger chunks, or mixed pieces in each case. An observation about representation does not establish how well a neural model performs arithmetic.

21.4.4 Unicode

- "café", using precomposed é at U+00E9.
- "cafe\u0301", using e followed by a combining acute accent.
- "東京".
- "👤".

Record Unicode code points and UTF-8 bytes for these fixtures. Visually similar text can have different underlying representations. The Unicode Consortium's normalization annex explains canonical equivalence; do not silently normalize your inputs during this experiment.

21.5 3. Collect the observations

For each case, save:

1. Input label and exact escaped text.
2. Unicode code-point count and UTF-8 byte count.
3. Token count, ordered token IDs, and tokenizer-reported token strings.
4. The result of decoding the complete ID sequence with special tokens retained.
5. An exact equality check between the original string and the decoded result.

The Encoding API documents the ID and token-string fields. Display strings with escapes so that spaces, tabs, and newlines cannot disappear in the presentation. The reported token string is an inspection representation; use the decoder to reconstruct text rather than concatenating those display strings.

When an equality check fails, preserve both strings and inspect their code points. Identify the first difference. Check settings and normalization before attributing the change to BPE itself. A mismatch is evidence to investigate, not a result to conceal by rewriting the input.

For one Unicode example, also decode each individual ID separately, then compare the concatenation of those individually decoded strings with a single decode of the complete sequence. Record what happens. Partial byte sequences may not form valid standalone characters, so per-token display can be misleading. Do not assume every token must be individually readable for full-sequence decoding to work.

21.6 4. Inspect special tokens separately

Find the declared special-token entries in the downloaded artifact. Locate GPT-2's <|endoftext|> marker and record its ID from the file or tokenizer, rather than copying a guessed number.

Encode "Before<|endoftext|>After" with the baseline settings. Decode its IDs once retaining special tokens and once skipping them. Compare the two results. Then encode "Before After" with automatic special-token insertion both enabled and disabled. Did that flag actually add anything for this artifact's configured post-processor?

Explain the distinction between recognizing an explicit marker, automatically inserting markers, and suppressing them during display. The mere existence of a flag does not guarantee that toggling it

changes every tokenizer’s output. Also explain why this tokenizer-only inspection cannot demonstrate how a chat model would obey role boundaries.

21.7 5. Separate encoding length from model context

Inspect `model_max_length` in the companion `tokenizer_config.json`. It records 1024 for this artifact, but loading `tokenizer.json` directly does not automatically load that separate wrapper configuration. Pinned configuration file

With truncation disabled, construct a repeated harmless input whose measured encoding exceeds 1024 tokens. Save the count and round-trip result. Do not run a neural model on it. Explain why a tokenizer accepting the input cannot establish that GPT-2 supports that context length.

21.8 6. Interpret and submit

Produce three artifacts: a run manifest, a machine-readable results file, and a short written interpretation. Keep predictions made before execution alongside the measurements.

Your interpretation should answer:

- Which single-character or formatting change had the most surprising effect?
- Which examples show that tokens, words, code points, and bytes are different units?
- Did complete-sequence round trips preserve exact input? Explain any failures.
- What changed when special tokens were skipped?
- What evidence would be needed before claiming one language tokenizes “better” than another?
- Which conclusions concern this pinned tokenizer, and which concern neural-model behavior you did not measure?

Avoid ranking languages using unrelated short strings or treating token counts as intelligence scores. One carefully explained counterexample is more useful than a large table without interpretation.

21.8.1 Completion criteria

The Lab is complete when all four input groups have recorded predictions and observations; the manifest identifies reproducible artifacts and settings; exact round-trip checks are preserved; and the write-up correctly separates tokenization from model inference. There are no supplied token-count answers to imitate. Unexpected outputs belong in the record, with an explanation or an explicit open question.

21.9 More Learning

- Hugging Face Tokenizers pipeline — Trace normalization, pre-tokenization, segmentation, and post-processing as separate operations.
- GPT-2 technical report, Section 2.2 — Connect the byte-level design to what you observed.
- Unicode normalization annex — Investigate why visually equivalent strings need not have identical code-point sequences.

21.10 References

Artifact and API sources were checked on 7 October 2026. The fixtures and investigation design are original course material. This Lab supplies no fabricated execution results.

- Hugging Face. Pinned GPT-2 artifact listing, Tokenizer API, Encoding API, and tokenization pipeline.

- Radford, A., et al. (2019). Language Models are Unsupervised Multitask Learners.
- Unicode Consortium. Unicode Standard Annex #15: Unicode Normalization Forms.

22 Tiny Network, Visible Computation

22.1 Goal

Build a forward-pass trace you can check line by line. Predict how an intervention changes the output, then explain the result using the computation rather than a story about what a neuron “wants.”

Prerequisite: Lesson 1.3, through the microscopic network example. Read its training section before the optional derivative check.

Time: about 45–75 minutes, including the written interpretation. The optional derivative check adds about 15 minutes.

Requirements: paper or a text editor. The coding route uses an existing Python 3 installation, ordinary CPU execution, and the standard library only. It requires no API, account, internet access, GPU, model download, or paid compute. A complete hand-worked submission is a valid alternative if Python is unavailable.

Expected artifact: a short `submission.md` containing predictions, arithmetic, observations, and interpretation. For the coding route, also keep `tiny_network.py` and a copy of its actual output. Do not substitute the answer key for observations from a run.

! Calculate before coding

Finish Part 1 and record the predictions in Part 2 before executing a program or asking an LLM to solve them. If you use an assistant, ask it to check your reasoning after your first attempt. The prediction is part of the evidence of learning.

22.2 The fixed network

Use exactly these values:

$$W = \begin{bmatrix} 1 & -1 \\ 0.5 & 1 \\ -1 & 2 \end{bmatrix}, \quad b = \begin{bmatrix} 0 \\ 1 \\ 0.5 \end{bmatrix}, \quad v = \begin{bmatrix} 2 \\ -1 \\ 1 \end{bmatrix}, \quad c = 0.5.$$

For every input column x , calculate

$$z = Wx + b, \quad h = \text{ReLU}(z), \quad \hat{y} = v^T h + c.$$

These parameters were chosen for transparent arithmetic. The network has not been fitted to a dataset. A successful Lab verifies computation and controlled changes, not predictive performance on an application.

22.3 Part 1 — Make a hand trace

Use the fresh input $x = [1, 2]^T$.

1. Write the shape of each object: x , W , b , z , h , v^T , and $\hat{y}$.
2. Expand each of the three weighted sums, including its bias.
3. Apply ReLU to obtain h .
4. Write each hidden unit's separate contribution $v_j h_j$ to the output. Add c once.
5. Count the weights and biases. Explain why the three computed values in h do not add three learned parameters.
6. Describe how the arithmetic would differ if ReLU were removed. Predict whether that changes the output for this input.

Keep your original attempt if you correct it later. A clear correction is useful evidence: name the particular sign, bias, or shape mistake rather than merely replacing the number.

22.4 Part 2 — Predict interventions

Return to the lesson's baseline input $x = [2, -1]^T$. Use the original parameters at the start of every case. Each case changes only what is named; do not accidentally accumulate edits.

Record a numerical prediction and one sentence of reasoning for each case:

1. **Ablation:** force $h_2 = 0$ after ReLU and before the output calculation.
2. **Weight change:** set $v_3 = 101$ while leaving every other parameter unchanged.
3. **Different input:** keep that same $v_3 = 101$ change, but now use $x = [0, 1]^T$. Compare with the original network on that input.
4. **Remove ReLU:** use $h = z$ for the baseline input, with all original parameters restored.

For the ablation, keep both the unmodified activation vector and the vector actually sent to the readout. This distinguishes observing an activation from changing it.

Which case do you expect to disprove the statement “if changing this weight has no effect once, this weight is useless”? Explain your prediction before checking the answer.

22.5 Part 3 — Implement or independently recalculate

22.5.1 Coding route

Implement the equations directly with lists, loops, multiplication, addition, and `max`. Avoid a machine-learning library for the first pass. The small scale makes it practical to see every operation.

The following is a **learner scaffold**, not a complete reference implementation. Its unfinished functions deliberately raise errors. Complete them before running the checks; the code shown here is not presented as an executed or verified solution.

```
from copy import deepcopy

PARAMS = {
    "W": [[1.0, -1.0], [0.5, 1.0], [-1.0, 2.0]],
    "b": [0.0, 1.0, 0.5],
    "v": [2.0, -1.0, 1.0],
    "c": 0.5,
}
```

```

def dot(a, b):
    # Reject mismatched lengths before computing the sum.
    raise NotImplementedError("Implement a checked dot product")

def forward(x, params, ablate_index=None, use_relu=True):
    # Check: len(x) == 2; W has 3 rows of length 2;
    # b and v each have length 3.
    # 1. Compute and retain z.
    # 2. Compute and retain the natural activation vector h.
    # 3. Copy h to readout_h before any intervention.
    # 4. If requested, set readout_h[ablate_index] to zero.
    # 5. Compute the scalar output from readout_h.
    # Return a dict containing z, h, readout_h, and output.
    raise NotImplementedError("Implement the forward trace")

# Python uses zero-based indices: hidden unit 2 has index 1.
# Use deepcopy(PARAMS) for each independent parameter change.
# Do not mutate PARAMS or an earlier run's activation vectors.

```

For `dot`, do not rely on an unchecked `zip`: it can silently truncate mismatched inputs. A deliberate shape error should raise a clear `ValueError`. Reject an invalid ablation index too.

Once the functions are complete:

1. Run both the fresh hand-worked input and the lesson baseline. Print every returned field.
2. Compare intermediate values before comparing the final output. An incorrect implementation can occasionally produce the right output through cancelling errors.
3. Run the four intervention cases with independent parameter copies.
4. Test $x = [1, 2, 3]$ and confirm that the shape check rejects it.
5. Run the baseline again and check that earlier interventions did not change it.

You can run your completed file with:

```
python3 tiny_network.py
```

Record the Python version and the actual command. Save the observed trace, including any failing checks you used to diagnose a problem. No benchmark or training claim is expected.

22.5.2 Paper route

Recalculate the same cases using a second arrangement of the arithmetic: compute Wx first, add b , apply ReLU, and only then form the readout. Compare this with your earlier unit-by-unit expansion. State that your results are hand-checked rather than program outputs.

22.5.3 Interpret the comparison

For each case, write the prediction, checked result, and explanation. If they differ, identify the earliest intermediate value that diverged. Answer these questions:

- Why can removing a positive activation increase the final output?
- Why does a weight change matter on one input and disappear on another?
- Did the ablation retrain the model?
- What would you need to know before assigning a real-world meaning to these hidden units?

22.6 Part 4 — Optional one-parameter training step

This part connects inference to training without implementing a training framework. Start again from the original parameters and input $[2, -1]^T$. Let the target be $t = 4$ and define $L = \frac{1}{2}(\hat{y} - t)^2$.

1. Calculate the original loss and the derivative $\partial L / \partial v_1$ by hand.
2. Predict the effect of changing only v_1 with learning rate 0.1. Freeze all other parameters.
3. Recalculate the output and loss with the new v_1 .
4. Explain why this improvement on one example does not establish generalization.

For an optional numerical derivative check, choose $\epsilon = 10^{-4}$. Evaluate the loss twice with $v_1 + \epsilon$ and $v_1 - \epsilon$, resetting the other parameters each time, and compute

$$g_{\text{finite difference}} = \frac{L(v_1 + \epsilon) - L(v_1 - \epsilon)}{2\epsilon}.$$

Compare that estimate with the hand derivative. Ordinary floating-point arithmetic can introduce small differences; report what you observe rather than promising bit-for-bit equality. This finite-difference check concerns one scalar parameter. It is not an efficient substitute for backpropagation in a large network.

Do not describe merely computing this derivative as training. The parameter update is a separate operation. PyTorch's optimization tutorial illustrates this separation in a framework.

22.7 Checkpoints and solutions

The values below are **analytically derived expectations**, not a record of a software run. Compare them only after making your own attempt.

Hand trace and interventions

For the fresh input $[1, 2]^T$:

- $z = [-1, 3.5, 3.5]^T$.
- $h = [0, 3.5, 3.5]^T$.
- The three contributions are 0, -3.5 , and 3.5 .
- The output is 0.5 after adding the output bias.
- Without ReLU, the first contribution becomes -2 , so the output is -1.5 .

Shapes are $x : 2 \times 1$, $W : 3 \times 2$, $b, z, h : 3 \times 1$, $v^T : 1 \times 3$, and $\hat{y}$: scalar. The parameter count is $6 + 3 + 3 + 1 = 13$. Activations are calculated from the input and parameters, not independently stored coefficients to be learned in this model.

For the baseline input $[2, -1]^T$, $z = [3, 1, -3.5]^T$, $h = [3, 1, 0]^T$, and the output is 5.5.

1. Ablating hidden unit 2 yields readout activations $[3, 0, 0]^T$ and output 6.5. The removed contribution was negative.
2. Changing v_3 to 101 leaves the baseline output at 5.5, because the third activation is zero.
3. On $[0, 1]^T$, $z = [-1, 2, 2.5]^T$ and $h = [0, 2, 2.5]^T$. The original output is 1; the changed output is 251. The same weight now multiplies a nonzero activation.
4. Removing ReLU at the baseline yields output 2. The negative third preactivation now contributes -3.5 .

Ablation changes an intermediate value for a controlled forward computation. It does not, by itself, fit parameters to data. The parameters and axes have no assigned application meaning here, so these effects do not justify labels such as a “truth neuron” or “planning neuron.”

i Optional derivative check

The original error is 1.5 and loss is 1.125. Since $h_1 = 3$, the derivative is $(\hat{y} - t)h_1 = 4.5$.

Updating only v_1 gives $2 - 0.1(4.5) = 1.55$. The new output is 4.15 and loss is 0.01125.

In exact arithmetic, the centered finite difference equals 4.5 here because the loss is quadratic in v_1 . An implemented floating-point estimate should be close; a tolerance such as 10^{-6} is a reasonable check for this specific calculation using ordinary Python floats. Its observed behavior still needs to be checked in the completed program.

22.8 Submit an explanation

Your `submission.md` should include:

- **Prediction:** your pre-run answers and reasoning;
- **What I did:** hand calculations or implementation details, with version and command if applicable;
- **Results:** checked intermediate values and the four intervention outcomes;
- **What surprised me:** or an explanation of why the results matched expectations;
- **Interpretation:** the causal role of the changed quantity in these examples;
- **Limits:** what these checks do not establish about training or real-world performance.

A strong submission makes it possible for someone else to reconstruct the arithmetic. It also distinguishes a parameter change from an activation intervention and avoids generalizing from a single zero effect.

22.9 More Learning

- Lesson 1.3 supplies the full mathematical and conceptual explanation.

- PyTorch Linear documents the corresponding affine layer if you later reproduce your checked implementation with a library.
- PyTorch Automatic Differentiation connects the optional hand derivative to computational graphs. Framework use is optional and outside the core Lab.

23 Reading the Original Transformer Paper

23.1 Goal

Use a research paper as a recurring reference while your understanding develops. Build an initial map before studying the residual stream, then revisit that map after learning attention and MLPs. You are not expected to understand the entire paper on your first pass.

Primary source: Vaswani and colleagues, *Attention Is All You Need*, originally published in 2017. The linked arXiv version provides accessible HTML; the PDF is an alternative for its figures and equations. Record which version you used.

Prerequisite: Module 1 for Part 1; Lessons 2.1–2.3 for Part 2.

Time: 20–30 minutes now, then about 40–60 minutes after Lesson 2.3.

Requirements: the paper and somewhere to write. No account, API, model download, GPU, paid compute, or programming is required. Keep the paper available locally if you want to do the reading offline.

Expected artifact: one short reading note with dated first-pass and return-pass sections, a simple sketch, source locations, and revised explanations. An incomplete but honest map is more useful than an elaborate summary you cannot explain.

23.2 Part 1 — Orient before Lesson 2.1

Read only the abstract, introduction, Figure 1 and its caption, and conclusion. Scan the section headings to know where further explanations live. Set a timer if you tend to stop at every unfamiliar term.

Write three kinds of notes separately:

- **What the authors claim:** attach a section or figure number.
- **My current interpretation:** explain it in your own words.
- **A question I cannot answer yet:** leave it open.

This distinction prevents a tentative interpretation from quietly becoming a statement attributed to the paper. Avoid copying sentences into your submission except for an occasional short phrase that genuinely needs quotation.

23.2.1 Find the problem and the proposed change

Answer these questions in one or two sentences each:

1. What task setting motivates the work? Identify an example from the paper rather than substituting a modern chatbot use case.
2. Which broad architectural approaches are the authors comparing against?
3. What do the authors propose changing about the main sequence-processing mechanism?
4. Which proposed benefits concern computation, and which concern measured task performance?

5. Does the title imply that the network literally contains nothing except attention? Record your initial answer and what in Figure 1 supports it.

You may leave technical details unresolved. The aim is to identify the research question and the shape of the argument.

23.2.2 Make an orientation sketch

Draw a small, original sketch with two stack-shaped boxes and the major arrows between them. Use your own arrangement rather than tracing or redistributing the paper's figure. Label where the input enters and where output predictions emerge.

Choose five labels from Figure 1. For each, write one of:

- I can explain this operation.
- I recognize its role but cannot calculate it.
- I do not know what it means yet.

Include residual addition or “Add & Norm” among the labels. Mark the connection whose purpose is least clear to you. You will revisit it after Lesson 2.3.

23.2.3 Stop deliberately

Write a provisional three-sentence explanation of the architecture. Then list up to five questions you want Module 2 to answer. Good questions name a missing connection: “Why does this addition need equal dimensions?” is more useful than “How do Transformers work?”

Now continue to Lesson 2.1. You do not need to derive attention, reproduce training, or resolve every symbol before moving on.

23.3 Part 2 — Return after Lesson 2.3

Reopen your initial note without replacing it. Read Sections 3.1–3.3 carefully, referring to Figure 1 and the attention figure. Read Sections 3.4–3.5 for the input and output boundaries. Treat details not yet covered in the course as questions for Lesson 2.4.

23.3.1 Trace rather than recite

Add to your sketch and answer:

1. Where are the residual additions? What shapes must agree at each one?
2. Where is normalization relative to the addition? How does this differ from Lesson 2.1's main pre-norm illustration?
3. Which operations allow information to move between positions, and which apply a transformation separately at each position?
4. Which attention operation reads information originating in the other stack? Trace both ends of that arrow.
5. Which positions must be unavailable when the decoder predicts a token, and how does the architecture enforce that restriction?
6. How can the feedforward component use an internal width different from the width of the surrounding stream?

For one attention equation, annotate the meaning and shape of each matrix under the paper’s conventions. Explain the final output shape in words. A shape check is more valuable here than memorizing the equation’s typography.

23.3.2 Compare with one documented decoder-only model

Use one primary reference, such as the LLaMA report’s architecture section, together with Lesson 2.1. Identify one component that is retained in broad form, one arrangement or operation that changes, and one component of the original encoder–decoder system absent from the chosen decoder-only architecture.

Be specific about the model and source. “Modern Transformers use pre-norm” is too broad: Lesson 2.1 already introduced architectures whose normalization arrangements need more careful description. If an implementation adds detail absent from a report, label which source establishes which fact.

Write one paragraph answering: What would go wrong if I treated Figure 1 as an exact wiring diagram for every current LLM?

23.3.3 Separate evidence from extrapolation

Choose one result reported in the paper. Record the task, evaluation measure, model variant, and location of the result. You do not need to reproduce it. Explain why that result does not, by itself, establish performance on an unrelated present-day task.

Then revise your original three-sentence explanation. Preserve the earlier version and identify two concrete improvements. Name one question still open after the second pass. Finishing the Lab does not require pretending the paper has become completely transparent.

23.4 Self-check and discussion guide

Compare only after writing your own answers.

i Orientation checks

The paper studies sequence transduction, especially machine translation. Its proposed architecture replaces recurrence and convolution in its main sequence model with attention-based processing. Figure 1 still contains embeddings, feedforward operations, residual additions, normalization, and output components.

The original model has encoder and decoder stacks; a decoder-only language model does not retain that complete two-stack arrangement. Post-norm placement in the original differs from the chapter’s sequential pre-norm example. Your comparison should identify a particular modern architecture rather than generalize from the word “Transformer.”

For the return pass, a strong answer explains dependencies, checks shapes, and cites locations. If you can name every box but cannot trace what supplies its input, return to that connection. If you can calculate an equation but cannot explain why a mask is needed, revisit the prediction setting.

23.5 Submit your reading note

Include your first-pass map, unresolved questions, annotated second-pass sketch, one model comparison, one bounded result interpretation, and revised explanation. Clearly distinguish the paper's reported results from experiments you ran; this reading Lab requires no executed model experiment.

23.6 More Learning

- The original paper remains the main source. Rereading a specific section with a sharper question often helps more than opening another general overview.
- LLaMA architecture provides one concrete comparison, including normalization and positional choices.
- Trace a Residual Stream turns one part of the wiring diagram into arithmetic. Return to your sketch afterward and mark exactly which operations that toy omits.

24 Trace a Residual Stream

24.1 Goal

Track the difference between a residual state and an update. Predict how changing an early branch affects a later calculation. Optionally record clearly identified residual boundaries in a small trained Transformer.

Prerequisite: Lesson 2.1.

Time: 45–60 minutes for the hand trace. Allow additional setup time for the optional real-model path.

Core requirements: paper or a text editor. The first path is fully offline and requires no API, account, paid compute, GPU, installed machine-learning library, or model download. A hand-worked submission completes the core Lab.

Expected artifact: a short `submission.md` or equivalent note containing predictions, calculations, explanations, and limitations. Distinguish analytically checked expectations from outputs you actually observed in software.

24.2 Part 1 — Predict a fresh residual sequence

Use the lesson's hand-constructed functions with a fresh input:

$$x = \begin{bmatrix} 1 \\ 2 \\ -1 \end{bmatrix}, \quad F(x) = \begin{bmatrix} -x_1 \\ 2 \\ 0 \end{bmatrix}, \quad G(y) = \begin{bmatrix} y_2 \\ -y_2 \\ 2y_1 \end{bmatrix}.$$

Calculate $y = x + F(x)$ and $z = y + G(y)$. The functions are affine or linear update rules chosen for arithmetic; neither implements attention or a Transformer MLP. There is no normalization in this toy.

Before calculating, predict:

1. Will the first coordinate of y depend on the first coordinate of x ?
2. Must each addition increase the stream's norm?
3. If you remove F , can you recover the changed output by subtracting the old $F(x)$ from the old z ?

Then make a trace with five labeled vectors: input, first update, intermediate stream, second update, and final stream. Check that every vector has shape 3×1 .

Calculate their Euclidean norms and the total change $z - x$. Explain why the norm of the total change need not equal the sum of the two update norms. A diagram of three-dimensional arrows is optional; coordinate arithmetic is sufficient.

24.3 Part 2 — Compare three different changes

Reset to the same fresh input for every case.

1. **Suppress the first branch:** replace its output with zero, and recompute the second branch from its changed input.
2. **Suppress the second branch:** retain the first update, then replace only the second update with zero.
3. **Parallel arrangement:** evaluate both branches on the original input, producing $z_{\text{parallel}} = x + F(x) + G(x)$.

Record a prediction for each before doing its arithmetic. Explain why case 3 changes the architecture rather than simply renaming an intermediate vector.

Use the scalar readout $s = [0, 0, 1]z$, which selects the final third coordinate. Compare the two baseline update norms with the effect of suppressing each branch on s . Does the larger update necessarily have the larger effect? Limit your conclusion to this specified calculation.

Finally, give the toy two positions x and $\tilde{x} = [-2, 0, 3]^T$. Arrange them as rows of a 2×3 array. Apply the same update rules separately to each row, transposing your individual-vector results as needed. Explain why these rules never move information from one position into the other.

24.4 Part 3 — Optional implementation of the toy

A standard-library Python implementation can reproduce the trace on a CPU. Implement checked vector addition, the two update rules, Euclidean norm, and independent intervention cases. Reject mismatched vector lengths rather than silently truncating them.

Retain copies of intermediate vectors. Confirm that running an intervention does not mutate the baseline input or stored baseline results. Compare every intermediate vector with your hand trace before checking just the readout.

If you use the supplied runner or a coding assistant implements this specification, inspect the arithmetic and save the actual run output, Python version, and command. Analytical answers below are not evidence that your script has run.

24.5 Part 4 — Optional trained-model observation

24.5.1 Use a bounded public specimen

Use EleutherAI/Pythia-14M at revision 94f7c35d5e9f2e9bac8ca839329f505b4d007d5d. This immutable publisher commit contains the final checkpoint. Its config specifies six blocks, width 128, and `use_parallel_residual: true`.

Download only `config.json`, `tokenizer.json`, `tokenizer_config.json`, `special_tokens_map.json`, `generation_config.json`, and `model.safetensors` from that revision. The weights file is 28,143,920 bytes, about 28.1 MB; tokenizer and configuration files are additional. Inspect the planned total before downloading. Do not download every training checkpoint, optimizer states, `.bin` weights, or custom Python files. Framework installation can consume substantially more space than these weights.

The model card records a February 2026 replacement of the formerly hosted deduplicated model. Pinning this revision avoids silently mixing those histories. Use the model for local inspection, not as a reliable assistant or factual source.

24.5.2 Local setup and commands

The toy runner supplies an optional standard-library implementation of Part 3. Save it locally and run `python3 trace_residual_toy.py` after making your hand trace. Its JSON output records actual Python version and independent cases. Compare the intermediate arrays with your calculations.

For the trained-model route, save the observation runner and pinned requirements in a new directory. Use a separate environment from Lab 02: these libraries require a newer Tokenizers version. This setup was tested with CPython 3.13.13, PyTorch 2.9.0, Transformers 4.57.1, Tokenizers 0.22.1, Safetensors 0.6.2 and Hugging Face Hub 0.35.3. The download client is CPython 3.13.13's standard-library `urllib.request`; model inference uses the explicit eager attention backend.

On macOS with Python 3.13 installed:

```
python3.13 -m venv .venv-residual
.venv-residual/bin/python -m pip install -r residual-requirements.txt
.venv-residual/bin/python inspect_residual_stream.py --plan
.venv-residual/bin/python inspect_residual_stream.py --artifacts pythia-artifacts
--output run-01
```

On Linux, install the CPU-only PyTorch wheel before the requirements, avoiding optional CUDA packages:

```
.venv-residual/bin/python -m pip install torch==2.9.0 --index-url https://
download.pytorch.org/whl/cpu
.venv-residual/bin/python -m pip install -r residual-requirements.txt
```

On Windows use `py -3.13 -m venv .venv-residual`, replace the Python executable with `.venv-residual\Scripts\python.exe`, and use the CPU-wheel command before installing requirements. Package installation can require substantially more disk space than the specimen files.

The plan prints exactly six files totaling **30,264,046 bytes**, including the 28,143,920-byte weights file. The runner verifies each fixed byte size and SHA-256 hash; it rejects extra artifact files and altered caches. Initial downloads retrieve only that scope, without an account. Text is fixed to the harmless sentence below and processed locally. Afterward repeat into a fresh directory without downloading:

```
.venv-residual/bin/python inspect_residual_stream.py --offline --artifacts pythia-
artifacts --output run-02
```

Open `manifest.json`, `observations.json` and `captures.json` in each result directory. These record versions, provenance, token IDs/pieces/positions, full cloned raw states and branch updates, selected-position metrics, separate final normalization, actual maximum numerical errors, repeated-input comparisons, and hook-removal checks. Undefined cosines are JSON `null`. The runner refuses to overwrite nonempty output directories. Retain unexpected observations; do not replace them with the analytical toy answers. The implementation checks are measured on the machine that runs it; no expected trained-model norms are supplied.

24.5.3 Implementation requirements

The following requirements describe the supplied runner and any independent implementation. Inspect them alongside your actual run; published requirements are not your measured results.

- Use CPU execution and float32 model computation, an explicit evaluation mode, inference mode without gradients, and `use_cache=False`. No inference API, GPU service, account, or payment is needed. Initial artifact and dependency downloads require internet access.
- Use Transformers' built-in `GPTNeoXForCausalLM` implementation and a fast tokenizer with `trust_remote_code=False`; load only safetensors weights. After the allowlisted download, load from that local directory with `local_files_only=True` so the observation run needs no network. Never enable remote custom code to solve a loading error.
- Pin and record tested versions of Python, PyTorch, Transformers, tokenizers, safetensors, and the download client. The inspected architecture reference is Transformers v4.57.1; record any implementation-version change and recheck its boundaries.
- Use a short nonprivate input, such as `The small red boat crossed the lake`. Record actual token IDs, decoded pieces, and positions before selecting a token. Do not assume words coincide with tokens.
- Record the first block's input, each of the six raw block outputs, and the separate output of the final LayerNorm. Clone captured tensors and remove observation hooks afterward.

At each block, this checkpoint adds attention and MLP updates calculated from separately normalized versions of the same input. Capture branch outputs at their addition boundaries as well if verifying that identity. Source inspection must establish the hook locations and whether a module returns a tensor or a tuple; do not guess from a generic hook recipe.

Do not label a returned `hidden_states` tuple as seven raw residual states without checking the implementation. In this architecture, final normalization changes the last returned boundary. Keep the raw last-block output and final normalized output separately labeled.

24.5.4 Observe and interpret

For one selected token position, report each raw state's norm, consecutive raw-state difference norm, and cosine similarity where both norms are nonzero. Mark undefined cosines explicitly. Keep final-normalization changes separate from block updates.

Verify expected shapes $(1, n, 128)$ and, if capturing branches, the parallel addition identity within a declared floating-point tolerance. Record maximum error rather than reporting only "passed." Repeat the unchanged input as a reproducibility check.

Describe one observed pattern and two explanations consistent with it. Explain what additional evidence would distinguish those explanations. Do not label a large update as new factual knowledge or successful reasoning. No expected trained-model numbers are provided: actual observations must come from your recorded run.

24.6 Checkpoints and solutions

i Analytically derived answers

The first intermediate coordinate is always $y_1 = x_1 - x_1 = 0$. For the fresh input:

- $F(x) = [-1, 2, 0]^T$ and $y = [0, 4, -1]^T$.
- $G(y) = [4, -4, 0]^T$ and $z = [4, 0, -1]^T$.
- Norms of input, first update, intermediate stream, second update, and final stream are $\sqrt{6}$, $\sqrt{5}$, $\sqrt{17}$, $\sqrt{32}$, and $\sqrt{17}$.
- $z - x = [3, -2, 0]^T$, with norm $\sqrt{13}$.

Suppressing the first branch gives $G(x) = [2, -2, 2]^T$ and output $[3, 0, 1]^T$. Subtracting the baseline first update from the baseline output would give $[5, -2, -1]^T$, which is a different calculation.

Suppressing the second branch leaves $y = [0, 4, -1]^T$. The parallel arrangement gives $[2, 2, 1]^T$.

The baseline readout is -1 . Suppressing the smaller first update changes it to 1 ; suppressing the larger second update leaves it at -1 . Update norm therefore does not rank these intervention effects.

For the second position $\tilde{x}$, the first update is $[2, 2, 0]^T$, its intermediate state is $[0, 2, 3]^T$, its second update is $[2, -2, 0]^T$, and its final state is $[2, 0, 3]^T$. Stack these results as rows; neither computation references the other row.

24.7 Submit an explanation

Include original predictions, corrected arithmetic, baseline and intervention traces, readout comparison, and the two-position explanation. If you executed code, attach its actual outputs and environment details. If you performed the optional model observation, include checkpoint revision, download scope, exact measurement boundaries, shape and numerical checks, observations, and limitations. A complete hand trace remains a valid core submission.

24.8 More Learning

- Lesson 2.1 explains the geometry and architecture choices.
- Pythia's pinned configuration and GPT-NeoX source establish the optional specimen's actual wiring.
- Reading the Original Transformer Paper connects this trace to a different historical architecture. Explain the differences before transferring your observation code to it.

25 Compute and Perturb Attention

25.1 Goal

Trace one attention head from projected vectors to output. Then change one part of the computation at a time to distinguish attention coefficients, value contributions, and mask effects.

Prerequisite: Lesson 2.2 — Attention, through the three-position example. Read its interpretation section before writing your conclusions.

Time: about 60–90 minutes. Allow another 20 minutes for the optional checks.

Requirements: paper or a text editor. The coding route uses an existing Python 3 installation, ordinary CPU execution, and the standard library only. No account, API, GPU, internet access, model download, package installation, or paid compute is needed. A hand-worked route is equally valid.

Expected artifact: a short `submission.md` containing predictions, calculations, comparisons, and interpretation. For the coding route, also retain `tiny_attention.py` and its actual output. Label hand-derived answers and observed program output separately.

! Predict before executing

Complete Parts 1 and 2 before running code or reading the reference solution. If you use an LLM, ask it to critique your attempt before showing you its own. The supplied scaffold is unfinished; the completed reference is a separate comparison aid.

25.2 The fixed head

All position vectors are rows. Use

$$X = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}, \quad W_Q = I_2, \quad W_K = \sqrt{2} \ln(2) I_2, \quad W_V = \begin{bmatrix} 2 & 0 \\ 0 & 4 \end{bmatrix}.$$

Compute $Q = XW_Q$, $K = XW_K$, $V = XW_V$, then

$$S = QK^T / \sqrt{2}, \quad A = \text{softmax}_{\text{sources}}(S + M), \quad Z = AV.$$

The causal mask has zero on and below the diagonal and negative infinity above it. There is one head, no bias, no dropout, no normalization, and no residual addition in this Lab's measured output. An identity output projection would leave Z unchanged. These choices isolate the mechanism; this is not a trained language model.

25.3 Part 1 — Compute before coding

1. Write the shapes of X , each projection matrix, Q , K , V , S , A , and Z .
2. Calculate the three projected matrices. Retain the exact factor $\sqrt{2} \ln 2$ instead of rounding it early.
3. Expand $q_2 \cdot k_1$ and $q_3 \cdot k_3$, then construct the complete scaled score matrix.
4. Write the masked scores. For each row, list the allowed sources and the denominator used by softmax.

5. Calculate A exactly as fractions. Verify the row sums and the entries above the diagonal.
6. Expand z_3 as three scaled value vectors. Check z_1 and z_2 too.
7. Explain why applying softmax across destinations rather than sources computes a different operation.

Now deliberately make a masking error on paper: apply softmax to every score in each row, then zero the future-position coefficients without renormalizing. Compute its row sums and output. Does a lower-triangular final matrix guarantee that the computation respected causality?

Do not correct your original prediction by silently overwriting it. If you discover an error, identify the first intermediate quantity that went wrong.

25.4 Part 2 — Predict controlled changes

Restart from the original projected matrices for every case. **Each intervention takes place after projection**, and changes only what is named. Do not change X as a shortcut: that could change queries, keys, and values together.

For each case, predict which rows of A change, which rows of Z change, and the exact new z_3 .

1. **Value ablation:** replace $v_2 = [0, 4]$ with $[0, 0]$. Leave Q , K , and the mask unchanged. Do not renormalize A .
2. **Query perturbation:** replace only q_3 with $[1, 0]$. Keep K , V , and the mask fixed, and recompute attention coefficients.
3. **Connection removal:** forbid only the connection from destination 3 to source 2. Leave the other mask entries, scores, and values unchanged. Recompute softmax over the remaining sources. This does not delete token 2 from the sequence.
4. **Future-key perturbation:** replace k_3 with twice its original vector, leaving Q and V unchanged. Predict whether any correct output at positions 1 or 2 can change. Also predict the result for position 1 in the deliberately buggy masking procedure from Part 1.

Before checking the answers, explain why value ablation and connection removal should not generally produce the same result.

25.5 Part 3 — Implement small arrays

25.5.1 Learner scaffold

Copy this into `tiny_attention.py`. The unfinished functions deliberately raise errors. Implement them using lists, loops, sum, and functions from math.

```
import math

X = [[1.0, 0.0], [0.0, 1.0], [1.0, 1.0]]
C = math.sqrt(2.0) * math.log(2.0)
WQ = [[1.0, 0.0], [0.0, 1.0]]
WK = [[C, 0.0], [0.0, C]]
WV = [[2.0, 0.0], [0.0, 4.0]]
ALLOWED = [[True, False, False],
            [True, True, False],
```

```

[True, True, True]]

def matmul(left, right):
    # Reject empty, ragged, or incompatible matrices.
    # Compute every output entry as a row-column dot product.
    raise NotImplementedError("Implement checked matrix multiplication")

def masked_softmax(scores, allowed):
    # Check matching nonzero lengths and at least one allowed source.
    # Reject nonfinite input scores and non-boolean mask entries.
    # Subtract the largest ALLOWED score before exponentiation.
    # A blocked source contributes exactly zero.
    raise NotImplementedError("Implement stable masked softmax")

def attention(Q, K, V, allowed):
    # Q/K widths must agree; K/V must have the same source count.
    # allowed must have one row per query and one column per key.
    # Build scaled scores, masked scores, row-wise weights, and Z.
    # Return a dict with scores, masked_scores, weights, and output.
    raise NotImplementedError("Implement the attention trace")

# Q = matmul(X, WQ); K = matmul(X, WK); V = matmul(X, WV)
# Print attention(Q, K, V, ALLOWED) after completing the functions.

```

The allowed convention is explicit: True permits a source. Some framework APIs use the opposite convention for a padding mask. This Lab does not import a framework or inherit its mask semantics.

Use fresh nested-list copies for interventions, such as `[row[:] for row in V]`. A shallow outer-list copy alone still shares the inner rows. Make sure later edits cannot change a previously recorded baseline.

25.5.2 Run and compare

Run your completed file with an appropriate command for your existing installation, such as `python3 tiny_attention.py`. Record the command and Python version.

Print all four returned arrays. Compare the masked pattern exactly and finite numerical values within an absolute tolerance of 10^{-9} . Floating-point approximations to $\ln 2$ and exponentiation need not produce exact fractions. Do not compare an expression such as infinity minus infinity to measure error; compare blocked positions separately.

Then run the four interventions. Record the observed differences alongside your predictions. Finally rerun the unchanged baseline to detect accidental mutation.

Your implementation should reject:

- a ragged matrix;

- incompatible query and key widths;
- different key and value source counts;
- a mask with the wrong dimensions;
- a row with every source blocked;
- nonfinite supplied query, key, value, or score entries.

This deliberately narrow implementation rejects all-masked rows instead of inventing a fallback. Production code may have additional policies, data types, batching, or numerical constraints; those are outside this Lab.

25.5.3 Hand-worked alternative

Calculate every intervention independently, using fractions where possible. Check the answer once by expanding sums and once by matrix multiplication. For the future-key perturbation, you need not recompute forbidden terms to prove earlier outputs unchanged in the correct computation. Record that these are analytical results rather than program observations.

25.6 Part 4 — Explain what changed

Write a short interpretation addressing all of the following:

1. Which intervention changed an output while preserving the entire attention map?
2. Which interventions changed only the third row of the map?
3. Why did connection removal redistribute weight while value ablation did not?
4. How can the buggy procedure leak a future key's information when its final coefficients above the diagonal are zero?
5. What did these experiments establish about this isolated head? What additional evidence would be needed to claim that a head explains a trained model's final answer?

A useful conclusion names the intervention, controlled quantities, and measured outcome. Avoid calling a coefficient a token's "importance" without specifying the meaning and test.

Analytically derived answer key

These values follow from the equations. They are reference expectations, not a report of an executed learner program. Compare them only after recording your attempt.

Shapes: X, Q, K, V, Z are 3×2 ; projection matrices are 2×2 ; S, A, M are 3×3 .

Baseline: $Q = X$, $K = \sqrt{2} \ln(2) X$, $V = [[2, 0], [0, 4], [2, 4]]$. The scaled scores are $\ln(2)[[1, 0, 1], [0, 1, 1], [1, 1, 2]]$. The causal softmax denominators are 2, 3, and 8 when using the unshifted exponentials.

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 1/3 & 2/3 & 0 \\ 1/4 & 1/4 & 1/2 \end{bmatrix}, \quad Z = \begin{bmatrix} 2 & 0 \\ 2/3 & 8/3 \\ 3/2 & 3 \end{bmatrix}.$$

Buggy post-softmax masking: coefficients are $[[2/5, 0, 0], [1/5, 2/5, 0], [1/4, 1/4, 1/2]]$. Row sums are $2/5, 3/5$, and 1; outputs are $[[4/5, 0], [2/5, 8/5], [3/2, 3]]$.

Value ablation: A is unchanged. The new outputs are $[[2, 0], [2/3, 0], [3/2, 2]]$. At position 3 the change is exactly $-A_{32}v_2 = [0, -1]$.

Query perturbation: only row 3 changes. Its scaled scores are $[\ln 2, 0, \ln 2]$, its weights are $[2/5, 1/5, 2/5]$, and its output is $[8/5, 12/5]$.

Connection removal: only row 3 changes. Its weights become $[1/3, 0, 2/3]$, giving $z_3 = [2, 8/3]$. This differs from value ablation because the remaining values receive larger coefficients.

Future-key perturbation: correct outputs at positions 1 and 2 are unchanged. Row 3 has exponentials $[2, 2, 16]$, weights $[1/10, 1/10, 4/5]$, and output $[9/5, 18/5]$. In the buggy procedure, row 1's full denominator changes from $2 + 1 + 2 = 5$ to $2 + 1 + 4 = 7$, so its output changes from $[4/5, 0]$ to $[4/7, 0]$ despite having its future coefficients zeroed afterward.

25.7 Completed reference implementation

This is a separate, complete reference listing for comparison after your own attempt. It expresses the intended algorithm; its presence does not certify a particular Python environment or substitute for recording actual execution. It omits automatic differentiation, batching, and optimized kernels.

```
import math
from numbers import Real

def matrix_shape(matrix, name):
    if not matrix or not matrix[0]:
        raise ValueError(f"{name} must be nonempty")
    width = len(matrix[0])
    for row in matrix:
        if len(row) != width:
            raise ValueError(f"{name} is ragged")
        if any(not isinstance(x, Real) or not math.isfinite(x)
               for x in row):
            raise ValueError(f"{name} needs finite real entries")
    return len(matrix), width

def matmul(left, right):
    rows, inner = matrix_shape(left, "left")
    right_rows, columns = matrix_shape(right, "right")
    if inner != right_rows:
        raise ValueError("matrix dimensions do not align")
    return [[sum(left[i][k] * right[k][j]
                  for k in range(inner))
             for j in range(columns)]
            for i in range(rows)]
```

```

def masked_softmax(scores, allowed):
    if not scores or len(scores) != len(allowed):
        raise ValueError("scores and mask must have matching lengths")
    if any(type(a) is not bool for a in allowed):
        raise ValueError("allowed entries must be boolean")
    if not any(allowed):
        raise ValueError("at least one source must be allowed")
    if any(not isinstance(s, Real) or not math.isfinite(s)
            for s in scores):
        raise ValueError("input scores must be finite")
    shift = max(s for s, a in zip(scores, allowed) if a)
    terms = [math.exp(s - shift) if a else 0.0
              for s, a in zip(scores, allowed)]
    denominator = sum(terms)
    return [term / denominator for term in terms]

def attention(Q, K, V, allowed):
    nq, dk = matrix_shape(Q, "Q")
    nk, key_width = matrix_shape(K, "K")
    nv, _ = matrix_shape(V, "V")
    if dk != key_width:
        raise ValueError("Q and K widths must agree")
    if nk != nv:
        raise ValueError("K and V source counts must agree")
    if len(allowed) != nq or any(len(row) != nk for row in allowed):
        raise ValueError("mask dimensions must be queries by sources")
    scale = math.sqrt(dk)
    scores = [[sum(Q[i][k] * K[j][k] for k in range(dk)) / scale
                for j in range(nk)] for i in range(nq)]
    weights = [masked_softmax(row, mask)
                for row, mask in zip(scores, allowed)]
    masked_scores = [[s if a else float("-inf")
                      for s, a in zip(row, mask)]
                     for row, mask in zip(scores, allowed)]
    output = matmul(weights, V)
    return {"scores": scores, "masked_scores": masked_scores,
            "weights": weights, "output": output}

def baseline_inputs():
    X = [[1.0, 0.0], [0.0, 1.0], [1.0, 1.0]]
    c = math.sqrt(2.0) * math.log(2.0)
    WQ = [[1.0, 0.0], [0.0, 1.0]]
    WK = [[c, 0.0], [0.0, c]]
    WV = [[2.0, 0.0], [0.0, 4.0]]
    allowed = [[True, False, False],
                [True, True, False],

```

```

        [True, True, True]]
    return matmul(X, WQ), matmul(X, WK), matmul(X, WV), allowed

if __name__ == "__main__":
    Q, K, V, allowed = baseline_inputs()
    baseline = attention(Q, K, V, allowed)
    for name, rows in baseline.items():
        print(name)
        for row in rows:
            print(row)

```

For example, the value-ablation case uses fresh inputs, sets $V[1] = [0.0, 0.0]$, and calls `attention` again. Position 2 has index 1 in Python. The query case sets $Q[2] = [1.0, 0.0]$; connection removal sets $allowed[2][1] = False$; the future-key case sets $K[2] = [2.0 * x \text{ for } x \text{ in } K[2]]$. Start with `baseline_inputs()` before each case.

25.8 Optional checks with explanatory value

A shared value shift. Add $c = [1, -2]$ to every value vector, holding scores and masks fixed. Predict the change in each output. Since the weights sum to one, every correct output increases by c . The buggy implementation increases row i by its surviving coefficient sum times c . This tests normalization through output behavior.

Different maps, equal mixtures. Use separate value vectors $[[2, 0], [2, 0], [4, 0]]$. Compare row weights $[1/4, 1/4, 1/2]$ and $[3/8, 1/8, 1/2]$. Both yield $[3, 0]$ because the first two values are identical. This is an explicitly constructed counterexample, not a claim that arbitrary changes to every attention map leave output unchanged.

Score offsets. Add 1000 to every finite score in one row before masking and softmax. A numerically stable implementation should preserve that row's weights within tolerance. Explain why multiplying every score by 1000 would be a different operation.

25.9 Completion criteria

Your submission should let another learner check:

- the original hand trace and predictions;
- correct normalization and causal masking, including the deliberately buggy comparison;
- the four independent interventions and actual outputs or clearly labeled analytical checks;
- rejection of the listed invalid inputs for the coding route;
- preservation of the baseline after interventions;
- an explanation distinguishing mixture coefficients from transmitted values;
- a conclusion limited to the computation you actually investigated.

Keep your first attempt, corrections, and remaining questions. A useful result is a trace you can explain, including why a mistaken implementation produced its particular wrong answer.

26 Inspect MLP Responses and Feature Geometry

26.1 Goal

Trace how hidden activations become a residual contribution, test a small feature dictionary, and practice limiting an interpretation to its evidence. An optional extension compares actual MLP responses across contrasting prompts in a trained Transformer.

Prerequisite: Lesson 2.3, including its two numerical constructions.

Time: 60–75 minutes for the hand-worked core. Allow additional setup and analysis time for the optional model observation.

Core requirements: paper or a text editor. Parts 1–3 need no account, API, network, model download, GPU, paid compute, or machine-learning package. Optional toy code uses only ordinary Python on a CPU.

Expected artifact: a short `submission.md` or equivalent note with predictions, calculations, a sketch, explanations, and limitations. If you run code, include the actual output and environment details. The provided analytical answers are not evidence of software execution.

26.2 Part 1 — Trace a new MLP input

Reuse the lesson’s matrices and biases, but use a fresh input:

$$x = \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix}, \quad W_{\text{in}} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & -1 & 1 \\ 1 & 0 & -1 \\ -1 & 1 & 1 \end{bmatrix}, \quad b_{\text{in}} = \begin{bmatrix} 0 \\ -1 \\ 1 \\ 1 \end{bmatrix},$$
$$W_{\text{out}} = \begin{bmatrix} 1 & 0 & -1 & 1 \\ 0 & 1 & 0 & -2 \\ 1 & -1 & 2 & 0 \end{bmatrix}, \quad b_{\text{out}} = \begin{bmatrix} 0 \\ 1/2 \\ 0 \end{bmatrix}.$$

As in the lesson’s toy, there is no normalization: $u = x$. Apply $z = W_{\text{in}}x + b_{\text{in}}$, $h = \text{ReLU}(z)$, $\Delta = W_{\text{out}}h + b_{\text{out}}$, and $y = x + \Delta$.

Before calculating, predict which units will be positive and whether the first residual coordinate will increase, decrease, or remain unchanged. Then record all four intermediate vectors and their shapes. Expand Δ into the separate column contributions $h_j c_j$ plus the output bias. Keep zero contributions visible.

For an immediate score, choose $s = y_1 + y_3$. Reset to baseline for each intervention:

1. Set only h_1 to zero after ReLU.
2. Set only h_3 to zero after ReLU.
3. Set only h_2 to zero after ReLU.

Predict the new y and s before computing them. Which equally activated units have unequal effects on the score? Explain using their output columns rather than their activation values alone. Does the zero effect of intervention 3 establish that unit 2 is irrelevant on other inputs?

These are activation replacements, not weight changes. Because this toy has only an affine output map after the replacement, its immediate effect can be calculated exactly by subtracting one contribution. In a deeper model, later computations must be rerun.

26.3 Part 2 — Test a dictionary under different feature combinations

Use the hand-designed dictionary

$$D = \begin{bmatrix} 1 & -1/2 & -1/2 \\ 0 & \sqrt{3}/2 & -\sqrt{3}/2 \end{bmatrix}, \quad r = Da, \quad \hat{a} = \text{ReLU}(D^T r).$$

The three entries of a are nonnegative feature strengths. They are known generating variables in this construction. The two entries of r are representation coordinates, not two semantic feature labels. No weights are learned in this part.

Sketch the three columns as arrows and verify their lengths and pairwise dot products. Predict which of these inputs will reconstruct exactly:

- $a = [0, 2, 0]^T$;
- $a = [1, 0, 1]^T$;
- $a = [1, 2, 0]^T$;
- $a = [1, 1, 1]^T$.

For each input, calculate r , the pre-ReLU readout $D^T r$, the reconstruction $\hat{a}$, and squared error $\| \hat{a} - a \|^2$. Do not hide the negative entries before explaining what ReLU removes.

Then answer:

1. Why does singleton recovery not contradict the fact that three independent directions cannot fit in two dimensions?
2. Which failure could potentially be improved by choosing a different decoder, and which pair of inputs creates an unavoidable ambiguity for any decoder receiving only r ?
3. If inputs are usually singletons but occasionally dense, what additional information is needed to calculate average reconstruction error?
4. Is a sparse feature-strength vector necessarily a sparse representation vector? Use a displayed example.

For question 3, specify probabilities and strength distributions; merely saying “usually sparse” is insufficient to calculate an expectation. You do not need to invent or fit such a distribution.

26.4 Part 3 — Separate a distributed code from superposition

Use the orthogonal dictionary with columns

$$p = \frac{1}{\sqrt{2}}[1, 1]^T, \quad q = \frac{1}{\sqrt{2}}[1, -1]^T.$$

Encode feature strengths $(2, 1)$ as $r = 2p + q$. Recover them with the two dot products. Explain why observing that the first coordinate responds to both generating features is insufficient evidence of

superposition. In this example, the complete representation has enough independent directions for the two features.

Write two short claims about the triangle construction: one that your calculations establish and one that would require observing a trained network. Avoid turning “this geometry can work on these inputs” into “language models learned this exact geometry.”

26.5 Part 4 — Optional standard-library implementation

Implement checked matrix–vector multiplication, ReLU, vector addition, dot products, and squared error. Reject incompatible shapes instead of silently truncating arrays. Preserve immutable baseline values or copies so one intervention cannot change another case’s starting point.

Verify every Part 1 vector against the hand trace. For Part 2, report numerical error against the analytical answers with a declared tolerance, such as absolute error 10^{-12} for these small double-precision calculations. Near-zero roundoff from $\sqrt{3}$ should not be mislabeled a newly discovered feature.

The downloadable toy runner implements these constructions using only the standard library. Save it locally and run `python3 trace_mlp_geometry.py` after making your predictions. Save the program, invocation, Python version, and actual JSON output. A passing test checks implementation against this construction; it does not establish that a trained language model uses the same representation.

26.6 Part 5 — Optional contrasting-prompt observation

26.6.1 Choose the documented specimen

Reuse the local artifact from Lab 05, if available: EleutherAI/Pythia-14M at revision 94f7c35d5e9f2e9bac8ca839329f505b4d007d5d. The publisher’s immutable commit specifies six blocks, residual width 128, MLP width 512, GELU, and parallel residuals. The model card documents a February 2026 checkpoint replacement; do not substitute an unrecorded floating revision.

If downloading for the first time, allow only `config.json`, `tokenizer.json`, `tokenizer_config.json`, `special_tokens_map.json`, `generation_config.json`, and `model.safetensors` from that revision. The weights alone occupy 28,143,920 bytes, roughly 28.1 MB; inspect the total including tokenizer and configuration files before downloading. Do not fetch optimizer state, every training checkpoint, `.bin` weights, or custom Python files. Dependency installation can require much more disk space than the model weights.

The following requirements describe the supplied runner and any independent implementation. No expected trained-model activation values are supplied here. Use CPU float32 computation, evaluation mode, inference mode without gradients, and `use_cache=False`. Use built-in `GPTNeoXForCausalLM`, `safetensors` weights, a fast tokenizer, and `trust_remote_code=False`. After the bounded download, load locally with `local_files_only=True`. No paid inference service or API is needed.

Pin and record compatible, tested versions of Python, PyTorch, Transformers, tokenizers, safetensors, and the download client. The inspected source reference is Transformers v4.57.1. Recheck module boundaries if using a different version; do not enable remote custom code to repair a loading failure.

26.6.2 Local runner and two observation phases

Reuse the separate environment, pinned requirements and verified artifact directory from Lab 05. Save the MLP observation runner and its sibling specimen helper together. Both must be in the same directory. The tested versions and CPU eager backend match Lab 05; its download plan and six-file hashes apply here too. This runner requires the already downloaded artifacts and loads them offline.

First save your block 0 and block 2 structural predictions in `my-structure.txt`. Then run only the discovery pairs:

```
.venv-residual/bin/python inspect_mlp_responses.py discover --artifacts pythia-artifacts --structural-predictions my-structure.txt --output mlp-discovery
```

Read `discovery.json` and its complete captures. The runner selects three indices using the stated discovery contrast and tie-break rule. It also creates `predictions-template.json`. Copy that template to `my-predictions.json`. Keep its selected indices unchanged; fill the three directions entries with -1, 0, or 1, then write your reasoning and `structural_predictions`. These are your hypotheses, not automatic labels. Save this file before evaluating held-out prompts.

Only then run:

```
.venv-residual/bin/python inspect_mlp_responses.py reveal --artifacts pythia-artifacts --discovery mlp-discovery --predictions my-predictions.json --output mlp-heldout
```

The reveal phase validates the frozen discovery selection, writes your predictions to its output directory before inference, and evaluates only pairs 3 and 4. It retains every selected-unit value and contrast sign, including failures, without reselection. Manifest files record versions, specimen hashes and settings. Full prompt/token records and signed preactivation, post-activation and MLP output arrays remain in the local JSON files. Nonempty result directories are never overwritten. The phase split supports an honest sequence; it cannot prove that a person never inspected held-out data elsewhere.

For the additional surface-pattern challenge below, write exactly two new contexts as a JSON array in `challenge-contexts.json`. Copy your prediction file, preserve selected indices, and record your predicted directions and rationale for this new contrast. Both contexts receive the same suffix automatically; the runner limits each to 256 tokens.

```
.venv-residual/bin/python inspect_mlp_responses.py challenge --artifacts pythia-artifacts --discovery mlp-discovery --predictions challenge-predictions.json --contexts challenge-contexts.json --output mlp-challenge
```

The first context minus the second defines the contrast. This is a user-designed challenge, not another selection set or an automatic validation claim.

No expected trained-model activations are published. Maximum boundary errors, hook cleanup, repeatability and same-ID checks are recorded from the actual machine run. A failed structural check should prompt inspection before interpretation.

26.6.3 Fix prompts and a position before looking at activations

Use these four code/prose pairs as a small exploratory set. Each prompt ends in the identical suffix `\nSummary:.` Here `\n` denotes one actual newline, not two literal characters. These examples are original prompts, not known activation triggers.

1. Discovery pair 1

- Code context: `x = 2 + 3\nprint(x)`
- Prose context: Mira added two and three and wrote the total.

2. Discovery pair 2

- Code context: `items = [1, 2, 3]\nlen(items)`
- Prose context: Mira counted three items on the desk.

3. Held-out pair 3

- Code context: `names = ['Ada', 'Lin']\nnames[0]`
- Prose context: Mira chose Ada from a list of two names.

4. Held-out pair 4

- Code context: `word = 'blue'\nword.upper()`
- Prose context: Mira rewrote the word blue in capital letters.

Use the prose text exactly as written after its label, including its final period. Append the common suffix to every code and prose context.

Tokenize each complete prompt with `add_special_tokens=False`, no chat template, and no truncation. Run prompts individually to avoid padding. Save exact text, token IDs, decoded pieces, and zero-based positions. Choose the final token of the common suffix; verify that its token ID is identical across all eight prompts. If it is not, resolve the alignment and record the change before observing activations.

Matching that final ID controls one variable. It does not equalize prefix length, token positions, vocabulary, punctuation, or task demands. Record these differences as possible explanations. Do not call the pairings a complete semantic control.

Before measuring, predict:

1. Should the final-position MLP activation in zero-indexed block 0 differ between prompts with the same final token ID?
2. Could zero-indexed block 2 differ even though its MLP still operates separately at each position?
3. If a unit looks code-selective on the discovery pairs, what held-out result would weaken that interpretation?

26.6.4 Name and verify the capture boundary

Inspect GPTNeoXMLP in the linked implementation. At the chosen block, capture:

- z : output of `m1p.dense_h_to_4h`, before GELU;
- h : input to `m1p.dense_4h_to_h`, after GELU;
- Δ : output of the whole `m1p`, before residual addition.

An observation-only forward pre-hook on `dense_4h_to_h` exposes h . It must leave inputs unchanged and return `None`. Clone recorded tensors, clear captured values between runs, and remove hook handles

in a finally block, including when a check fails. Verify the actual return types rather than assuming every module returns a tuple.

Use a discovery prompt for the initial hook checks and repeatability tests; do not inspect held-out activations yet. Capture block 0 for the structural check and block 2 for the comparison; do not search all layers for the most flattering result. Expect shapes $(1, n, 512)$ for z, h and $(1, n, 128)$ for Δ . Check $h = \text{GELU}(z)$ using the model’s actual activation function, and reconstruct Δ with the stored output map and bias. Report maximum absolute discrepancies and declared tolerances, not just a pass label. Repeat one unchanged prompt and report the largest difference between runs.

GELU can produce negative values. Preserve signed activations; do not silently replace it with ReLU or treat all nonzero coordinates as equivalent active features.

26.6.5 Explore, freeze a prediction, then check

For block 2 and the aligned final position, calculate each unit’s mean paired discovery contrast:

$$\delta_j = \frac{1}{2} \sum_{k=1}^2 (h_{j,\text{code},k} - h_{j,\text{prose},k}).$$

Choose the three units with largest $|\delta_j|$, resolving ties by smaller zero-based unit index. Keep their signs. This is an exploratory selection from 512 candidates, not a significance test.

Before opening the held-out activations, save the selected indices, discovery values, and predicted contrast direction for each. Evaluate those same indices on pairs 3 and 4 without reselection. Report each pair’s values and signed difference, including failures. If held-out observations were viewed while selecting units, relabel the analysis exploratory and reserve genuinely fresh prompts for a later check.

Next write one additional contrast for a selected unit that challenges a surface explanation: for example, code-like punctuation in ordinary prose, or the same simple operation described without programming syntax. Save your prediction before running it. One such check is a useful challenge, not a validation suite.

Report one observation, two plausible explanations, and a test that would distinguish them. Do not claim that three selected coordinates are three discovered semantic features, that an activation contrast proves superposition, or that this tiny checkpoint represents larger models’ behavior. Real-model ablation is not required here; the toy already provided a fully accountable intervention.

26.6.6 Explain the first-block control

In this checkpoint, the initial residual state comes from token embeddings, and rotary position handling occurs inside attention. Parallel residuals make block 0’s MLP read a normalized version of that initial state, before the same block’s attention contribution is added. With the same target token ID and evaluation settings, its MLP therefore receives the same target vector across these prompts. This predicts matching target activations within numerical tolerance. A discrepancy should trigger a boundary, tokenization, or implementation check before a semantic story.

Block 2 reads states that preceding blocks have already contextualized. Its activations may differ. This comparison tests your understanding of the actual wiring as well as your hooks. Neither matching nor differing values have been supplied as observed results.

26.7 Analytically derived checkpoints

i Core solutions

For Part 1, $z = h = [2, 0, 2, 0]^T$. The nonzero column contributions are $2c_1 = [2, 0, 2]^T$ and $2c_3 = [-2, 0, 4]^T$. Thus $\Delta = [0, 1/2, 6]^T$ and $y = [2, 1/2, 7]^T$; baseline $s = 9$.

- Zero h_1 : $y = [0, 1/2, 5]^T$ and $s = 5$.
- Zero h_3 : $y = [4, 1/2, 3]^T$ and $s = 7$.
- Zero h_2 : baseline is unchanged, since $h_2 = 0$ on this input.

Units 1 and 3 both activate at two, but their score contributions are four and two. Unit 2's zero effect is conditional on this input and intervention.

For Part 2:

- $[0, 2, 0]$: $r = [-1, \sqrt{3}]^T$; readout before ReLU $[-1, 2, -1]^T$; reconstruction $[0, 2, 0]^T$; squared error 0.
- $[1, 0, 1]$: $r = [1/2, -\sqrt{3}/2]^T$; readout $[1/2, -1, 1/2]^T$; reconstruction $[1/2, 0, 1/2]^T$; squared error $1/2$.
- $[1, 2, 0]$: $r = [0, \sqrt{3}]^T$; readout $[0, 3/2, -3/2]^T$; reconstruction $[0, 3/2, 0]^T$; squared error $5/4$.
- $[1, 1, 1]$: $r = [0, 0]^T$; readout and reconstruction are zero; squared error 3.

Singleton recovery uses a restricted input family. All-one and all-zero inputs give the same representation, so no alternative decoder can distinguish them from r alone. Decoder improvements for selected mixtures do not remove this ambiguity. B alone supplies an example of sparse feature strengths with two nonzero representation coordinates.

For Part 3, $r = [3/\sqrt{2}, 1/\sqrt{2}]^T$. Its projections onto p and q recover two and one. The code is distributed but needs no overcomplete dictionary.

26.8 Submit evidence and an explanation

Include your original predictions, corrected MLP trace, three intervention outcomes, dictionary calculations, sketch, and distributed-code comparison. State explicitly that the numerical constructions were hand-designed.

For executed toy code, add its source and real output. For optional trained-model work, include checkpoint revision, dependency versions, settings, token records, hook boundaries, shape and numerical checks, discovery selections, predictions made before held-out inspection, complete held-out values, and limitations. Save the full final-position activation vectors so the selected-unit report can be checked against them.

A complete hand-worked core is a valid submission. After this Lab, return to Lab 04, Part 2 to connect the MLP's internal calculation to the original Transformer's architecture.

26.9 More Learning

- Lesson 2.3 supplies the distinction between units, features, and superposition.

- Toy Models of Superposition studies learned toy representations. Identify the training setup before comparing it with this Lab’s chosen dictionary.
- Pythia-14M’s pinned publisher commit and GPT-NeoX source establish the optional specimen and capture locations.

27 Read Model Configurations and Estimate Memory

27.1 Goal

Read two small public configuration sources, translate their fields into architecture, and estimate weight and KV-cache payloads. The central skill is separating what a file states, what an implementation establishes, what a calculation assumes, and what an actual run would have to measure.

Prerequisite: Lesson 2.4 — From the Original Transformer to Modern LLMs, especially its head-count and memory sections.

Time: about 60–90 minutes, with another 20–30 minutes for the optional calculator.

Requirements: a browser or previously saved copies of the public text sources, a text editor, and ordinary arithmetic. Optional coding uses an existing Python 3 installation and the standard library on a CPU. No model weights, tokenizer download, machine-learning library, account, API credential, GPU, or paid compute is needed.

Expected artifact: `submission.md` containing your original predictions, evidence table, architecture sketches, memory worksheet, and conclusions. If you implement the optional calculator, also retain its source, input data, and actual output.

! Predict before calculating

Save your predictions before reading the answer section or running a calculator. Keep incorrect predictions and explain their corrections. A numerical answer copied from this Lab is an analytical reference, not evidence of a software run.

27.2 Part 1 — Establish exactly which sources you read

Use these fixed references:

1. **GPT-2 configuration:** open `config.json` in the configuration repository at revision 7979bb7. The full revision is 7979bb77dc27f20ed20949aa4b91db714c64751f. Read or save only `config.json`, a 665-byte text file. Do not use the page’s model-loading examples or download the repository’s weights.
2. **Gemma 3 1B configuration:** read Google’s `gemma/config.py` at revision cb7c015. The full revision is cb7c0152a369e43908e769eb09e1ce6043afe084. Locate `get_config_for_1b`, then inspect any needed `GemmaConfig` defaults and the `get_model_config` caller. This is a Python configuration factory; read it as text. Do not import or execute downloaded source.
3. **GPT-2 interpretation reference:** use OpenAI’s `src/model.py` at revision c2dae27, full revision c2dae27c1029770cea40978813f17a5fd545b883. Read the `attn`, `block`, and `model` definitions only as needed to interpret the configuration. No TensorFlow installation is needed.

4. **Gemma interpretation reference:** read Google’s `gemma/model.py` at the same `cb7c015` revision when a claim requires implementation evidence. `GemmaAttention`, `Gemma2DecoderLayer`, `GemmaModel`, and the cache-allocation portion of `generate` establish the relevant boundaries. Despite its class name, that decoder-layer implementation is also selected for Gemma 3. Read only; no PyTorch installation or execution is required.

The two config sources have different forms. GPT-2’s file is serialized model metadata. Gemma’s file constructs variant configurations from defaults and overrides. Do not treat an uncalled factory as a loaded model or copy a different variant’s fields. The schemas are not interchangeable: the Hugging Face JSON names `vocab_size` and `n_positions`, while the historical OpenAI implementation constructs its token and position tables using `n_vocab` and `n_ctx`. Map these by meaning rather than passing the JSON unchanged to that code. OpenAI’s generic `n_vocab=0` is a placeholder; our specimen supplies vocabulary size 50257. The selected JSON’s position fields and the chosen position-table length agree at 1024.

If a source cannot be retrieved, record that limitation. You may complete the arithmetic using the lesson’s stated values, but label that route as analysis of supplied values rather than direct source verification. Do not silently switch to a similarly named model or an unpinned main revision.

Make an evidence table with these columns:

Claim or quantity	Exact field or source location	Value	Evidence category	Limitation
Example: cache bytes per scalar	Worksheet assumption	2	Assumed	Does not measure a running cache

Use four evidence categories: **stated in source**, **established by implementation**, **derived**, and **assumed**. An observed runtime category is available only if you actually performed a relevant measurement; this core Lab loads no model.

27.3 Part 2 — Translate keys into a computation

For both specimens, record:

- residual width and block count;
- query-head count, KV-head count, and head width;
- vocabulary size and position-limit field;
- positional mechanism and normalization type;
- local/global attention pattern, if any;
- the source of any dtype value or default you report.

For GPT-2, mark facts that require the implementation rather than a dedicated JSON key. The absence of `num_key_value_heads` does not itself prove a value. For Gemma, distinguish fields set inside `get_config_for_1b` from values supplied by its caller or inherited from the configuration class. Do not run the factory to discover them.

Sketch each model’s attention projections with row-vector matrix dimensions. Show the residual width, total query width, total key width, and the width returned by the output projection. Label whether each uses multi-head, grouped-query, or multi-query attention under the definitions in Lesson 2.4.

Answer these questions before checking your sketch:

1. Must residual width equal query-head count times head width?
2. Does sharing a KV pair force the associated query heads to have equal attention probabilities?
3. Which configuration value would determine how many learned positions GPT-2 can index? Why does a RoPE model require a different positional explanation?
4. Does the `get_model_config` caller’s default dtype prove the dtype of a model loaded by another program?
5. If the factory declares a local-attention pattern, does that prove that the serving engine evicts old local KV entries?

For the mixed Gemma pattern, number the layers from 1 through 26 and mark each local or global. Keep this list: it will determine the memory calculation.

27.4 Part 3 — Predict how cache memory changes

Before calculating bytes, predict the effect of each change while keeping the other worksheet assumptions fixed:

- batch size increases from one independent sequence to two;
- stored sequence length doubles under full-length storage;
- KV heads increase from one to four at unchanged head width;
- weights become four-bit while the cache remains BF16;
- local layers retain only their sliding window rather than full-length arrays.

Distinguish a change in the arithmetic model from a valid modification of an existing trained checkpoint. In particular, replacing a KV-head count in a worksheet does not convert a model’s learned projections.

Use the following cache assumptions throughout the baseline:

- keys and values both have the specified head width;
- two bytes per stored scalar;
- no quantization metadata, alignment, temporary tensors, or allocator overhead;
- no cross-sequence prefix sharing, device replication, or additional hypotheses such as beams;
- batch one, unless a question changes it;
- all mentioned cache entries have already been processed and are retained.

27.4.1 A — GPT-2 full-length payload

Use 12 layers, 12 KV heads, width 64, and 1024 stored positions. Write the logical shape of one layer’s key array and value array. Compute total bytes, MiB, and GiB from

$$M_{KV} = 2BLS h_{kv} d_h b.$$

Then change only cache precision to four bytes per scalar. Does the parameter count change? Does this calculation establish which cache precision any particular loading command would choose?

27.4.2 B — Gemma full-length payload

Use 26 layers, one KV head, width 256, and 32,768 stored positions. Calculate full-length payload in bytes and GiB. Repeat with four KV heads as a clearly labeled hypothetical architecture. Explain why substituting four **query** heads in the original calculation would be an error.

27.4.3 C — Gemma with bounded local retention

Now assume an engine keeps only the latest 512 positions in every local layer, but retains the full prefix in each global layer. Use your layer list to compute

$$M_{KV} = 2Bh_{kv}d_hb [L_gS + L_\ell \min(S, 512)].$$

Evaluate this at $S = 1024$ and $S = 32768$. Compare it with full-length allocation at the same two lengths. Predict which part continues growing once the local layers have reached 512 entries.

These are two different storage assumptions for the same attention pattern. The lesson documents that Google’s pinned reference generator allocates full-length arrays. The bounded calculation estimates a retention policy that another suitable engine could implement; it is not a measurement of that reference program.

27.5 Part 4 — Estimate weights without loading them

Use the GPT-2 specimen to count unique learned scalars. Its residual width is $d = 768$, vocabulary size is $V = 50257$, position-table length is $T = 1024$, and layer count is $L = 12$. The source has a fourfold MLP expansion, ordinary attention projections with biases, two affine LayerNorms per block, a final affine LayerNorm, and a vocabulary readout tied to the token embedding.

Derive this count instead of treating the marketed size as exact:

$$P = Vd + Td + L(12d^2 + 13d) + 2d.$$

Explain every term:

- token and learned-position embeddings;
- four attention matrices and two MLP matrices per block;
- attention and MLP biases;
- normalization scales and offsets;
- the absence of an additional independently stored vocabulary readout matrix.

Compute weight-only payload for FP32, FP16/BF16, ideal packed eight-bit, and ideal packed four-bit storage. Use integer byte counts before converting to GiB. For the quantized rows, explicitly exclude metadata, mixed-precision exceptions, and padding.

Add the FP16/BF16 weight payload to the two-byte GPT-2 cache payload from Part 3A. Explain why even this combined number remains insufficient as a minimum hardware recommendation.

Finally, consider a separate toy quantization layout with 1024 scalars: four-bit values packed without padding, groups of 64 values, one two-byte scale per group, and no zero points or other metadata.

Calculate value bytes, scale bytes, and their total. Which assumption would fail if some tensors stayed FP16?

27.6 Part 5 — Optional standard-library calculator

Implement a small calculator only after finishing your hand worksheet. The requirements below also describe the supplied standard-library calculator. Save it after completing your worksheet; the published analytical answers remain separate from your actual output.

Create your own compact JSON input files containing the normalized architecture fields needed for the calculations and a source URL plus revision for each specimen. Preserve the original field names in your evidence table. Label your JSON as a learner-created transcription; it is not a publisher's complete config.

The calculator should:

1. Read only these local JSON files; perform no downloads and no model loading.
2. Require explicit `query_heads`, `kv_heads`, and `head_dim`. Do not infer Gemma's head width from division. Require positive integer dimensions and reject non-divisible query/KV grouping for this simplified calculator.
3. Calculate full-length and mixed-retention KV payloads with an explicit per-scalar byte count, layer counts, batch, and sequence length. Require nonnegative local and global layer counts, allow zero local layers for an all-global model, and check that their sum equals the positive total layer count.
4. Calculate ideal weight payload from an explicit unique-parameter count and bits per scalar. Reject non-byte-aligned packed totals or state an explicit ceiling rule rather than silently discarding fractional bytes.
5. Print assumptions, exact bytes, MiB, and GiB separately. Label all results as calculated payloads, not measured RAM or VRAM.
6. Test that batch doubling doubles cache bytes, changing one KV head to four quadruples them, and changing weight bits alone leaves a separately specified cache unchanged.
7. Test a short sequence below the local window: at $S = 128$, bounded-local and full-length payloads should match under otherwise identical dimensions.

27.6.1 Running the supplied tools

Save `calculate_memory.py` into your Lab directory. The normalized JSON schema uses `residual_width`, `layers`, `query_heads`, `kv_heads`, `head_dim`, `batch`, `sequence_length`, `cache_bytes_per_scalar`, `local_layers`, and `global_layers`. Add `local_window` when local layers exist. For weight estimates, supply `unique_parameters` and `weight_bits_per_scalar` together. Include source with `url` and the complete revision. Each numerical field must be an integer under the validation rules above. These fields are your transcription and assumptions; they are not a publisher config.

```
python3 calculate_memory.py my-normalized-config.json > calculated-payloads.json
```

The program was tested on CPython 3.13.13 and uses no dependencies or network. JSON output separates exact bytes, MiB, GiB, logical projection shapes and exclusions. A zero local-layer count is valid for GPT-2; residual width need not equal total query width.

For optional retrieval, save `read_config_sources.py`. It retrieves exactly five pinned text files: the two configurations, the two interpretation references above, and Lab 09's gpt-oss config. The total is **48,054 bytes**. It checks each fixed size and SHA-256, retains original bytes and embedded notices, and writes provenance. It never imports the Python reference files or retrieves weights/tokenizer files. Read downloaded `.py` files in your text editor.

```
python3 read_config_sources.py --plan
python3 read_config_sources.py --output config-references
python3 read_config_sources.py --offline --output config-references
```

Cached references are verified rather than silently replaced; altered or unexpected files fail. Initial retrieval requires internet; the calculator and verified cached references work offline. Consult each publisher's full license in its pinned repository.

Retain your command, Python version, actual output, and any corrections. A match confirms your calculator against these cases; it does not benchmark an inference engine. Never execute the downloaded model implementation as a shortcut.

27.7 Checkpoints and analytical answers

i Compare after completing your worksheet

Architecture. GPT-2 has residual width 768, 12 query and KV heads, and head width 64. Its W_Q, W_K, W_V, W_O each have logical shape 768×768 . Gemma 3 1B has residual width 1152, four query heads, one KV head, and explicit head width 256. Its W_Q is 1152×1024 , W_K, W_V are 1152×256 , and W_O is 1024×1152 . Library storage may transpose these logical matrices or fuse Q, K, and V into one projection. Count the same learned scalars only once.

Positions and layers. GPT-2 uses a learned position table. The selected JSON reports `n_positions: 1024`; the paired historical OpenAI implementation uses `n_ctx` to size that table. These names identify the same 1024-position assumption here, not a shared input-file schema. Gemma's factory specifies 32,768 positions and a repeating local/local/local/local/local/global pattern. With one-based numbering, global layers are 6, 12, 18, and 24. The other 22 are local. Query heads sharing KV can still produce different distributions.

Cache A. Each GPT-2 key or value array has logical shape (1, 12, 1024, 64). Across 12 layers at two bytes per scalar, the total is 37,748,736 bytes, 36 MiB, or 0.03515625 GiB. Four-byte cache scalars double this to 75,497,472 bytes. Neither calculation changes learned parameter count.

Cache B. Gemma full-length storage is 872,415,232 bytes, or 0.8125 GiB. The hypothetical four-KV-head version gives 3,489,660,928 bytes, or 3.25 GiB. These are analytical payloads, not observations.

Cache C. At $S = 1024$, bounded local retention gives 15,728,640 bytes, or 15 MiB; full-length storage gives 27,262,976 bytes, or 26 MiB. At $S = 32768$, bounded retention gives 145,752,064 bytes, or 139 MiB, versus 832 MiB for full-length storage. Beyond the local window, only the four global layers' retained payload keeps growing under the bounded policy.

Weights. Attention contributes $4d^2$ matrix scalars per block, and a width- $4d$ two-matrix MLP contributes $8d^2$. Their biases contribute $4d + 5d$, and two LayerNorms contribute $4d$, totaling $13d$. The embedding, positional, and final-normalization terms complete $P = 124,439,808$ unique learned scalars.

Weight assumption	Exact payload bytes	Approximate GiB
FP32	497,759,232	0.463574
FP16 or BF16	248,879,616	0.231787
Ideal packed eight-bit	124,439,808	0.115894
Ideal packed four-bit	62,219,904	0.057947

Two-byte weights plus the two-byte GPT-2 cache total 286,628,352 bytes, approximately 0.266943 GiB. This excludes activations, workspace, runtime overhead, and any additional tensors or copies. It is not measured process memory.

Toy quantization overhead. Packed values occupy 512 bytes. Sixteen two-byte scales add 32 bytes, giving 544 bytes total. Uniform four-bit storage fails if some tensors remain FP16; count those separately. Quantizing weights does not change an independently specified cache dtype.

Metadata limits. A dtype field, a cache flag, and a local-attention pattern do not establish the loaded tensor dtype, actual cache allocation, or eviction policy. Those require matching implementation evidence and, for an observed-state claim, a real measurement.

27.8 Submit a bounded conclusion

Finish with three short paragraphs:

1. Explain one architectural difference whose memory consequence you calculated.
2. Identify one tempting inference that the configuration alone does not justify.
3. State which additional evidence you would need to estimate peak inference memory or compare speed on a specific machine.

Include the original predictions, exact revisions, evidence categories, dimensions, assumptions, and units. If your optional calculator disagrees with the hand calculation, show the first divergent quantity and the correction. Do not invent loaded-model outputs for a Lab that never loads one.

27.9 More Learning

1. **GPT-2 configuration snapshot** Compare the small configuration file with the much larger checkpoint artifacts without downloading the latter.
2. **Google Gemma configuration factory** Practice resolving one variant's overrides and inherited defaults.
3. **OpenAI GPT-2 implementation** Verify the tied vocabulary readout and the parameter-count terms from their tensor shapes.
4. **Transformers cache strategies, v4.51.3** Compare growing, static, offloaded, and quantized cache policies. Separate memory placement and allocation from the model's attention mask.

28 Trace and Perturb Expert Routing

28.1 Goal

Produce a routing log that connects each token position and layer to router logits, selected experts, combining coefficients, expert outputs, and the residual update. Then distinguish four interventions and inspect a small, pinned gpt-oss configuration.

Prerequisite: Lesson 2.5 — Mixture of Experts, through the worked example. Read its accounting and interpretation sections before drawing conclusions.

Time: about 90–120 minutes. Allow another 20 minutes for the optional extension.

Requirements: paper or a text editor. The coding route uses an existing Python 3 installation, a CPU, and the standard library. The toy requires no network, model weights, API, account, package installation, GPU, or paid compute. Configuration inspection needs only a public text file; an attributed excerpt is supplied for an offline route.

Expected artifact: `submission.md`, a routing log, and an exact run specification. If coding, save `tiny_moe.py`, its generated `routing-log.jsonl`, and `run-spec.json`. If working by hand, reproduce the same information in a legible table. Keep predictions separate from results and distinguish hand-derived values from actual program output.

! Predict before executing

Complete Parts 1 and 2 before running the code or reading the solution. The scaffold intentionally raises errors. Compare your implementation with the reference functions only after attempting it.

28.2 The fixed two-layer toy

This is an invented arithmetic system. It has no tokenizer, attention, normalization, trained weights, vocabulary output head, or language-generation ability. We call its three indexed inputs “token positions” to practise the logging convention used for a real model. Its token IDs are arbitrary labels, not entries in a published tokenizer.

Each layer has four experts and selects two. Vectors are columns mathematically; Python stores their coordinates in ordinary lists. The experts in this Lab are linear maps rather than full MLPs. Every layer computes

$$z = W_r x + b_r, \quad S = \text{TopKIndices}(z, 2), \quad \alpha_S = \text{softmax}(z_S),$$

$$f_e(x) = A_e x, \quad m = \sum_{e \in S} \alpha_e f_e(x), \quad x_{\text{next}} = x + m.$$

Sort by descending logit, breaking equal scores by ascending expert ID. Selected-only softmax is mandatory. There is no router noise, expert capacity limit, token dropping, or parameter sharing between layers. Some specified coefficients are equal or zero; those are still separate stored coefficient slots.

Use these inputs:

Position	Toy token ID	Initial vector
0	11	$[1, 1]^T$
1	11	$[1, 0]^T$
2	17	$[0, 1]^T$

The repeated ID with different vectors is an explicit fixture. We are not simulating the context that could have produced those vectors in a trained model.

28.2.1 Layer 0

Use the lesson's router:

$$W_r^{(0)} = \begin{bmatrix} \ln 4 & 0 \\ 0 & \ln 3 \\ \ln 2 & 0 \\ 0 & 0 \end{bmatrix}, \quad b_r^{(0)} = [0, 0, 0, 0]^T.$$

Its experts are

$$\begin{aligned} f_0(x) &= [2x_1, 0]^T, & f_1(x) &= [0, 3x_2]^T, \\ f_2(x) &= [-x_1, x_2]^T, & f_3(x) &= [x_2, x_1]^T. \end{aligned}$$

28.2.2 Layer 1

Feed each layer-0 residual output into this layer. Use a zero 4×2 router matrix and

$$b_r^{(1)} = [0, \ln 2, \ln 4, 0]^T.$$

This deliberately input-independent router makes the second hand calculation small. Its expert functions differ from layer 0:

$$\begin{aligned} f_0(x) &= [x_2, 0]^T, & f_1(x) &= [x_1, 0]^T, \\ f_2(x) &= [0, 2x_2]^T, & f_3(x) &= [-x_1, -x_2]^T. \end{aligned}$$

Layer 1's expert 1 is therefore not layer 0's expert 1. Keep the layer index in every record.

28.3 Part 1 — Compute a baseline trace

For every position in layer 0:

1. Compute all four logits and the full softmax distribution p .
2. Rank the experts, naming any tie and its resolution.
3. Record the two selected IDs and their selected-only coefficients.
4. Evaluate only those two expert functions.
5. Compute the combined update and residual output.

Then compute layer 1 using the actual layer-0 outputs. There should be six routing records. Each record must contain:

- position, toy token ID, and layer;
- input vector and full router logits;
- full-router probabilities;
- selected expert IDs in rank order and selected-only coefficients;
- expert outputs before any output intervention;
- expert outputs actually combined;
- combined update and residual output;
- intervention name, or baseline.

Predict first: will the two occurrences of toy token 11 select the same experts in layer 0? Will a selected expert with a zero output disappear from the selection log? Can different expert outputs produce different updates while the route and coefficients remain fixed?

For each layer, count **assignments**, including both selected experts for every position. Explain why the counts sum to six, rather than three or one.

28.4 Part 2 — Specify and predict interventions

Restart from the original inputs for each scenario. Apply the intervention **only at layer 0, position 0**. All other positions and the layer-1 computation retain their baseline rules. Continue the altered residual vector through layer 1.

1. **Prohibit expert 0.** Remove ID 0 from the eligible set before selecting the best two remaining experts. Renormalize over those selected experts. Do not zero its output after ordinary selection.
2. **Force experts 2 and 3.** Override the selected set with exactly those IDs, retaining their original logits to calculate the selected-only weights. This is not equal weighting.
3. **Boost expert 2.** Add $\ln 4$ to only its router logit before selection. Recompute the ranking and selected-only weights. Leave expert matrices unchanged.
4. **Ablate expert 0's output.** Preserve baseline selection and coefficients. Replace only its selected output vector with zero. Do not renormalize or substitute another expert.

For each scenario, predict the selected IDs, coefficients, layer-0 update, and both residual outputs. Say which later router logits will change in **this particular toy**, and why. Do not generalize from layer 1's constant router to trained routers.

A control: add the same constant to every router logit. Predict what changes in raw scores, full probabilities, selection, and combined output.

28.5 Part 3 — Implement and save the trace

28.5.1 Data and learner scaffold

Copy the following into `tiny_moe.py`. Complete the three numerical functions; then follow the tracing recipe below.

```
import json
import math
import sys
from pathlib import Path
```

```

SPEC_ID = "tiny-moe-routing-v1"
K = 2
INPUTS = [
    {"position": 0, "token_id": 11, "x": [1.0, 1.0]},
    {"position": 1, "token_id": 11, "x": [1.0, 0.0]},
    {"position": 2, "token_id": 17, "x": [0.0, 1.0]},
]
LAYERS = [
    {
        "router_W": [
            [math.log(4), 0.0], [0.0, math.log(3)],
            [math.log(2), 0.0], [0.0, 0.0],
        ],
        "router_b": [0.0, 0.0, 0.0, 0.0],
        "experts": [
            [[2.0, 0.0], [0.0, 0.0]],
            [[0.0, 0.0], [0.0, 3.0]],
            [[-1.0, 0.0], [0.0, 1.0]],
            [[0.0, 1.0], [1.0, 0.0]],
        ],
    },
    {
        "router_W": [[0.0, 0.0] for _ in range(4)],
        "router_b": [0.0, math.log(2), math.log(4), 0.0],
        "experts": [
            [[0.0, 1.0], [0.0, 0.0]],
            [[1.0, 0.0], [0.0, 0.0]],
            [[0.0, 0.0], [0.0, 2.0]],
            [[-1.0, 0.0], [0.0, -1.0]],
        ],
    },
]

def matvec(matrix, vector):
    """Return matrix @ vector without mutating either input."""
    raise NotImplementedError("Compute a dot product for each row")

def softmax(values):
    """Subtract the largest input before exponentiating."""
    raise NotImplementedError("Return positive weights summing to one")

def choose(logits, eligible, k):
    """Descending score, then ascending ID; return exactly k IDs."""
    raise NotImplementedError("Validate the eligible count and sort")

```

The tracing function should perform these steps, in this order:

1. Calculate the raw affine router logits and preserve a copy.
2. For `boost_2`, add $\ln 4$ to logit 2. Preserve these effective logits separately.
3. Compute the diagnostic full softmax over all effective logits. In `prohibit_0`, this diagnostic still includes expert 0; the eligibility restriction affects selection, not this recorded diagnostic.
4. Set eligible IDs to `[1, 2, 3]` for `prohibit_0`, `[2, 3]` for `force_23`, and `[0, 1, 2, 3]` otherwise. Select two by the stated ranking rule.
5. Compute a new softmax on the selected effective logits only.
6. Calculate and save the selected expert outputs. For `ablate_0`, replace expert 0's output with zero in a separate copy used for combination.
7. Sum the weighted vectors, add the input residual, and return the complete record.

Run all three positions through both layers for each of five scenarios: baseline and the four interventions. Apply a scenario's intervention only to position 0 in layer 0. Keep the overall scenario label on the downstream records even when their local intervention is baseline.

Save one JSON object per line in `routing-log.jsonl`. The complete run should have 30 records. Save the input fixtures, layer coefficients, top- k , normalization convention, tie rule, scenario list, and Python version in `run-spec.json`. Save the source file too: floating-point values alone do not document every rule of the experiment.

28.5.2 Verification checks

Do not rely only on a final vector matching a reference. Check:

- four full probabilities sum to one for each record;
- two distinct, eligible expert IDs are selected;
- two combining coefficients sum to one;
- combining weights are paired with the correct expert outputs;
- the combined update reproduces the weighted sum;
- the output is the input plus that update;
- layer-1 inputs equal the corresponding layer-0 outputs in every scenario;
- all scenarios leave positions 1 and 2 unchanged;
- the zero-output selected expert at baseline layer 0, position 2 remains in the log;
- repeated runs give the same records, apart from deliberately recorded execution metadata.

Use a small floating-point tolerance, such as 10^{-10} , when comparing numerical results. Our exact fractions describe the mathematical answers; binary floating point may store nearby values.

28.6 Part 4 — Inspect one real configuration without loading a model

Open only the pinned `gpt-oss-20b config.json`. The repository is `openai/gpt-oss-20b`; the revision is `f81fef1ddd90d214968e951a76834f1ded130a18`. Reading the file in a browser is sufficient. Downloading this one small JSON file is also sufficient. Do not run model-loading examples or download weight shards for this Lab.

For an offline route, use this **selected-field transcription** from that pinned file. It is not the complete configuration or an independently observed local download:

```
{
  "model_type": "gpt_oss",
  "hidden_size": 2880,
  "intermediate_size": 2880,
  "num_hidden_layers": 24,
  "num_local_experts": 32,
  "num_experts_per_tok": 4,
  "experts_per_token": 4,
  "output_router_logits": false,
  "tie_word_embeddings": false,
  "vocab_size": 201088
}
```

Record whether you inspected the complete pinned file or used the excerpt. If you save the complete file, keep its source URL, revision, retrieval date, and SHA-256 digest. A digest documents the bytes you retained; an unverified digest does not independently prove publisher identity. There is no need to install a model library merely to read JSON.

Answer:

1. How many layer-specific expert networks exist across all 24 MoE layers? How many expert evaluations lie on one token’s full forward path?
2. What percentage of expert networks are selected at each layer? Why is this not the model’s whole active-parameter percentage?
3. Using three $d \times m$ -equivalent matrices per gated expert, calculate total and selected expert-matrix coefficients. Explicitly exclude biases and non-expert parameters.
4. Why can the configuration not tell you which expert a particular real prompt selects?
5. What additional evidence would be required to claim that router logits were actually captured?

Compare your counts with Table 1 of the gpt-oss model card. Record its published 20b totals and its active-count convention. Do not equate “32 experts per layer” with “32 complete language models.”

For the toy, count stored coefficient slots directly: each layer has a 4×2 router matrix, four router biases, and four 2×2 expert matrices. Count zeros as stored coefficients. How many slots are total? How many belong to the router and selected expert matrices along one token’s two-layer path? The calculation assumes generic dense evaluation of each selected matrix, even when it contains zeros or produces zero output.

Finally, estimate a hypothetical packed float32 storage size for all toy coefficients. Explain why that number is not the actual memory consumption of Python lists, nor the memory consumption of a trained-model runtime.

28.7 Reference functions

Compare only after attempting the scaffold and tracing recipe. These functions implement the toy specification; their output is not trained-model evidence. Retain your own run output when you execute them.

```

def matvec(matrix, vector):
    return [sum(a * b for a, b in zip(row, vector))
            for row in matrix]

def softmax(values):
    peak = max(values)
    exp_values = [math.exp(value - peak) for value in values]
    total = sum(exp_values)
    return [value / total for value in exp_values]

def choose(logits, eligible, k):
    if len(set(eligible)) != len(eligible) or len(eligible) < k:
        raise ValueError("Need at least k distinct eligible experts")
    return sorted(eligible, key=lambda e: (-logits[e], e))[:k]

def trace_one(x, layer_id, position, token_id, scenario):
    local = scenario if layer_id == 0 and position == 0 else "baseline"
    layer = LAYERS[layer_id]
    projection = matvec(layer["router_W"], x)
    raw = [a + b for a, b in zip(projection, layer["router_b"])]
    logits = raw[:]
    if local == "boost_2":
        logits[2] += math.log(4)

    eligible = list(range(4))
    if local == "prohibit_0":
        eligible = [1, 2, 3]
    elif local == "force_23":
        eligible = [2, 3]
    selected = choose(logits, eligible, K)
    weights = softmax([logits[e] for e in selected])
    original = [matvec(layer["experts"][e], x) for e in selected]
    used = [vector[:] for vector in original]
    if local == "ablate_0":
        used[selected.index(0)] = [0.0, 0.0]
    update = [sum(w * vector[j] for w, vector in zip(weights, used))
              for j in range(2)]
    output = [a + b for a, b in zip(x, update)]
    return {
        "scenario": scenario, "local_intervention": local,
        "position": position, "token_id": token_id,
        "layer": layer_id, "input": x[:],
        "raw_logits": raw, "effective_logits": logits,
        "full_probabilities": softmax(logits),
        "eligible": eligible, "selected": selected,
    }

```

```

        "combine_weights": weights,
        "expert_outputs_original": original,
        "expert_outputs_used": used,
        "update": update, "output": output,
    }

SCENARIOS = ["baseline", "prohibit_0", "force_23",
             "boost_2", "ablate_0"]
records = []
for scenario in SCENARIOS:
    for item in INPUTS:
        x = item["x"][:]
        for layer_id in range(len(LAYERS)):
            record = trace_one(x, layer_id, item["position"],
                              item["token_id"], scenario)
            records.append(record)
            x = record["output"][:]

spec = {
    "spec_id": SPEC_ID, "python": sys.version,
    "top_k": K, "layers": LAYERS, "inputs": INPUTS,
    "tie_rule": "descending logit then ascending expert ID",
    "combine_rule": "softmax over selected effective logits",
    "normalization": "none", "attention": "none",
    "capacity_limit": None, "scenarios": SCENARIOS,
    "intervention_site": {"layer": 0, "position": 0},
}
Path("run-spec.json").write_text(
    json.dumps(spec, indent=2) + "\n", encoding="utf-8")
Path("routing-log.jsonl").write_text(
    "".join(json.dumps(row) + "\n" for row in records),
    encoding="utf-8")
print("Saved", len(records), "routing records")

```

Keep the numerical definitions from the first block and replace its unfinished functions with these completed functions. The code writes files in the directory where you run it; use a new Lab directory so you do not replace an earlier result accidentally. Run `python3 tiny_moe.py`, then perform the verification checks on the saved records. Generating a file is not itself evidence that those checks passed.

28.7.1 Running the supplied completed reference

After attempting the scaffold, save `tiny_moe.py`. It uses the exact numerical functions and fixtures above with an output-directory wrapper. It was tested with CPython 3.13.13 and requires no packages or network.

```
python3 tiny_moe.py --output moe-run-01
```

Inspect all 30 actual records in `moe-run-01/routing-log.jsonl` and its `run-spec.json`. The wrapper refuses a nonempty directory. Choose a new directory for another retained run; use `--overwrite` only to replace the two named result files deliberately. Raw/effective logits and original/combined expert outputs are distinct fields. Preserve failed predictions in your submission.

The optional text-only source helper described in Lab 08 retrieves the pinned gpt-oss config together with four small architecture references; it never loads a model. Its verified config is 1,806 bytes with SHA-256 `3a2a26ded679375b7928ddeca59764df7cea83220c1961035f6d6e232659e9ce`. Record whether you used this actual retrieval, a browser view or the supplied excerpt.

28.8 Solutions and interpretation

28.8.1 Baseline

For layer 0, the exact results are:

Position	Full probabilities in ID order	Selected IDs	Combining weights
0	[4, 3, 2, 1]/10	[0, 1]	[4/7, 3/7]
1	[4, 1, 2, 1]/8	[0, 2]	[2/3, 1/3]
2	[1, 3, 1, 1]/6	[1, 0]	[3/4, 1/4]

The corresponding layer-0 vectors are:

Position	Update	Residual output
0	[8/7, 9/7]	[15/7, 16/7]
1	[1, 0]	[2, 0]
2	[0, 9/4]	[0, 13/4]

At position 2, expert 1 wins; experts 0, 2, and 3 tie for the second slot. The specified tie rule selects 0, whose output is zero for this input. It still receives a routing assignment. The layer-0 counts are [3, 2, 1, 0].

Layer 1 has full probabilities [1, 2, 4, 1]/8 for every input and selects [2, 1] with weights [2/3, 1/3]. For input v , its update is $[v_1/3, 4v_2/3]^T$, and its residual output is $[4v_1/3, 7v_2/3]^T$.

Its final outputs at positions 0, 1, and 2 are respectively

$$[20/7, 16/3]^T, \quad [8/3, 0]^T, \quad [0, 91/12]^T.$$

Its assignment counts are [0, 3, 3, 0]. The toy IDs do not determine routing independently of the input vectors.

28.8.2 Interventions at position 0

The following are hand-derived predictions to check against your run:

Scenario	Selected IDs	Weights	Layer-0 update
Prohibit 0	[1, 2]	[3/5, 2/5]	[−2/5, 11/5]
Force 2 and 3	[2, 3]	[2/3, 1/3]	[−1/3, 1]
Boost 2	[2, 0]	[2/3, 1/3]	[0, 2/3]
Ablate output 0	[0, 1]	[4/7, 3/7]	[0, 9/7]

Residual outputs include the bypass additions:

Scenario	After layer 0	After layer 1
Prohibit 0	[3/5, 16/5]	[4/5, 112/15]
Force 2 and 3	[2/3, 2]	[8/9, 14/3]
Boost 2	[1, 5/3]	[4/3, 35/9]
Ablate output 0	[1, 16/7]	[4/3, 16/3]

Layer 1’s logits, routes, and combining weights remain fixed because its router matrix is zero. Its expert outputs and residual outputs change. In a trained network with input-dependent routers, an earlier intervention can change later routes too. Our system contains no cross-position mixing, so it also cannot demonstrate how attention would spread an intervention across positions.

Adding a common constant changes the raw logits but none of the probabilities, rankings, or vector outputs in exact arithmetic. Subtracting the largest logit before exponentiation exploits this identity for numerical stability.

28.8.3 Parameter accounting

The 20b architecture has $24 \times 32 = 768$ layer-specific experts and $24 \times 4 = 96$ expert evaluations along one token’s forward path. Four of 32 is 12.5%. This fraction applies to equal-sized routed expert banks, not the whole model.

Ignoring biases and non-expert components, the gated expert-matrix totals are 19,110,297,600 stored coefficients and 2,388,787,200 selected coefficients per token path. The model card’s whole-model figures are 20.91 billion total and 3.61 billion active; it counts unembedding parameters but not embeddings as active. The differences are not errors in the multiplication: the quantities include different components.

The toy stores $8 + 4 + 16 = 28$ coefficient slots per layer, or 56 total. Each token uses the 12 router coefficients plus two four-coefficient expert matrices per layer: $2 \times (12 + 8) = 40$ active slots under our stated convention. Hypothetical packed float32 storage is $56 \times 4 = 224$ bytes. Python objects, activations, and interpreter memory are excluded. Storing only the 40 active slots would not preserve all possible routes.

The gpt-oss config has no learned tensors or prompt activations. It cannot produce a trained-model routing trace. An actual trace requires a specific checkpoint, matching tokenizer and formatting, a specified implementation, instrumented intermediate tensors, and evidence of a real execution.

28.9 Optional future extension — Trace a trained model

This is a separate advanced activity, not a required download or setup step for this Lab. Before attempting it, choose a compatible checkpoint and runtime, estimate disk and memory requirements including trace storage, and establish whether the implementation exposes or permits instrumentation of router logits and selected indices.

A normal text-generation API may return no internal routing data. Do not infer expert selections from answer text, ask a model to invent its expert IDs, or relabel this toy's arrays as real measurements.

For a properly instrumented experiment, record checkpoint and code revisions, token IDs and positions, layer indices, prompt formatting, precision, routing rules, and whether tracing covers prompt processing or generation. Validate a small baseline before collecting a large dataset. Compare categories using matched lengths and controlled tokens, and distinguish selection frequency from output contributions. Forced routing can move representations outside an expert's usual training distribution; effects need controls and a defined outcome measure.

28.10 Submission

Include your original predictions, exact baseline calculations, saved or hand-worked routing records, configuration source and inspection method, parameter accounting, and an interpretation of no more than 300 words.

Explain one initial mistake or uncertainty and how you checked it. Conclude with one finding established by the toy and one claim about trained experts that it cannot establish. You have completed the core when another learner can reproduce your routes and outputs from the specification and understand the limits of your conclusion.

28.11 More Learning

1. **Pinned gpt-oss-20b configuration — OpenAI.** Identify the fields used in your accounting and the information missing for an actual routing trace.
2. **gpt-oss reference implementation — OpenAI, pinned commit.** Locate plausible measurement boundaries without loading or running the model.
3. **gpt-oss Model Card — OpenAI, version 1.** Compare your explicitly partial matrix count with the card's whole-model accounting.
4. **Mixtral of Experts — Jiang et al., Section 5.** Design one control that would strengthen an association between input categories and expert selection.