Mathematical Foundations of Artificial Intelligence: A Focused Review of Core Concepts and Practical Applications

Connecting School-Level Mathematics with Modern Artificial Intelligence Systems

Vadlamudi Sai Chanakya
Student, Hyderabad, India

October 2026

Abstract

Artificial intelligence (AI) systems appear complex, yet their core operations rest on a small set of mathematical ideas that are accessible to students with Class 11–12 mathematics. This narrative review explains those ideas and shows where each one appears in practice. Vectors and matrices represent data and parameters; derivatives and gradients measure how a model's error changes; optimization uses gradients to learn; probability and statistics describe uncertainty and prediction; eigenvalues and principal component analysis compress information; and entropy and cross-entropy supply the loss functions of classification and language modelling. The review then shows how these tools combine in embeddings, vector search, retrieval-augmented generation and Transformer attention. Each topic follows one pattern: concept, formula, symbol definitions, a fully worked numerical example, an AI application and a statement of limitations. All calculations were recomputed independently, and advanced material is deliberately simplified or labelled as an extension. The paper is not a systematic review and reports no original experiments.

Keywords: *artificial intelligence, mathematics, vectors, matrices, gradient descent, probability, eigenvalues, principal component analysis, entropy, embeddings, retrieval-augmented generation, Transformers*

1. Introduction: Mathematics as the Foundation of AI

1.1 Why mathematics is essential to AI

Artificial intelligence covers systems that classify images, translate languages, recommend products and generate text. Whatever the task, a computer cannot work with a photograph or a sentence directly; it works with numbers. The central idea of this paper is that an AI system is a sequence of mathematical transformations:

data → numerical representation → model → learning → prediction or generation.

Each arrow in this chain is a piece of mathematics that a senior secondary student has already met in some form. A list of numbers describing an object is a vector. A table of such lists is a matrix. The rule that tells a model how wrong it is can be differentiated, and the slope of that rule tells the model how to improve. Uncertainty about an answer is expressed as a probability. The aim here is a compact route through them using only school-levelmathematics, with university-level ideas introduced gently.

Mathematics performs six connected functions in AI: it *represents* data, *measures* similarity and error, *drives learning* through optimization, *quantifies uncertainty*, *infers* outputs from inputs, and *generates* new content by sampling from probability distributions.

Figure 1.1 shows how the disciplines covered in this paper map onto the stages of the pipeline above.

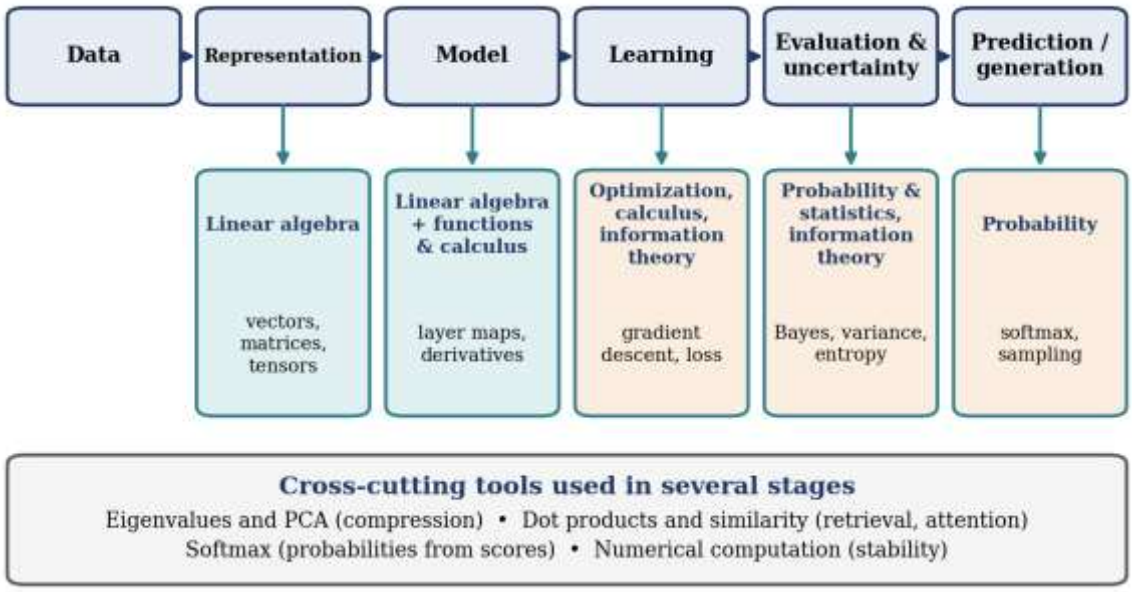


Figure 1.1. Mathematical disciplines mapped to the stages of an AI pipeline. Linear algebra represents the data and, with functions and calculus, defines the model; optimisation, calculus (gradients) and information theory (the loss) drive learning; probability and statistics, with information theory, describe uncertainty and evaluation, and probability (softmax, sampling) produces the prediction or generation. The figure is an original conceptual diagram prepared for this paper.

1.2 Research objectives and scope

The paper has four objectives:

(i) to explain the mathematical concepts that most directly underlie modern AI;

(ii) to show each concept through a small, fully calculated numerical example;

(iii) to connect each concept to a concrete AI mechanism; and

(iv) to state clearly where simplified presentations stop being accurate.

The scope is deliberately focused. Seven areas are covered: vectors, matrices, calculus, optimization, probability and statistics, eigenvalues with dimensionality reduction, and information theory. A final technical section shows how they combine in embeddings, retrieval-augmented generation (RAG) and Transformer attention. Topics that need university-level mathematics, such as singular-value decomposition theory, convergence proofs, graph neural networks, reinforcement learning and the detailed mathematics of generative image models, are omitted or mentioned only briefly and labelled as *extensions*.

Assumed background. Readers should know Class 11–12 algebra, coordinate geometry, vectors, matrices, differentiation, probability and basic statistics. Every symbol is defined where it first appears, and Appendix A (Table A.1) collects the notation. Standard conventions are used throughout: lowercase bold letters such as denote vectors, uppercase bold letters such as denote matrices, and italic letters such as or denote scalars.

1.3 Review methodology and relationship to earlier work

This is a *narrative review*, not a systematic review and not an experimental study. No search protocol, screening procedure or quantitative synthesis is claimed. The paper is derived from the author's longer manuscript (September 2026), which surveyed a wider range of topics. For this paper that manuscript was re-read in full and its content was sorted into material to retain, simplify, merge or omit according to its usefulness for a reader at the stated level. The sources cited here are drawn from the earlier manuscript's reference list: foundational textbooks and papers for definitions and established methods, and a small number of widely cited papers for modern

applications. Every citation in the text has a matching entry in the reference list, and sources that no longer support any statement were dropped.

All worked examples were designed by the author and recomputed independently by script, with one exception: the five data points of Worked example 7.2 come from the PCA tutorial of Smith [28], and the calculations on them are the author's own. Figures are original diagrams prepared for this paper. Because the paper reuses ideas from an earlier manuscript by the same author, the prior work is disclosed on the title page and here, and any venue should be informed of it according to its own policy on prior work.

Established facts versus interpretations. Definitions, identities and calculations are mathematical facts. Statements about what a quantity "means" for an AI system, such as the claim that a vector captures meaning, are conceptual interpretations, and the text marks them as such.

Relationship to existing treatments. Deisenroth, Faisal and Ong [29] present a structured mathematical foundation for machine learning, including linear algebra, analytic geometry, matrix decompositions, vector calculus, optimisation, probability and statistics. Their textbook derives several central machine-learning methods and includes worked examples; it is intended for readers building the mathematical skills needed to study machine learning in greater depth. Strang [30] develops linear algebra from vectors, matrices and linear systems through vector spaces, eigenvalues and singular-value decomposition, with applications to data science and learning. Goodfellow, Bengio and Courville [1] provide a more advanced treatment of deep-learning foundations, including the mathematical ideas behind neural-network training. The present review does not replace these resources or claim greater mathematical depth. Its narrower educational contribution is to reorganise a selected subset of these established ideas around a Class 11–12 entry point: each topic is introduced through a short explanation, a defined formula, a hand-checkable example, an AI mechanism and an explicit limitation. It then traces how the same tools reappear in embeddings, retrieval-augmented generation and Transformer attention. This is an instructional synthesis, not a new mathematical method or an exhaustive survey.

Educational contribution and intended reader. The review is designed for senior-secondary students who know introductory algebra, vectors, matrices, differentiation and probability. It does not assume prior university-level linear algebra or machine learning. The sequence is cumulative: vectors and matrices establish numerical representation; calculus and optimisation explain learning; probability and information theory explain prediction and loss; eigenvalues and PCA introduce dimensionality reduction; and the final section combines these ideas in modern AI examples. The intended outcome is conceptual and mathematical literacy, not the ability to implement or independently evaluate a production-scale AI system.

1.4 How mathematical concepts connect to AI applications

The pattern *mathematical concept* → *operation* → *AI mechanism* → *application* organises the rest of the paper. Table 1.1 previews the mappings; each row is developed in the section named in its last column.

Table 1.1. Preview of how mathematical concepts become AI mechanisms.

Concept	Operation	AI mechanism	Section
Vector, dot product	Cosine similarity	Semantic search, embeddings	Section 2
Matrix	Linear transformation	Neural-network layer	Section 3
Derivative, chain rule	Gradient	Backpropagation	Section 4
Gradient, loss	Parameter update	Gradient descent training	Section 5
Conditional probability	Bayes' rule, softmax	Classification, language models	Section 6
Eigenvector	Covariance analysis	PCA, data compression	Section 7
Logarithm, probability	Cross-entropy	Classification loss	Section 8
Dot product, softmax	Scaled attention	Transformers, RAG	Section 9

Reading the paper in order is recommended, because later sections reuse earlier ones: the dot product of Section 2 reappears in attention, and the gradient of Section 4 drives the learning of Section 5.

1.5 A first complete example: one learning step

Before the individual tools are developed, a single learning step shows how they fit together. Suppose a model predicts a number from one input using the rule , where is the input, is the one learnable parameter and is the prediction. The model is given one training example with input and correct output , and its parameter starts at . The loss is the squared error , and the learning rate is (the size of each correction step).

> **Worked example 1.1: one training step.** *Representation:* the input is the number . *Prediction:* . *Loss:* . *Gradient:* differentiating with respect to gives , so increasing would reduce the loss. *Update:* . *Check:* the new prediction is and the new loss is , much smaller than .

Every stage of this small example is developed later in the paper: the representation of data as vectors (Section 2), the model as a matrix operation (Section 3), the gradient (Section 4), the update rule (Section 5), and, for classification and language tasks, probabilities and cross-entropy in place of squared error (Section 6, Section 8). Repeating this step over many examples and many parameters is what is meant by training an AI model.

2. Vectors and Geometric Representations

2.1 Vectors and vector operations

A **vector** is an ordered list of numberThe set of all such vectors with real components is written . In coordinate geometry a vector in is an arrow from the origin to the point ; in AI the same idea extends to hundreds or thousands of dimensions, where each component records one measured or learned feature. A house described by (area, rooms, age) is a vector in ; a word represented by 768 learned numbers is a vector in .

Two operations are used constantly. **Vector addition** and **scalar multiplication.**

A **linear combination** of vectors is any vector . Every vector in can be written as a linear combination of the two *standard basis vectors* and , since . This idea, that a vector is built from a few fixed directions, returns in Section 3.3 and in the principal components of Section 7.3.

In machine learning, the choice of how to turn an object into a vector is a design decision. Categorical data are often written as **one-hot vectors**, with a single marking the category and elsewhere, and text can be written as a **bag-of-words** vector that counts how often each vocabulary word occurs. These simple vectors have as many components as there are categories or words, and most entries are zero, whereas learned embeddings (Section 2.4) are shorter and dense.

2.2 Magnitude, norms and distance

The **magnitude** (Euclidean norm or norm) of generalises Pythagoras' theorem:

The **distance** between two vectors is the norm of their difference, . Other norms are also used: the norm adds absolute values, and is less sensitive to a single large component. A vector with is called a *unit vector*, and dividing any non-zero vector by its norm produces one.

> **Worked example 2.1: norms.** For : and . The unit vector in the same direction is , whose norm is . The distance between and is , the length of the hypotenuse of a — right triangle.

2.3 Dot product and cosine similarity

The **dot product** of is the sum of products of matching components. It also equals the product of the magnitudes and the cosine of the angle between the vectors:

Rearranging gives the **cosine similarity**, which depends only on direction:

A value near means the vectors point the same way, means they are orthogonal (at a right angle; this describes direction only and does not mean that the items are statistically independent or share no information), and means they are opposite. Figure 2.1 shows a two-dimensional case, and the worked example below uses three dimensions so that the three measures can be compared.

> **Worked example 2.2: similarity search.** Let a query be and let three stored items be , and . First, . For : and , so and . For : , , so . For : , , so , while .

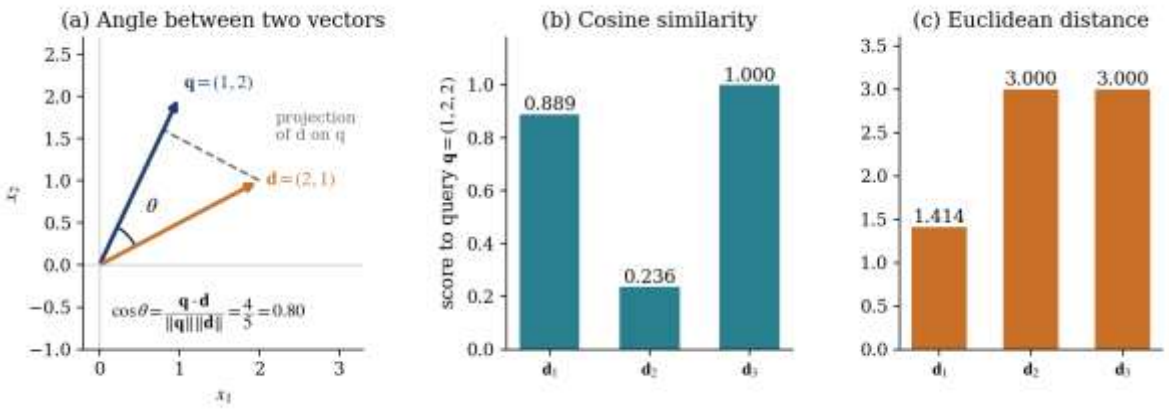


Figure 2.1. (a) Two vectors in the plane with the angle between them; for and , . (b, c) Cosine similarity and Euclidean distance of the three items in Worked example 2.2 relative to . Original figure.

The example shows why the choice of measure matters. By cosine similarity, is the best match (it has exactly the direction of), whereas by Euclidean distance is closest, because is longer. Table 2.1 compares the measures. For vectors scaled to unit length the two rankings coincide, because in that case; this is why many systems normalise embeddings before searching.

Table 2.1. Common vector measures and their behaviour.

Measure	Sensitive to length?	Typical use
Dot product	Yes	Scores in neural layers, attention
Cosine similarity	No	Text and image retrieval
Euclidean distance	Yes	Clustering, nearest neighbours
Distance	Yes	Robust comparisons

A geometric reading of the dot product is also useful. The **projection** of onto is the component of that lies along , namely . For and this is , shown as the dashed segment in Figure 2.1(a). When two vectors are perpendicular the projection is zero, which is why a zero dot product means the vectors are perpendicular.

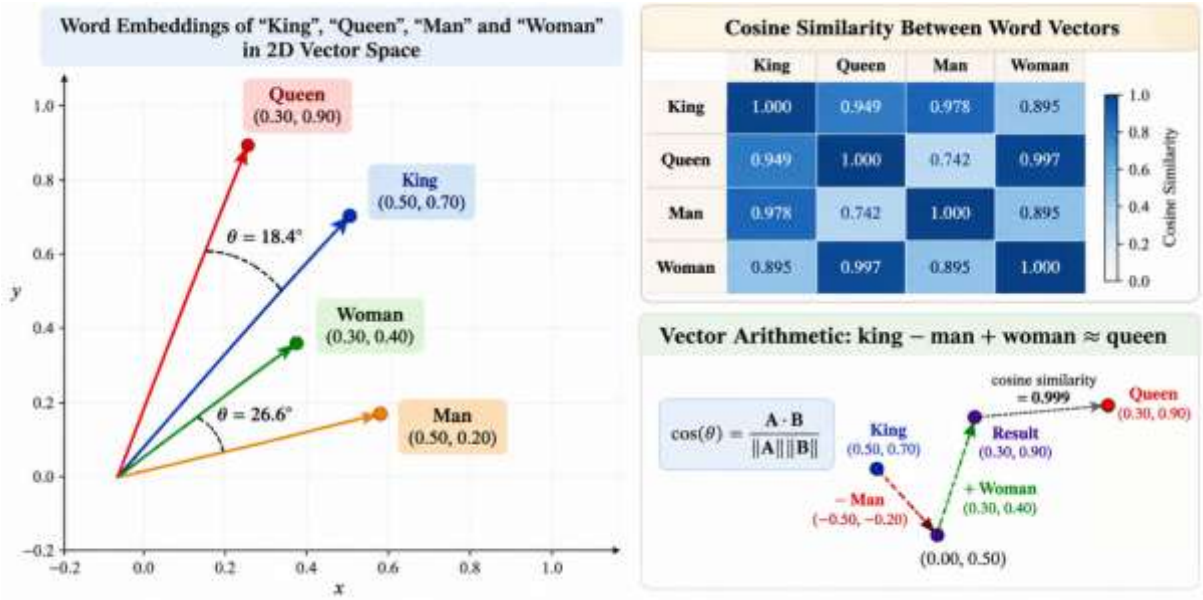
Table 2.2 gives reference values of the cosine for common angles. A cosine of about or above corresponds to an angle of or less, so the vectors are broadly aligned. Cosine similarity is also the sample correlation coefficient of Section 6.3 when the two vectors have had their means subtracted.

Table 2.2. Cosine similarity for representative angles between two vectors.

Angle	0	45	90	120	180
Meaning	same direction	similar	orthogonal	dissimilar	opposite
Cosine Similarity	1	0.707	0	-0.5	-1

2.4 Embeddings, vector representations and similarity search

An **embedding** is a function that maps an object (a word, sentence, image or user) to a vector , chosen so that objects that behave similarly receive nearby vectors. Early word-embedding methods learned such vectors from word co-occurrence, and the resulting geometry supported vector arithmetic such as *king* − *man* + *woman* ≈ *queen* . Contextual models such as BERT produce a different vector for a word in each sentence , and sentence-embedding models are trained so that cosine similarity reflects the similarity of whole sentences .



Worked example 2.3: vector arithmetic in a toy embedding space. To see why vector arithmetic can express relationships, build a hand-made two-dimensional space whose axes are (royalty, gender), with the values chosen for illustration (these are not real learned embeddings): , , , . Then . In real embeddings the equality holds only approximately, and the nearest vector to the result is found by cosine similarity. This analogy is a famous but over-simplified illustration: in standard evaluations the input words are excluded from the nearest-neighbour search, and later studies have questioned how well such analogies hold [36].

Worked example 2.4: comparing two sentences. Take the vocabulary (the, cat, sat, ran). The sentence "the cat sat" is the bag-of-words vector and "the cat ran" is . Then and , so . The sentences share two of their three words, and the cosine reflects this overlap. Bag-of-words vectors, however, ignore word order and meaning: "dog" and "cat" would have cosine even though both are animals, which is the weakness that learned embeddings address.

The dot product has three properties that are used repeatedly later: it is *symmetric* (), *linear* in each argument (), and equal to when both arguments are the same. Linearity is what allows a whole layer of a neural network to be written as a matrix product in Section 3.

Application. In similarity search, every stored item is embedded once, the query is embedded at search time, and the items with the highest cosine similarity (or smallest distance) are returned, exactly as in Worked example 2.2 but with hundreds of dimensions and millions of items. Exact search compares the query with all stored vectors, which costs about multiplications; large systems therefore use approximate nearest-neighbour indexes [8], [9], [10] (see Section 9).

Limitations. Similarity in an embedding space is a statistical regularity learned from data, not proof that two items are factually related. In very high dimensions, distances between random points tend to become more alike, which reduces the contrast between near and far items, and the quality of a search is limited by the quality of the embedding model. [31]

3. Matrices and Linear Transformations

3.1 Matrix notation and basic operations

A **matrix** is a rectangular array of numbers. A matrix has rows and columns, and denotes the entry in row and column . In AI, a dataset of examples, each with features, is stored as an matrix whose rows are the examples. Matrices of equal size are added entry by entry, and multiplying a matrix by a scalar multiplies every entry. The **transpose** swaps rows and columns, so .

Worked example 3.1: a dataset as a matrix. Three houses are described by two features, area (in units of) and number of rooms, giving the data matrix (size , one row per house). A simple model predicts a price score with the weight vector as : the three predictions are , and , so . One matrix–vector product has therefore evaluated the model on every example at once.

3.2 Matrix multiplication

If and , the product is defined only when the inner dimensions agree, and its entries are dot products of a row of with a column of : Matrix multiplication is generally not commutative (), but it is associative. Because every entry is a dot product, one matrix multiplication computes many similarity scores at once, which is why modern hardware is designed around this operation.

> **Worked example 3.2: a matrix product.** Let (size) and (size). The product has size . Entry ; entry ; entry ; entry . Hence .

Multiplying matrices corresponds to *composing* transformations: , so acts first and second. The order matters, which is the geometric meaning of non-commutativity.

> **Worked example 3.3: composing a scaling and a rotation.** Let double every length, and let rotate vectors by anticlockwise, since and . For , first and then . The product matrix gives the same result directly: .

A square matrix may have an **inverse** with , where is the *identity matrix* (ones on the diagonal, zeros elsewhere); it exists exactly when . For the inverse is . Inverses solve linear systems: if then .

> **Worked example 3.4: solving a linear system.** Let and . Here , so and . Check: . If the transformation flattens the plane onto a line, information is lost, and no inverse exists.

3.3 Linear transformations

Multiplying a vector by a matrix, , is a **linear transformation**: it maps vectors to vectors while preserving addition and scaling. The columns of are the images of the standard basis vectors and , so the matrix completely describes how the plane is stretched, rotated, sheared or reflected. The **determinant** of a matrix, for , is the factor by which areas change: its absolute value is the area (in three dimensions, volume) scaling factor, and a negative sign means that orientation is reversed, as in a reflection. If the determinant is zero, the transformation collapses the plane onto a line or a point, so no inverse exists.

> **Worked example 3.5: transforming the unit square.** Take . Then and , the two columns of . The corner maps to . The unit square, of area , becomes a parallelogram of area (Figure 3.1).

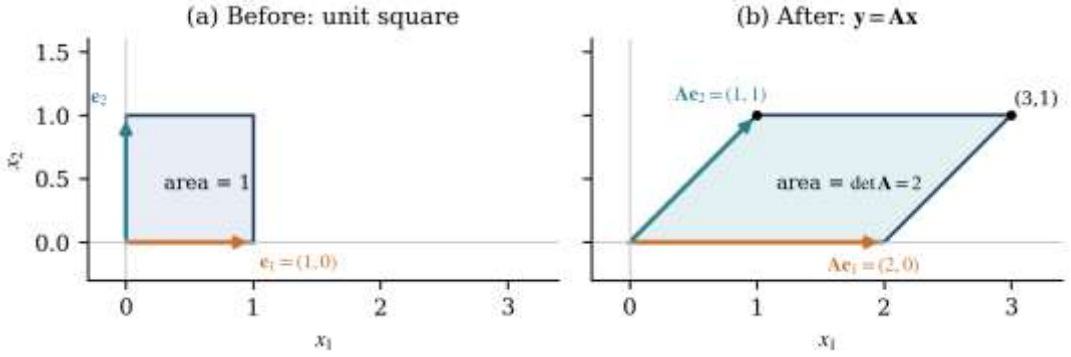


Figure 3.1. A matrix as a linear transformation. The unit square (a) is mapped by to a parallelogram (b); the images of and are the columns of , and the area is multiplied by . Original figure.

3.4 Matrices in neural networks

A basic neural-network layer applies a linear transformation followed by a simple nonlinear function ,where is the input, is the *weight matrix*, is the *bias vector* and is the output of units. A common activation is the rectified linear unit, , applied to each component. Stacking layers composes such maps, and the nonlinearity is essential: without it, a product of matrices would collapse into a single matrix (more precisely, a single affine map when biases are included) and the network could represent only linear relationships [1]. Figure 3.2 shows the shapes involved in one layer.

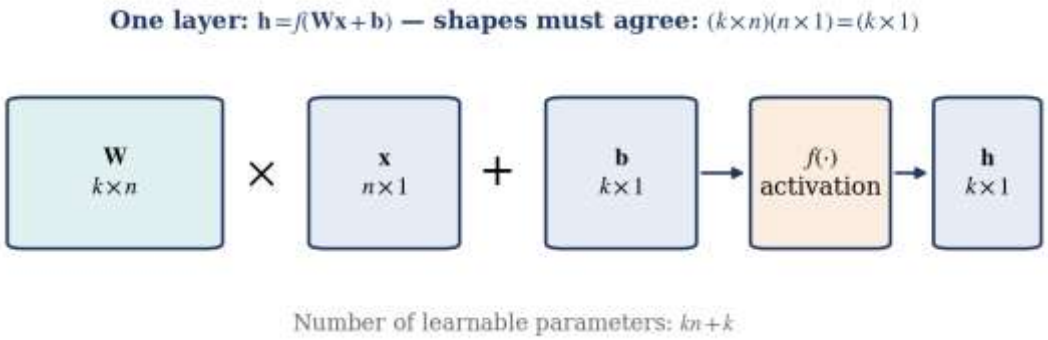


Figure 3.2. A neural-network layer as a matrix operation, with the shape of each object. The shapes must satisfy , and the layer has learnable parameters. Original figure.

When many inputs are processed together, they are stacked as the rows of a matrix (a *batch* of examples), and the whole batch passes through the layer as , where is a column of ones that copies the bias to every row. This is the form in which graphics processors execute networks, since one large matrix product replaces many small ones.

> **Worked example 3.6: one layer.** Let , and . Then , so and . The first unit is switched off because its input is negative.

The number of learnable parameters of such a layer is . For a layer mapping inputs (a image flattened into a vector) to units, this is parameters. Learning means choosing good values for all of these entries (Section 5).

3.5 Introductory tensors

A **tensor** is the generalisation of vectors and matrices to more indices: a scalar is a 0-dimensional tensor, a vector is 1-dimensional, a matrix is 2-dimensional, and so on. The number of indices is the tensor's *order*, and the list of sizes along each index is its *shape*. A colour image has shape (height, width, 3), one value for each of three colour channels, and a batch of 32 such images has an extra index. Convolutional networks, introduced for recognising handwritten digits [11], process image tensors by sliding small weight arrays over them. Text is stored as a matrix of shape (number of tokens, embedding dimension), the form used by the attention mechanism of Section 9.

Table 3.1 lists common tensor shapes. Reading the shape of every object is the single most useful habit when working with AI code, because most errors are shape mismatches that the rule detects at once.

Table 3.1. Typical tensor shapes in AI computations.

Object	Order	Typical shape	Meaning of the indices
Feature vector	1		feature
Data matrix	2		example, feature
Greyscale image	2		row, column
Colour image	3		row, column, colour channel
Batch of images	4		example, row, column, channel
Token matrix	2		token, embedding component

Limitations. A matrix describes only *linear* behaviour; the expressive power of AI models comes from alternating matrices with nonlinear functions. In practice, computations also use finite-precision floating-point numbers, so very large or very small values can cause rounding problems.

4. Functions and Calculus in AI

4.1 Functions and mathematical models

A **function** assigns to each input exactly one output. A machine-learning *model* is a function that maps an input to a prediction , where stands for the *parameters* (the numbers, such as the weights of Section 3.4, that are adjusted during learning). The simplest model is a straight line, , with parameters (slope) and (intercept). A *loss function* is a second function that measures how far the predictions are from the correct answers. Learning will mean changing so that becomes small, and calculus tells us in which direction to change it.

Table 4.1 lists functions that appear constantly in AI models, together with their derivatives, which are developed in the next subsection.

Table 4.1. Common functions in AI models and their derivatives.

Function	Role in AI
Linear	Simplest model
Square	Squared-error loss
Exponential	Softmax, likelihoods
Logarithm	Cross-entropy loss
Sigmoid	Probability output
ReLU	Hidden-layer activation

Worked example 4.1: a model, a loss and a slope. Fit to the three points using the mean squared error . At the predictions are and the residuals are , so . Differentiating, . The slope is negative, so the loss falls if is increased from ; the best value, found by setting , is .

4.2 Derivatives and rates of change

The **derivative** of at is the instantaneous rate of change of , the slope of its tangent line:

For example, has , so at the slope is : increasing slightly increases about six times as fast. A positive derivative means the function is rising, a negative one that it is falling, and zero marks a possible minimum, maximum or flat point. The *second derivative* , the derivative of , measures curvature: for it equals , so the graph is bowl-shaped and the point where (namely) is a minimum. The sigmoid function , used to turn any number into a value between and , has the convenient derivative .

Derivatives follow a few rules that suffice for most of this paper: the derivative of a sum is the sum of the derivatives, for a constant , and . The definition in Eq. (4.1) can be checked directly: for , as .

A practical consequence is the **numerical derivative check**: choosing a tiny (for example) and evaluating approximates closely, and is routinely used to test that a hand-written gradient is correct. Worked example 4.4 below uses exactly this check.

4.3 Partial derivatives and gradients

Models have many parameters, so the loss depends on several variables. The **partial derivative** is the derivative with respect to while all other variables are held constant. Collecting all partial derivatives of in one vector gives the **gradient**:

The gradient points in the direction in which increases fastest, and its magnitude is the steepness in that direction. Consequently points in the direction of fastest decrease, which is the basis of gradient descent in Section 5.

Figure 4.1 illustrates both ideas: the slope of a tangent line for one variable, and the gradient arrow pointing across the contour lines for two.

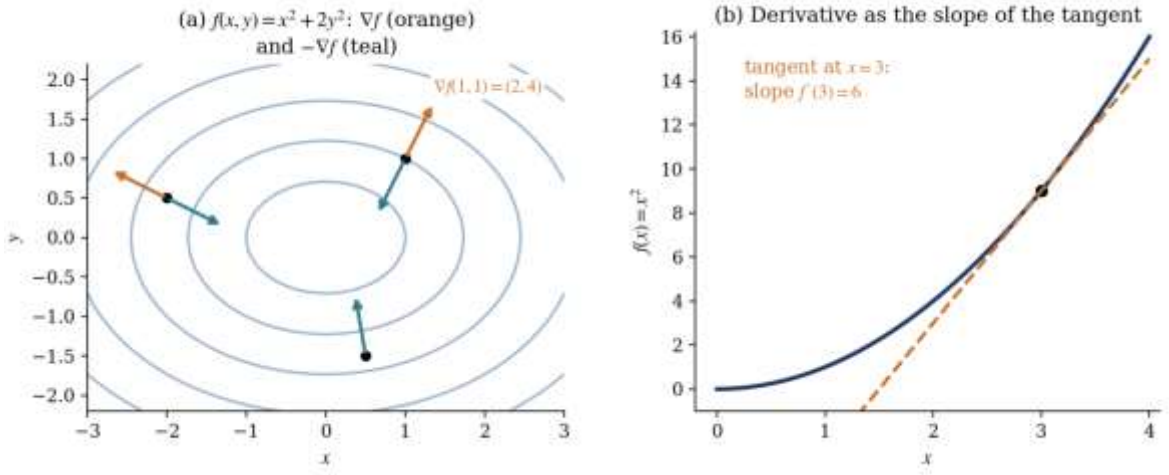


Figure 4.1. (a) Contour lines of with the gradient (orange) and the direction of fastest decrease (teal) at three points; the gradient is perpendicular to the contour line through each point where the gradient is non-zero. (b) The derivative of at is the slope of the tangent line. Original figure.

Worked example 4.2: a gradient. Let . Treating as a constant gives ; treating as a constant gives . At the point the gradient is , with magnitude . Moving a small step in the direction increases fastest.

Worked example 4.3: gradient with respect to two parameters. A model has two parameters, so its loss on one example, , has a two-component gradient. By the chain rule, and . For , , and , the prediction is and , so . Both components are negative, so increasing either or reduces the loss, and the weight is more influential than the bias because the input multiplies it.

4.4 The chain rule and backpropagation

Most models are *compositions* of functions: the output of one step is the input of the next. The **chain rule** states that the derivative of a composition is the product of the derivatives of its steps. If depends on , depends on , and depends on , then

Backpropagation is the systematic application of the chain rule to a neural network: values are computed forward through the layers, and derivatives are then accumulated backward from the loss to every parameter, reusing intermediate results so that all gradients are obtained in one backward sweep [1], [12]. Modern software performs this automatically ("automatic differentiation"), but the underlying mathematics is exactly Eq. (4.3) applied repeatedly. For a two-layer network with , the output error is passed back through the transpose of the downstream weight matrix, ($\odot$ is element-wise multiplication), and the gradients are .

Worked example 4.4: backpropagation through one neuron. Consider one neuron with , output and squared-error loss , where is the input, the target, and the parameters are and . *Forward pass:* ; ; . *Local derivatives:* ; ; . *Chain rule:* . The negative sign says that increasing slightly would decrease the loss. As an independent check, evaluating at and taking the central difference gives , in agreement (Figure 4.2).

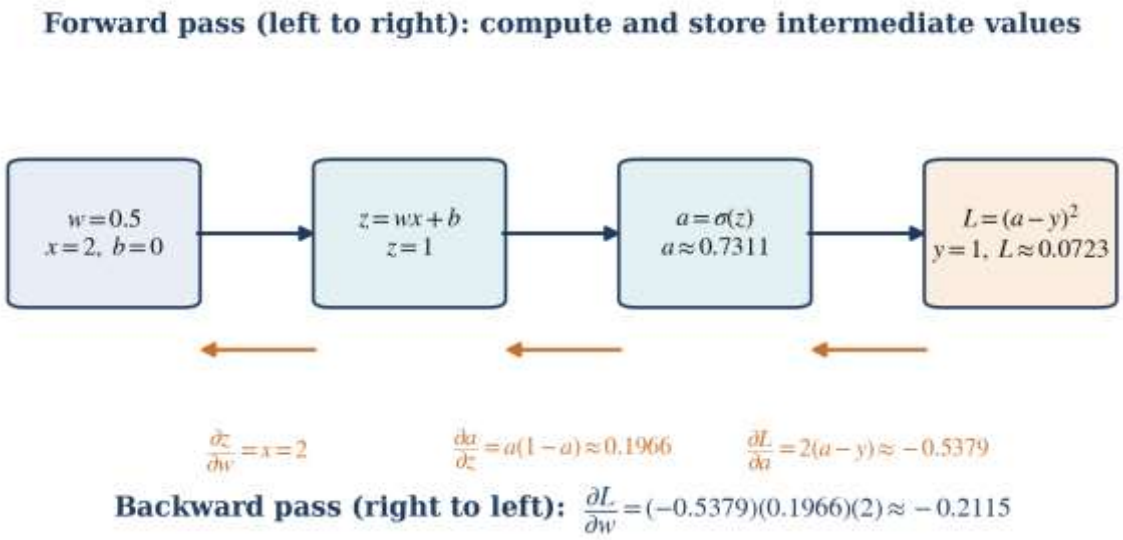


Figure 4.2. Backpropagation through the one-neuron model of Worked example 4.4. The forward pass (top) stores intermediate values; the backward pass (bottom) multiplies the local derivatives along the path from the loss back to . Original figure.

Application and limitations. The gradient computed above is precisely the quantity that gradient descent uses to update (Section 5). The chain rule also explains a practical difficulty: in a very deep network the product in Eq. (4.3) contains many factors, and if they are typically smaller than (as is) the product shrinks towards zero, so early layers learn slowly. This "vanishing gradient" effect is one reason activations such as ReLU are widely used. [35] Calculus also requires the functions to be differentiable; at isolated points such as for ReLU, a convention is chosen for the derivative.

5. Optimization and How AI Learns

5.1 Objective and loss functions

Learning is framed as an *optimisation problem*: find the parameters that make an objective as small as possible,

where is the **loss function**, a non-negative number measuring how badly the model fits the training data. For regression, a standard choice is the *mean squared error*, , the average squared gap between predictions and targets over examples. For classification the usual choice is cross-entropy (Section 8). The loss is chosen so that smaller values mean better behaviour on the task.

> **Worked example 5.1: gradient descent for a regression problem.** Use the data and loss of Worked example 4.1: with . Start at , where and . With : . At the residuals are , so and ; hence and . The loss has fallen from to to , and is approaching the exact minimiser , where .

5.2 Gradient descent and the learning rate

Since points in the direction of fastest decrease (Section 4.3), a sensible strategy is to take repeated small steps in that direction. **Gradient descent** updates the parameters by

where counts iterations and is the **learning rate**, the step size. If is too small, progress is slow; if it is too large, the steps overshoot the minimum and the loss may oscillate or even grow.

> **Worked example 5.2: a complete gradient-descent run.** Minimise , whose gradient is and whose minimum is at . Start at with . The update is . *Step 1:* , so and . *Step 2:* , so and . *Step 3:* , so and . *Step 4:* and . *Step 5:* and . The distance to the minimum shrinks by the factor at every step, so and the iterates approach without reaching it exactly. The loss declines at the same geometric rate (Figure 5.1).

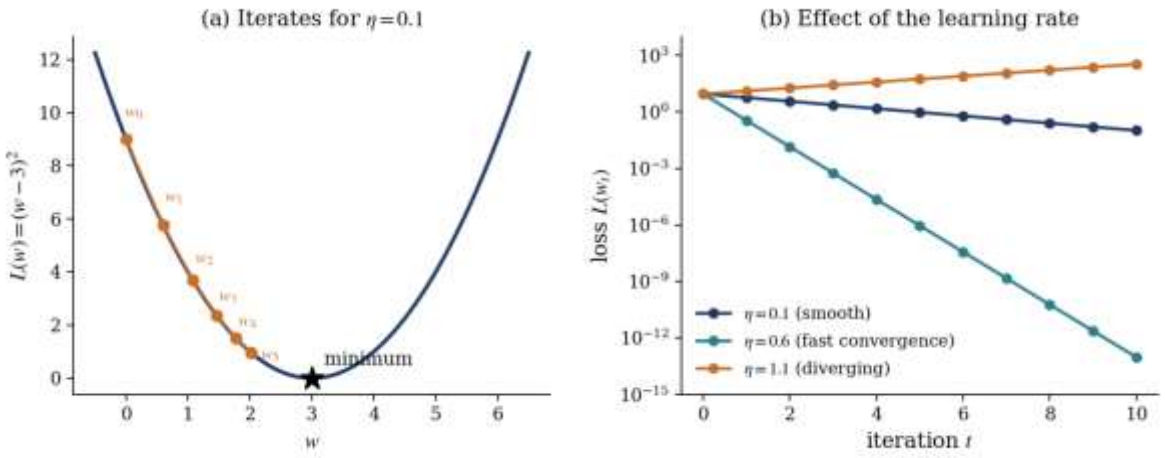


Figure 5.1. Gradient descent on starting from . (a) The first five iterates for move down the parabola towards the minimum. (b) Loss against iteration for three learning rates (logarithmic vertical axis): decreases steadily, oscillates around the minimum but converges rapidly (error factor per step), and diverges (error factor). Values computed from the update rule. Original figure. For η = 0.6 the error w_t − 3 alternates in sign, but a squared loss hides this.

The learning-rate effect in Figure 5.1 can be derived exactly for this loss. The error is multiplied by at each step, so the iteration converges when , that is, for . With the minimum is reached in one step, and gives a factor of magnitude , so the error grows. For the least-squares objective , the gradient is , and full-batch gradient descent with a constant step converges when , where is the largest eigenvalue of (assumed positive). A minimiser always exists, because the normal equations are always solvable; it is unique exactly when is invertible, that is, when has full column rank. In Worked example 5.1 the second derivative of is , so the condition is , which satisfies. For real neural networks no such formula exists, and the learning rate is chosen by experiment. A common refinement is to let the step size shrink during training, for example ; with and this gives at and at , taking large steps early and small, careful steps near the end. Training is usually stopped when the loss on validation data stops improving or when the gradient norm becomes very small, since marks a stationary point.

With more than one parameter, the shape of the loss surface matters. Consider , whose gradient is : the surface is four times steeper in than in . Starting at with , the first step is , the second is and the third is , with decreasing as . The steep coordinate is almost solved after one step, while the shallow coordinate shrinks by only a factor each step. Such uneven curvature is the main reason why practical optimisers rescale the step for each parameter or accumulate momentum.

Not every loss has a single minimum. Take , with , which is zero at . The points are minima () and is a local maximum. Starting at with , , so , and the iterates move towards ; starting at they move towards by symmetry. The result depends on the starting point, a simple model of what happens in neural networks, where the starting values of the parameters are chosen at random.

5.3 Stochastic gradient descent

In practice the loss is an average over a very large training set, and computing its exact gradient at every step is expensive. **Stochastic gradient descent (SGD)** instead estimates the gradient from one randomly chosen example or a small random *mini-batch*, a method whose theoretical roots lie in stochastic approximation [13]. Each step is noisier but far cheaper, so many more steps can be taken in the same time. The noise also means the loss does not decrease smoothly. Common refinements add *momentum* (an average of past gradients) or adapt the learning rate for each parameter; they change the update rule but not the underlying idea of following a gradient estimate downhill. [32]

> **Worked example 5.3: a mini-batch gradient estimate.** In the regression problem of Worked example 5.1 at , the full-data gradient is . Suppose a mini-batch contains only the first and third examples. Its residuals are with inputs , and the batch gradient is . The estimate is close to, but not equal to, the true value ; a different batch would give a different estimate, and on average the estimates equal the full gradient.

Table 5.1 compares the three common ways of computing the gradient used in a step.

Table 5.1. Batch, stochastic and mini-batch gradient descent compared.

Variant	Examples per step	Cost per step	Gradient noise	Typical use
Batch	All	High	None	Small datasets
Stochastic	One	Very low	High	Online learning, theory
Mini-batch	(e.g. 32–512)	Moderate	Moderate	Almost all deep learning

5.4 Regularisation and overfitting

A model with many parameters can fit the training data almost perfectly by memorising noise, and then predict poorly on new data. This is **overfitting**; its opposite, a model too simple to capture the pattern, is *underfitting*. A model should be judged on *validation data* that were not used for training. **Regularisation** reduces overfitting by discouraging unnecessarily large parameters. In regularisation (weight decay) a penalty is added to the loss:

where controls the strength of the penalty. Other methods include stopping training early and *dropout*, which randomly switches off units during training [33].

Figure 5.2 illustrates the trade-off on synthetic data generated for this paper (noisy samples of a sine curve; the numbers shown are computed from that synthetic sample and are not experimental results on real data). A straight line is too simple and fits the training points poorly; a cubic captures the pattern; a degree-8 polynomial passes almost exactly through the ten training points, yet its error on fresh points from the same curve is far larger.

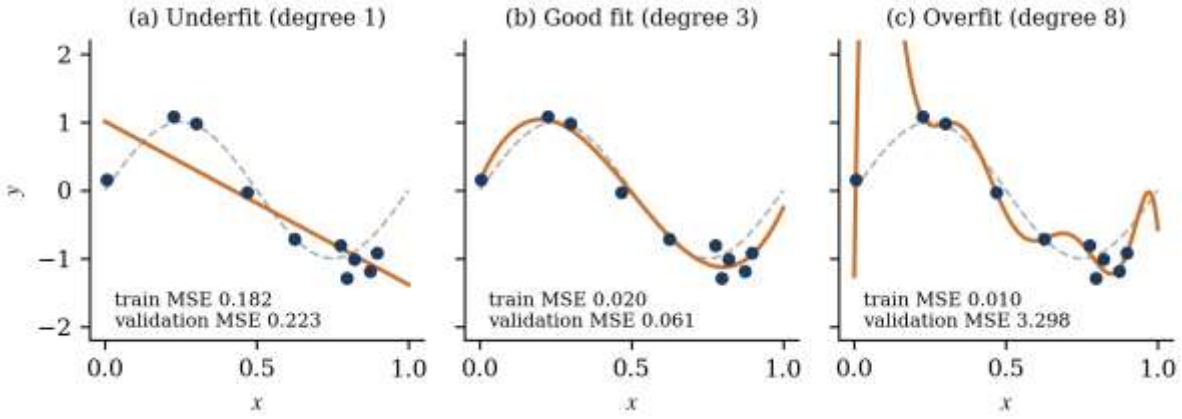


Figure 5.2. Underfitting and overfitting, shown on synthetic data (10 noisy training points from a sine curve; dashed grey line: true curve; orange: fitted polynomial). Training error falls as the polynomial degree rises, but validation error, computed on 40 new points, rises sharply for degree 8. Original illustration generated for this paper. The large degree-8 validation error is dominated by the sharp swings near x = 0, in the gap between the first two training points; six of the ten training points lie at x > 0.6.

Worked example 5.4: the effect of regularisation. Take and , so . Setting the derivative to zero, , gives , so . In general : the regularised minimiser is pulled from towards , and larger pulls it further. The model accepts a slightly worse fit to the training loss (here instead of) in exchange for smaller parameters.

Limitations. The guarantees seen above apply to a simple convex loss with a single minimum. The loss surfaces of deep networks are non-convex, so gradient descent can reach only a point where the gradient is small, not necessarily the best possible solution. Choosing , the batch size and remains largely empirical, and a low training loss alone never proves that a model will generalise.

6. Probability and Statistics in AI

6.1 Random variables and probability distributions

The **probability** of an event is a number between and , with for the certain event and probabilities of mutually exclusive outcomes adding up. A **random variable** assigns a number to each outcome of a random process. A *discrete* random variable takes separate values and is described by a probability mass function with . A *continuous* one is described by a density whose area between two points is the probability of lying between them. The most important continuous distribution is the **Gaussian (normal) distribution** with mean and standard deviation :

About of its probability lies within one standard deviation of the mean (Figure 6.1), and about within two. Table 6.1 lists the distributions most often met in AI.

Table 6.1. Common probability distributions and where they appear in AI.

Distribution	Variable type	Parameters	Typical AI use
Bernoulli	Binary (0 or 1)	success probability	Yes/no classification
Categorical	One of classes	probabilities	Multi-class output, next token
Uniform	Continuous or discrete	range	Random initialisation, sampling
Gaussian	Continuous	mean , variance	Noise models, weight initialisation

In AI, a classifier outputs a discrete distribution over classes, and the **softmax** function converts a vector of real scores (called *logits*) into one:

Every is positive and the sum to . For logits the exponentials are , and , which sum to , giving probabilities .

> **Worked example 6.1: a Gaussian and the -score.** Suppose exam scores follow , so and . The *-score* counts standard deviations from the mean. A score of has . Since about of values lie within of the mean, about lie outside, and by symmetry about lie above . Standardising features in this way (subtracting the mean, dividing by the standard deviation) is a routine preparation of data for learning algorithms.

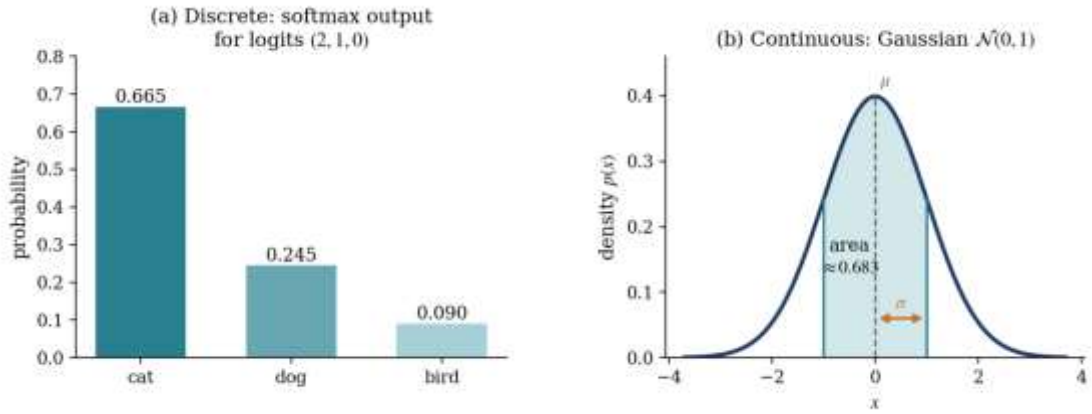


Figure 6.1. (a) A discrete distribution: the softmax output for logits over three classes. (b) A continuous distribution: the standard Gaussian density, with the region within one standard deviation of the mean shaded (area). Original figure.

> **Worked example 6.2: a numerically stable softmax.** For logits , evaluating overflows ordinary floating-point arithmetic. Because softmax is unchanged when the same constant is subtracted from every logit, , subtract to obtain . Then and , so . Every implementation of softmax uses this device.

For two random variables, the *joint* probability describes both together, and the *marginal* probability is obtained by summing over the other variable, . The denominator in Bayes' theorem below is exactly such a marginalisation.

6.2 Conditional probability and Bayes' theorem

The **conditional probability** of given is for . Combining with the same identity for yields **Bayes' theorem**, which reverses the direction of conditioning:

Here is the *prior* probability, the *likelihood*, the *posterior*, and means "not ". Classification is naturally conditional: a model estimates , the probability of label given input .

Two events are *independent* if , that is, if knowing does not change the probability of . For two fair coin flips, . The assumption of independence between features, used by naive Bayes classifiers, which extend Worked example 6.3 from one word to several words, is rarely exactly true, but it makes the calculation of tractable [2].

> **Worked example 6.3: a spam filter.** Suppose of emails are spam, so . The word "prize" appears in of spam emails and in of other emails: and . The overall chance of seeing the word is . By Bayes' theorem, . Observing the word raises the probability of spam from to .

A related idea is **maximum likelihood estimation**: choose the parameter value under which the observed data are most probable. If a coin is tossed times and shows heads, the likelihood of a head-probability is ; setting the derivative of its logarithm, , to zero gives . Minimising the cross-entropy loss of Section 8 is a maximum-likelihood problem of exactly this kind [3].

6.3 Expectation, variance and covariance

The **expectation** (mean) of a discrete random variable is , the probability-weighted average. The **variance** measures spread, and its square root is the standard deviation. For two variables, the **covariance** is positive when they tend to rise together and negative when one rises as the other falls. From observations, the sample covariance is , where and are the sample means.

> **Worked example 6.4: moments.** For a fair die, and , so . For the paired data and : and ; the deviations are and , whose products sum to . Hence the sample covariance is , with and . The correlation coefficient is , a strong positive relationship.

From samples to estimates. In practice a distribution is not known but estimated from data. The sample mean estimates , and the sample variance estimates ; the divisor rather than removes a small bias that arises because is itself computed from the same data. The larger the sample, the closer these estimates are to the true values, a statement known as the law of large numbers.

Evaluating a classifier. A classifier's predictions on a test set are summarised in a *confusion matrix*. With true positives, false positives, false negatives and true negatives, the *accuracy* is and the other standard summaries are

where is precision, is recall, and the score is their harmonic mean.

> **Worked example 6.5: classifier metrics.** Suppose a spam filter is tested on emails with , , and . Then accuracy , precision , recall and . Although accuracy looks high, one third of the spam is missed; this is why a single number such as accuracy can mislead when classes are unequal.

6.4 Probability in classification and language models

A classifier turns scores into the distribution through softmax (Eq. (6.2)) and predicts the most probable class. A **language model** assigns probabilities to sequences of tokens. By the chain rule of probability, the probability of a sequence is built one token at a time,

so the model's task is to output a distribution over the vocabulary for the next token, given the previous ones. As an illustration with invented probabilities, if , and , then . Such products become extremely small for long texts, which is why implementations add logarithms instead of multiplying probabilities. Large Transformer models trained in this way, such as the GPT family, are generative in exactly this sense [14], [15]. Text is produced by *sampling* from, or choosing the most probable token of, each such distribution (Section 9).

6.5 Uncertainty in AI predictions

A probability output is only useful if it is trustworthy. A model is **well calibrated** if, among all predictions made with confidence, about are correct; studies of modern neural networks have found that their confidence is often higher than their accuracy [16]. Practical ways of estimating predictive uncertainty include averaging the predictions of several independently trained networks (ensembles) [17] and keeping dropout active at prediction time as an approximation to Bayesian inference [18]. *Limitations:* a probability expresses the model's own estimate under its assumptions and training data; it can be badly wrong for inputs unlike anything seen in training.

7. Eigenvalues, Eigenvectors and Dimensionality Reduction

7.1 The eigenvalue equation

Most vectors change direction when multiplied by a matrix. Certain special vectors do not: they are only stretched or shrunk. For a square matrix , a non-zero vector is an **eigenvector** with **eigenvalue** if

Rewriting as , where is the identity matrix, shows that a non-zero solution exists only when , the *characteristic equation*. Geometrically, an eigenvector marks a direction that the transformation leaves unchanged, and the eigenvalue is the stretch factor along it: stretches, shrinks, and a negative also reverses the direction.

Worked example 7.1: eigenvalues of a matrix. Let . The characteristic equation is , that is , so and . For , gives , so ; check: . For , , so ; check: . The two eigenvectors are perpendicular. This is not an accident: the eigenvectors of a *symmetric* matrix () with different eigenvalues are always perpendicular.

Figure 7.1 shows the geometry of Worked example 7.1: the unit circle is mapped to an ellipse whose axes lie along the eigenvectors, stretched by the eigenvalues and . This picture is the key to PCA: the long axis of the ellipse is the direction of greatest variation.

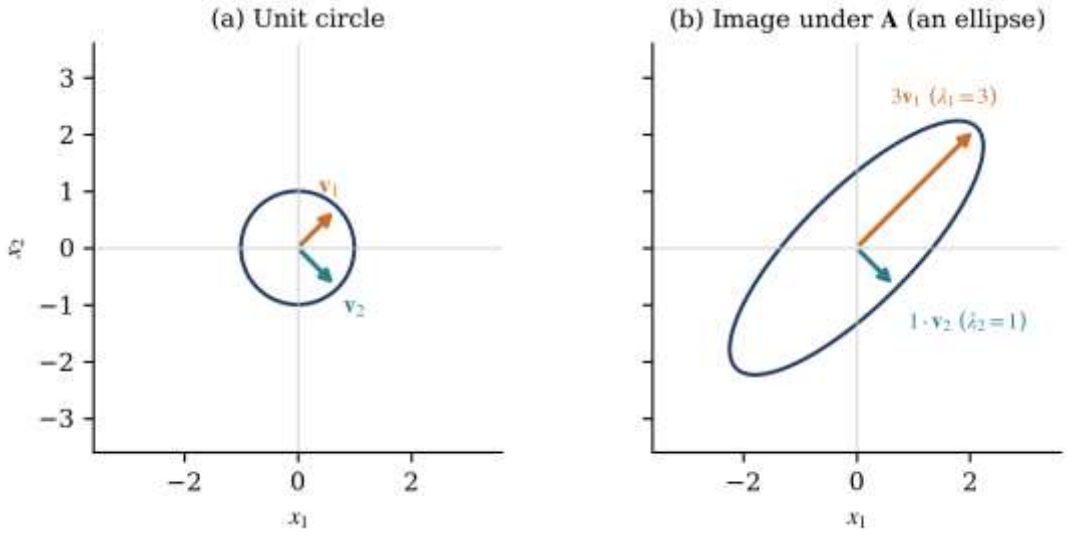


Figure 7.1. Geometry of the eigenvectors of . (a) Unit circle with the unit eigenvectors and . (b) Its image under : an ellipse whose axes are stretched by the eigenvalues, along and along . Original figure.

A symmetric matrix can be written in terms of its eigenvectors and eigenvalues. If the columns of are the unit eigenvectors and is the diagonal matrix of eigenvalues, then

For Worked example 7.1, and , and multiplying out gives , confirming the decomposition. Eigenvalues also describe repeated application: , so . A quantity multiplied by a factor at every step grows rapidly, and by a factor shrinks; this is the same mechanism as the exploding and vanishing gradients mentioned in Section 4.4, and explains why the eigenvalues of weight matrices matter in the analysis of deep and recurrent networks. [34]

7.2 Covariance matrices

For data with features, the **covariance matrix** collects the variances of the features on its diagonal and the covariances between pairs of features off the diagonal (Section 6.3). If holds examples whose feature means have been subtracted (the data are *centred*), then

The matrix is symmetric, so its eigenvalues are real and its eigenvectors can be chosen perpendicular, and the trace of (the sum of its diagonal entries) equals the total variance of the data.

7.3 Principal component analysis

Principal component analysis (PCA) finds new perpendicular axes, the *principal components*, ordered by how much of the data's variance they capture [19]. The key fact is that these axes are the eigenvectors of , and the variance along each axis equals the corresponding eigenvalue. The procedure is: (1) centre the data; (2) compute ; (3) find its eigenvalues and unit eigenvectors ; (4) keep the first eigenvectors; (5) project each centred example onto them, . The **proportion of variance explained** by the first components is

Worked example 7.2: PCA on five points. The data, the first five points of the example in [28], are , , , and (,). *Centre:* the means are and , so the centred points are , , , and . *Covariance* (divisor): , and , so . *Eigenvalues:* the characteristic equation is (trace , determinant), giving and . *Eigenvectors:* for , gives , so after normalising and . *Variance explained:* , so the first component retains about of the variance.

Figure 7.2 illustrates the result, and Table 7.1 lists the projections. Keeping only the first component replaces each two-number example by one number, with a total squared reconstruction error of , which equals , the variance discarded.

Worked example 7.3: reconstruction from one component. Keeping only PC1, each point is rebuilt as . For point 1, , so , compared with the original . For point 2, gives against the original . The squared errors over all

five points are , , , and , which sum to , in agreement with . Storing one score per point instead of two numbers halves the storage while keeping of the variance.

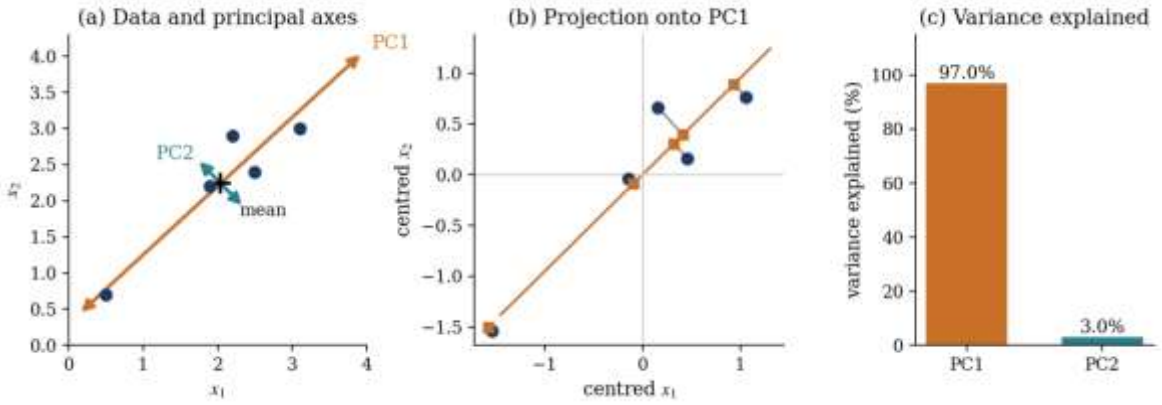


Figure 7.2. PCA of the five points in Worked example 7.2. (a) The data, their mean and the two principal axes (arrow length proportional to). (b) Projection of the centred points onto PC1; dashed segments show what is discarded. (c) Proportion of variance explained. Original figure.

In practice, the number of components is selected using the cumulative proportion of variance explained. Table 7.2 uses five illustrative eigenvalues (5, 3, 1, 0.7 and 0.3), whose total is 10. The first two components explain 80% of the variance, and four components are required to exceed a 95% retention threshold.

Table 7.2. Cumulative variance explained for five illustrative eigenvalues (5, 3, 1, 0.7 and 0.3; total = 10).

Components retained	1	2	3	4	5
Cumulative variance explained	50%	80%	90%	97%	100%

Worked example 7.4: why features should be standardised. Suppose a dataset has two features: height in centimetres, with variance , and weight in kilograms, with variance , and they are uncorrelated. The covariance matrix is , and PCA would report the height direction as explaining of the variance, only because of the units. If height is expressed in metres its variance becomes , and the weight direction would now dominate with . Dividing each feature by its standard deviation (Section 6.1) gives both unit variance and removes this artefact.

7.4 Applications, and limitations

PCA is used to reduce the number of features before training a model, to compress images and embeddings, to remove noise (small-variance directions are often noise), and to visualise high-dimensional data in two or three dimensions. In practice the number of components is chosen so that reaches a target such as . PCA is closely related to the singular-value decomposition, a more general factorisation of matrices that is mentioned here only as an *extension* and not developed.

Limitations. PCA captures only *linear* structure and ranks directions by variance, which is not always the same as usefulness for a prediction task. Features measured on very different scales should be standardised first, because otherwise the feature with the largest numbers dominates the variance.

8. Information Theory in AI

8.1 Information and entropy

Information theory, founded by Shannon [20], quantifies uncertainty. An outcome of probability carries *surprise* : rare events are more surprising. The **entropy** of a discrete random variable with probabilities is the average surprise,

With base-2 logarithms entropy is measured in bits; with natural logarithms, in nats. Entropy is largest when all outcomes are equally likely and zero when one outcome is certain [21]. For equally likely classes , which is about bits for : identifying one of equally likely classes needs about ten yes/no questions.

Worked example 8.1: entropy. A fair coin has bit. A heavily biased coin with has bits, because its outcome is almost predictable. Similarly, a classifier output has bits, whereas has bits, close to the maximum .

8.2 Cross-entropy and classification loss

Cross-entropy measures the average surprise when outcomes follow a true distribution but are scored using a model distribution :

It splits as , where the Kullback–Leibler divergence [22] is zero only when . In classification the true distribution is *one-hot*: for the correct class and elsewhere, so and the loss reduces to

the negative log of the probability the model assigned to the correct class. Minimising this loss is equivalent to maximising the likelihood of the training labels, and it is paired with the softmax of Eq. (6.2). The same quantity trains language models: the loss for a text is the average of over its tokens, applying Eq. (6.5) [14].

> **Worked example 8.2: cross-entropy loss.** The correct class is . Table 8.1 shows the loss (natural logarithm) for three predicted probabilities of that class.

Table 8.1. Cross-entropy loss for different probabilities assigned to the correct class.

Predicted	Loss (nats)	Loss (bits)	Interpretation
			Fairly confident and correct
			Weak belief in the right class
			Confidently wrong: large penalty

Eq. (8.2) also shows what is being measured when the true distribution is not one-hot.

> **Worked example 8.3: cross-entropy and KL divergence.** Let the true distribution be and the model's be . Entropy: bits. Cross-entropy: bits. The difference, bits, is the extra cost of using in place of ; direct calculation of gives the same value.

A closely related quantity for language models is **perplexity**, the exponential of the average per-token cross-entropy loss, ; it can be read as the effective number of equally likely choices the model faces at each step. If a model assigns probabilities to the four tokens of a sentence, the average loss is bits, and the perplexity is .

The loss grows without bound as , so the model is penalised heavily for confident mistakes, and its gradient with respect to the logits has the simple form (derived in Appendix B).

Application and limitations. Entropy also measures uncertainty in a model's predictions; a recent method estimates the entropy of the *meaning* of several generated answers to flag likely errors in language models [23]. Cross-entropy rewards correct probabilities but, on its own, does not guarantee calibrated confidence (Section 6.5), and entropy values depend on the chosen logarithm base.

9. Mathematics Behind Modern AI

This section shows how the preceding tools combine in three widely used mechanisms: vector retrieval, retrieval-augmented generation and Transformer attention. No new mathematics is introduced; each step reuses an equation from an earlier section.

9.1 Embeddings, vector databases and nearest-neighbour retrieval

A **vector store**, the core of a vector database, holds the embedding of each of items (for example, text chunks), where is an embedding model with parameters . Stacking the vectors gives a matrix . For a query embedding , one matrix–vector product yields all dot products at once; if the rows are normalised these are cosine similarities (Eq. (2.4)), and the items with the highest scores are returned (*top- retrieval*). Exact search costs about operations per query, so large systems use *approximate* nearest-neighbour methods, such as compressed vector codes [8] and graph-based indexes [9], implemented in libraries such as FAISS [10]. These trade a small loss of accuracy for a large gain in speed. Four terms are easy to confuse: an embedding model turns each item into a vector; a vector index is the data structure that makes search fast, such as the graph-based [9] or compressed-code [8] indexes above; a library such as FAISS [10] implements such indexes; and a vector database combines an index with storage, metadata filtering and updates. The matrix E above is the simplest case, exact search with no index. Dense retrieval with learned encoders for questions and passages was shown to be effective for open-domain question answering [24].

> **Worked example 9.1: cost of exact search.** For stored vectors of dimension , one exact query needs about multiplications and additions. This is feasible on modern hardware for one query, but not for thousands of

queries per second, which is the motivation for approximate indexes. Approximate indexes return the true nearest neighbours only with some recall below 100%, so they trade retrieval accuracy for speed.

9.2 Mathematical workflow of retrieval-augmented generation

Retrieval-augmented generation (RAG) combines such a retriever with a generative language model so that answers can use external, updatable text [25], [26]. In its commonly used form, a RAG system retrieves the top-k chunks, places them in the prompt, and then generates the answer. The workflow of Figure 9.1 is this retrieve-then-generate composition

where is the user's query, the retrieved chunks, the generated answer and the language model's conditional distribution. Retrieval is a similarity computation, and generation is probability, token by token: .

The original formulation in [25] differs: it marginalises over retrieved documents z, roughly $P(y \mid q) \approx \Sigma_z \; p_\eta(z \mid q) \cdot p_\theta(y \mid q, z)$, with a trained retriever p_η. In words: each retrieved document contributes to the answer in proportion to how relevant the retriever judges it.

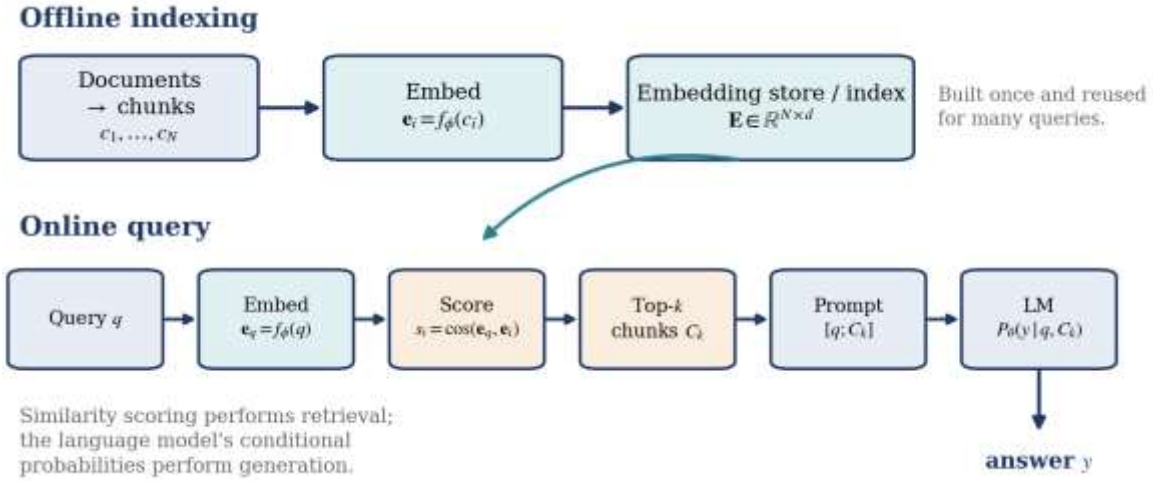


Figure 9.1. Mathematical workflow of RAG. Offline, text chunks are embedded and stored in an embedding store or index (in the simplest case, the rows of a matrix). Online, the query is embedded, scored against every row by cosine similarity, the top- chunks are combined with the query in a prompt, and the language model generates the answer from its conditional distribution. Original diagram illustrating the retrieve-then-generate pipeline; the original formulation in [25] differs: it marginalises over retrieved documents, roughly P(y | q) ≈ Σz pη(z | q) pθ(y | q, z), with a trained retriever.

Worked example 9.2: top-2 retrieval. Let and three chunk embeddings , , . With : ; ; . The top-2 chunks are , which are placed in the prompt together with . *Limitation:* retrieval finds chunks that are *close* to the query in the embedding space, not necessarily chunks that contain the correct answer, so retrieval errors pass directly into the generated text.

9.3 Neural-network computation, queries, keys and values

A deep network is a composition of the layers of Eq. (3.2), and its parameters are learned by backpropagation and gradient descent (Section 4.4, Section 5.2). The Transformer architecture [27] adds one specific operation, **self-attention**, which lets each token gather information from the other tokens. Let hold token vectors as rows. Three learned weight matrices produce *queries*, *keys* and *values*:

Informally, a query asks "what am I looking for?", a key says "what do I contain?", and a value is the information that is passed on. Scaled dot-product attention is

where is the key dimension. The entry of is the dot product of query with key (Eq. (2.3)) and is called an attention score; softmax (Eq. (6.2)) is applied to each row of the scaled scores to give the attention weights, so row of the weights is a probability distribution over the input positions (not over the vocabulary); and multiplying by forms, for each token, a probability-weighted average of the value vectors, which is that token's output vector. The division by keeps the scores at a stable size: if query and key components are independent with mean and variance , the dot product has variance (Appendix B), and unscaled large scores would make the softmax almost one-hot with very small gradients.

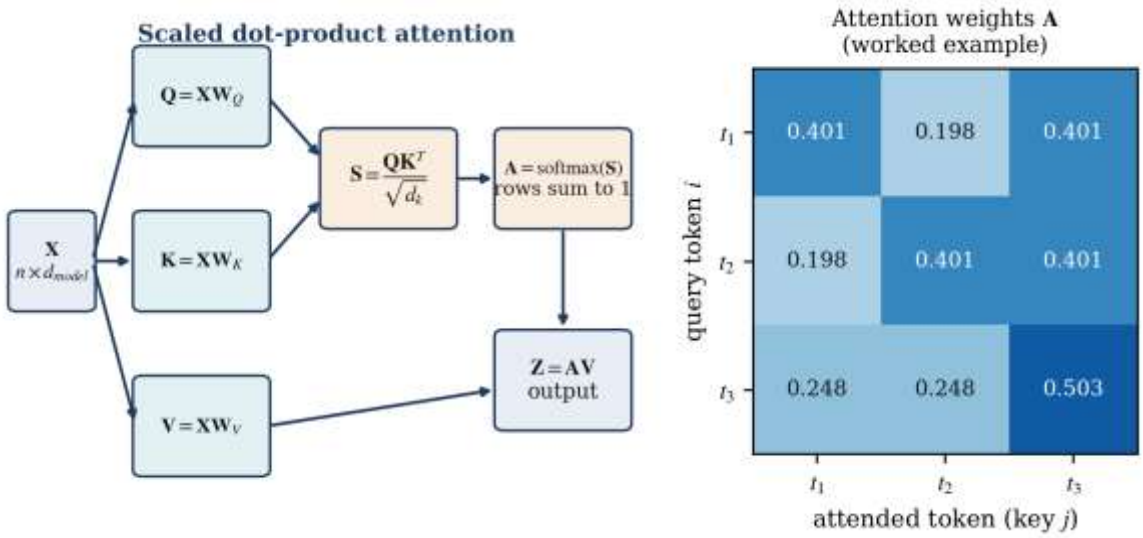


Figure 9.2. Self-attention. Left: from token matrix to queries, keys and values, scaled scores, row-wise softmax and the output (shapes follow [27]). Right: the attention weights computed in Worked example 9.3. Original figure.

In practice the Transformer runs several attention computations in parallel, called **heads**, each with its own matrices ; their outputs are joined side by side and mixed by a further weight matrix . In the base model of the original paper, , there are heads and [27]. Each of the three projection matrices of one head then has entries, the heads together have , and adds , for attention parameters in one layer (biases ignored).

A complete Transformer layer combines attention with a position-wise feed-forward network, which is two linear maps with a nonlinearity between them as in Eq. (3.2), and wraps each part in a residual connection (adding the input to the output) and layer normalisation. Self-attention without positional information is permutation-equivariant: reordering the input tokens reorders the output vectors in the same way, so word order is not otherwise represented. Position information is therefore added to the token vectors before the first layer [27].

> **Worked example 9.3: attention for three tokens.** Take and, to keep the arithmetic visible, let and . Then , and dividing by gives first-row scores . The exponentials are with sum , so the first attention row is . The output for token 1 is . Likewise, row 2 of the weights is , giving output ; and row 3 is , giving . Each weight row sums to (Figure 9.2).

Table 9.1 summarises where each mathematical tool of the earlier sections appears in these mechanisms.

Table 9.1. Roles of the mathematical foundations in retrieval, RAG and attention.

Mathematical tool	Section	Role in retrieval, RAG and Transformers
Vectors, cosine similarity	Section 2	Embeddings and similarity scores
Matrices	Section 3	Embedding tables, , layers
Chain rule, gradients	Section 4	Training of every parameter
Gradient descent	Section 5	Parameter updates during training
Softmax, conditional probability	Section 6	Attention weights, next-token distribution
Cross-entropy	Section 8	Training loss of language models

9.4 Probability-based language generation

A language model outputs logits over its vocabulary, and the next-token distribution is a softmax. Dividing the logits by a **temperature** before the softmax, , controls randomness. Greedy decoding always picks the largest ; sampling draws a token at random according to . Repeating this step token by token realises Eq. (6.5).

> **Worked example 9.4: temperature.** For logits : at the probabilities are (Section 6.1). At the scaled logits are , with exponentials , and (sum), so : the distribution sharpens. At the scaled logits are , giving : it flattens.

Limitations. A fluent output is a high-probability sequence under the model, which is not the same as a verified fact; scale and training data determine the quality of the learned distribution [15]. The attention cost grows with ,

because is an matrix. Attention weights show how strongly each value contributes to one layer's output; on their own they do not give a causal explanation of the model's prediction.

10. Discussion and Limitations

Connections among the disciplines. The sections above are separate only for teaching. A single training step of a neural network uses all of them: linear algebra computes the forward pass (Eq. (3.2)), probability turns scores into a distribution (Eq. (6.2)), information theory scores it against the truth (Eq. (8.3)), calculus supplies the gradient by the chain rule (Eq. (4.3)), and optimisation applies the update (Eq. (5.2)). Eigenvalue methods (PCA) and similarity search operate on the same vectors that these steps produce, and Transformer attention (Eq. (9.3)) is a dot product, a softmax and a matrix product in sequence. Understanding any one idea therefore deepens the understanding of the others.

Limitations of the simplifications. The worked examples use two or three dimensions, a single neuron or a single parameter so that every number can be checked by hand. Real systems differ in important ways. Their loss surfaces are non-convex, so the convergence seen in Worked example 5.2 is not guaranteed. Their data are high-dimensional, where intuitions from the plane, such as "close" and "far", become unreliable. Their numbers are stored with finite precision. Probabilities produced by models are estimates, not guarantees (Section 6.5). PCA is linear, and cosine similarity measures direction only. Several topics that matter in current research were omitted on purpose because they need university-level mathematics: singular-value decomposition theory, formal convergence analysis, graph neural networks, reinforcement learning, and the mathematics of generative image models.

Scope of the review. This is a narrative review: it selects and explains foundational material and does not claim exhaustive coverage or a systematic search of the literature. It reports no experiments; all numerical examples are illustrative calculations, not empirical results. Statements about what vectors or probabilities "represent" are interpretations of established mathematics, not additional mathematical facts, and statements about recent systems rely on the cited sources and may change as the field develops.

Critical interpretation and practical limits. The mathematical links presented here are useful abstractions, but they should not be read as complete descriptions of deployed systems. An embedding is learned from a training objective and corpus; it is not a literal coordinate map of meaning. Nearest-neighbour quality depends on the representation, distance function, index and evaluation data, and distance concentration can weaken intuitive geometric interpretations in some high-dimensional settings [31]. PCA preserves directions of high variance, not necessarily features that are most useful for a downstream prediction. Gradient-based optimisation is sensitive to architecture, initialisation, data, learning-rate schedules and numerical precision; adaptive methods such as Adam alter the update dynamics rather than guaranteeing a global optimum [32]. Likewise, dropout and other regularisers can reduce overfitting but do not remove the need for representative validation data [33]. Finally, attention weights describe a computational weighting within a layer and should not automatically be interpreted as a complete explanation of a model's reasoning. These qualifications delimit what the simplified examples establish and what would require empirical testing.

11. Conclusion

Main insights. This review showed that the working of modern AI can be followed with school-level mathematics. Data become vectors and matrices (Section 2, Section 3). Calculus converts the question "how wrong is the model, and which way should it change?" into a gradient, and the chain rule makes that gradient computable layer by layer (Section 4). Gradient descent repeats small steps along the negative gradient, and its behaviour depends strongly on the learning rate (Section 5). Probability turns scores into predictions and measures uncertainty (Section 6), eigenvectors of the covariance matrix give the principal directions of data (Section 7), and cross-entropy gives the standard loss for classification and language models (Section 8). Embeddings, retrieval-augmented generation and Transformer attention are then compositions of these same tools (Section 9).

Why mathematical understanding matters. A reader who knows these ideas can interpret what a model's outputs mean, anticipate why training may fail (for example, with a learning rate that is too large), judge the limits of similarity scores and probabilities, and read technical papers without treating the equations as a barrier.

Mathematical understanding also supports responsible use: a probability is an estimate, a similarity score is not proof of relevance, and a fluent answer is not a verified one.

Opportunities for further study. Natural next steps are: singular-value decomposition as a generalisation of PCA; convergence theory and adaptive optimisers; Bayesian inference and maximum-likelihood estimation in depth; the mathematics of convolutional and recurrent networks; and graph-based, reinforcement-learning and generative models. Practising each worked example with different numbers, and then implementing it in a short program, is the most reliable way to consolidate the material.

Appendices

Appendix A. Essential notation and formula sheet

Table A.1. Notation used throughout the paper, with the equation or section where each item is introduced.

Symbol	Meaning	Where introduced
	Vectors (lowercase bold); is the -th component	Section 2.1
	Space of real vectors with components	Section 2.1
	Matrices (uppercase bold); is the transpose	Section 3.1
	Euclidean norm (Eq. (2.2))	Section 2.2
	Dot product (Eq. (2.3))	Section 2.3
	Cosine similarity (Eq. (2.4))	Section 2.3
	Determinant: signed area (volume) scale factor	Section 3.3
	Neural-network layer (Eq. (3.2))	Section 3.4
	Model parameters	Section 4.1
	Loss function	Section 5.1
	Derivative, partial derivative	Section 4.2
	Gradient vector (Eq. (4.2))	Section 4.3
	Learning rate (Eq. (5.2))	Section 5.2
	Regularisation strength (Section 5.4); also an eigenvalue (Section 7.1)	Section 5.4 (regularisation); Section 7.1 (eigenvalue)
	Probability, conditional probability	Section 6.2
	Expectation, variance	Section 6.3
	Covariance	Section 6.3
	Gaussian distribution (Eq. (6.1))	Section 6.1
	Softmax (Eq. (6.2))	Section 6.1
	Eigenvalue equation (Eq. (7.1))	Section 7.1
	Covariance matrix (Eq. (7.3))	Section 7.2
	Entropy, cross-entropy (Eq. (8.1), Eq. (8.2))	Section 8.1
	Kullback–Leibler divergence	Section 8.2
	Embedding vector; matrix of embeddings	Section 9.1
	Query, key, value matrices (Eq. (9.2))	Section 9.3

Appendix B. Selected supplementary derivations

B.1 Gradient of softmax cross-entropy. Let and with . Differentiating the softmax gives , where if and otherwise. Then

so the gradient with respect to each logit is simply the predicted probability minus the target. For Worked example 8.2 with the gradient at the correct logit is .

B.2 Variance of a scaled dot product. Let be independent of each other, each with independent components of mean and variance . Each product satisfies and . Variances of independent terms add, so

which is the reason for the factor in Eq. (9.3).

B.3 Distance and cosine for unit vectors. Expanding the squared norm, . If this equals , so ranking by smallest distance and by largest cosine give the same order, as used in Section 2.3.

References

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.

[2] C. M. Bishop, *Pattern Recognition and Machine Learning*. New York, NY, USA: Springer, 2006.

[3] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. Cambridge, MA, USA: MIT Press, 2012.

[4] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *Proc. International Conference on Learning Representations (ICLR)*, 2013.

[5] J. Pennington, R. Socher, and C. D. Manning, "GloVe: Global vectors for word representation," in *Proc. 2014 Conf. Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1532–1543, 2014.

[6] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. 2019 Conf. North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 4171–4186, 2019, doi: 10.18653/v1/N19-1423.

[7] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence embeddings using Siamese BERT-networks," in *Proc. 2019 Conf. Empirical Methods in Natural Language Processing and 9th Int. Joint Conf. Natural Language Processing*, pp. 3982–3992, 2019, doi: 10.18653/v1/D19-1410.

[8] H. Jégou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 1, pp. 117–128, 2011, doi: 10.1109/TPAMI.2010.57.

[9] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 4, pp. 824–836, 2020, doi: 10.1109/TPAMI.2018.2889473.

[10] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with GPUs," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2021, doi: 10.1109/TBDATA.2019.2921572.

[11] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998, doi: 10.1109/5.726791.

[12] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 1986, doi: 10.1038/323533a0.

[13] H. Robbins and S. Monro, "A stochastic approximation method," *The Annals of Mathematical Statistics*, vol. 22, no. 3, pp. 400–407, 1951, doi: 10.1214/aoms/1177729586.

[14] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," OpenAI, San Francisco, CA, USA, Tech. Rep., 2019.

[15] T. B. Brown *et al.*, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, vol. 33, 2020.

[16] C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger, "On calibration of modern neural networks," in *Proc. 34th International Conference on Machine Learning (ICML)*, pp. 1321–1330, 2017.

[17] B. Lakshminarayanan, A. Pritzel, and C. Blundell, "Simple and scalable predictive uncertainty estimation using deep ensembles," in *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[18] Y. Gal and Z. Ghahramani, "Dropout as a Bayesian approximation: Representing model uncertainty in deep learning," in *Proc. 33rd International Conference on Machine Learning (ICML)*, pp. 1050–1059, 2016.

[19] I. T. Jolliffe and J. Cadima, "Principal component analysis: A review and recent developments," *Philosophical Transactions of the Royal Society A*, vol. 374, no. 2065, Art. no. 20150202, 2016, doi: 10.1098/rsta.2015.0202.

[20] C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.

[21] T. M. Cover and J. A. Thomas, *Elements of Information Theory*, 2nd ed. Hoboken, NJ, USA: Wiley, 2006.

[22] S. Kullback and R. A. Leibler, "On information and sufficiency," *The Annals of Mathematical Statistics*, vol. 22, no. 1, pp. 79–86, 1951.

[23] S. Farquhar, J. Kossen, L. Kuhn, and Y. Gal, "Detecting hallucinations in large language models using semantic entropy," *Nature*, vol. 630, pp. 625–630, 2024.

[24] V. Karpukhin, B. Oguz, S. Min, P. Lewis, L. Wu, S. Edunov, D. Chen, and W.-t. Yih, "Dense passage retrieval for open-domain question answering," in *Proc. 2020 Conf. Empirical Methods in Natural Language Processing (EMNLP)*, pp. 6769–6781, 2020, doi: 10.18653/v1/2020.emnlp-main.550.

[25] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Advances in Neural Information Processing Systems*, vol. 33, 2020.

[26] Y. Gao *et al.*, "Retrieval-augmented generation for large language models: A survey," arXiv:2312.10997, 2023.

[27] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[28] L. I. Smith, "A tutorial on principal components analysis," Dept. of Computer Science, University of Otago, Dunedin, New Zealand, Tech. Rep. OUCS-2002-12, Feb. 2002.

[29] M. P. Deisenroth, A. A. Faisal, and C. S. Ong, Mathematics for Machine Learning. Cambridge: Cambridge University Press, 2020. doi: 10.1017/9781108679930.

[30] G. Strang, Introduction to Linear Algebra, 6th ed. Wellesley, MA, USA: Wellesley-Cambridge Press, 2023.

[31] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, "When is "nearest neighbor" meaningful?," in Database Theory—ICDT'99, Lecture Notes in Computer Science, vol. 1540, pp. 217–235, 1999. doi: 10.1007/3-540-49257-7_15.

[32] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in Proc. International Conference on Learning Representations (ICLR), 2015. arXiv:1412.6980.

[33] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," Journal of Machine Learning Research, vol. 15, pp. 1929–1958, 2014.

[34] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in Proc. 30th International Conference on Machine Learning, Proc. Machine Learning Research, vol. 28, no. 3, pp. 1310–1318, 2013.

[35] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in Proc. 14th International Conference on Artificial Intelligence and Statistics, Proc. Machine Learning Research, vol. 15, pp. 315–323, 2011.

[36] A. Rogers, A. Drozd, and B. Li, "The (too many) problems of analogical reasoning with word vectors," in Proc. 6th Joint Conference on Lexical and Computational Semantics (*SEM 2017), pp. 135–148, 2017. doi: 10.18653/v1/S17-1017.