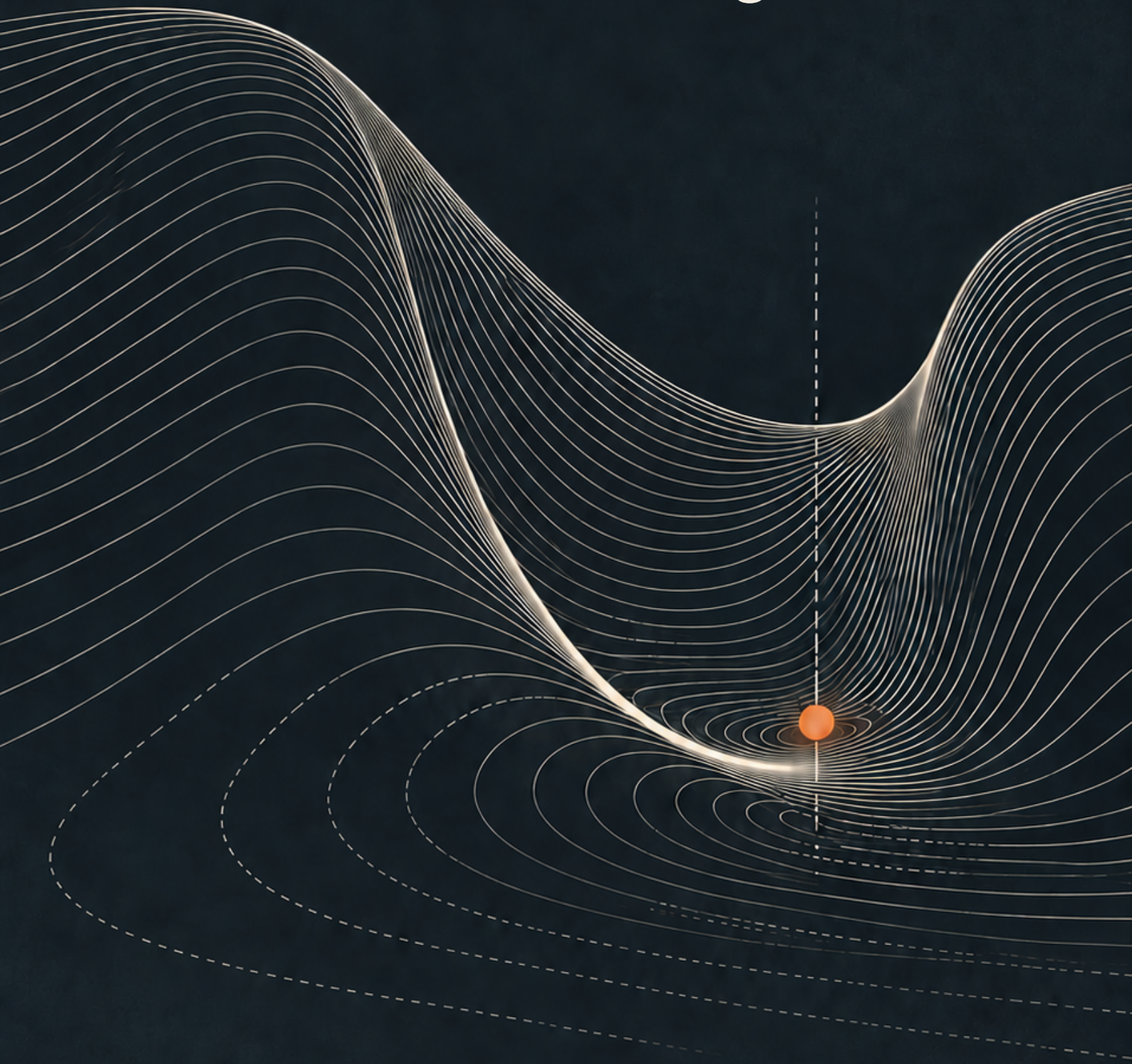
VOLUME I

Foundations of LLMs and Training



Frank Gonzalez, Tom Liu

Preface and Notation

This course packet establishes the common background expected before students divide into Inputs, Core Architecture, Memory, Pre-Training, and Post-Training research teams. It is intentionally narrower than a complete machine-learning course. The goal is practical reading fluency: after finishing the packet, a student should be able to follow the central equations in an LLM paper, identify which component a proposed method changes, and understand how representations, architecture, data, loss, gradients, and optimization fit together.

Basic calculus and matrix multiplication are assumed. When a new symbol or term appears, it is introduced before it is used in a more complex argument. Optional links appear only where a reader may reasonably want greater mathematical depth. Visual companions are short videos by 3Blue1Brown: use a chapter-opening recommendation as context before reading or alongside the chapter, and follow an in-lesson [watch] link when a visual treatment would clarify the nearby idea.

0.1 Notation Used Throughout

Each symbol below is introduced here once and then reused consistently.

Scalar An italic lowercase letter denotes one number.

$\eta = 3 \times 10^{-4}$ may denote a learning rate.

Vector A bold lowercase letter denotes an ordered column of numbers.

$\mathbf{x} = [2, -1, 4]^{\mathsf{T}} \in \mathbb{R}^3$.

Transpose The superscript T exchanges rows and columns.

$[2, -1, 4]$ is a row, while $[2, -1, 4]^{\mathsf{T}}$ is a column.

Matrix A bold uppercase letter denotes a rectangular array.

$\mathbf{W} \in \mathbb{R}^{2\times 3}$ has two rows and three columns.

Tensor A calligraphic uppercase letter denotes an array with three or more axes.

$\mathcal{X} \in \mathbb{R}^{B\times T\times d}$ stores a batch of token representations.

Index A subscript selects an entry or position.

x_j selects coordinate j of $\mathbf{x}$.

x_t selects the token at sequence position t.

Batch size B is the number of sequences processed together.

$B = 32$ means one batch contains 32 sequences.

Sequence length T is the number of token positions in one sequence.

$T = 1024$ means each packed sequence has 1024 positions.

Model width d is the length of one hidden representation.

$\mathbf{x}_t \in \mathbb{R}^d$ is the representation at position t.

Vocabulary size V is the number of token types.

$\mathbf{z}_t \in \mathbb{R}^V$ contains one output score for each token.

Parameters θ denotes the full collection of trainable values.

$\mathcal{L}(\theta)$ is the loss produced by the current parameters.

Probability $p(a \mid b)$ is the probability of a given b.

$p(x_{t+1} \mid x_{\leq t})$ is the next-token probability after a prefix.

Loss ℓ denotes one local loss and $\mathcal{L}$ denotes an aggregated objective.

$$\mathcal{L} = B^{-1} \sum_{i=1}^{B} \ell_i.$$

Gradient $\nabla_\theta \mathcal{L}$ collects the partial derivatives of the loss with respect to all parameters.

Contents

Preface and Notation **i**

0.1 Notation Used Throughout i

1 Introduction to Neural Networks **1**

1.1 Numbers and Shapes 1

1.2 Linear Layers and Affine Transformations 2

1.3 Adding Nonlinearity 3

1.4 Measuring Error 4

1.5 How Learning Changes the Model 5

1.6 Summary 6

2 How Language Models Represent Text **7**

2.1 Why Text Must Become Numbers 7

2.2 Tokens and Vocabularies 7

2.3 Embeddings and Position 8

2.4 Predicting the Next Token 10

2.5 Generating Text 11

2.6 Summary 11

3 How Transformers Work **12**

3.1 Context and Hidden Representations 12

3.2 Self-Attention 13

3.3 Causal Attention 16

3.4 Multi-Head Attention 16

3.5 MLPs and Residual Updates 17

3.6 A Complete Decoder Block 18

3.7 Summary 19

4 Pretraining: How Base Models Learn **20**

4.1 From Corpus to Batches 20

4.2 The Next-Token Objective 21
4.3 One Training Step 21
4.4 Hyperparameters and Schedules 22
4.5 Reading Training Curves 24
4.6 Summary 25

5 Post-Training: Shaping Model Behavior 26
5.1 Why Post-Training Exists 26
5.2 Supervised Fine-Tuning 26
5.3 Learning from Preferences 27
5.4 Keeping Behavior Stable 28
5.5 Summary 29

Sources Used 30

Chapter 1

Introduction to Neural Networks

A neural network is a function built from numerical operations. It receives an input, produces a prediction, measures how wrong that prediction was, and changes its internal parameters. The same cycle later trains language models; only the data and architecture become larger.

1.1 Numbers and Shapes

The first task is learning to read the objects that move through a model. A scalar is one number. A vector is an ordered list. A matrix arranges values into rows and columns. A tensor extends this organization across additional axes.

The panel below pairs the general notation for each object with one concrete instance. In the general forms, n is a vector length, m and n are the numbers of matrix rows and columns, and $d_1, \ldots, d_k$ are the sizes of a tensor's k axes.

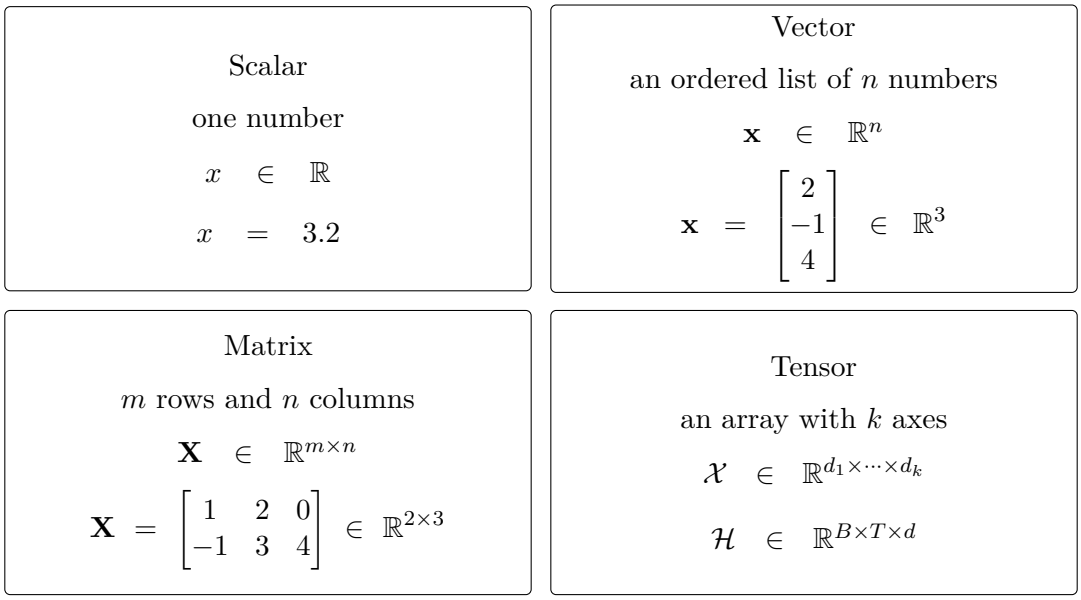


Figure 1.1: General notation and concrete examples for four numerical object types.

For a geometric picture of a vector as both an arrow and an ordered list of coordinates, see *Vectors, what even are they?* [watch].

The **shape** of an object records the size of every axis. Shapes are not decoration. They tell us whether an operation is valid and what kind of output it produces.

Consider

$$\mathbf{W} = \begin{bmatrix} 1 & 2 & 0 \\ -1 & 3 & 4 \end{bmatrix} \in \mathbb{R}^{2\times 3}, \qquad \mathbf{x} = \begin{bmatrix} 2 \\ 1 \\ -1 \end{bmatrix} \in \mathbb{R}^3. \tag{1.1}$$

The 3 columns of $\mathbf{W}$ match the 3 entries of $\mathbf{x}$, so the product is valid:

$$\mathbf{W}\mathbf{x} = \begin{bmatrix} 1(2) + 2(1) + 0(-1) \\ -1(2) + 3(1) + 4(-1) \end{bmatrix} = \begin{bmatrix} 4 \\ -3 \end{bmatrix} \in \mathbb{R}^2. \tag{1.2}$$

Each output entry is a weighted sum of the inputs.

The geometric meaning of this operation is developed in *Linear transformations and matrices* [watch]. If the way several matrix operations combine is unfamiliar, *Matrix multiplication as composition* [watch] shows why multiplying matrices corresponds to applying transformations one after another.

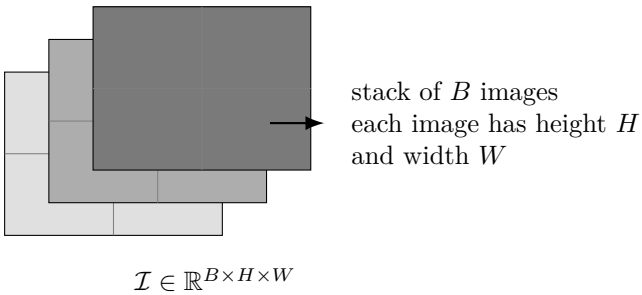


Figure 1.2: A tensor can be understood as familiar arrays stacked along another axis.

A tensor is not a mysterious new number system. It is a way to keep several axes organized. Later, a language-model tensor may use one axis for examples in a batch, one for token positions, and one for the features stored at each position.

1.2 Linear Layers and Affine Transformations

A model becomes trainable when its numerical rule contains parameters that can change. The most common starting point is

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}. \tag{1.3}$$

The multiplication $\mathbf{W}\mathbf{x}$ forms weighted combinations of the input. The vector $\mathbf{b}$ then shifts those combinations. Together, this is called an **affine transformation**.

The terms have specific meanings:

$\mathbf{x}$	input vector,
$\mathbf{W}$	weight matrix,
$\mathbf{b}$	bias vector,
$\mathbf{z}$	output of the affine transformation.

A strictly linear map has the form $\mathbf{W}\mathbf{x}$ and must send the zero vector to the zero vector. Adding $\mathbf{b}$ allows the whole mapping to shift away from the origin. Neural-network libraries still commonly call this a **linear layer**, even though the full operation is affine.

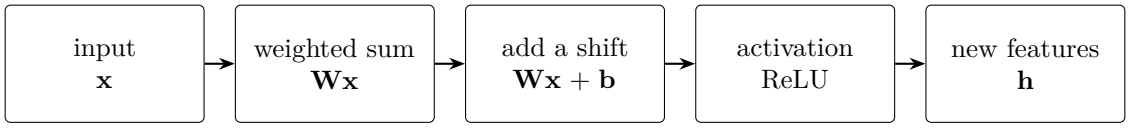


Figure 1.3: An affine transformation mixes the input and then shifts it. A nonlinear activation follows.

For example, let

$$\mathbf{W} = \begin{bmatrix} 2 & -1 \\ 1 & 3 \end{bmatrix}, \qquad \mathbf{x} = \begin{bmatrix} 4 \\ 2 \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}. \tag{1.4}$$

Then

$$\mathbf{Wx} + \mathbf{b} = \begin{bmatrix} 6 \\ 10 \end{bmatrix} + \begin{bmatrix} 1 \\ -2 \end{bmatrix} = \begin{bmatrix} 7 \\ 8 \end{bmatrix}. \tag{1.5}$$

The weights determine how the input features are mixed. The bias adds the same learned shift each time the layer is used.

1.3 Adding Nonlinearity

If we stack affine transformations without anything between them, the result can still be rewritten as one affine transformation. The model would remain limited to one family of straight, flat relationships. Neural networks become more expressive by placing a nonlinear **activation function** between layers.

A common activation is the rectified linear unit:

$$\mathrm{ReLU}(z) = \max(0, z). \tag{1.6}$$

It keeps a positive input unchanged and replaces a negative input with zero. Applied to a vector, ReLU acts on each entry separately.

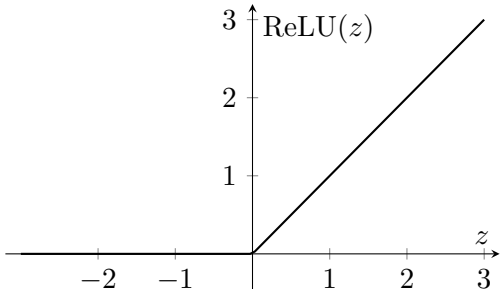


Figure 1.4: ReLU keeps positive values and replaces negative values with zero.

A small multilayer perceptron, usually called an **MLP**, can now be written as

$$\begin{aligned} \mathbf{h} &= \mathrm{ReLU}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1), \\ \widehat{y} &= \mathbf{w}_2^{\top}\mathbf{h} + b_2. \end{aligned} \tag{1.7}$$

The vector $\mathbf{h}$ is a **hidden representation**. It is not supplied by the dataset; it is produced inside the model. The symbol $\widehat{y}$ denotes the prediction, while y will denote the true target.

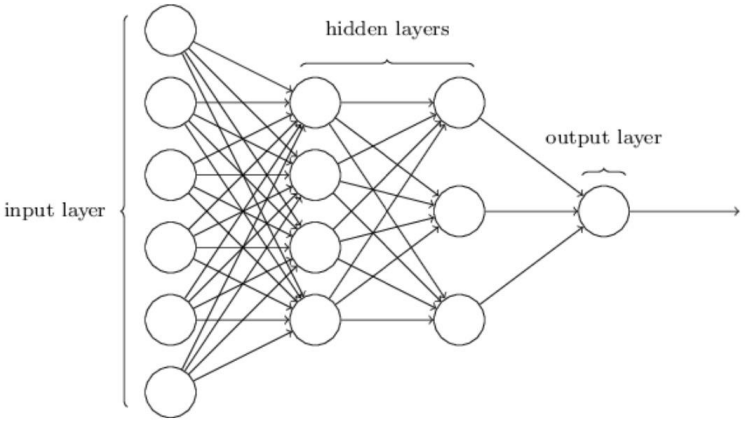


Figure 1.5: A multilayer perceptron passes inputs through learned hidden layers to an output.

The superscript T means transpose. It turns the column vector $\mathbf{w}_2$ into a row vector so that $\mathbf{w}_2^{\mathsf{T}}\mathbf{h}$ is a dot product and produces one number.

The important idea is composition. Each layer is simple. The model gains flexibility because the output of one layer becomes the input to another, with nonlinear activations between them.

See *But what is a neural network?* [watch] to connect weighted sums, biases, activations, hidden layers, and predictions in one visual model.

Go deeper. Dive into Deep Learning, Multilayer Perceptrons, for a longer treatment of affine layers, activations, and hidden representations. [read]

1.4 Measuring Error

A prediction alone does not tell the model how to improve. Training needs a number that measures how far the prediction is from the target. This number is called the **loss**.

Suppose a model predicts house prices, measured in hundreds of thousands of dollars. A true price of $y = 5.0$ means \$500,000. If the prediction is $\widehat{y} = 4.4$, the model predicted \$440,000 and missed by \$60,000.

For a numerical prediction, a simple choice is squared error:

$$\ell = (\widehat{y} - y)^2. \tag{1.8}$$

Here ℓ is the loss for one example. In the house-price example,

$$\ell = (4.4 - 5.0)^2 = 0.36. \tag{1.9}$$

The square makes the result nonnegative and gives larger errors more influence.

Models usually process a **batch**, which is a small set of examples handled together. If the batch contains B examples with losses $\ell_1, \ell_2, \ldots, \ell_B$, the average loss is

$$\mathcal{L} = \frac{1}{B}\sum_{i=1}^{B} \ell_i. \tag{1.10}$$

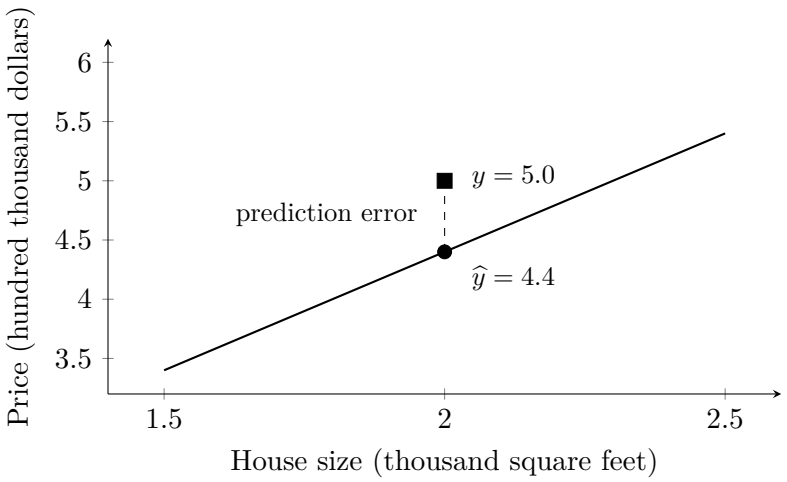


Figure 1.6: A prediction can be compared with a known target before the model is updated.

The symbol i indexes the examples, B is the batch size, and $\mathcal{L}$ is the one scalar used to guide the next update.

1.5 How Learning Changes the Model

The weights and biases are the model's **parameters**. We use θ as a compact name for the whole collection. Training changes θ so that the loss becomes smaller.

The **gradient** measures how the loss changes when each parameter is nudged. If increasing a parameter would increase the loss, the update should move that parameter in the opposite direction. The basic update is

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \, \nabla_\theta \mathcal{L}. \tag{1.11}$$

The symbols mean:

θ_{old}	current parameters,
$\nabla_\theta \mathcal{L}$	gradient of the loss with respect to the parameters,
η	learning rate, controlling the step size,
θ_{new}	parameters after the update.

The surface in the figure is a visual metaphor. Real models have far more than two parameters, so their loss landscape cannot be drawn directly. The principle is unchanged: use the local slope to move toward lower loss.

See *Gradient descent, how neural networks learn* [watch] to visualize the negative-gradient direction, learning-rate step size, and movement toward lower loss.

Backpropagation computes the required gradients efficiently. It follows the model's operations backward and repeatedly applies the chain rule from calculus. The update rule is then carried out by an **optimizer**. For a visual explanation of what the backward flow accomplishes, see *What is backpropagation really doing?* [watch]. Readers who want the corresponding calculus can continue to *Backpropagation calculus* [watch].

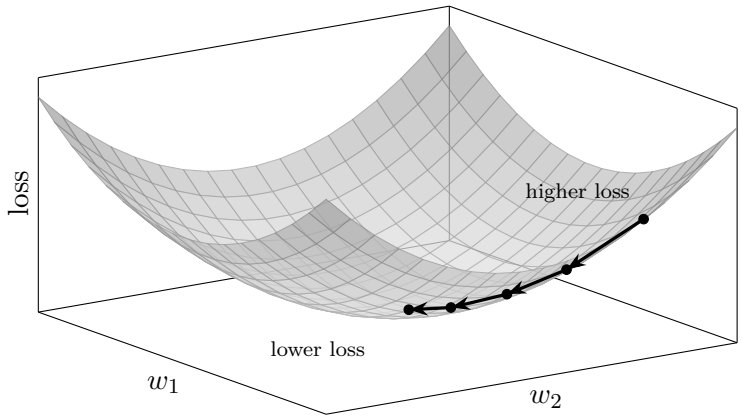


Figure 1.7: Gradient descent can be pictured as taking local downhill steps on a loss surface.

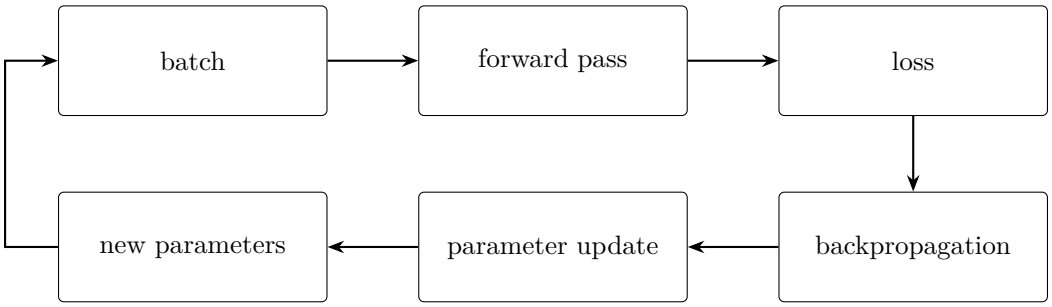


Figure 1.8: Training repeats the same prediction, error, gradient, and update cycle.

During the forward pass, the current parameters produce predictions. The loss measures error. Backpropagation computes gradients, and the optimizer changes the parameters. Repeating this loop over many batches is training.

Go deeper. Michael Nielsen, How the Backpropagation Algorithm Works, for a slower explanation of the chain rule and gradient flow. [read]

1.6 Summary

Neural networks transform numerical inputs through affine layers and nonlinear activations. A loss turns prediction error into one scalar. Backpropagation computes how that loss depends on each parameter, and gradient-based updates improve the model one step at a time. These ideas remain in place when the inputs become tokens and the architecture becomes a Transformer.

Chapter 2

How Language Models Represent Text

The previous chapter described a model that accepts numerical inputs. Language begins as characters and words, so an additional representation pipeline is needed. A language model first breaks text into tokens, maps those tokens to vectors, and then learns to predict which token should come next.

Visual companion. *Large Language Models explained briefly* [watch]. This eight-minute overview previews tokens, next-token probabilities, autoregressive generation, parameters, pretraining, and the role of Transformers.

2.1 Why Text Must Become Numbers

A neural network cannot directly multiply the word "model" by a matrix. The text must first be converted into discrete identifiers and then into numerical vectors.

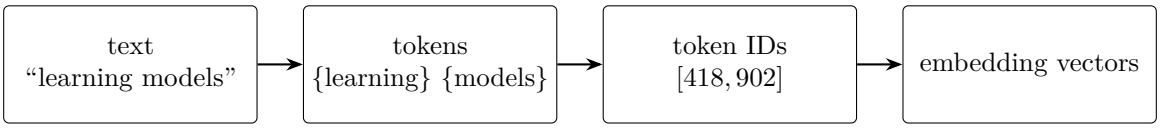


Figure 2.1: Text must be converted into discrete token IDs and then continuous vectors before a neural network can process it.

The pipeline separates two jobs. **Tokenization** decides which pieces of text the model treats as discrete units. The **embedding table** then assigns each unit a learned vector.

A token does not have to be a complete word. It may be a word, part of a word, punctuation, a space pattern, or a byte. This flexibility allows a finite vocabulary to represent an open-ended language.

2.2 Tokens and Vocabularies

A **tokenizer** converts a string into a sequence of tokens. A **vocabulary** is the list of token types the tokenizer knows. Each vocabulary entry receives an integer ID.

For example, one possible tokenizer might produce

$$\text{"unbelievable"} \longrightarrow [\text{"un"}, \text{"believ"}, \text{"able"}] \longrightarrow [71, 1842, 309].$$

The specific pieces and IDs depend on the tokenizer. The IDs themselves do not carry meaning; they are addresses used to look up vectors.

Subword tokenizers balance two extremes. A word-level vocabulary would need an enormous entry list and would struggle with unseen words. A character-level vocabulary is small but creates very long sequences. Subwords reuse common pieces while keeping sequences manageable. Byte-pair encoding, or BPE, is one common method: it begins with small units and repeatedly merges adjacent pairs that occur often.

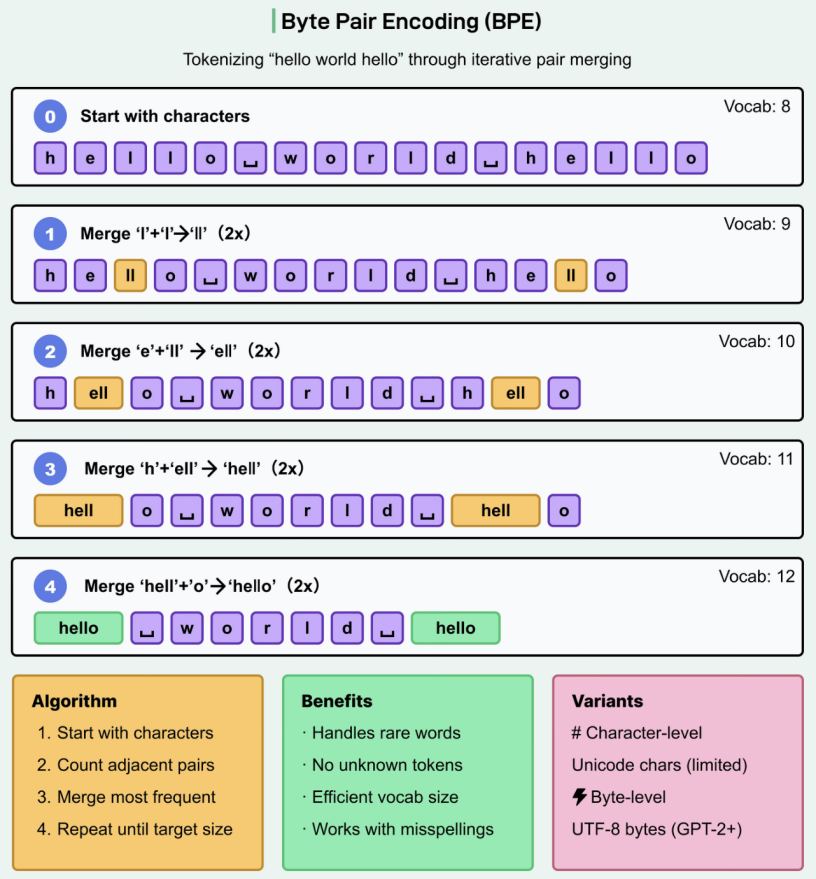


Figure 2.2: Byte-pair encoding repeatedly merges frequent adjacent pairs into longer tokens.

Tokenization changes the cost and difficulty of learning. If one language is split into many more tokens than another, the model must process longer sequences to read the same amount of content. This is why tokenizer design belongs to the model's input architecture, not merely data cleaning.

Go deeper. Jurafsky and Martin, Words and Tokens, for Unicode, subwords, byte-pair encoding, and multilingual tokenization. [read]

2.3 Embeddings and Position

A token ID is only an integer label. To give the model a useful numerical representation, we use an embedding matrix

$$\mathbf{E} \in \mathbb{R}^{V \times d}. \tag{2.1}$$

Here V is the vocabulary size and d is the embedding dimension. Row i of $\mathbf{E}$ is the vector assigned to token ID i.

If token ID i appears in a sequence, its embedding is

$$\mathbf{e}_i = \mathbf{E}_{i,:}^{\mathsf{T}} \in \mathbb{R}^d. \tag{2.2}$$

The notation $i,:$ means row i and all columns. The transpose writes those d row entries as the column vector $\mathbf{e}_i$. The embedding values are learned during training, just like the weights of a neural-network layer.

A one-hot vector has length V and contains zeros everywhere except for a single 1 at the selected token ID. Multiplying that vector by the transposed embedding matrix is another way to express the same row selection.

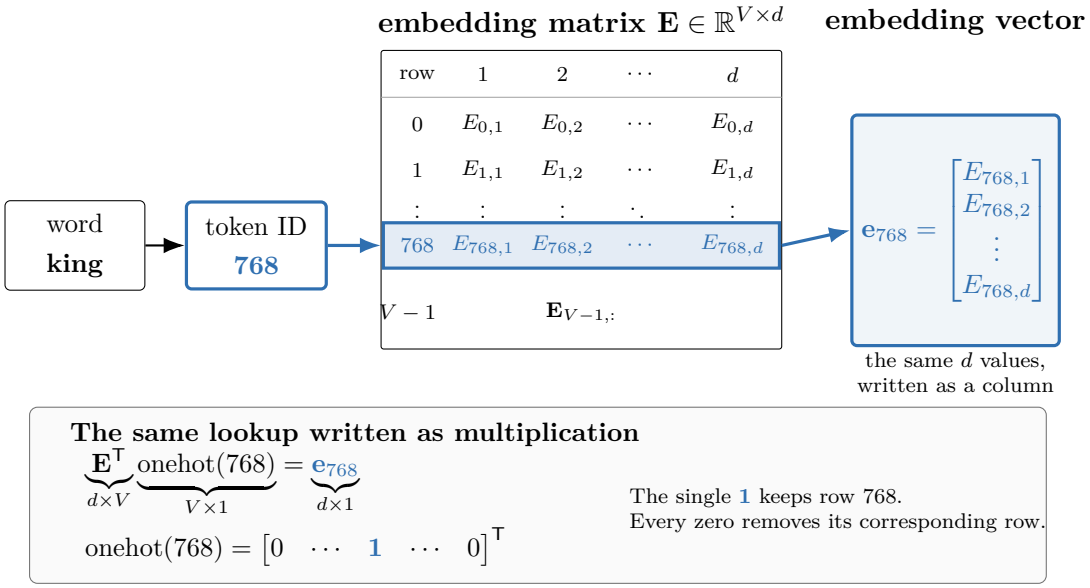


Figure 2.3: Token ID 768 selects one row of the embedding matrix. A one-hot vector with its single 1 at the same index performs the identical selection through matrix multiplication.

Embeddings allow similar uses of language to develop related vector representations. This does not mean that each coordinate has a clean human label. Meaning is distributed across the vector and changes as the vector passes through the model.

The opening portion of *Transformers, the tech behind LLMs* [watch] gives a visual account of embedding vectors, vocabulary size, and the projection from a final hidden vector back to next-token probabilities.

A sequence also needs information about order. Without position information, the same tokens arranged differently would give the model the same collection of vectors. A model therefore combines token content with a position representation:

$$\mathbf{h}_t^{(0)} = \mathbf{e}_{x_t} + \mathbf{p}_t. \tag{2.3}$$

The token at position t has ID x_t, token embedding $\mathbf{e}_{x_t}$, position vector $\mathbf{p}_t$, and initial hidden representation $\mathbf{h}_t^{(0)}$.

Models can encode position in several ways. Some learn a position vector directly. Others modify the hidden features according to position. The details differ, but the purpose is stable: the model must know not only which token it sees, but where that token occurs.

2.4 Predicting the Next Token

A language model learns a probability distribution over possible next tokens. For a sequence $x_1, x_2, \ldots, x_T$, its probability is

$$p(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} p(x_t \mid x_1, \ldots, x_{t-1}). \tag{2.4}$$

The large symbol $\prod$ is called **product notation**. It means "multiply one term for every value of the index." The lower label $t = 1$ says to begin at the first position, and the upper label T says to stop at the final position. If a_t denotes one number associated with position t, then

$$\prod_{t=1}^{3} a_t = a_1 a_2 a_3. \tag{2.5}$$

In the language-model equation, every factor is one next-token probability. Multiplying the factors builds the probability assigned to the full sequence. The vertical bar means "given." For instance,

$$p(x_4 \mid x_1, x_2, x_3)$$

is the probability assigned to the fourth token after seeing the first three.

For each position, the model outputs one score for every vocabulary item. These raw scores are called **logits**. If the vocabulary has size V, the logits form a vector

$$\mathbf{z}_t \in \mathbb{R}^V. \tag{2.6}$$

A softmax converts those scores into probabilities:

$$p_{t,j} = \frac{e^{z_{t,j}}}{\sum_{k=1}^{V} e^{z_{t,k}}}. \tag{2.7}$$

Here $p_{t,j}$ is the probability of vocabulary item j at position t, and $z_{t,j}$ is its logit. The denominator adds the exponentiated scores for all V vocabulary items, so the probabilities sum to 1.

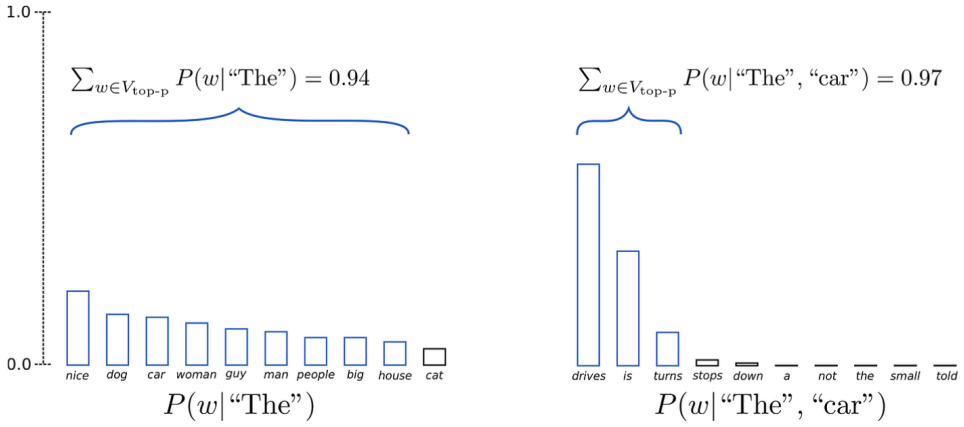


Figure 2.4: Different contexts produce different next-token probability distributions.

Training rewards the model for assigning high probability to the token that actually followed. For the correct token x_t, the loss at position t is

$$\ell_t = -\log p(x_t \mid x_1, \ldots, x_{t-1}). \tag{2.8}$$

Here log is the natural logarithm. We use it because sequence probabilities multiply, while logarithms turn multiplication into addition. The model can therefore add and average the losses from many token positions. The minus sign makes the direction intuitive: assigning probability 1 to the correct token gives a loss of 0, while assigning probability 0.1 gives $-\log(0.1) \approx 2.30$. Very small correct-token probabilities receive a large penalty.

The losses are averaged across positions and examples before backpropagation.

2.5 Generating Text

Training and generation use the same next-token distribution in different ways. During training, the correct next token is already known and provides the target. During generation, the model chooses a token from its predicted distribution, appends that token to the context, and predicts again.

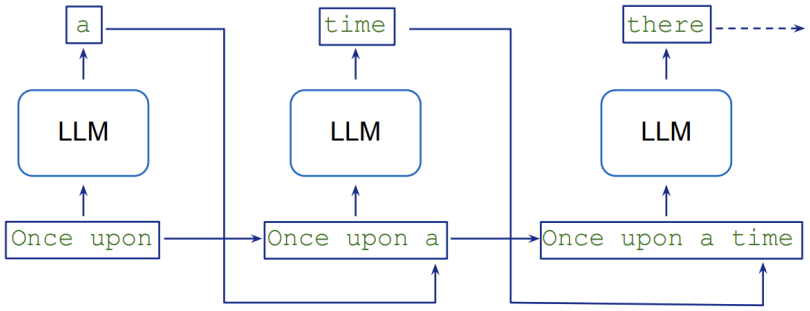


Figure 2.5: Generation predicts a token, appends it to the context, and repeats.

Choosing the most likely token every time is called greedy decoding. Sampling introduces variation by drawing from the distribution. A temperature setting can make the distribution sharper or flatter before sampling. At this stage, the important point is not the full menu of decoding methods. It is that generation repeatedly turns model probabilities into an extended sequence.

A **context window** limits how many previous tokens the model can use at once. Longer contexts provide more information but require more computation and memory. Later architectural work asks how efficiently a model can use that available context.

2.6 Summary

Language models operate on token sequences, not raw words. Tokenizers create discrete units, embedding tables turn those units into vectors, and position representations preserve order. The model then predicts a probability distribution over the next token. Training compares that distribution with the observed continuation, while generation repeatedly selects from it.

Chapter 3

How Transformers Work

A next-token model must combine information from many earlier positions. The Transformer does this through repeated blocks. Each block contains attention, which shares information across positions, and an MLP, which transforms the features stored at each position.

Visual companion. *Transformers, the tech behind LLMs* [watch]. Watch this chapter-level overview before the equations below. It shows how tokens, embeddings, attention, MLPs, and the final vocabulary prediction fit into one model.

3.1 Context and Hidden Representations

The embedding at a position begins as a representation of one token plus its position. That is not enough. The meaning of a token depends on its context: "bank" means something different near "river" than near "loan."

A Transformer therefore maintains one vector for every token position. These vectors are called **hidden states**. As the vectors pass through successive blocks, they become contextual representations that include information gathered from earlier tokens.

If each hidden vector has dimension d, then the vector at position t is

$$\mathbf{h}_t \in \mathbb{R}^d. \tag{3.1}$$

The subscript t identifies the token position. A sequence of length T therefore has the vectors $\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_T$.

To process every position together, we transpose the column vectors and stack them as the rows of one matrix:

$$\mathbf{H} = \begin{bmatrix} \mathbf{h}_1^\mathsf{T} \\ \mathbf{h}_2^\mathsf{T} \\ \vdots \\ \mathbf{h}_T^\mathsf{T} \end{bmatrix} \in \mathbb{R}^{T \times d}. \tag{3.2}$$

The transpose symbol T turns each column vector into a row. Consequently, row t of $\mathbf{H}$ is $\mathbf{h}_t^\mathsf{T}$. The matrix changes from layer to layer while keeping T rows and d columns.

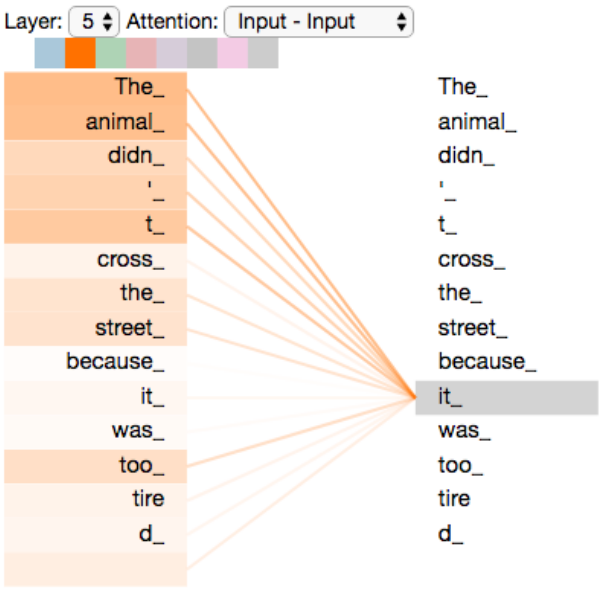


Figure 3.1: Attention lets a token gather context from other positions.

The evolving matrix $\mathbf{H}$ is often called the **residual stream**. Attention and MLP sublayers compute updates, and those updates are added back into the stream rather than replacing it entirely.

3.2 Self-Attention

Self-attention lets each position form a weighted combination of information from other positions. The weights are not fixed. They are computed from the current hidden states, so different sequences produce different patterns of information flow.

Begin with one position i. Its hidden state $\mathbf{h}_i$ is transformed into three new vectors:

$$\begin{aligned} \mathbf{q}_i &= \mathbf{W}_Q \mathbf{h}_i, \\ \mathbf{k}_i &= \mathbf{W}_K \mathbf{h}_i, \\ \mathbf{v}_i &= \mathbf{W}_V \mathbf{h}_i. \end{aligned} \tag{3.3}$$

The learned matrices $\mathbf{W}_Q$, $\mathbf{W}_K$, and $\mathbf{W}_V$ perform the three projections. The outputs $\mathbf{q}_i$, $\mathbf{k}_i$, and $\mathbf{v}_i$ are the query, key, and value for position i. Their matrix shapes are

$$\mathbf{W}_Q, \mathbf{W}_K \in \mathbb{R}^{d_k \times d}, \qquad \mathbf{W}_V \in \mathbb{R}^{d_v \times d}, \tag{3.4}$$

so each matrix accepts a hidden vector of length d and produces a vector of the required output length.

The names describe their roles. A **query** represents what a position is looking for. A **key** represents what a position offers for matching. A **value** contains the information that can be transferred. If queries and keys have length d_k and values have length d_v, then

$$\mathbf{q}_i, \mathbf{k}_i \in \mathbb{R}^{d_k}, \qquad \mathbf{v}_i \in \mathbb{R}^{d_v}. \tag{3.5}$$

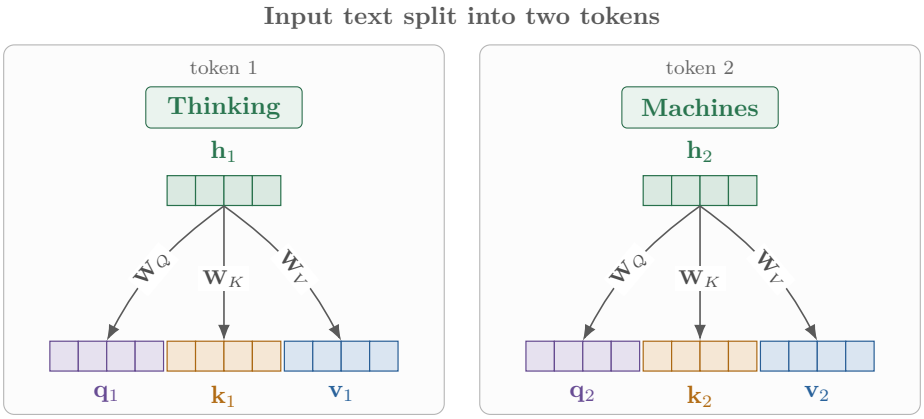


Figure 3.2: The input text is split into tokens. Each token's hidden state is then projected into a query, key, and value using the same learned matrices at every position. The next figure follows $\mathbf{q}_1$ through one complete attention calculation.

Attention now builds one updated vector for position i. First, its query is compared with the key at every position j:

$$s_{ij} = \frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_k}}. \tag{3.6}$$

Here i is the position receiving information and j is a position it may receive information from. The dot product $\mathbf{q}_i^\top \mathbf{k}_j$ produces one matching score. Dividing by $\sqrt{d_k}$ keeps the scores at a manageable scale.

Next, softmax converts the scores into nonnegative weights that sum to 1:

$$\alpha_{ij} = \frac{e^{s_{ij}}}{\sum_{r=1}^{T} e^{s_{ir}}}. \tag{3.7}$$

The symbol α_{ij} is the weight that position i gives to position j. The index r simply runs across all T positions in the denominator.

Finally, position i receives a weighted sum of the value vectors:

$$\mathbf{o}_i = \sum_{j=1}^{T} \alpha_{ij} \mathbf{v}_j \in \mathbb{R}^{d_v}. \tag{3.8}$$

A large α_{ij} makes $\mathbf{v}_j$ contribute strongly to the output $\mathbf{o}_i$. These three equations describe attention one query position at a time.

The comparison between a query and a key is a dot product. For geometric intuition about why dot products measure directional alignment, see *Dot products and duality* [watch].

The same computation can be written compactly for the whole sequence. Stack the query, key, and

value vectors as rows:

$$\mathbf{Q} = \begin{bmatrix} \mathbf{q}_1^\mathsf{T} \\ \vdots \\ \mathbf{q}_T^\mathsf{T} \end{bmatrix} \in \mathbb{R}^{T \times d_k}, \qquad \mathbf{K} = \begin{bmatrix} \mathbf{k}_1^\mathsf{T} \\ \vdots \\ \mathbf{k}_T^\mathsf{T} \end{bmatrix} \in \mathbb{R}^{T \times d_k}, \qquad \mathbf{V} = \begin{bmatrix} \mathbf{v}_1^\mathsf{T} \\ \vdots \\ \mathbf{v}_T^\mathsf{T} \end{bmatrix} \in \mathbb{R}^{T \times d_v}. \tag{3.9}$$

The full attention calculation is then

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\mathsf{T}}{\sqrt{d_k}}\right)\mathbf{V}. \tag{3.10}$$

The softmax is applied across each row. Row i of the resulting weight matrix contains $\alpha_{i1}, \ldots, \alpha_{iT}$, and row i of the final output is $\mathbf{o}_i^\mathsf{T}$.

Figure 3.3 traces the whole calculation with the two positions introduced in Figure 3.2. It follows $\mathbf{q}_1$ from its comparisons with $\mathbf{k}_1$ and $\mathbf{k}_2$ through scaling, softmax, value weighting, and the final sum.

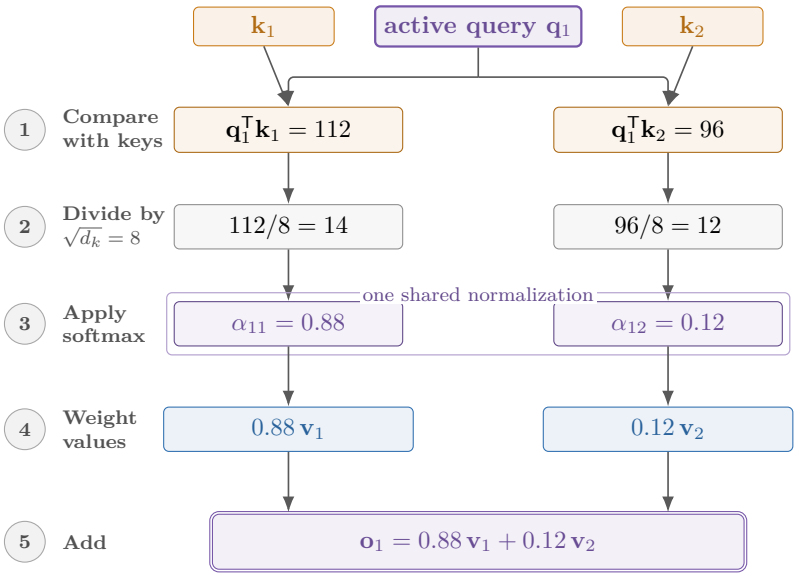


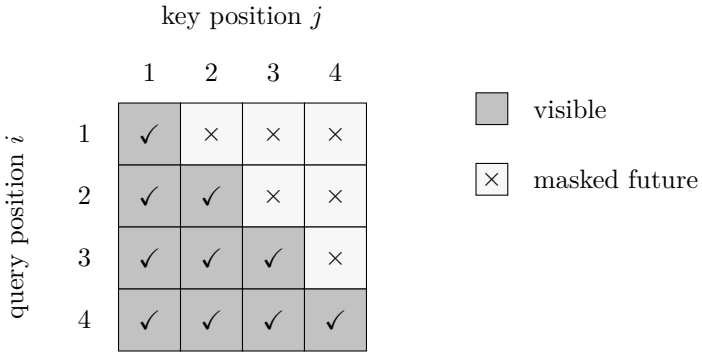
Figure 3.3: One query follows the complete attention procedure: compare with keys, scale the scores, normalize them, weight the values, and add the contributions. The example uses $d_k = 64$, so $\sqrt{d_k} = 8$.

See *Attention in transformers, step-by-step* [watch] for a visual walkthrough of queries, keys, dot-product scores, softmax weights, masking, and value-vector updates.

Go deeper. Jurafsky and Martin, Transformers, for a gradual derivation of causal self-attention and the complete decoder block. [read]

3.3 Causal Attention

A model trained to predict the next token must not look at future tokens when producing an earlier hidden state. Otherwise, the answer would leak into the input. A **causal mask** removes future positions before the softmax.



Query position i may use key position j only when $j \leq i$.

Figure 3.4: Causal attention exposes only the current and earlier positions.

Each row shows the keys available to one query position. The visible cells form a lower triangle: position 1 can use only itself, position 2 can use positions 1 and 2, and the final position can use the full prefix.

3.4 Multi-Head Attention

One attention calculation may be asked to represent several kinds of relationships at once. Transformers therefore use **multi-head attention**. Each head has its own query, key, and value projections. The heads compute attention in parallel, their outputs are concatenated, and another learned projection combines them.

Let h be the number of attention heads and let d_h be the feature dimension used by one head. For head r, where r ranges from 1 to h, three learned projection matrices produce

$$\mathbf{Q}_r = \mathbf{H}\mathbf{W}_Q^{(r)}, \quad \mathbf{K}_r = \mathbf{H}\mathbf{W}_K^{(r)}, \quad \mathbf{V}_r = \mathbf{H}\mathbf{W}_V^{(r)}. \tag{3.11}$$

The superscript (r) labels which head owns a projection matrix; it is not an exponent. If $\mathbf{H} \in \mathbb{R}^{T \times d}$ and each projection matrix is in $\mathbb{R}^{d \times d_h}$, then $\mathbf{Q}_r$, $\mathbf{K}_r$, and $\mathbf{V}_r$ are all in $\mathbb{R}^{T \times d_h}$.

Each head performs the attention calculation introduced in the previous section (3.10):

$$\text{head}_r = \text{Attention}(\mathbf{Q}_r, \mathbf{K}_r, \mathbf{V}_r) \in \mathbb{R}^{T \times d_h}. \tag{3.12}$$

The h head outputs are then joined side by side and projected back to the model width:

$$\text{MultiHead}(\mathbf{H}) = \text{Concat}\big(\text{head}_1, \ldots, \text{head}_h\big)\mathbf{W}_O. \tag{3.13}$$

Here Concat means concatenate, or place the feature columns from the heads next to one another. The joined matrix has shape $T \times (hd_h)$. The learned output matrix $\mathbf{W}_O \in \mathbb{R}^{(hd_h) \times d}$ returns the result to shape $T \times d$, matching the hidden-state matrix that entered the layer.

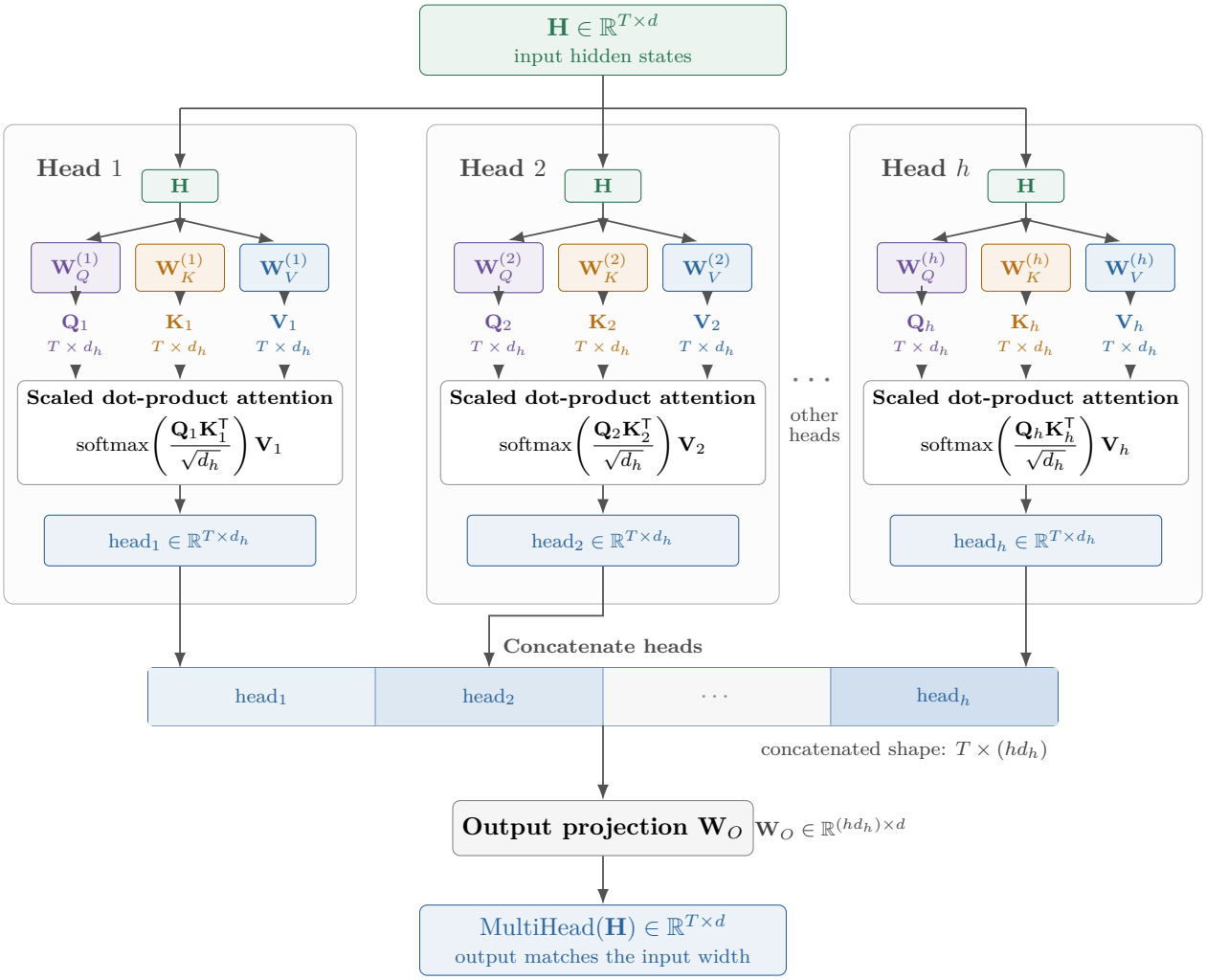


Figure 3.5: Multi-head attention projects one hidden-state matrix into several independent attention heads, concatenates their outputs, and applies one output projection that restores the model width.

Heads are not assigned fixed human meanings. Some may specialize in positional patterns, syntax, copying, or other useful relationships, but those behaviors emerge from training rather than manual labels.

3.5 MLPs and Residual Updates

Attention communicates across token positions. The MLP performs a different role: it transforms the features within each position independently. The same MLP weights are applied to every row of the hidden-state matrix.

A simplified Transformer MLP is

$$\mathrm{MLP}(\mathbf{h}) = \mathbf{W}_2\,\sigma(\mathbf{W}_1\mathbf{h} + \mathbf{b}_1) + \mathbf{b}_2. \tag{3.14}$$

The first affine transformation usually expands the feature dimension. The activation σ introduces nonlinearity. The second affine transformation returns to the model dimension.

See *How might LLMs store facts?* [watch] to visualize the two matrix multiplications and activation inside a Transformer MLP, followed by an intuitive example based on factual recall.

Attention and the MLP are inserted through residual updates. In simplified form,

$$\begin{aligned} \mathbf{H}' &= \mathbf{H} + \text{Attention}(\mathbf{H}), \\ \mathbf{H}_{\text{next}} &= \mathbf{H}' + \text{MLP}(\mathbf{H}'). \end{aligned} \tag{3.15}$$

The additions let each sublayer contribute a change while preserving a direct path for the representation already present.

Normalization is placed around these sublayers to keep activation scales manageable. Modern decoder-only models commonly normalize before attention and before the MLP. The exact normalization rule matters to specialists, but its broad purpose is stable training.

3.6 A Complete Decoder Block

A decoder block combines the mechanisms introduced above. Hidden states are normalized, updated by causal attention, normalized again, and updated by an MLP. Residual paths carry the existing representation around each sublayer.

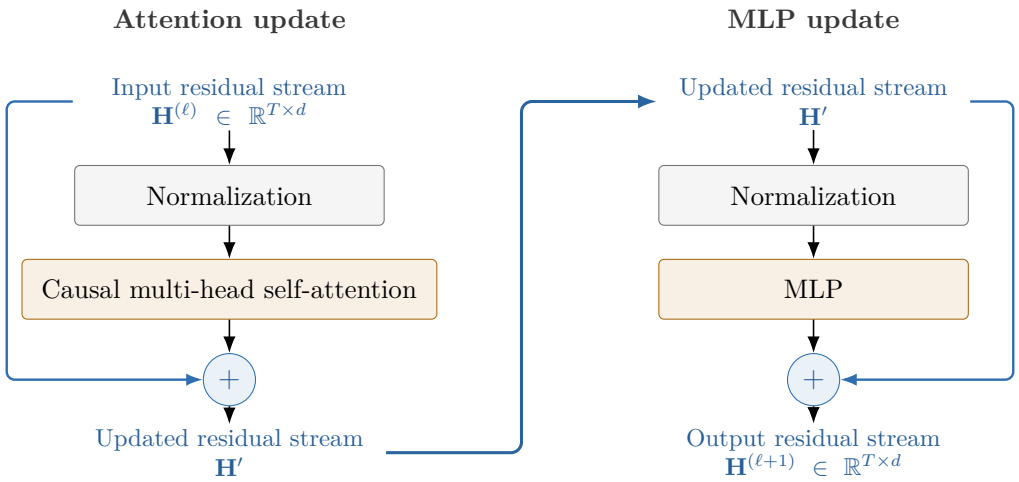


Figure 3.6: A two-column decoder block reads downward within each sublayer. The central arrow carries the attention-updated residual stream $\mathbf{H}'$ upward into the MLP column.

A full language model stacks many blocks. After the final block, each position's hidden state is projected to vocabulary logits. The model is therefore one continuous path:

$$\text{tokens} \longrightarrow \text{embeddings} \longrightarrow \text{Transformer blocks} \longrightarrow \text{vocabulary logits}.$$

The final projection produces one logit for every vocabulary item. Softmax then converts those logits into the next-token probability distribution.

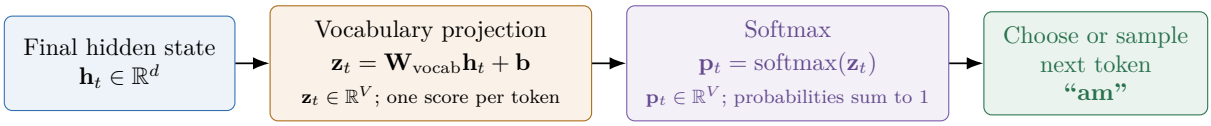


Figure 3.7: The final hidden state is projected to one logit per vocabulary item, normalized into probabilities, and used to choose or sample the next token.

The parameter count is spread across embedding tables, attention projections, MLP weights, normalization parameters, and the final vocabulary projection. Architecture research changes these components or the way they interact, but the decoder-block picture remains the common baseline. Figure 3.8 places the repeated decoder stack and its output head in one end-to-end view.

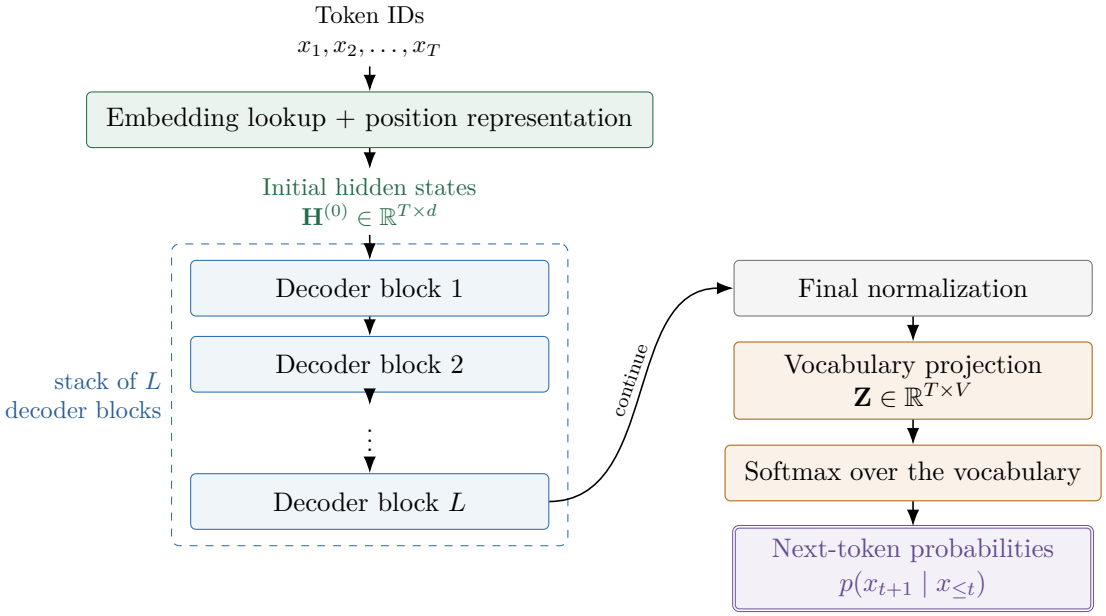


Figure 3.8: A decoder-only model first builds contextual hidden states, then maps them through an output head to vocabulary logits and next-token probabilities.

3.7 Summary

Transformers maintain a hidden vector for every token position. Self-attention shares information across positions by matching queries with keys and blending values. Causal masking prevents future information from leaking backward. MLPs transform features at each position, while residual connections preserve and update the shared stream. Stacking these blocks produces the contextual representations used for next-token prediction.

Chapter 4

Pretraining: How Base Models Learn

A Transformer architecture begins as a collection of random parameters. Pretraining turns it into a base language model by repeating next-token prediction across a very large text corpus. The model gradually acquires statistical regularities, factual associations, language patterns, and reusable capabilities.

4.1 From Corpus to Batches

A **corpus** is the collection of text used for training. It may combine web pages, books, code, scientific writing, conversations, and other domains. The mixture matters because the model can only learn from patterns that appear in its training distribution.

Raw documents are cleaned, filtered, deduplicated, and tokenized. The tokens are then packed into fixed-length training sequences. Several sequences form a batch.

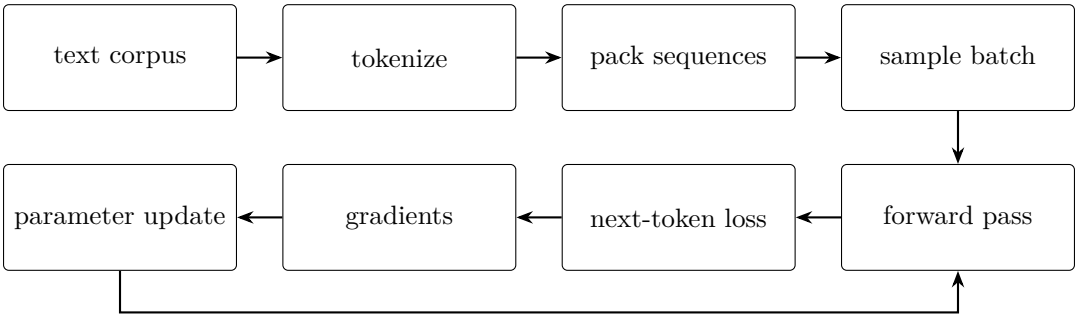


Figure 4.1: Pretraining repeatedly turns raw text into a next-token learning signal.

Sequence packing attempts to use the context window efficiently. Short documents may be joined with boundary tokens, while long documents may be divided across sequences. The model sees token IDs and prediction targets, not the original file structure unless that structure is explicitly represented.

Data quality and data quantity interact. More tokens create more learning opportunities, but repeated, corrupted, or low-value text may waste compute or teach undesirable patterns. Pretraining research therefore includes corpus composition, filtering, sampling, and ordering, not only model architecture.

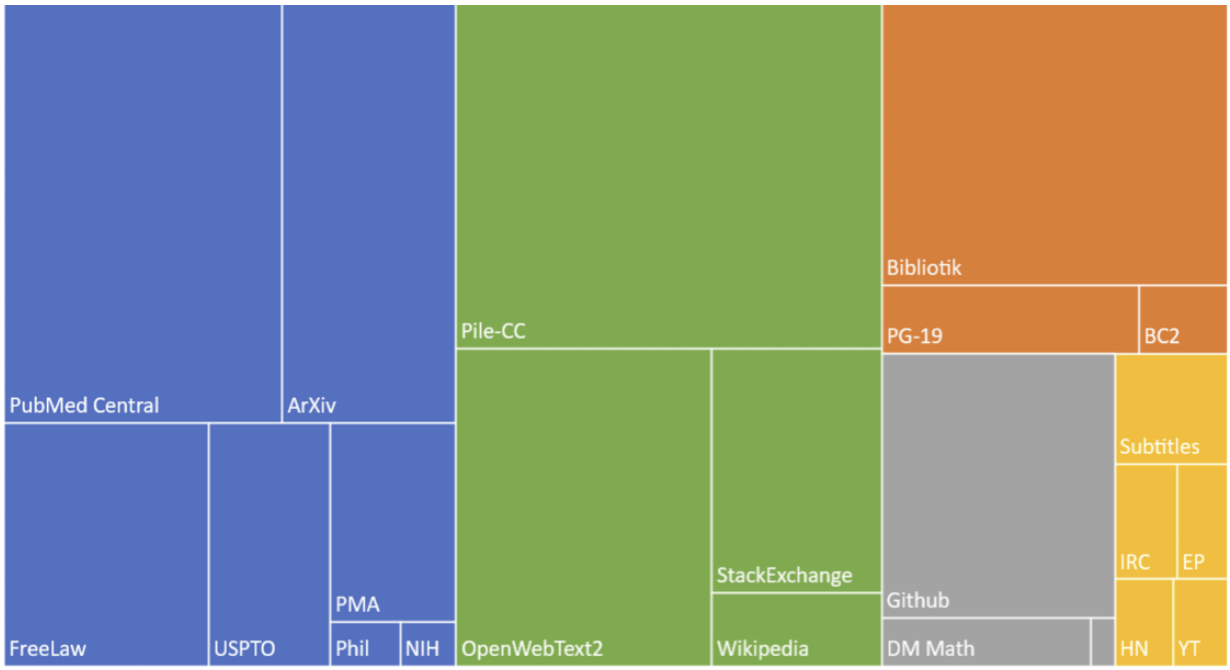


Figure 4.2: Rectangle area shows each source's share in one historical Pile mixture.

4.2 The Next-Token Objective

For a sequence $x_1, x_2, \ldots, x_T$, the model predicts token x_t from the prefix $x_1, \ldots, x_{t-1}$. The average training loss is

$$\mathcal{L} = -\frac{1}{T-1} \sum_{t=2}^{T} \log p_\theta(x_t \mid x_1, \ldots, x_{t-1}). \tag{4.1}$$

The subscript θ reminds us that the probabilities depend on the model parameters. The sum includes the prediction loss at every position after the first, and division by $T-1$ forms an average. The minus sign and logarithm turn a higher correct-token probability into a smaller loss, while assigning a very low probability to the correct token produces a large penalty.

This objective is called self-supervised because the text supplies its own labels. No person needs to annotate the next word. The target at one position is simply the token that already follows in the corpus.

One sequence therefore produces many learning signals. If a sequence contains 1,024 tokens, the model can be trained on roughly 1,023 next-token predictions during the same forward pass.

The loss does not directly say "learn grammar" or "memorize this fact." It rewards better prediction. Broader capabilities can emerge because good prediction requires the model to represent many recurring structures in the data.

4.3 One Training Step

A training step begins with a batch. The model performs a forward pass and produces logits (raw scores for every vocabulary token) at each token position. The next-token loss compresses those predictions into one scalar. Backpropagation computes gradients, and the optimizer updates the

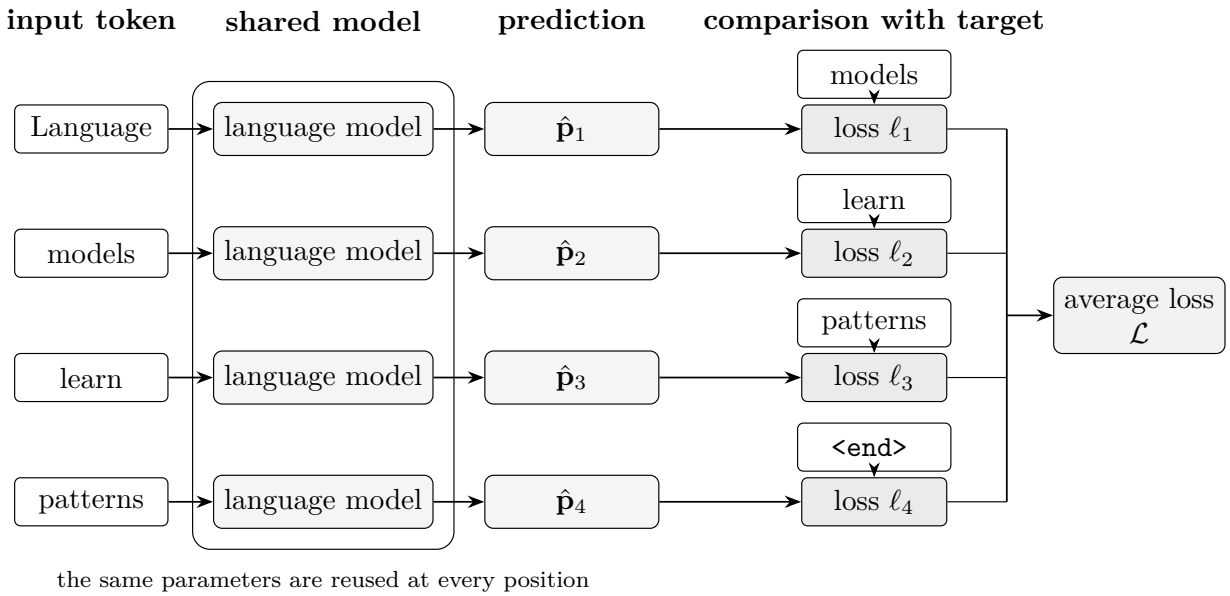


Figure 4.3: Each position predicts the next token; the resulting losses are averaged before backpropagation.

parameters.

The update still follows the principle introduced in Chapter 1 (1.11):

$$\theta_{s+1} = \theta_s - \eta_s \nabla_\theta \mathcal{L}_s. \tag{4.2}$$

Here s is the training-step number, θ_s is the current parameter collection, $\mathcal{L}_s$ is the loss at that step, $\nabla_\theta \mathcal{L}_s$ is its gradient with respect to the parameters, and η_s is the learning rate.

In practice, training systems may accumulate gradients across several smaller batches before updating. They may also split model computations across many accelerators. Those systems change how the calculation is executed, but not the conceptual loop.

A **checkpoint** is a saved copy of the model parameters and training state. Checkpoints allow a run to recover after failure and let researchers compare how capabilities develop at different stages.

4.4 Hyperparameters and Schedules

A **hyperparameter** is a training choice set by researchers rather than learned by the model. These choices determine how large the model is, what it sees, and how its parameters are updated. They interact, so changing one often requires reconsidering several others.

Learning rate. The learning rate controls the size of each parameter update. A value that is too large can overshoot useful settings and make the loss unstable. A value that is too small can make learning unnecessarily slow.

Optimizer choice. The optimizer converts gradients into parameter updates. Different optimizers smooth and rescale gradient information in different ways. The common foundation is still the same: follow the gradient toward lower loss while keeping updates stable.

Batch size. Batch size determines how many sequences contribute to one update. Larger batches average over more examples and can use hardware efficiently, but they require more memory and may need a different learning rate.

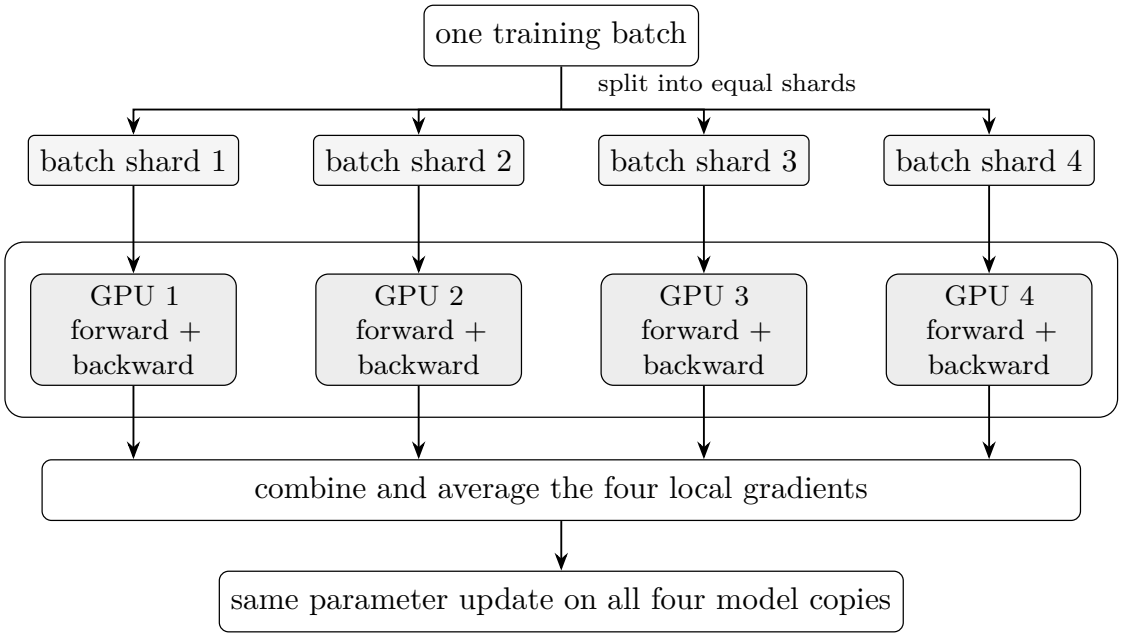


Figure 4.4: Data parallelism splits a batch across model copies, averages gradients, and applies a shared update.

Sequence length. Sequence length determines how much context appears in one training example. Longer sequences expose longer-range relationships, but attention becomes more expensive as the sequence grows.

Model depth and width. Depth is the number of Transformer blocks. Width is the size of each hidden representation. Increasing either usually increases model capacity, parameter count, and compute per token.

Data mixture. A data mixture sets the proportions of sources, languages, domains, and formats seen during training. It changes what the model practices, even when the architecture and total token count stay fixed.

Training tokens. The total number of training tokens measures how much text the optimizer processes. More tokens provide more learning opportunities, but only if data quality and compute remain adequate.

Weight decay. Weight decay gently discourages parameters from growing without bound. It is one form of regularization, meaning a training choice intended to improve behavior beyond the exact examples in the training batches.

Gradient clipping. Gradient clipping limits an unusually large gradient before it produces a destructive update. It is a safety mechanism for rare unstable steps, not a substitute for choosing a sensible learning rate.

A **learning-rate schedule** changes the learning rate over time. Training often begins with a short warmup, where the rate rises gradually from a small value. After reaching its main level, the rate usually decreases so that later updates become more cautious.

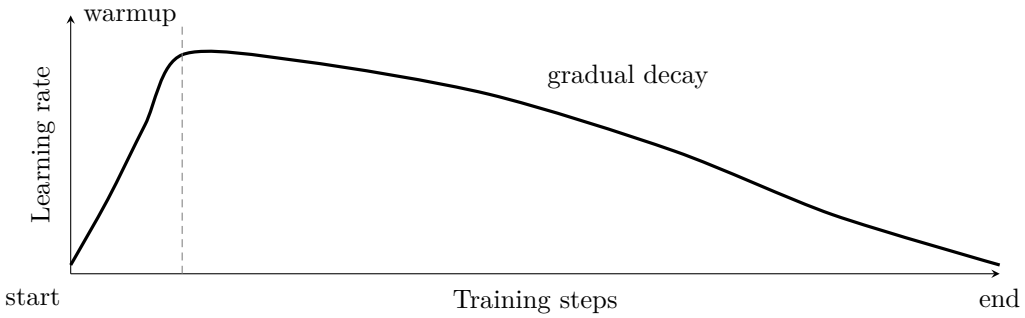
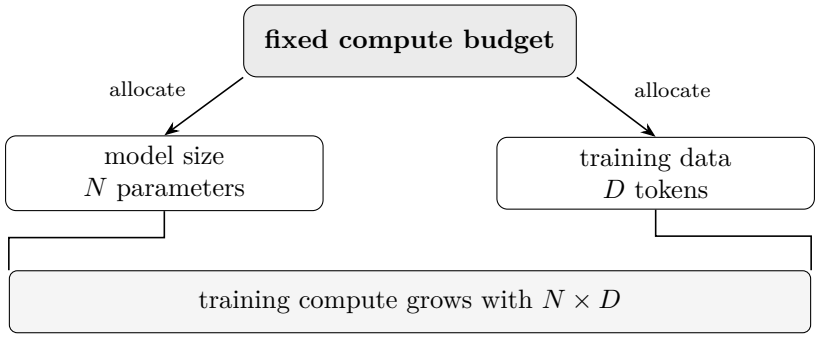


Figure 4.5: The learning rate rises during warmup and then gradually decays.

Warmup and decay. Warmup reduces the risk of large, erratic updates at the beginning of training, when the model's representations are still unstructured. Decay gradually reduces the learning rate later so that training can make finer adjustments. The figure shows the qualitative shape. Exact schedules vary across training recipes.

All of these choices share one budget. A larger model costs more per token. Longer sequences use more memory. More training tokens require more steps. Researchers therefore select a balanced training recipe rather than maximizing each quantity independently.



A fixed budget forces an allocation choice.

Figure 4.6: A fixed compute budget requires balancing model size and training tokens.

Go deeper. Stanford CS336, especially the architecture, hyperparameter, and scaling lectures, for a deeper treatment of pretraining design choices. [read]

4.5 Reading Training Curves

Training produces streams of measurements. The most basic is training loss, which should generally decrease as the model becomes better at predicting its batches. A held-out **validation set** measures the same objective on text that is not used for parameter updates.

If both curves remain high, the model may need more training, greater capacity, better data, or a different setup. If training loss keeps falling while validation loss worsens, the model may be fitting the training distribution without improving on held-out text.

A sudden spike or divergence can indicate an unstable learning rate, an unusual batch, numerical problems, or a system failure. Researchers therefore monitor loss together with gradient size, throughput, hardware utilization, and evaluation checkpoints.

Loss is not a complete measure of usefulness. Two models with similar validation loss may differ on reasoning, factual recall, multilingual behavior, or safety. Pretraining runs are therefore judged through both learning curves and downstream evaluations.

The next figure places those two views together. Individual downstream capabilities can improve at different rates and may fluctuate even while the general training objective improves.

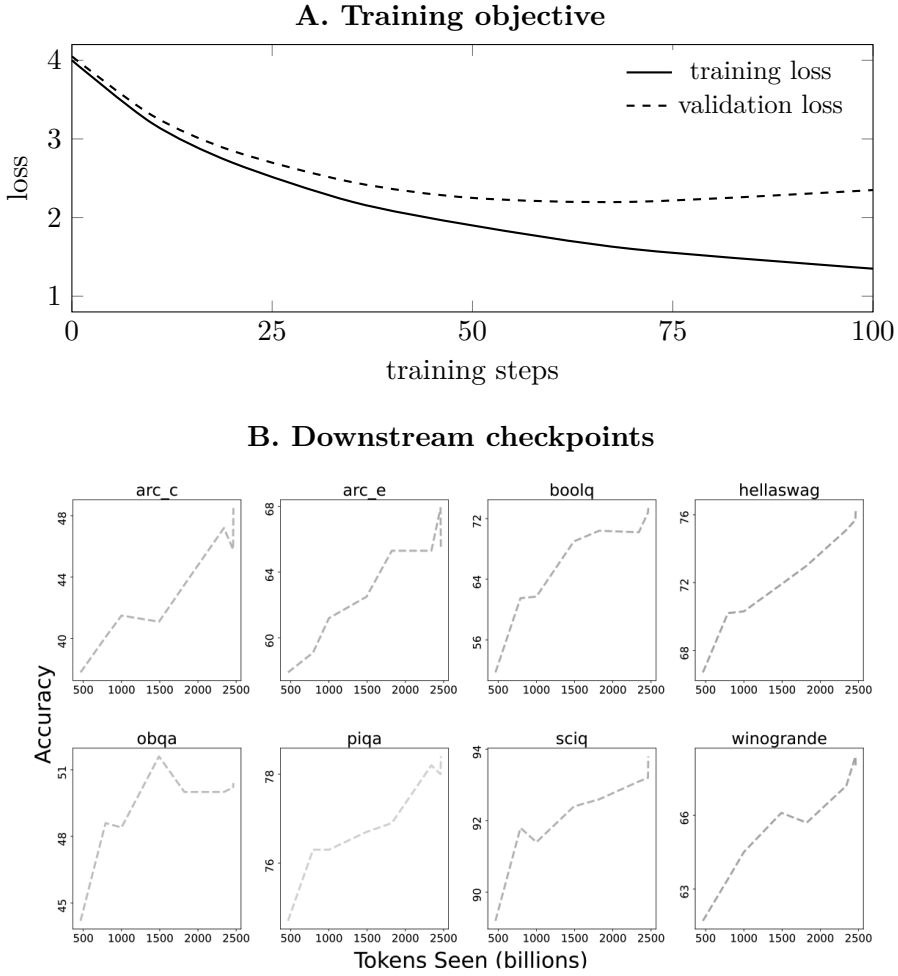


Figure 4.7: Loss tracks the training objective; checkpoint evaluations track individual capabilities.

4.6 Summary

Pretraining repeatedly turns corpus text into next-token learning signals. Data is tokenized and packed into batches, the model predicts each continuation, and gradients update the parameters. Hyperparameters control the scale and stability of this process. Learning-rate schedules make updates large enough to learn early and smaller later, while training and validation curves reveal whether the run is progressing normally.

Chapter 5

Post-Training: Shaping Model Behavior

Pretraining produces a model that is good at continuing text. A useful system must also recognize instructions, adopt consistent response conventions, and avoid behaviors that are undesirable for its intended setting. Post-training changes the model toward those goals using smaller, more targeted datasets.

5.1 Why Post-Training Exists

A base model has learned from many kinds of text: articles, dialogue, code, lists, fiction, and fragments of web pages. When given a request, it may continue the request, imitate surrounding text, or produce an answer. Pretraining alone does not clearly specify which behavior is preferred.

Post-training supplies that specification. It does not create the model's entire knowledge from scratch. Instead, it changes how existing capabilities are selected, formatted, and applied.

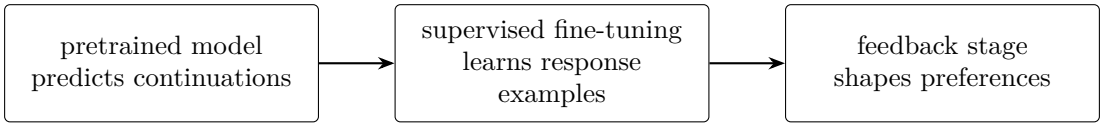


Figure 5.1: Post-training changes a broad continuation model into a model that follows a narrower behavioral specification.

The boundary is conceptual rather than absolute. Continued pretraining may adapt a base model to a domain using more next-token data. Post-training usually uses instruction examples or feedback that directly expresses desired behavior.

5.2 Supervised Fine-Tuning

Supervised fine-tuning, or SFT, trains the model on examples of an input followed by a desired response. The model still uses next-token prediction, but the sequences are deliberately constructed to demonstrate instruction-following behavior.

A training example may include a system instruction, a user request, and an assistant response. Special tokens or formatting boundaries tell the model where each part begins and ends.

The loss is often applied mainly to the target response rather than the user prompt. This focuses the update on learning what the model should generate. If the response tokens occupy positions in a set

Instruction: Explain why the sky appears blue in two sentences.

Target response: Short-wavelength blue light is scattered more strongly by the atmosphere than longer wavelengths. That scattered light reaches our eyes from many directions.

Figure 5.2: Supervised fine-tuning uses examples of the input and response behavior the model should imitate.

R, the simplified objective is

$$\mathcal{L}_{\text{SFT}} = -\frac{1}{|R|} \sum_{t \in R} \log p_\theta(x_t \mid x_1, \ldots, x_{t-1}). \tag{5.1}$$

The notation $|R|$ means the number of response-token positions. The rest of the equation is the same next-token loss used in pretraining (4.1).

The quality of SFT depends heavily on the examples. A narrow dataset can make behavior rigid. Inconsistent demonstrations can teach conflicting conventions. High-quality examples should cover the intended tasks, response styles, difficulty levels, and failure cases.

Few-Shot Learning for QA

Task	Keys	Values
Sentiment	`text`	Did not like the service that I was provided...
	`label`	0
	`text`	It sounds like a great plot, the actors are first grade, and...
	`label`	1
NLI	`premise`	No weapons of mass destruction found in Iraq yet.
	`hypothesis`	Weapons of mass destruction found in Iraq.
	`label`	2
	`premise`	Jimmy Smith... played college football at University of Colorado.
	`hypothesis`	The University of Colorado has a college football team.
	`label`	0
Extractive Q/A	`context`	Beyoncé Giselle Knowles-Carter is an American singer...
	`question`	When did Beyoncé start becoming popular?
	`answers`	{ `text`: ['in the late 1990s'], `answer_start`: 269 }

Figure 5.3: Different task formats can become instruction-tuning examples.

Go deeper. Jurafsky and Martin, Post-Training, for a broader map of instruction tuning, preferences, and modern post-training methods. [read]

5.3 Learning from Preferences

Demonstrations show one acceptable response, but sometimes the easier signal is a comparison. Given two candidate responses to the same prompt, a reviewer may indicate which one is better.

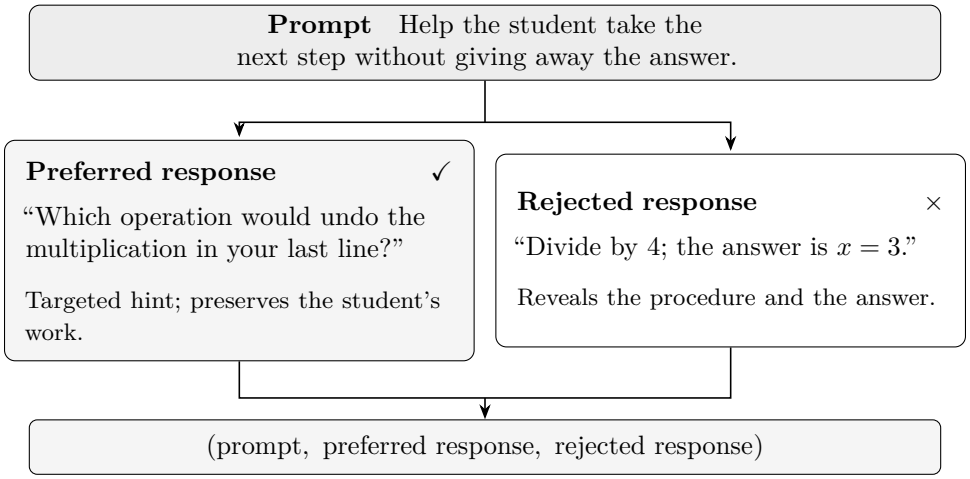


Figure 5.4: A preference example identifies which response better matches the desired tutoring behavior.

Preference data can be used in several ways. One approach trains a separate reward model to predict which responses people would prefer. Another directly updates the language model so that preferred responses become more likely than rejected ones. Reinforcement-learning and direct-preference methods are different implementations of this broader feedback idea.

At this level, the important object is the comparison:

$$(\text{prompt}, \text{preferred response}, \text{rejected response}).$$

The training signal says which behavior should be favored. It does not automatically explain why. Careful rubric design and reviewer agreement therefore matter.

Preference signals can also come from verifiable outcomes. A mathematical answer can be checked, code can be tested, and a structured prediction can be compared with a known target. These settings reduce reliance on subjective judgments, but they introduce a different risk: the model may exploit weaknesses in the checker rather than learn the intended behavior.

5.4 Keeping Behavior Stable

Post-training improves some behaviors while potentially weakening others. A model may become more helpful on instruction tasks but lose diversity, become overly cautious, or forget capabilities that were rare in the adaptation data.

Evaluation should therefore compare the model before and after each stage. Useful measurements include task accuracy, instruction following, confidence calibration, refusal behavior, and performance on capabilities that should remain unchanged. Confidence calibration asks whether the model is more confident when it is more likely to be correct.

Figure 5.5 gives a simple picture of this tradeoff. It is conceptual rather than a report of measured results: post-training is intended to improve the behaviors it targets, but unrelated capabilities still need to be checked for regressions.

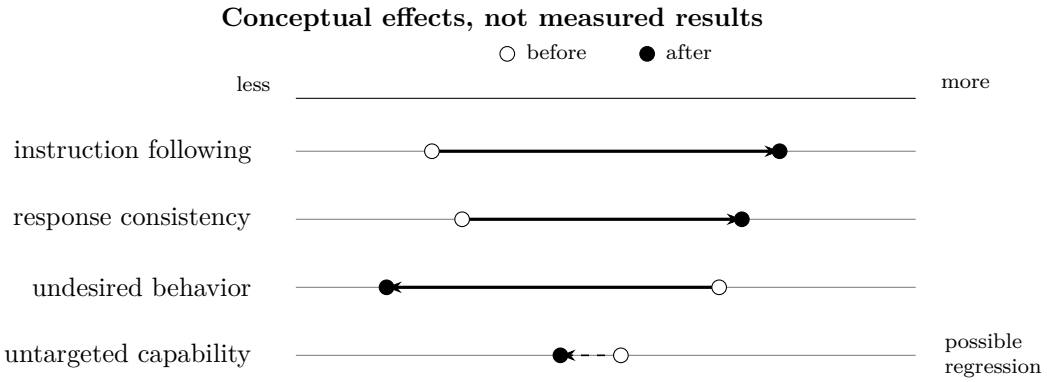


Figure 5.5: Post-training can improve targeted behaviors while leaving other capabilities unchanged or weaker.

The open circles mark behavior before post-training, and the filled circles mark behavior afterward. Instruction following and response consistency ideally move toward "more," while undesired behavior moves toward "less." The final row shows the main warning: a capability that was not represented in the adaptation data may drift slightly in the wrong direction.

Post-training data can encode the preferences and blind spots of its creators. For an educational model, a response that looks polished is not enough. The system should be evaluated on whether its feedback is correct, whether hints preserve student work, and whether interaction improves later independent performance.

The central lesson is that post-training defines behavior through examples and feedback. It must be paired with evaluation because the model will optimize the signal it receives, including any imperfections in that signal.

5.5 Summary

Post-training turns a broad continuation model toward a more specific behavioral role. Supervised fine-tuning teaches from desired responses. Preference data teaches which outputs should be favored. These stages can improve instruction following and consistency, but they may also introduce regressions or reward shortcuts. The quality of the data and the evaluation design are therefore part of the training method itself.

Sources Used

The books, course materials, explanatory notes, and research papers below informed the curriculum. Each title links to the corresponding book, course, or paper.

[1] Daniel Jurafsky and James H. Martin. *Speech and Language Processing*. Third edition, online manuscript released January 6, 2026.

[2] Sebastian Raschka. *Build a Large Language Model (From Scratch)*. Manning Publications, 2024.

[3] Tengyu Ma, Anand Avati, Kian Katanforoosh, and Andrew Ng. *CS229 Lecture Notes: Deep Learning*. Stanford University.

[4] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[5] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*. Cambridge University Press, 2023.

[6] Stanford CS336 course staff. *Language Modeling from Scratch*. Stanford University course materials.

[7] Jay Alammar. "The Illustrated Transformer." 2018. Figures reused under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

[8] Long Ouyang et al. "Training Language Models to Follow Instructions with Human Feedback." *Advances in Neural Information Processing Systems*, 2022.

[9] Leo Gao et al. "The Pile: An 800GB Dataset of Diverse Text for Language Modeling." 2021.

[10] Dirk Groeneveld et al. "OLMo: Accelerating the Science of Language Models." 2024.

3Blue1Brown Visual Companions

Individual videos are linked where they are most relevant in the lessons. The complete source playlists are listed here.

[V1] Grant Sanderson. *Essence of Linear Algebra*. 3Blue1Brown.

[V2] Grant Sanderson. *Neural Networks*. 3Blue1Brown.