Generative Models

From First Principles to Frontier Research

Yogesh P

March 2026

Overview

All chapters at a glance

Ch. 0 Introduction

Three ways to think about generating images: the pixel view, the feature view, and the noise view.

Part I: Explicit Likelihood Models

Ch. 1 Autoregressive Models

From FVSBN to Transformers: fixed-length models over images and audio, and variable-length sequence models.

Contents

— Part I: Explicit Likelihood Models — **5**

1 Autoregressive Models **6**

1.1 The Problem: Modeling Joint Distributions . . . 6
1.1.1 The Chain Rule . . . 6
1.2 Structural Assumptions . . . 7
1.2.1 Bayesian Networks . . . 7
1.2.2 The Markov Assumption . . . 7
1.2.3 Why Structure Alone Is Not Enough . . . 8
1.3 Parametric Modeling of Conditional Distributions . . . 8
1.4 Autoregressive Models . . . 9
1.5 Fixed-Length Autoregressive Models . . . 10
1.5.1 FVSBN . . . 11
1.5.2 NADE: Neural Autoregressive Distribution Estimators . . . 13
1.5.3 Autoencoders: A Necessary Digression . . . 16
1.5.4 MADE: Masked Autoencoder for Distribution Estimation . . . 18
1.5.5 The Fixed-Length Ceiling . . . 22
1.5.6 Summary: Fixed-Length Models . . . 22
1.6 Variable-Length Autoregressive Models . . . 22
1.6.1 Words to Vectors . . . 22
1.6.2 RNNs: The Hidden State as Context . . . 31
1.6.3 Attention . . . 39
1.6.4 Transformer . . . 49
1.6.5 Summary: Variable-Length Models . . . 62

2 Large Language Models **64**

2.0.1 Mixture of Experts . . . 64

3 Latent Variable Models **67**

3.1 Representation: Directed Latent Variable Models . . . 68
3.1.1 The Graphical Model . . . 69
3.1.2 Hypothesis Class . . . 69
3.2 Shallow Latent Variable Models . . . 70
3.2.1 Factor Analysis and Probabilistic PCA . . . 70
3.2.2 Gaussian Mixture Models . . . 71
3.2.3 Hidden Markov Models . . . 72
3.2.4 The Capacity Wall . . . 74
3.3 Learning Directed Latent Variable Models . . . 75
3.3.1 Maximum Marginal Likelihood . . . 75
3.3.2 Intractability . . . 75
3.4 Variational Inference and the ELBO . . . 75
3.4.1 Variational Families . . . 76

3.4.2 Information-Theoretic Bridge to the ELBO . . . 76
3.4.3 Derivation of the Evidence Lower Bound . . . 77
3.4.4 Tightness of the Bound . . . 79
3.4.5 Learning via ELBO Maximisation . . . 80
3.4.6 The ELBO as Two KL Divergences . . . 80
3.5 Black-Box Variational Inference . . . 81
3.5.1 The BBVI Algorithm . . . 82
3.5.2 Gradient Estimation . . . 82
3.5.3 REINFORCE Trick proof (Discrete z) . . . 82
3.6 The Reparameterisation Trick . . . 84
3.6.1 Change of Variables . . . 84
3.6.2 Low-Variance Gradient Estimator . . . 84
3.7 Deep Latent Variable Models . . . 85
3.8 Parameterising Distributions with Neural Networks . . . 85
3.8.1 Prior Distribution . . . 86
3.8.2 Decoder: Parameterising $p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z})$ . . . 86
3.8.3 Encoder: Parameterising the Variational Family . . . 86
3.9 Amortised Variational Inference . . . 86
3.9.1 Classical vs. Amortised Inference: The Key Distinction . . . 87
3.9.2 Learning the Inference Mapping . . . 87
3.10 Variational Autoencoders . . . 88
3.10.1 Architecture . . . 88
3.10.2 Training Objective . . . 88
3.10.3 Training via the Reparameterised ELBO . . . 89
3.11 Challenges and Limitations of the VAE . . . 91
3.11.1 Posterior Collapse . . . 91
3.11.2 The Expressiveness of the Variational Family . . . 91
3.11.3 The ELBO is Not the Likelihood . . . 92
3.12 Normalizing Flows . . . 92
3.12.1 The Change-of-Variables Formula . . . 92
3.12.2 Composing Flows . . . 93
3.12.3 Architectural Constraints: Making the Jacobian Cheap . . . 93
3.12.4 Flows Inside the Variational Posterior . . . 94
3.12.5 VAE vs. Normalizing Flows: Two Philosophies . . . 94
3.13 Summary . . . 95
3.14 Problems . . . 96
3.15 Solutions . . . 97

PART I

Explicit Likelihood Models

These models define a tractable probability distribution $p_\theta(x)$ explicitly and train by maximizing the log-likelihood of the data. The key question in each chapter is: how do we make the joint distribution over high-dimensional data both expressive and computationally feasible?

Chapters in this part:

Chapter 1 Autoregressive Models

Chapter 2 Large language Models*(coming soon)*

Chapter 3 Latent Variable Models *(coming soon)*

Chapter 4 something *(coming soon)*

Chapter 1

Autoregressive Models

Fully Visible Models: Fixed-Length and Variable-Length

1.1 The Problem: Modeling Joint Distributions

An image can be viewed as a collection of random variables, one per pixel. More generally, any observation can be represented as a vector.

Generative modeling asks a single question:

> *How do we model the joint distribution $p(x_1, x_2, \ldots, x_n)$?*

If we can model this distribution well, we can:

- sample new data,
- condition on partial observations or prompts,
- infer hidden structure from raw inputs.

For simplicity, We start with binary variables for the first few models. (The same ideas extend to discrete or continuous settings.)

A direct approach is to represent the joint distribution explicitly as a **lookup table**: for every possible configuration of x, we store its probability.

If the variables are binary, there are 2^n possible configurations. Since probabilities must sum to one, the number of independent parameters is

$$2^n - 1.$$

Thus the parameter complexity grows exponentially with n. For high-dimensional data such as images, this representation is computationally impossible.

The central challenge of generative modeling is therefore:

> *How can we represent the joint distribution without exponential cost?*

1.1.1 The Chain Rule

A natural first step is to rewrite the joint distribution using the **probability chain rule**:

$$p(x_1, \ldots, x_n) = \prod_{i=1}^{n} p(x_i \mid x_1, \ldots, x_{i-1}).$$

This decomposition is always exact — no assumptions are introduced.

However, decomposition is not compression.

If each conditional distribution is represented as a lookup table, then for binary variables the total number of parameters becomes

$$1 + 2 + 4 + \cdots + 2^{n-1} = 2^n - 1,$$

which is still exponential in n.

The chain rule changes the form of the problem but not its complexity. To make progress, we must introduce additional structure.

1.2 Structural Assumptions

The exponential cost arises because each variable is conditioned on *all* previous variables. In many real systems, however, only a small subset of those dependencies may actually matter.

This motivates a first strategy:

> *Reduce complexity by restricting which variables each conditional depends on.*

1.2.1 Bayesian Networks

Definition 1.1: Bayesian Network

A **Bayesian network** is a directed graphical model in which the joint distribution factorizes as

$$p(x_1, \ldots, x_n) = \prod_{i=1}^{n} p(x_i \mid \text{Parents}(x_i)).$$

Each variable depends only on a designated set of **parents**.

If each variable has at most k parents and variables are binary, each conditional table has 2^k entries. The total parameter count becomes

$$O(n \cdot 2^k),$$

which is linear in n when k is small.

Structural assumptions reduce complexity by limiting dependencies.

1.2.2 The Markov Assumption

We can push this idea to its extreme by assuming each variable depends only on the immediately preceding one.

Definition 1.2: First-Order Markov Assumption

The **first-order Markov assumption** states

$$x_{i+1} \perp \{x_1, \ldots, x_{i-1}\} \mid x_i.$$

The joint distribution factorizes as

$$p(x_{1:n}) = p(x_1) \prod_{i=1}^{n-1} p(x_{i+1} \mid x_i).$$

Here each variable has only one parent, so only local transition probabilities are required.

The Markov assumption is a modeling choice: we intentionally restrict dependencies to obtain tractability.

Both Bayesian networks and Markov chains follow the same philosophy: reduce complexity by narrowing *what we condition on*.

1.2.3 Why Structure Alone Is Not Enough

Structural assumptions help, but they do not fully solve the problem.

Even when dependencies are sparse, lookup tables scale exponentially with the size of the conditioning context. If the parent set becomes large, tabular representations again become intractable.

This suggests a second, fundamentally different strategy.

1.3 Parametric Modeling of Conditional Distributions

Instead of reducing dependencies, we can change how conditionals are represented.

Lookup tables memorize a probability for every possible configuration. But we can instead *compute* probabilities using a parameterized function.

Definition 1.3: Parametric Conditional Model

A **parametric conditional model** represents conditional probabilities using a parameterized function:

$$p(x_i \mid x_{<i}) = f_{\theta_i}(x_{<i}).$$

The parameters may be independent for each conditional or shared across conditionals.

The key difference is conceptual:

- Lookup tables memorize each context independently.
- Functions share parameters and generalize across contexts.

Complexity is now controlled by the function class rather than by the number of possible

inputs.

This allows us to keep the full chain-rule factorization while avoiding exponential storage.

Generative models become tractable through two complementary design dimensions.

1. Structural assumptions simplify the dependency structure by restricting which variables each conditional distribution depends on:

Approach	**Dependency**	**Parameters**
Raw chain rule	All previous variables	$O(2^n)$
Bayesian network	Small parent set (k parents)	$O(n \cdot 2^k)$
Markov chain	Previous variable only	$O(n)$

2. Parametric modeling simplifies representation by replacing explicit probability tables with parameterized functions:

- Tabular models store probabilities independently for each context.
- Parametric models compute probabilities using shared parameters.

Structural assumptions control *which dependencies are modeled*, while parametric modeling controls *how conditional distributions are represented*.

These two dimensions are complementary rather than mutually exclusive. Modern generative models often combine both: structural assumptions define the factorization of the joint distribution, while parametric function approximation makes high-dimensional conditionals tractable.

1.4 Autoregressive Models

The name **autoregressive** originates from classical time-series analysis and is composed of two parts:

- **Regression** refers to predicting a variable from other variables. In statistics, regression simply means modeling a quantity as a function of observed inputs.
- **Auto** means *self*. In an autoregressive model, the inputs used for prediction come from the same sequence being modeled.

Combining the two ideas, an autoregressive model performs *self-regression*: each variable is predicted from previously observed variables in the same sequence. Conceptually, the model learns how the past explains the future.

Formally, this idea is expressed by modeling each element x_i conditioned on the preceding elements $x_{<i}$.

Definition 1.4: Autoregressive Model

An **autoregressive model** applies the chain rule factorization exactly:

$$p(x_1, \ldots, x_n) = \prod_{i=1}^{n} p(x_i \mid x_1, \ldots, x_{i-1})$$

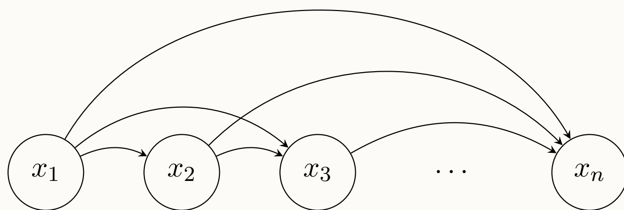

Figure 1.1: Autoregressive model with full past dependencies

but represents each conditional with a **learned function** f_{θ_i} rather than a lookup table. No independence assumptions are introduced — the full dependency structure of the chain rule is preserved.

Sampling proceeds sequentially:

1. Sample $x_1 \sim p(x_1)$
2. Sample $x_2 \sim f_{\theta_2}(x_1)$
3. Sample $x_3 \sim f_{\theta_3}(x_1, x_2)$
4. Continue until x_n

By choosing a specific function class, we implicitly restrict which distributions the model can represent. But this is a far more flexible restriction than hard independence assumptions — and it scales gracefully as the function class becomes more expressive.

Example: Autoregressive model

Consider a sequence of binary variables x_1, x_2, x_3 with learned conditionals:

$$p(x_1 = 1) = 0.6$$
$$p(x_2 = 1 \mid x_1) = f_{\theta_2}(x_1)$$
$$p(x_3 = 1 \mid x_1, x_2) = f_{\theta_3}(x_1, x_2)$$

To sample a sequence:

1. Draw x_1: get $x_1 = 1$ with probability 0.6
2. Feed x_1 into f_{θ_2} to obtain $p(x_2 = 1 \mid x_1)$, then sample x_2
3. Feed (x_1, x_2) into f_{θ_3} to obtain $p(x_3 = 1 \mid x_1, x_2)$, then sample x_3

At no point do we enumerate all $2^3 = 8$ configurations. The function does the work.

1.5 Fixed-Length Autoregressive Models

We said that instead of storing $p(x_i \mid x_1, \ldots, x_{i-1})$ as a lookup table, we could *compute* it using a learned function f_θ. But we never said what f_{θ_i} actually looks like.

This chapter builds up the answer — starting from the simplest possible parameterization

and progressively replacing its weaknesses with better ideas.

Section 1.5 covers fixed-length models. Throughout, we use MNIST as our running example: $28 \times 28 = 784$ binary pixels in raster scan order, $x_1, \ldots, x_{784} \in \{0, 1\}$.

1.5.1 FVSBN

A common and conceptually simple choice is to model each conditional distribution using logistic regression. In this setting, the conditional probability is parameterized as

$$p_\theta(x_i = 1 \mid x_{<i}) = \sigma(W_i x_{<i} + b_i)\,,$$

where $\sigma(\cdot)$ denotes the logistic sigmoid function. so each variable is predicted using a sigmoid applied to a linear function of the previous variables.. The name **Fully Visible Sigmoid Belief Network (FVSBN)** reflects three properties: the model defines probability directly over the observed variables (fully visible), each conditional distribution is parameterized with a sigmoid (logistic regression), and the joint distribution is factorized as a directed probabilistic model — historically called a belief network.

For a conditional depending on m binary variables, a lookup table requires $2^m - 1$ independent parameters. If the conditional is instead represented by a parameterized function, such as a logistic function, the number of parameters grows only linearly with m. For example, a linear logistic parameterization requires $m+1$ parameters. The reduction in complexity comes from restricting the space of possible conditional distributions to those expressible by the chosen function.

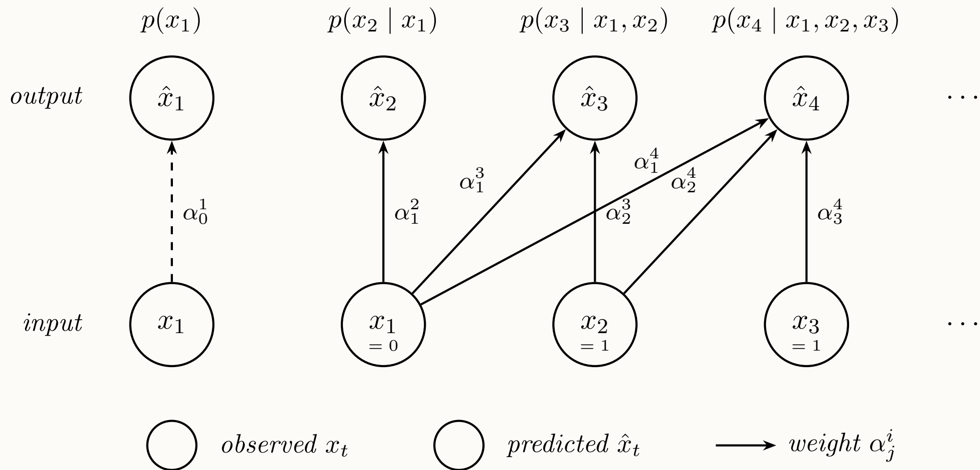

Figure 1.2: FVSBN: logistic regression per position. Each output $\hat{x}_i = \sigma(\alpha_i^0 + \sum_{j<i} \alpha_i^j x_j) = p(x_i{=}1 \mid x_{<i})$.

Definition 1.4: Fully Visible Sigmoid Belief Network (FVSBN)

A **Fully Visible Sigmoid Belief Network** parameterizes each conditional as:

$$\hat{x}_i = p(x_i = 1 \mid x_{<i};\, \alpha^i) = \sigma\left(\alpha_0^i + \sum_{j=1}^{i-1} \alpha_j^i\, x_j\right)$$

where $\sigma(z) = 1/(1 + e^{-z})$ and α^i are learned parameters for position i. The joint:

$$p(x_1, \ldots, x_n) = \prod_{i=1}^{n} p(x_i \mid x_{<i})$$

Parameters: $1 + 2 + \cdots + n = O(n^2/2)$.

Example: Evaluating and Sampling from FVSBN

Define the conditional probabilities using the parameter matrix

$$\hat{x}_i = p(x_i = 1 \mid x_{<i};\, \alpha^i) = \sigma\left(\alpha_0^i + \sum_{j=1}^{i-1} \alpha_j^i\, x_j\right).$$

Collect the parameters into a lower–triangular matrix

$$A = \begin{bmatrix} \alpha_0^1 & 0 & 0 & 0 \\ \alpha_0^2 & \alpha_1^2 & 0 & 0 \\ \alpha_0^3 & \alpha_1^3 & \alpha_2^3 & 0 \\ \alpha_0^4 & \alpha_1^4 & \alpha_2^4 & \alpha_3^4 \end{bmatrix}, \qquad \tilde{x} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}.$$

Then the vector of conditional probabilities can be written compactly as

$$\hat{\mathbf{x}} = \sigma(A\tilde{x}),$$

where the sigmoid function is applied element-wise.
To evaluate $p(X_1 = 0,\, X_2 = 1,\, X_3 = 1,\, X_4 = 0)$:

1. Compute all $\hat{x}_i$ using the matrix form above.
2. Multiply: $(1 - \hat{x}_1) \times \hat{x}_2 \times \hat{x}_3 \times (1 - \hat{x}_4)$

To sample:

1. Sample $x_1 \sim \text{Bernoulli}(\hat{x}_1)$
2. Sample $x_2 \sim \text{Bernoulli}(\hat{x}_2(x_1))$, and so on

The asymmetry of autoregressive models: Autoregressive models separate evaluation and generation in an important way. Given a known datapoint, the likelihood $p(x)$ can often be computed efficiently, since the conditional terms $p(x_i \mid x_{<i})$ are evaluated with inputs that

are already known. In contrast, sampling is inherently sequential: each variable must be generated before later conditionals can be evaluated. This difference between efficient likelihood evaluation and sequential generation is a defining property of autoregressive modeling.

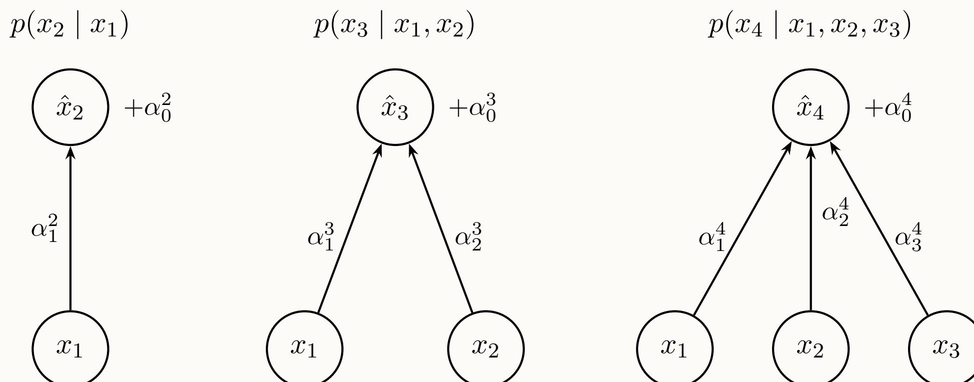

Figure 1.3: Conditionals as logistic regression.

Let the conditional distribution be parameterized by a function

$$f_{\theta_i}(x_{<i}) = \sigma\left(\alpha_0^i + \sum_{j=1}^{i-1} \alpha_j^i \, x_j\right),$$

which is simply a logistic regression model for variable x_i.
Then each conditional can be written compactly as

$$p(x_i = 1 \mid x_{<i}) = f_{\theta_i}(x_{<i}).$$

Using the autoregressive factorization, the joint distribution becomes

$$p(x_1, \dots, x_n) = \prod_{i=1}^{n} p(x_i \mid x_{<i}) = \prod_{i=1}^{n} f_{\theta_i}(x_{<i}).$$

FVSBN is valid but limited, since the function f_θ is a logistic regression and therefore models only linear dependencies. To obtain richer, more expressive conditionals, we next replace f_θ with neural networks.

1.5.2 NADE: Neural Autoregressive Distribution Estimators

NADE strengthens autoregressive modeling by replacing the linear conditional predictor of FVSBN with a one-hidden-layer neural network. For each position i, a hidden representation is computed and then mapped to a conditional probability:

$$h_i = \sigma\big(A^i x_{<i} + c\big), \qquad \hat{x}_i = \sigma\big(\alpha^i h_i + b_i\big).$$

The hidden layer allows the model to capture nonlinear interactions among observed variables, something a purely linear predictor cannot do.

The parameter-scaling problem. Assigning a separate weight matrix $A^i \in \mathbb{R}^{d\times(i-1)}$ to every conditional is costly. Summing over all positions gives

$$\sum_{i=1}^{n} d(i-1) = d\,\frac{n(n-1)}{2} = O(n^2 d)$$

parameters for the input-to-hidden mappings alone. Beyond the raw count, this design is statistically wasteful: the inputs to adjacent conditionals differ by only a single observation, yet each position learns entirely independent features. There is no mechanism for the model to transfer what it learns about earlier variables to later ones.

Weight sharing as the key idea. NADE resolves this by tying the input-to-hidden weights across all positions. A single shared matrix

$$W \in \mathbb{R}^{d\times n}$$

replaces the family $\{A^i\}$, and the i-th conditional uses only the first $i-1$ columns:

$$h_i = \sigma(W_{:,<i}\, x_{<i} + c)\,, \qquad \hat{x}_i = p(x_i = 1 \mid x_{<i}) = \sigma\big(\alpha^i h_i + b_i\big)\,.$$

Because the same column $W_{:,j}$ appears in every conditional that conditions on x_j, the model is forced to learn features that are useful across the entire sequence, not just at a single position. The input-to-hidden parameter count drops to $dn = O(nd)$, and the shared structure enables an efficient left-to-right recurrence: the pre-activation at position $i+1$ is obtained by augmenting the pre-activation at position i with a single rank-one update, rather than recomputing from scratch.

Definition 1.5: Neural Autoregressive Distribution Estimator (NADE)

NADE learns a single shared weight matrix $W \in \mathbb{R}^{d\times n}$ so that every conditional reuses the same input-to-hidden mapping:

$$h_i = \sigma(W_{:,<i}\, x_{<i} + c)\,, \qquad \hat{x}_i = p(x_i = 1 \mid x_{<i}) = \sigma\big(\alpha^i h_i + b_i\big)\,.$$

Columns of W are shared across positions, so hidden pre-activations can be updated incrementally from left to right. Compared with unshared FVSBN-style networks, NADE reduces input-to-hidden parameter count from $O(n^2 d)$ to $O(nd)$ while retaining nonlinear expressiveness.

Extensions. The NADE framework extends naturally beyond binary variables by modifying the output distribution while retaining the same autoregressive hidden representation.

Discrete variables. For non-binary discrete random variables $x_i \in \{1, \dots, K\}$ (e.g., pixel intensities ranging from 0 to 255), the output $\hat{x}_i$ parameterizes a categorical distribution rather than a Bernoulli. The hidden representation is computed as

$$h_i = \sigma(W_{\cdot,<i} x_{<i} + c),$$

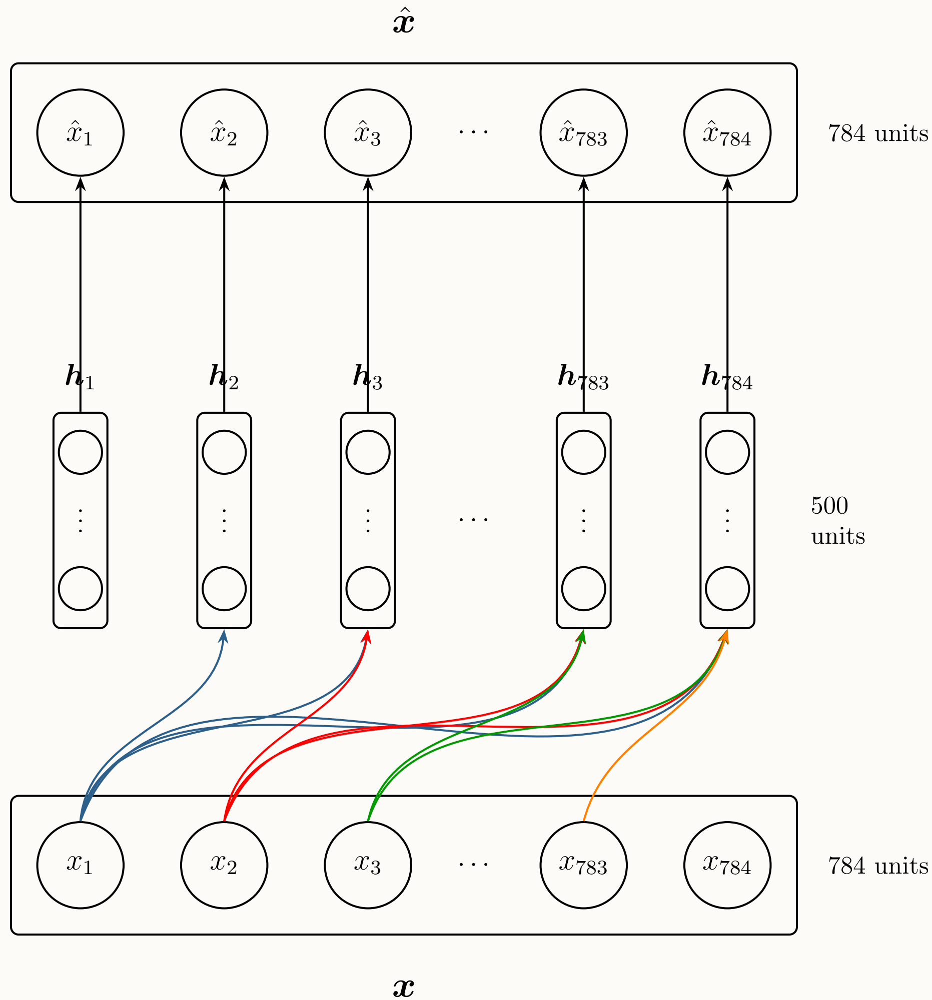

Figure 1.4: Neural Autoregressive Distribution Estimation (NADE). The input $\boldsymbol{x}$ is a 784-dimensional binary vector. Each hidden layer $\boldsymbol{h}_d$ receives only the inputs $x_1, \ldots, x_{d-1}$ (autoregressive masking), and produces the output $\hat{x}_d$ estimating the conditional probability $p(x_d = 1 \mid \boldsymbol{x}_{<d})$.

and the conditional distribution is modeled as

$$p(x_i \mid x_{<i}) = \mathrm{Cat}(p_i^1, \dots, p_i^K),$$

where

$$\hat{x}_i = (p_i^1, \dots, p_i^K) = \mathrm{softmax}(A_i h_i + b_i).$$

The softmax function generalizes the sigmoid by transforming a vector of real-valued scores into a valid probability distribution over K outcomes, ensuring non-negativity and unit normalization.

Continuous variables (RNADE). For real-valued variables $x_i \in \mathbb{R}$ (e.g., speech signals), each conditional can be modeled using a continuous distribution. A common choice is a mixture of Gaussians, giving rise to the Real-valued NADE (RNADE). In this case, the conditional distribution takes the form

$$p(x_i \mid x_{<i}) = \sum_{j=1}^{K} \pi_i^j \, \mathcal{N}\Big(x_i; \mu_i^j, \sigma_i^j\Big)\,,$$

where π_i^j are mixture weights and (μ_i^j, σ_i^j) are the mean and standard deviation of the j-th Gaussian component. These parameters are predicted from the hidden state:

$$h_i = \sigma(W_{\cdot,<i} x_{<i} + c), \qquad \hat{x}_i = (\pi_i^1, \dots, \pi_i^K, \mu_i^1, \dots, \mu_i^K, \sigma_i^1, \dots, \sigma_i^K) = f(h_i).$$

In practice, the standard deviations are typically parameterized using an exponential transform to ensure $\sigma_i^j > 0$. This extension allows NADE to model complex continuous densities while preserving the autoregressive factorization and efficient shared-parameter structure.

Remark: Hidden Units as Deterministic Features Although h_i is commonly called a *hidden layer*, it is more precise to regard it as a **deterministic feature vector** derived from the observations. Given $x_{<i}$, the value of h_i is fixed:

$$h_i = \sigma(W_{:,<i}\, x_{<i} + c)\,.$$

NADE therefore contains no latent random variables. The h_i serve purely as nonlinear summaries of the observed context, used to parameterize each conditional distribution.

1.5.3 Autoencoders: A Necessary Digression

Before MADE, we need to understand autoencoders — and why they cannot be used as generative models without modification.

Definition 1.6: Autoencoder

An **autoencoder** maps input x through an encoder f_{enc} to a representation z, then through a decoder f_{dec} to a reconstruction $\hat{x}$:

$$z = f_{\text{enc}}(x), \qquad \hat{x} = f_{\text{dec}}(z), \qquad \mathcal{L} = \sum_{x \in \mathcal{D}} \sum_i [-x_i \log \hat{x}_i - (1 - x_i) \log(1 - \hat{x}_i)]$$

The bottleneck at z forces the network to learn a compressed representation. But a vanilla autoencoder is **not** a generative model: it does not define a distribution over x we can sample from.

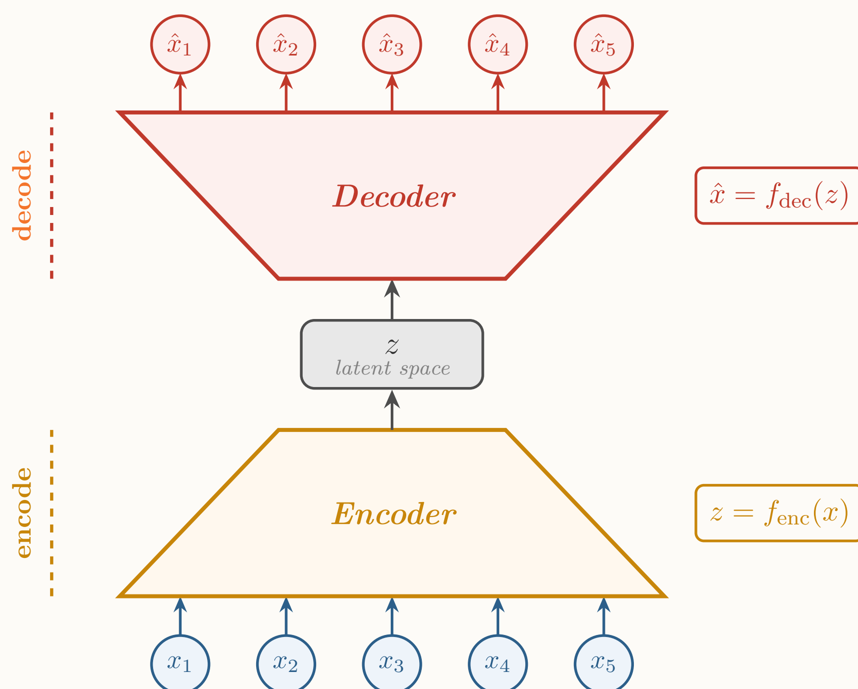

Figure 1.5: Autoencoder

On the surface, FVSBN and NADE already look like autoencoders: they take x as input and produce $\hat{x}$ with the same cross-entropy loss. The problem is that a standard autoencoder lets output $\hat{x}_i$ see input x_i directly — it can simply copy it, achieving zero reconstruction error while learning nothing useful.

The cheating problem: Output $\hat{x}_i$ must depend *only* on $x_1, \ldots, x_{i-1}$ to be useful as a generative model. At generation time we do not have x_i to copy — that is what we are trying to produce. A standard autoencoder does not enforce this constraint.

1.5.4 MADE: Masked Autoencoder for Distribution Estimation

An ordinary feed-forward autoencoder can be transformed into an autoregressive density estimator by masking its weight matrices. The goal is to preserve the autoregressive factorization

$$p(x) = \prod_{i=1}^{n} p(x_i \mid x_{<i}),$$

while computing all conditionals in a **single forward pass**.

Challenge. A standard autoencoder is fully connected, meaning each output $\hat{x}_i$ depends on all inputs. This violates the autoregressive constraint, which requires $\hat{x}_i$ to depend only on variables preceding x_i in a chosen ordering.

Key idea: masking as a structural constraint. MADE enforces a directed acyclic graph (DAG) inside the autoencoder by introducing binary masks on the weight matrices. These masks remove invalid connections so that each output obeys autoregressive dependencies. Given an ordering (or permutation) π of the variables, the joint distribution factorizes as

$$p(x) = \prod_{i=1}^{n} p(x_{\pi_i} \mid x_{\pi_{<i}}).$$

For example, if $\pi = (x_2, x_3, x_1)$, then

$$p(x_1, x_2, x_3) = p(x_2)\, p(x_3 \mid x_2)\, p(x_1 \mid x_2, x_3).$$

The output unit producing $\hat{x}_2$ is therefore not allowed to depend on any input, while the unit producing $p(x_3 \mid x_2)$ may depend only on x_2, and so forth.

Hidden-unit degrees. Instead of manually designing masks, MADE assigns each hidden unit k an integer degree

$$m(k) \in \{1, \dots, n-1\}.$$

The degree specifies the largest input index that the hidden unit is allowed to depend on. Intuitively, it determines how far into the input ordering the unit can "see".

Mask construction. Masks are constructed using these degrees:

- Input $\to$ hidden: a hidden unit with degree $m(k)$ connects only to inputs with index $j \le m(k)$.
- Hidden $\to$ hidden: connections satisfy

  $$m(\text{prev}) \le m(\text{next}),$$

  ensuring autoregressive consistency across layers.
- Hidden $\to$ output: output $\hat{x}_i$ receives connections only from hidden units with degree

  $$m < i.$$

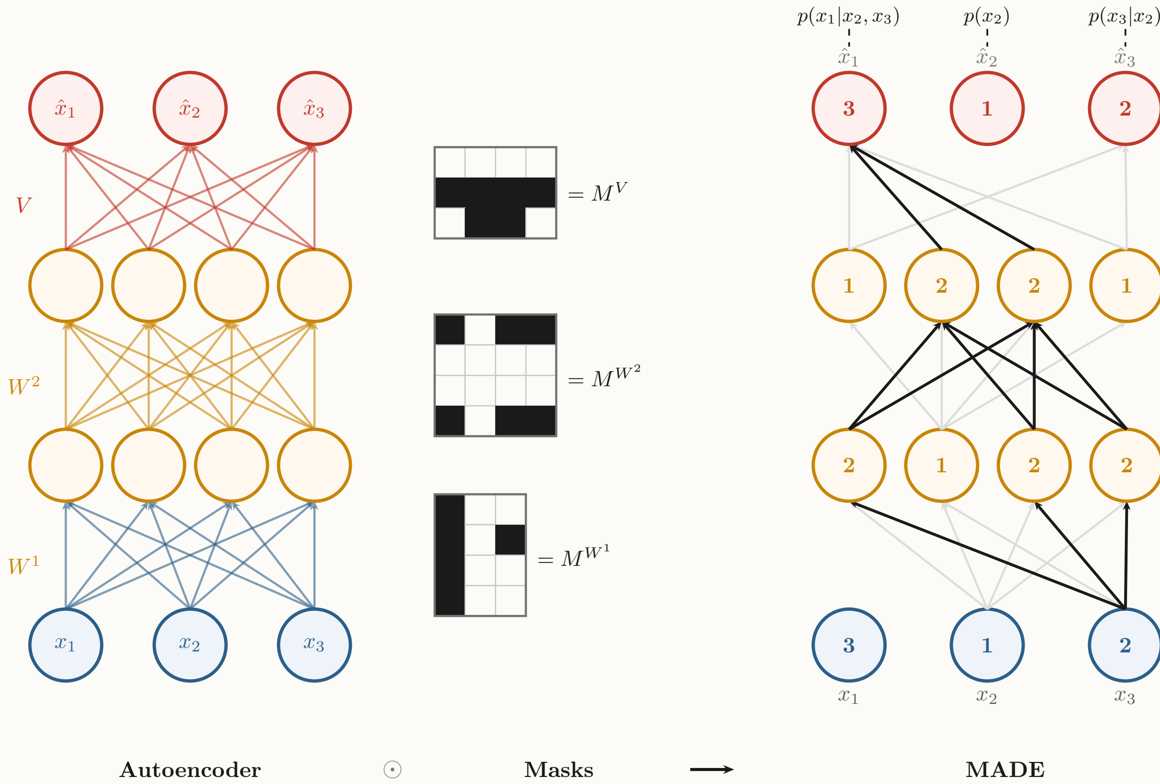

Figure 1.6: MADE: Masked Autoencoder for Distribution Estimation. Left: fully-connected autoencoder. Centre: binary masks applied via $\odot$. Right: resulting masked network with autoregressive order numbers shown in each unit.

These constraints guarantee that

$$\hat{x}_i \text{ depends only on } x_{<i},$$

so the network represents a valid autoregressive model.

Here is the paragraph you can add to introduce the masked forward pass equations:

Masked forward pass. Once the degrees are assigned and the masks are constructed, the autoregressive constraints are baked directly into the network weights via the **Hadamard product** (element-wise multiplication, denoted $\odot$). For a single hidden layer, the masked forward pass is written as

$$h(x) = g\big(b + (W \odot M^W)\, x\big)\,,$$

$$\hat{x} = \sigma\big(c + (V \odot M^V)\, h(x)\big)\,,$$

where g is a nonlinear activation function, σ denotes the sigmoid, b and c are bias vectors, and W, V are the weight matrices for the input-to-hidden and hidden-to-output layers respectively. The binary matrices M^W and M^V are the masks derived from the degree assignments described above: an entry of $M^W_{k,j} = 1$ if and only if $m(k) \geq j$, and an entry of $M^V_{i,k} = 1$ if and only if $m(k) < i$. By zeroing out the corresponding entries in the weight matrices, the

Hadamard product enforces the connectivity constraints structurally, so that no gradient or information can flow through a forbidden path. The entire responsibility of satisfying the autoregressive property therefore rests with M^W and M^V, and the remaining parameters W, V, b, c are free to be optimized by standard backpropagation without any additional constraints.

Computational structure. NADE and MADE both exploit shared computations, but differ in how the autoregressive constraint is enforced. NADE computes conditionals through a left-to-right recurrence that incrementally updates hidden pre-activations across positions. MADE instead encodes the same autoregressive dependencies directly in the network connectivity via masks, allowing all outputs to be produced by a single masked forward pass without an explicit sequential recurrence.

Order-agnostic training. Autoregressive models usually assume a fixed ordering of variables (e.g., raster scan for images). This introduces an artificial bias, since variables appearing early in the order cannot depend on later ones.

MADE removes this limitation by randomly changing the variable ordering during training and regenerating the corresponding masks. Each ordering defines a different valid autoregressive factorization of the same joint distribution.

Training across many orderings reduces reliance on any single ordering and encourages the model to learn more robust dependencies. This can be viewed as implicitly training an ensemble of autoregressive models while sharing the same parameters.

An additional benefit is flexibility: by choosing an appropriate ordering, inference tasks such as imputing missing values become easier.

At sampling time, however, a single ordering is chosen, and variables are generated sequentially according to that order.

Definition 1.7: Masked Autoencoder for Distribution Estimation (MADE)

MADE is an autoregressive model obtained by applying binary masks to the weight matrices of an autoencoder. Hidden units are assigned ordering degrees that determine allowed connections, ensuring that each output $\hat{x}_i$ depends only on preceding variables. This allows all conditional distributions to be computed in a single forward pass while preserving autoregressive structure.

Example: MADE with ordering $x_2 \to x_3 \to x_1$

Consider a binary vector

$$x = (x_1, x_2, x_3), \qquad x_i \in \{0, 1\},$$

and assume the autoregressive ordering

$$x_2 \to x_3 \to x_1.$$

Autoregressive factorization. The joint distribution becomes

$$p(x) = p(x_2)\, p(x_3 \mid x_2)\, p(x_1 \mid x_2, x_3).$$

Training (stochastic gradient descent). Given a training example (x_1, x_2, x_3), MADE receives the full binary vector as input. Due to masking, the outputs represent the conditionals:

$$\hat{x}_2 = p(x_2), \qquad \hat{x}_3 = p(x_3 \mid x_2), \qquad \hat{x}_1 = p(x_1 \mid x_2, x_3).$$

Each output is a Bernoulli probability rather than a reconstructed binary value. Parameters are updated using stochastic gradient descent by minimizing the negative log-likelihood (binary cross-entropy).

Sampling (generation). After training, sampling follows the chosen ordering, requiring one forward pass per variable. We trace through each pass below.

Forward Pass 1. The input is zero-initialised since nothing has been sampled yet:

$$\text{input} = [x_1, x_2, x_3] = [0,\ 0,\ 0].$$

MADE runs forward. By construction of the masks, the output $\hat{x}_2$ has no dependencies. We read off $p(x_2)$ and sample

$$x_2 \sim \text{Bernoulli}(p(x_2)), \qquad \text{e.g. } x_2 = 1.$$

Forward Pass 2. The sampled value of x_2 is inserted into the input:

$$\text{input} = [0,\ 1,\ 0].$$

MADE runs forward again. The masks now allow $\hat{x}_3$ to attend to x_2. We read off $p(x_3 \mid x_2 = 1)$ and sample

$$x_3 \sim \text{Bernoulli}(p(x_3 \mid x_2 = 1)), \qquad \text{e.g. } x_3 = 0.$$

Forward Pass 3. Both sampled values are inserted into the input:

$$\text{input} = [0,\ 1,\ 0].$$

MADE runs forward a final time. The masks allow $\hat{x}_1$ to attend to both x_2 and x_3. We read off $p(x_1 \mid x_2 = 1,\ x_3 = 0)$ and sample

$$x_1 \sim \text{Bernoulli}(p(x_1 \mid x_2 = 1,\ x_3 = 0)), \qquad \text{e.g. } x_1 = 1.$$

The final sampled tuple is

$$(x_1, x_2, x_3) = (1, 1, 0),$$

which forms a new generated example. Although MADE computes all conditionals in one forward pass during training, generation requires D forward passes (one per variable) because each newly sampled variable changes the conditioning context. Only the output corresponding to the next variable in the ordering is read at each step; the remaining outputs are discarded.

1.5.5 The Fixed-Length Ceiling

Every model in the previous section was designed for data with a **fixed, known dimensionality**. A 28×28 image always contains exactly 784 pixels. The model is therefore constructed with exactly n variables: the masks are built for n inputs, and the network produces n conditional outputs.

This design works well for images and audio segments, where the dimensionality is known in advance. Language, however, breaks this assumption. A sentence may contain 3 words or 3000. A paragraph, a document, or an entire codebase has no fixed length known at design time. If the number of variables is not known ahead of time, we cannot build an architecture that allocates one conditional distribution for each of them.

Handling such data therefore requires a different architectural idea: instead of constructing n separate conditional models, we need a single conditional model that can be applied repeatedly as the sequence grows.

1.5.6 Summary: Fixed-Length Models

Model	Key idea	Parameters	Parallel training
FVSBN	Logistic regression per position	$O(n^2)$	✓
NADE	Shared weights, n passes	$O(nd)$	✓
MADE	Masking, single pass	$O(nd)$	✓

1.6 Variable-Length Autoregressive Models

Machine Translation: From Images to Language Language introduces two challenges that did not arise with images.

First, sequences have variable length. A sentence may contain three words or three thousand, and the model must handle both cases with the same parameters.

Second, language is symbolic rather than numerical. Pixels already live in a numeric space: each pixel value is an integer in $\{0, \ldots, 255\}$. Words, however, are discrete symbols without any inherent numerical representation.

Before we can model sequences of words, we must therefore answer a more fundamental question:

> *How do we represent a word as a mathematical object?*

1.6.1 Words to Vectors

The Naive Approach and Why It Fails

The most obvious answer to "how do we represent a word as a mathematical object" is to assign each word a unique integer. Fix a vocabulary $\mathcal{V}$ of size V — the set of all distinct words the model will ever encounter — and map each word to an index:

$$\text{coffee} \mapsto 1, \quad \text{tea} \mapsto 2, \quad \text{the} \mapsto 3, \quad \ldots$$

This is compact and unambiguous, but it smuggles in a false assumption: that numbers are ordered. The model would be forced to treat "tea" as *between* "coffee" and "the" in some meaningful sense. It is not. Any arithmetic on these integers would be nonsense.

Second attempt: one-hot vectors. Instead of a single integer, give each word a vector of length V that is all zeros except for a single 1 at the position corresponding to that word:

$$\mathbf{e}_{\text{coffee}} = (1, 0, 0, \ldots), \quad \mathbf{e}_{\text{tea}} = (0, 1, 0, \ldots), \quad \mathbf{e}_{\text{the}} = (0, 0, 1, \ldots).$$

Formally, for a word w with vocabulary index $i(w)$:

$$(\mathbf{e}_w)_j = \begin{cases} 1 & j = i(w) \\ 0 & \text{otherwise.} \end{cases}$$

This removes the false ordering. But it creates a new problem.

One-hot vectors have no geometry.

Every pair of distinct words is exactly equidistant:

$$\|\mathbf{e}_w - \mathbf{e}_{w'}\| = \sqrt{2} \qquad \text{for all } w \neq w'.$$

Equivalently, every dot product between distinct words is zero:

$$\mathbf{e}_{\text{coffee}}^\top \mathbf{e}_{\text{tea}} \;=\; \mathbf{e}_{\text{coffee}}^\top \mathbf{e}_{\text{the}} \;=\; 0.$$

"Coffee" is exactly as far from "tea" as it is from "the." The representation contains no information about which words are related.

This means there is no generalization. A model learning that "coffee" is a drink cannot apply that knowledge to "tea" — from the representation's perspective, they are unrelated objects.

These are not minor inconveniences. They mean that any model operating on one-hot vectors must learn the behavior of every word from scratch, with zero transfer between related words. For a vocabulary of hundreds of thousands of words, this is a fundamental barrier.

What We Actually Need

The failure of one-hot vectors is precise: they treat every word as equally dissimilar. What we need is a representation where **similar words are close together** in some geometric space. But close according to what?

We need a notion of word similarity that can be extracted from data, without anyone hand-labeling which words are related. The key insight comes from an observation in linguistics:

> **The distributional hypothesis:** words that appear in similar contexts tend to have similar meanings.

Consider two words you have never seen before: *grutt* and *blarpen*. You encounter them in the following sentences:

"I drank a hot *grutt* this morning."
"She ordered a *blarpen* at the café."
"The *grutt* was bitter and dark."
"He prefers *blarpen* to tea."

You do not know what these words mean, but you know they mean something similar — because they appear near the same words: *drank, hot, morning, café, bitter, tea.* The context is the meaning.

This gives us a concrete strategy: represent each word by the *company it keeps.*

Count Vectors and Their Limits

The most direct implementation: scan a large corpus and count how often each word appears near every other word. Define a *co-occurrence matrix* $C \in \mathbb{R}^{V \times V}$ where C_{ij} records how often word j appears within a fixed window of word i.

The row $C_{i,:}$ then describes the full distribution of linguistic contexts for word i — a vector representation that embodies the distributional hypothesis directly. Words with similar rows have similar contexts, and therefore similar meanings.

This works. In practice, words like "coffee" and "tea" do end up with similar rows. But it creates two practical problems.

> **Two failures of count vectors.**
>
> **Scale.** With $V = 100{,}000$ words, C has 10^{10} entries. Storing and manipulating this matrix is expensive. Most entries are zero — the matrix is extremely sparse.
>
> **Rigidity.** The matrix must be recomputed from scratch whenever new text is available. There is no way to update it incrementally.

The standard fix for the scale problem is **singular value decomposition (SVD)**: approximate C by a low-rank matrix, keeping only the d most important dimensions. This compresses each row from a V-dimensional sparse vector into a dense d-dimensional vector, with $d \in \{50, 100, 300\}$. The result is a compact, dense representation for each word that still captures co-occurrence structure.

This works, but it remains a two-stage pipeline: collect counts, then factorize.

> Can we learn the same dense representations directly, in a single pass over text?

Learning Vectors Directly: Word2Vec

> Instead of counting contexts and then compressing, train a model to *predict* contexts directly. The vectors that emerge from this prediction task will encode the same co-occurrence structure — but learned online, in a single pass, without ever building the full matrix.

Each word w is assigned a trainable vector $\mathbf{v}_w \in \mathbb{R}^d$. All vectors are collected into an

embedding matrix:

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

where column i holds the vector for the i-th vocabulary word. Given a word w with one-hot vector $\mathbf{e}_w$, its embedding is:

$$\mathbf{v}_w = E\,\mathbf{e}_w = E_{:,\,i(w)}.$$

This is just column selection — the one-hot vector picks out a single column of E. The matrix E is learned end-to-end during training.

A note on terminology. Each word appears in two roles: sometimes it is the word being predicted *from* (the center word), sometimes it is the word being predicted (the context word). We give each word two separate vectors — $\mathbf{v}_w$ for when it acts as center, $\mathbf{u}_w$ for when it acts as context — and learn both independently. This works better in practice than sharing a single vector.

The Skipgram Objective

The training task is:

given a center word, predict which words appear nearby.

For a sentence like "I drank hot coffee this morning," treating **coffee** as the center word with window size $k = 2$ generates the training pairs:

$$(\text{coffee, hot}), \quad (\text{coffee, drank}), \quad (\text{coffee, this}), \quad (\text{coffee, morning}).$$

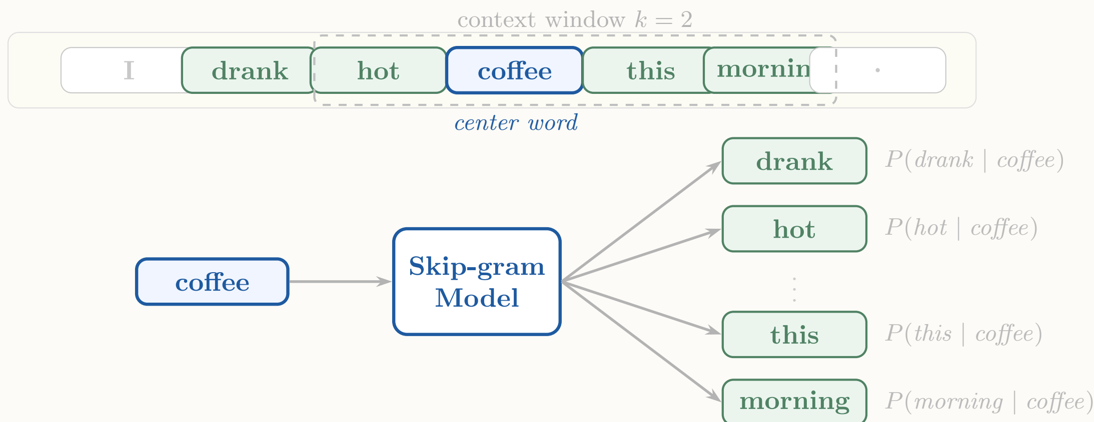

Figure 1.7: The Skip-gram model: given center word *coffee*, the model predicts each context word within a window of size $k = 2$. The training pairs are $(coffee, drank)$, $(coffee, hot)$, $(coffee, this)$, and $(coffee, morning)$.

For each pair, the model should assign high probability to the observed context word and low probability to words that did not appear. To score how compatible a center word c and context word o are, we use their dot product:

$$\mathbf{u}_o^\top \mathbf{v}_c.$$

When two vectors point in the same direction, their dot product is large. Training will push the vectors of co-occurring words to align — which is exactly what encodes "these words belong together." Converting scores to probabilities via softmax:

$$p(o \mid c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{w \in \mathcal{V}} \exp(\mathbf{u}_w^\top \mathbf{v}_c)}.$$

Training maximizes this probability across all observed center–context pairs in the corpus.

What the Gradient Actually Does

The gradient of the log-probability with respect to the center vector $\mathbf{v}_c$ is:

$$\nabla_{\mathbf{v}_c} \log p(o \mid c) = \underbrace{\mathbf{u}_o}_{\substack{\text{observed}\\\text{context}}} - \underbrace{\sum_{x \in \mathcal{V}} p(x \mid c)\, \mathbf{u}_x}_{\substack{\text{model's current}\\\text{expected context}}}.$$

This is a correction signal. The update moves $\mathbf{v}_c$ **toward the vector of the word that actually appeared** and **away from the weighted average of all words the model currently predicts**. If the model is already confident about the right context word, the correction is small. If it is confidently wrong, the correction is large.

Repeat this over every center–context pair in the corpus — billions of small corrections, each one nudging vectors toward or away from each other.

Words that share contexts get pulled toward each other. Words that never share contexts drift apart. The geometry of the embedding space becomes a map of linguistic relatedness.

CBOW:

Skipgram asks: *given this word, what surrounds it?* **Continuous Bag of Words (CBOW)** flips the question: *given the surroundings, what is the missing word?* The context vectors are averaged and fed through the same softmax to predict the center.

Averaging multiple context signals makes CBOW faster and more stable for common words. But averaging also smooths out the contribution of any individual context word — rare words, which appear in few contexts, receive weaker gradient signal. Skipgram, treating each center–context pair as a separate training example, gives rare words more updates.

	Skipgram	**CBOW**
Training speed	Slower	Faster
Rare words	Better	Worse
Frequent words	Worse	Better
Analogy tasks	Better	Worse

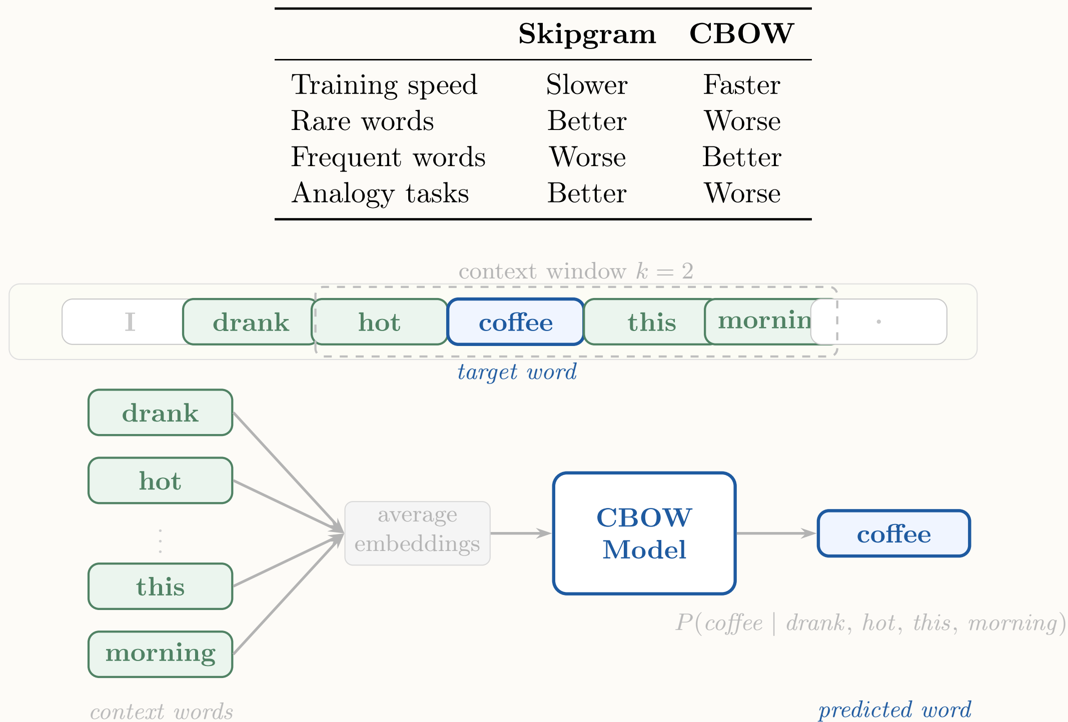

Figure 1.8: The CBOW model: all context words within a window of size $k = 2$ — *drank*, *hot*, *this*, *morning* — are averaged and used to predict the target word *coffee*.

What the Geometry Encodes

After training, the embedding space shows a simple but useful structure: some word relationships appear as consistent shifts (offsets) between vectors. This behavior is not explicitly programmed—it emerges from the training process.

A well-known example is the gender–royalty relationship:

$$\mathbf{v}_{\text{king}} - \mathbf{v}_{\text{man}} + \mathbf{v}_{\text{woman}} \approx \mathbf{v}_{\text{queen}}.$$

This example is often highlighted, but it should not be overinterpreted. The similarity is only moderate, and such relationships do not always hold reliably across all word combinations.

More reliable patterns appear in simpler and more regular relationships, especially in language structure. In these cases, the same vector difference shows up repeatedly across many examples:

Consistent Patterns in Static Embeddings

Relationship	Observed pattern
Verb forms	$\mathbf{v}_{\text{walking}} - \mathbf{v}_{\text{walk}} \approx \mathbf{v}_{\text{running}} - \mathbf{v}_{\text{run}} \approx \mathbf{v}_{\text{swimming}} - \mathbf{v}_{\text{swim}}$
Country–capital	$\mathbf{v}_{\text{Paris}} - \mathbf{v}_{\text{France}} \approx \mathbf{v}_{\text{Berlin}} - \mathbf{v}_{\text{Germany}} \approx \mathbf{v}_{\text{Rome}} - \mathbf{v}_{\text{Italy}}$
Comparatives	$\mathbf{v}_{\text{bigger}} - \mathbf{v}_{\text{big}} \approx \mathbf{v}_{\text{taller}} - \mathbf{v}_{\text{tall}} \approx \mathbf{v}_{\text{faster}} - \mathbf{v}_{\text{fast}}$

These patterns are more convincing because they appear across many examples, not just one. When the same offset works repeatedly, it suggests that the model has captured a real structure in language rather than a coincidence.

> This structure is what makes embeddings useful. They do more than store words: they capture relationships between them. For example, if a model learns several country–capital pairs, it effectively learns the idea of a "capital city" as a vector shift that can be reused across different countries.

Two qualifications constrain how far this picture extends. First, the linearity is approximate: offsets are consistent in direction but not magnitude, and the analogy method degrades for relation types that are sparse in the training corpus or that require compositional rather than associative reasoning. Second, this analysis applies to *static* embedding spaces. In transformer-based models, token representations are dynamically contextualized at every layer—the same word occupies different positions in representational space depending on its context. The clean geometric picture described here does not straightforwardly transfer to those settings.

What We Are Really Trying to Do

A word is a symbol. Behind it sits a concept — a thought, a feeling, a texture of meaning — that exists in the mind before language does. When we speak or write, we reach into a shared vocabulary and try to compress that concept into symbols, hoping it reassembles correctly in someone else's mind.

There is a useful analogy here, not a precise biological claim, but an intuition worth holding. When you encounter a concept — "coffee," "loss," "home" — something happens in the brain that is not a single neuron firing but a *combination*: a pattern spread across many neurons, each partially active, whose collective state encodes the meaning. Change the context and the pattern shifts. "Bank" beside "river" and "bank" beside "investment" are the same symbol but, in some sense, a different mental event. Meaning lives not in the symbol but in the pattern it evokes.

A word vector borrows this structure. Instead of neurons, dimensions. Instead of activation levels, real-valued coordinates. A dense vector $\mathbf{v} \in \mathbb{R}^d$ is a pattern across d numbers — some large, some small, some negative — whose combination encodes a concept in a way that no single number could.

Word2Vec was a first attempt to learn these patterns from data. It worked surprisingly well, but with one fundamental limitation: each word received a single fixed vector regardless of

context. “Bank” always activated the same combination of dimensions, whether the sentence was about rivers or money.

What we actually want is a representation that shifts with context — the same symbol producing a different pattern depending on what surrounds it. Building that is the problem we turn to next.

Relaxing orthogonality increases capacity. To see the effect concretely, consider an embedding space of dimension $n = 1000$. If we require vectors to be exactly orthogonal, then at most 1000 independent directions can exist. Now relax this constraint slightly and allow vectors to be nearly orthogonal, meaning their pairwise inner products satisfy

$$|\langle v_i, v_j \rangle| \leq \varepsilon.$$

High-dimensional geometry shows that the number of such vectors scales roughly as

$$M \approx 2^{\Theta(\varepsilon^2 n)}.$$

For example, if we allow a moderate deviation from orthogonality corresponding to angles of about 85° (so $\varepsilon \approx 0.087$), then $\varepsilon^2 n \approx 7.5$. This implies that the number of nearly independent directions can be on the order of

$$M \sim 2^{7.5} \approx 180$$

times larger than the strictly orthogonal limit. Instead of only 1000 independent directions, the space may support on the order of hundreds of thousands of approximately independent ones. As the dimension increases further, this growth becomes extremely rapid.

The implication for word vectors is important: embedding spaces do not need one dimension per concept. By tolerating small overlap between directions, a moderate-dimensional space can represent vastly more words or semantic features than its raw dimensionality would suggest, while still keeping them geometrically distinguishable.

From Words to Tokens

So far, we have treated words as the basic units of language. This was sufficient for models such as Word2Vec and GloVe, which assign a single vector to each word in a fixed vocabulary.

However, this choice has clear limitations. Natural language is productive: new words appear constantly, and many words are rare. Treating each word as an atomic unit forces the model to either ignore unseen words or map them to a generic unknown symbol.

Modern models relax this assumption by operating not on whole words, but on **tokens**.

A token is a unit of text chosen by the model. It may correspond to:

- a full word (“coffee”),
- part of a word (“drink”, “ing”),
- or even punctuation.

For example, the sentence *I am drinking coffee* might be represented as

$$(\text{I}, \ \text{am}, \ \text{drink}, \ \text{ing}, \ \text{coffee}).$$

Definition 1.8: Token

A **token** is a discrete unit of text used as input to a language model. It may correspond to a full word, a subword fragment, or a punctuation symbol.

The set of all possible tokens is called the **vocabulary**, denoted $\mathcal{V}$. In modern architectures, text is therefore modeled as a sequence of tokens:

$$(y_1, y_2, \ldots, y_N), \quad y_i \in \mathcal{V}.$$

This shift from words to tokens allows models to handle rare and unseen words, share structure across related forms, and scale more effectively to open-ended language.

> In practice, we no longer construct word vectors using skip-gram, CBOW, GloVe, or any explicit co-occurrence objective. We initialize E randomly and let the model learn it end-to-end — gradient descent discovers the same structure implicitly, as a byproduct of learning to do something harder. What these methods gave us is not a recipe but an understanding: dense vectors can encode meaning, and context is how meaning is learned.

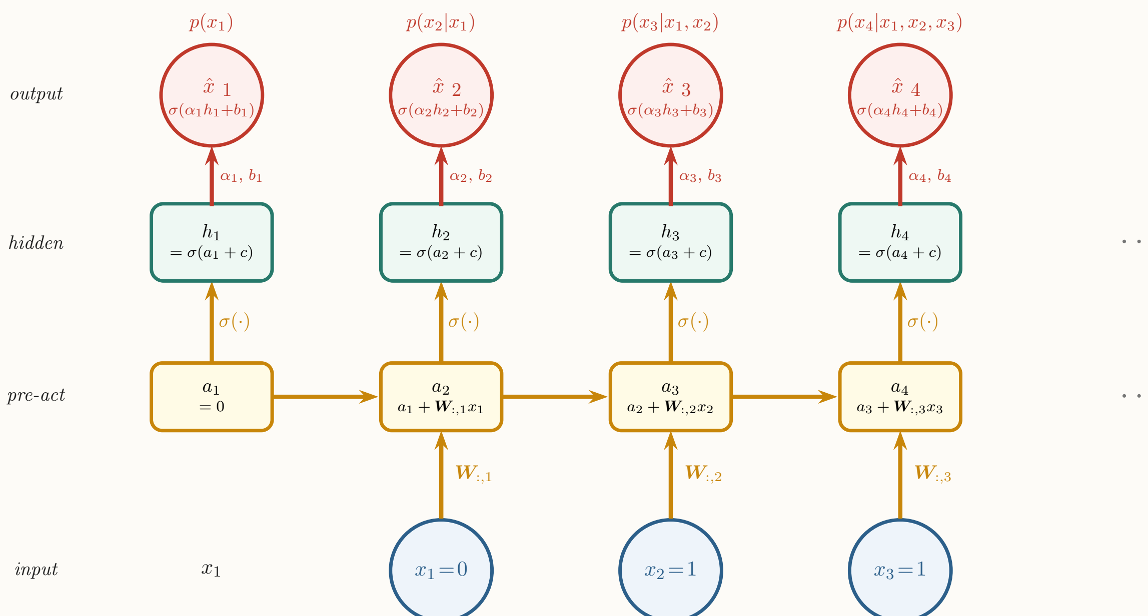

Figure 1.9: NADE: Neural Autoregressive Distribution Estimator. Pre-activations a_i chain horizontally; each hidden unit $h_i = \sigma(a_i + c)$ has its own bias c; outputs give $p(x_i \mid x_{<i}) = \sigma(\alpha_i h_i + b_i)$.

Bridging NADE to Recurrent Computation. NADE already contains a computational pattern that resembles recurrent computation. Instead of recomputing a new network for every conditional distribution, it incrementally updates a running quantity,

$$a_t = a_{t-1} + W_{:,t-1} x_{t-1},$$

which accumulates information about previously observed variables. Each conditional distribution is produced from this accumulated pre-activation.

The vector a_t therefore summarizes the prefix $x_{<t}$. As new variables are observed, the representation is updated incrementally rather than recomputed from scratch. In effect, the model maintains a running state that carries information from earlier variables to later ones.

However, the accumulation rule in NADE is limited. The update is purely additive and linear: the model learns *what* contribution each variable makes through the weight columns $W_{:,t-1}$, but not *how* the accumulated state itself should evolve in response to its current value. Moreover, the weight columns are position-specific, so the update rule is not shared across steps.

A natural generalization is to allow the state update itself to be learned. Instead of a fixed additive rule, we introduce a parameterized transition that determines how the previous state and the new input interact,

$$h_t = f(h_{t-1}, x_t).$$

This form of state evolution is familiar in control theory and dynamical systems, where a system is described by a state that evolves over time according to

$$s_t = f(s_{t-1}, u_t).$$

Applying this idea to neural networks yields a model that processes sequences by maintaining a learned state representation that evolves as new inputs arrive.

This leads to the architecture known as a *recurrent neural network*, where the transition function is typically implemented as

$$h_t = \sigma(W_h h_{t-1} + W_x x_t).$$

From this perspective, NADE performs a simple linear recurrence, while recurrent neural networks generalize the idea by learning the dynamics of the state itself. The resulting model maintains a hidden state that summarizes the sequence prefix. This state acts as the context used to make predictions at each step.

1.6.2 RNNs: The Hidden State as Context

While feed-forward networks process inputs independently, many real-world data sources exhibit sequential structure.

Many forms of data are inherently ordered. Language unfolds token by token, audio evolves continuously over time, and even images can be processed as sequences of pixels or patches. In such settings, the prediction at time t should depend not only on the current input x_t but also on the history $x_{<t}$.

A standard neural network treats inputs independently and therefore has no mechanism for remembering past observations. RNNs address this by introducing a *hidden state*, a vector

that is updated at each time step and acts as a compressed representation of everything seen so far. In this sense, the hidden state provides the contextual summary needed for sequential prediction.

Definition 1.9: Recurrent Neural Network (RNN)

A **Recurrent Neural Network** processes a sequence one element at a time, updating a hidden state h_t according to

$$h_t = \phi_1(W_{hh}h_{t-1} + W_{xh}x_t + b_h),$$

and producing an output

$$\hat{y}_t = \phi_2(W_{hy}h_t + b_y),$$

which is typically interpreted as the parameterization of a conditional distribution

$$p(y_t \mid h_t).$$

The parameters W_{hh}, W_{xh}, W_{hy} and biases b_h, b_y are shared across all time steps, allowing sequences of arbitrary length to be processed with a constant number of parameters.

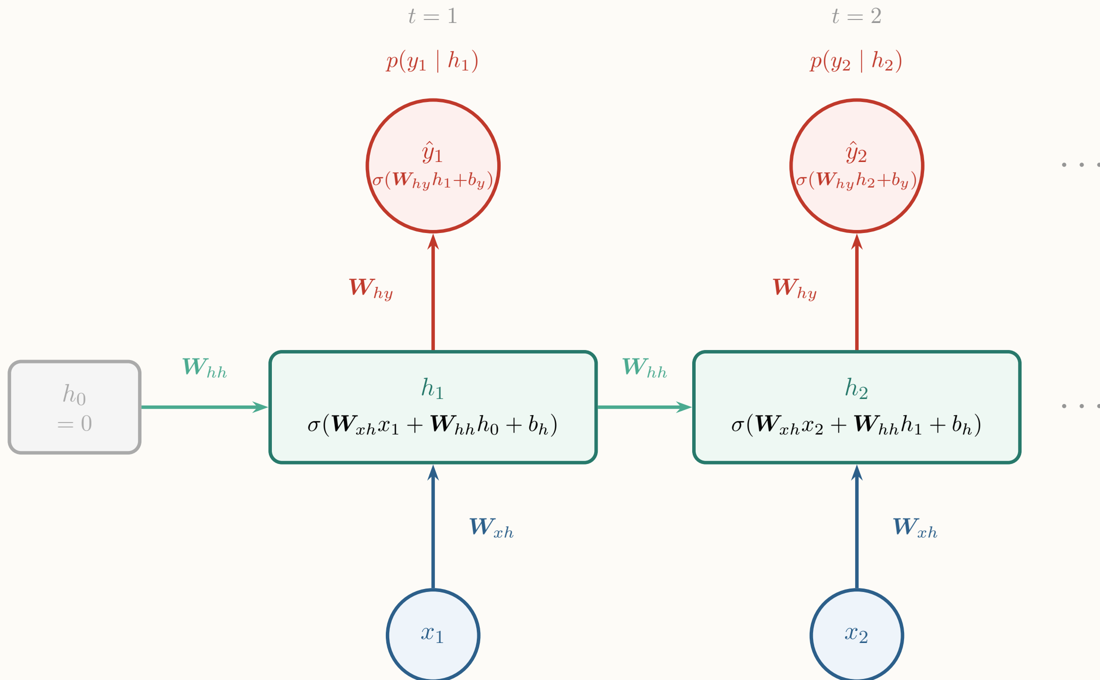

Figure 1.10: RNN (unrolled). Hidden state $h_t = \sigma(\boldsymbol{W}_{xh}x_t + \boldsymbol{W}_{hh}h_{t-1} + b_h)$ passes recurrently; outputs give $p(y_t \mid h_t) = \sigma(\boldsymbol{W}_{hy}h_t + b_y)$.

The key idea is simple: rather than explicitly storing all previous inputs, the model learns a function that compresses past information into a fixed-dimensional state. This compact

representation enables flexible sequence modeling — but it also introduces a fundamental optimization challenge.

Gradient Flow Through Time

Training an RNN requires learning how information should persist across many steps. This immediately raises a question: *how does learning signal propagate backward through a long sequence?*

Because each hidden state depends on the previous one, gradients must travel through the entire temporal chain. The behavior of this backward flow determines whether the model can learn long-range dependencies.

Assume a sequence of length T, with a loss defined at each step:

$$L_{\text{sequence}} = \sum_{t=1}^{T} L_t.$$

To study gradient propagation, it is sufficient to examine the contribution of a single future loss term L_k $(k \geq t)$ to an earlier hidden state h_t. The full gradient is obtained by summing such contributions.

Our goal is therefore to understand how

$$\frac{\partial L_k}{\partial h_t}$$

changes as the temporal distance $(k - t)$ grows.

Backpropagation Through Time Since hidden states form a temporal chain

$$h_t \to h_{t+1} \to \cdots \to h_k,$$

the chain rule yields

$$\frac{\partial L_k}{\partial h_t} = \frac{\partial L_k}{\partial h_k} \prod_{j=t+1}^{k} \frac{\partial h_j}{\partial h_{j-1}}.$$

Thus, gradients are obtained by repeatedly multiplying Jacobians — one for each recurrent step. The central question becomes: what happens when this multiplication is repeated many times?

Jacobian of a Recurrent Step From the update equation,

$$h_j = \phi_1(W_{hh} h_{j-1} + W_{xh} x_j + b_h),$$

the Jacobian with respect to the previous hidden state is

$$\frac{\partial h_j}{\partial h_{j-1}} = D_j W_{hh},$$

where

$$D_j = \operatorname{diag}\big(\phi_1'(a_j)\big), \qquad a_j = W_{hh} h_{j-1} + W_{xh} x_j + b_h.$$

Substituting,

$$\frac{\partial L_k}{\partial h_t} = \frac{\partial L_k}{\partial h_k} \prod_{j=t+1}^{k} D_j W_{hh}.$$

The same transformation is therefore applied repeatedly as gradients move backward through time. This repeated reuse of the recurrent dynamics is the source of both the power and the instability of RNN training.

A Linearized View: Why Gradients Grow or Shrink To build intuition, we first isolate the effect of recurrence itself. Suppose the activation derivatives vary slowly across time, so that locally the network behaves approximately linearly. In this regime, we may treat

$$D_j \approx I,$$

not as an exact assumption, but as a simplifying lens that reveals the dominant dynamics of recurrence.

Under this approximation,

$$\frac{\partial L_k}{\partial h_t} \approx \frac{\partial L_k}{\partial h_k} W_{hh}^{(k-t)}.$$

The problem reduces to repeated powers of the recurrent matrix. If W_{hh} is diagonalizable,

$$W_{hh} = Q\Lambda Q^{-1}, \qquad \Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_n),$$

then each component aligned with eigenvector i scales as

$$\lambda_i^{(k-t)}.$$

This immediately reveals the core phenomenon:

- If $|\lambda_i| < 1$, gradients decay exponentially.
- If $|\lambda_i| > 1$, gradients grow exponentially.

Stable propagation requires eigenvalues whose magnitudes remain close to 1, a condition that is difficult to maintain during training.

Returning to the Nonlinear Case The linearized picture provides intuition, but the full network also includes activation derivatives. Taking norms and using submultiplicativity,

$$\left\| \frac{\partial L_k}{\partial h_t} \right\| \le \left\| \frac{\partial L_k}{\partial h_k} \right\| \prod_{j=t+1}^{k} \|D_j\| \, \|W_{hh}\|.$$

For common activations such as sigmoid or tanh,

$$|\phi_1'(x)| \le \alpha < 1,$$

which implies $\|D_j\| \le \alpha$. Activation derivatives therefore introduce an additional contractive effect, reinforcing the tendency for gradients to shrink over long horizons.

Combined with the recurrent dynamics, this leads to two characteristic pathologies:

- **Vanishing gradients:** early time steps receive almost no learning signal, making long-term dependencies difficult to learn.
- **Exploding gradients:** gradient norms grow uncontrollably, producing unstable parameter updates.

Why Exploding Gradients Are Easier to Handle In practice, exploding gradients are more manageable than vanishing ones. Explosions manifest as sudden large norm spikes, making them detectable during training. A simple and widely used remedy is *gradient clipping*, which rescales gradients whenever they exceed a threshold τ:

$$g \leftarrow g \cdot \frac{\tau}{\|g\|} \qquad \text{if } \|g\| > \tau.$$

Vanishing gradients, by contrast, are silent: the learning signal simply disappears, leaving the model unable to adjust parameters that influence distant past states.

> Backpropagation through time repeatedly applies the same recurrent transformation. Small deviations from unit magnitude accumulate exponentially: some directions shrink, others grow. As sequence length increases, gradients either vanish or explode, making long-range learning unstable in vanilla RNNs.

The limitation of vanilla RNNs is not representational power but optimization.

A recurrent network can, in principle, model complex sequential dependencies using a fixed set of shared parameters. In practice, however, learning is constrained by unstable gradient flow and by the need to compress an ever-growing history into a finite-dimensional hidden state. These two issues — fragile optimization and memory compression — motivate the architectural developments that follow.

LSTM and GRU: Fixing the Memory

The core issue is that memory is updated multiplicatively at every step, so gradients must pass through repeated products. Long-range dependencies — the kind that matter most in language — are exactly what vanilla RNNs cannot learn. The LSTM was designed specifically to fix this.

Definition 1.10: Long Short-Term Memory (LSTM)

The **LSTM** introduces a **cell state** c_t alongside h_t, controlled by three learned gates:

$$\begin{aligned}
f_t &= \sigma(W_f[h_{t-1}, x_t] + b_f) \quad \text{(forget gate: what to erase from } c_t\text{)} \\
i_t &= \sigma(W_i[h_{t-1}, x_t] + b_i) \quad \text{(input gate: what new info to write)} \\
o_t &= \sigma(W_o[h_{t-1}, x_t] + b_o) \quad \text{(output gate: what to expose as } h_t\text{)}
\end{aligned}$$

$$\begin{aligned}
c_t &= f_t \odot c_{t-1} + i_t \odot \tanh(W_c[h_{t-1}, x_t] + b_c) \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}$$

The cell state c_t is updated by **addition** rather than multiplication. Gradients flowing through c_t do not shrink exponentially, allowing the LSTM to learn dependencies spanning hundreds of steps. Both h_t and c_t are deterministic.

The **Gated Recurrent Unit (GRU)** is a streamlined variant that merges c_t and h_t into a single state with two gates instead of three. Computationally cheaper, and in practice often matches LSTM performance.

LSTMs and GRUs dominated sequence modeling from their introduction through the mid-2010s, producing strong results in machine translation, speech recognition, and language modeling. Their fundamental limitation remains: processing position t requires the output of position $t-1$. Training is inherently sequential and cannot be parallelized across the time dimension.

Linear accumulation revisited. The LSTM brings back an idea that first appeared in NADE: a linear additive path for carrying information forward. RNNs replaced linear accumulation with nonlinear state updates to gain expressive power, but this came at a cost: gradients shrink as they pass through many transformations, making long-range dependencies hard to learn. The LSTM addresses this by separating two roles that vanilla RNNs conflate. The hidden state handles nonlinear computation at each step, while the cell state follows a controlled additive update, gated by what the model has learned to remember or forget, giving gradients a stable path to flow backward across many steps. The key difference from NADE is that the linearity is not fixed: the forget and input gates decide at each step how much of the additive path to use. Uncontrolled linearity limits what a model can represent; controlled linearity, combined with nonlinear computation elsewhere, is what makes long-range memory possible.

Bidirectional RNNs

A standard RNN reads left to right. At position t, the hidden state h_t summarizes everything seen so far — $x_1, \ldots, x_t$ — and nothing after it. For generation this is unavoidable: you cannot condition on tokens you have not yet produced. But when the full input sequence is already available, discarding the right context is a waste.

Consider labeling each word in a sentence with its part of speech. The word "bank" is ambiguous in isolation, but not in "the bank of the river" versus "the bank approved the loan." A left-to-right RNN processing "the bank" has not yet seen the disambiguating context. A model that could see both directions simultaneously would have no such problem.

A **bidirectional RNN** runs two RNNs over the same sequence:

$$\overrightarrow{h}_t = \mathrm{RNN}\Big(x_t,\ \overrightarrow{h}_{t-1}\Big) \quad \text{(left to right)} \tag{1.1}$$

$$\overleftarrow{h}_t = \mathrm{RNN}\Big(x_t,\ \overleftarrow{h}_{t+1}\Big) \quad \text{(right to left)} \tag{1.2}$$

The two hidden states are concatenated at each position:

$$h_t = \Big[\ \overrightarrow{h}_t\ ;\ \overleftarrow{h}_t\ \Big] \tag{1.3}$$

Now h_t encodes the full sequence context around position t — everything to its left and everything to its right. This makes bidirectional RNNs the natural choice whenever the task is to *encode* an input rather than *generate* an output. We will see this distinction become critical in the encoder-decoder architecture below.

Stacked RNNs and LSTMs

A single RNN layer maps an input sequence to a sequence of hidden states. Those hidden states are themselves a sequence — so there is nothing stopping us from feeding them into another RNN.

$$h_t^{(1)} = \mathrm{RNN}^{(1)}\left(x_t,\, h_{t-1}^{(1)}\right) \tag{1.4}$$
$$h_t^{(2)} = \mathrm{RNN}^{(2)}\left(h_t^{(1)},\, h_{t-1}^{(2)}\right) \tag{1.5}$$
$$\vdots$$
$$h_t^{(L)} = \mathrm{RNN}^{(L)}\left(h_t^{(L-1)},\, h_{t-1}^{(L)}\right) \tag{1.6}$$

Lower layers capture local, surface-level patterns; upper layers build more abstract representations over longer spans. The final output at each timestep is $h_t^{(L)}$, the top-layer hidden state. The same principle applies directly to LSTMs — each layer receives the hidden states of the layer below as its input sequence, and maintains its own cell state independently.

Sequence-to-Sequence Learning: The Encoder-Decoder Architecture

Every model so far models a single sequence: $p(x_1, \ldots, x_n)$. But many tasks involve two sequences — an input and an output that are related but distinct. Machine translation is the clearest example: given an English sentence of length n, produce a German sentence of length m, where $n \neq m$ in general and neither is known in advance.

What we need is a conditional model:

$$p(y_1, \ldots, y_m \mid x_1, \ldots, x_n) \tag{1.7}$$

The natural way to build this is to split the job in two. One network reads the input and builds a representation of it. Another network generates the output, conditioned on that representation. This is the **encoder-decoder** architecture.

Encoder. The encoder reads the source sequence $x_{1:n}$ and transforms it into a single fixed-length representation called the **context vector**. At each step the encoder updates its hidden state using the current input token and the previous state:

$$h_t = f_W(x_t, h_{t-1}) \tag{1.8}$$

After the final token is processed, the last hidden state becomes the summary of the entire input:

$$c = h_n \tag{1.9}$$

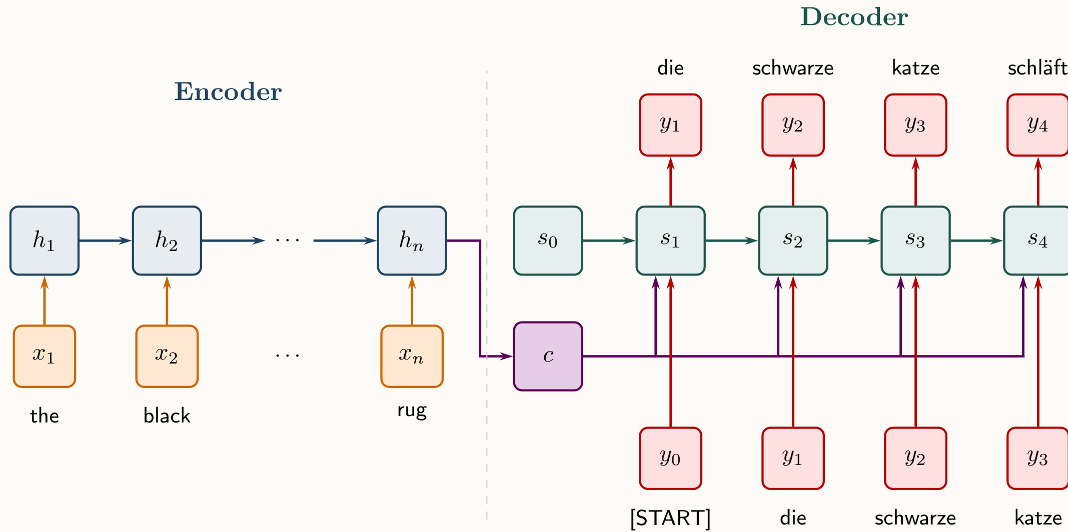

Figure 1.11: Sequence-to-sequence encoder–decoder architecture translating the English sentence "the black cat sleeps on the rug" into German.

Decoder. The decoder generates the output sequence one token at a time. Its hidden state depends on the previously generated token, the previous decoder state, and the context vector produced by the encoder:

$$s_t = g_U(y_{t-1}, s_{t-1}, c) \tag{1.10}$$

The probability of the next token is obtained by applying a softmax to the decoder state:

$$p(y_t \mid y_{<t}, x_{1:n}) = \mathrm{softmax}(W s_t + b) \tag{1.11}$$

The model therefore factorizes the conditional probability of the output sequence as

$$p(y_{1:m} \mid x_{1:n}) = \prod_{t=1}^{m} p(y_t \mid y_{<t}, x_{1:n}) \tag{1.12}$$

The bottleneck. There is a quiet problem in this architecture. The context vector c is fixed-size. It does not matter whether the source sentence has 5 words or 50 — the encoder must compress everything into the same vector. For short sentences, this works well. For long sentences, it does not: information gets squeezed out, and translation quality degrades visibly with source length.

The problem is not the LSTM — a more powerful encoder will not fix it. The problem is structural. A single vector is a hard bottleneck, and no encoder, however deep, can overcome a fixed-size information channel when the input is arbitrarily long.

The fix is to stop forcing the decoder to read from a single summary. Instead, let the decoder look back at *all* the encoder hidden states $h_1, \ldots, h_n$ and decide, at each generation step, which parts of the source to focus on. That mechanism is **attention**, and it is the subject of the next section.

1.6.3 Attention

Dynamic Retrieval: Let the Decoder Look Back

The root cause of the seq2seq bottleneck is that the decoder is cut off from the source sentence. Once the encoder finishes, the entire input must be reconstructed from a single fixed vector.

The natural fix is simple: keep all the encoder hidden states available and let the decoder look at them directly.

Let the encoder produce a sequence of hidden states

$$(h_1, h_2, \dots, h_{T_x}),$$

where each h_j represents the j-th English word in the context of its neighbours. These vectors are not discarded after encoding. Instead, the decoder can access all of them at every step.

When generating the i-th German word, the decoder constructs a *dynamic context vector* by taking a weighted combination:

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} \, h_j.$$

The weights α_{ij} determine how relevant each English word is to the current decoding step. They are computed by scoring the compatibility between the decoder state s_{i-1} and each encoder state h_j, then normalising via softmax:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_k \exp(e_{ik})}.$$

The weights form a probability distribution over the source positions. When the decoder generates the German verb *schläft*, it places high weight on the English word *sleeps* and low weight on unrelated words such as *the* or *rug*. When generating a noun, the focus shifts accordingly.

The retrieval is dynamic and input-dependent. At each decoding step the model decides which parts of the source sentence matter most.

This mechanism is *attention*.

Definition 1.11: Attention

Attention converts a set of static representations into a dynamic contextual representation by computing a relevance-weighted mixture of those representations.

The bottleneck disappears. The decoder no longer extracts everything from a single vector; it selectively retrieves information from the entire encoded sequence. This raises a key question:

How should compatibility be scored? Everything hinges on computing the compatibility score e_{ij} well. This is the central design question of attention.

First Design: Score with a Network

The most expressive option is to learn a small neural network that takes the decoder state s and encoder state h and outputs a score:

$$e = v^\top \tanh(W_1 s + W_2 h).$$

The network learns directly from translation data what compatibility means. It can capture complex nonlinear interactions between the decoder and encoder representations.

This approach works well, but it has a structural cost. The network must be evaluated for *every pair* of decoder and encoder positions. Because decoder states are generated sequentially, the computation cannot be parallelised easily. For long sentences this becomes expensive.

Second Design: Score with a Dot Product

A much simpler alternative is to replace the neural network with a dot product:

$$e = s^\top h.$$

The dot product requires no additional parameters and is extremely cheap to compute. Surprisingly, it performs competitively in practice.

But taking dot products between raw hidden states introduces two problems.

The relevance problem. A dot product measures raw similarity between vectors. Two states will score highly if their representations happen to be close, even if the words are not actually relevant to each other in this particular sentence. Similarity is not necessarily relevance.

The scale problem. When the hidden state dimension d is large, dot products tend to grow in magnitude proportionally to d. This pushes the softmax into saturation, where gradients become very small and training slows down.

Both issues arise because the vectors are being compared in the wrong space.

The Fix: Project Before You Compare

The solution is to project the states into a learned interaction space before computing the dot product.

Consider the English sentence

"The black cat sleeps on the rug."

Suppose the decoder has produced the German phrase *"Die schwarze Katze"* and must now generate the next word. It needs to locate the English verb *sleeps*.

The decoder state s encodes everything known so far about the German sentence. The encoder states $h_1, \ldots, h_n$ encode the English words. But these vectors live in different representation spaces. Comparing them directly is like comparing measurements expressed in different units.

The fix is to learn projections that place them in a common space.

The decoder state is projected into a **query**:

$$q = sW_Q.$$

Each encoder state is projected into a **key**:

$$k_j = h_j W_K.$$

Both projections map into the same d_k-dimensional space. The compatibility score becomes

$$e_j = \frac{q^\top k_j}{\sqrt{d_k}}.$$

The scaling factor $\sqrt{d_k}$ prevents dot products from growing too large and stabilises training.

Now the decoder query for the next German verb aligns strongly with the key of the English word *sleeps* and weakly with unrelated words.

The projections solve **Relevance**: the model learns how queries and keys should align.

What Gets Retrieved: The Value Projection

Once attention weights are computed, the model takes a weighted sum of vectors from the encoder. But a weighted sum of *what*?

Using the key vectors themselves would mix two roles that should remain separate: how a word is *found* and what it *contributes* once found.

Keys are optimised for discoverability. But the decoder ultimately needs the meaning of the word.

To separate these roles, each encoder state is also projected into a **value** vector:

$$v_j = h_j W_V.$$

Queries and keys determine how much each word contributes. Values determine what information is actually retrieved.

The resulting context vector is

$$c_i = \sum_j \alpha_{ij} v_j.$$

The word *sleeps* is located because its key matches the query. What is retrieved, however, is its value — a representation encoding its semantic and grammatical content.

What projection means geometrically For each English word the encoder produces a hidden state h_j. This state is projected into a key k_j, while the decoder state s is projected into a query q.

Seven English words therefore produce seven keys. The decoder produces one query. Their dot products measure how well each source word matches the decoder's current need.

Softmax converts these scores into weights, and the decoder retrieves a weighted mixture of value vectors.

Geometrically, the projections isolate the components of the representations that are relevant for interaction. The query isolates the features that express what the decoder is searching for. The keys isolate the features that advertise what each word contains.

The hidden states themselves are simply the vectors that feed this interaction.

This observation leads to an important question. Attention does not depend on how the vectors $h_1, \ldots, h_{T_x}$ were produced. It only requires that each position in the sentence is associated with a vector that can be projected into queries, keys, and values.

In the encoder–decoder architecture these vectors come from a recurrent encoder:

$$h_j = f(x_j, h_{j-1}).$$

But once attention allows the decoder to retrieve information directly from any position in the source sentence, the role of the recurrent chain changes. In the original encoder, the dependence on the previous state was necessary because contextual information had to be carried forward through the sequence. Each hidden state inherited information from the previous one so that earlier words could influence later representations.

Attention removes this requirement. The decoder can now access every position in the source sentence directly and retrieve whatever information it needs through the attention weights. Context no longer has to be transported step by step through the recurrent chain.

Once this mechanism exists, the dependence on the previous hidden state becomes unnecessary. The representation associated with each word no longer needs to carry the entire sentence history forward. The encoder transformation can therefore be simplified to

$$h_j = f(x_j).$$

At this point the role of f itself becomes unclear. Word embeddings x_j are learned parameters of the model. If a particular vector representation of a word is useful for the attention mechanism, the training process can move the embedding in that direction.

In effect, the embedding itself can serve as the vector that attention operates on. The model can simply learn the embedding of each word to be the representation that best supports attention. We can therefore identify the hidden state with the word vector itself,

$$h_j = x_j.$$

The same reasoning applies on the decoder side. The decoder state s was previously the representation used to form queries. But if the model already maintains a vector representation for each generated token, that representation can directly serve as the query source. Attention therefore operates on the word representations themselves rather than on hidden states produced by a recurrent network.

Once we adopt this view, attention can be written directly in terms of the word representations in the encoder and decoder sequences.

The decoder representation at position i produces a query, while the encoder representations produce keys and values:

$$q_i = y_i W_Q, \qquad k_j = x_j W_K, \qquad v_j = x_j W_V.$$

The decoder query is compared with all encoder keys to determine which source positions are relevant, and the resulting weights are used to retrieve a weighted combination of the corresponding values. Written in matrix form, this mechanism becomes the standard definition of scaled dot-product attention.

Definition 1.12: Scaled Dot-Product Attention

For queries Q, keys K, and values V stacked into matrices:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V.$$

- **Query** (Q): what each position is looking for.
- **Key** (K): what each position is advertising it contains.
- **Value** (V): what each position contributes once found.
- $\sqrt{d_k}$: scaling factor that prevents dot products from growing large and saturating the softmax.

The output at each position is a weighted sum of values, where the weights are determined entirely by query–key compatibility.

The mechanism developed so far is *cross-attention*: the German decoder attends to the English encoder. The query comes from the German side; the keys and values come from the English side. The two sequences are different.

Definition 1.13: Cross-Attention

Cross-attention applies the attention operator to two different sequences. Queries are produced from the decoder representations, while keys and values are produced from the encoder representations.
Let Y be the matrix of decoder representations and X the matrix of encoder representations. Cross-attention is defined as

$$\text{CrossAttention}(Y, X) = \text{Attention}(YW_Q,\ XW_K,\ XW_V).$$

Each decoder position forms a query that searches over the encoder representations to retrieve the information most relevant to the next prediction.

Example: Parallel Cross-Attention

The encoder produces representations

$$X_{enc} = (x_1, \ldots, x_7)$$

for the English words

$$(\text{The},\ \text{black},\ \text{cat},\ \text{sleeps},\ \text{on},\ \text{the},\ \text{rug}).$$

Suppose the decoder currently contains the partial German sequence

$$Y = (\text{Die},\ \text{schwarze},\ \text{Katze},\ \text{schläft})$$

with representations

$$Y = (y_1, y_2, y_3, y_4).$$

The query, key, and value projections are computed as

$$Q = YW_Q, \qquad K = X_{enc}W_K, \qquad V = X_{enc}W_V.$$

The i-th row of Q is the query vector q_i, while the j-th rows of K and V correspond to the key k_j and value v_j associated with the j-th source word. All query–key compatibilities are computed simultaneously:

$$= \frac{QK^\top}{\sqrt{d_k}}.$$

Applying softmax row-wise produces the attention matrix

$$A = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right).$$

	The	black	cat	sleeps	on	the	rug
Die	0.62	0.05	0.04	0.02	0.02	0.03	0.02
schwarze	0.03	0.68	0.05	0.02	0.02	0.02	0.03
Katze	0.02	0.05	**0.70**	0.03	0.03	0.02	0.02
schläft	0.02	0.03	0.06	**0.72**	0.05	0.03	0.04

Each row corresponds to a decoder query, and each column corresponds to an encoder key. The rows form probability distributions over the source positions.

The matrix shows how different German words attend to different parts of the English sentence. For example:

- *Die* attends strongly to *The*.
- *schwarze* attends strongly to *black*.
- *Katze* attends strongly to *cat*.
- *schläft* attends strongly to *sleeps*.

Because the matrix $QK^\top$ is computed using matrix multiplication, all these alignments are obtained *in parallel* in a single operation.

The context vectors for all decoder positions are then computed simultaneously as

$$C = AV.$$

Computing the context vectors The context matrix is obtained by multiplying the attention matrix with the value matrix:

$$C = AV.$$

$$C = \begin{bmatrix} 0.62 & 0.05 & 0.04 & 0.02 & 0.02 & 0.03 & 0.02 \\ 0.03 & 0.68 & 0.05 & 0.02 & 0.02 & 0.02 & 0.03 \\ 0.02 & 0.05 & 0.70 & 0.03 & 0.03 & 0.02 & 0.02 \\ 0.02 & 0.03 & 0.06 & \mathbf{0.72} & 0.05 & 0.03 & 0.04 \end{bmatrix} \quad \overset{V}{\begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \end{bmatrix}}$$

The highlighted row corresponds to the decoder word *schläft*. Multiplying this row with the value matrix produces

$$c_{\text{schläft}} = 0.02v_1 + 0.03v_2 + 0.06v_3 + 0.72v_4 + 0.05v_5 + 0.03v_6 + 0.04v_7.$$

Self-Attention: Turning the Mechanism Inward

But now consider the English sentence itself, before translation begins. The meaning of each English word depends on the words around it. The word *bank* means something different beside *river* than beside *loan*. A recurrent encoder captures this by passing information forward step by step, but this forces early words to carry their influence through a long chain of hidden states to reach distant ones — a slow, sequential, and lossy process.

The same attention mechanism can be applied within a single sequence. Each English word produces its own query, key, and value, and attends over all other words in the same sentence. This is *self-attention*: the sequence attending to itself.

The effect is significant. Any two positions interact directly in a single operation — *bank* can immediately attend to *river* or *loan* to resolve its meaning, without the relationship having to propagate through every intermediate hidden state. Long-range dependencies are no harder to capture than short-range ones, and the entire computation runs in parallel.

Definition 1.14: Self-Attention

The queries, keys, and values are all derived from the same input sequence:

$$\text{SelfAttention}(X) = \text{Attention}(XW_Q,\ XW_K,\ XW_V).$$

Each position attends over all other positions in the same sequence, allowing direct interaction between any two tokens regardless of their distance. This enables the model to refine the representation of each token using information from the entire sentence.

Self-attention in the encoder allows every position to attend to every other position in the

sentence. The decoder, however, generates the target sentence one token at a time. When predicting the token at position i, the model must not see future tokens.

Masked Self-Attention

In the encoder, every token can attend to every other token in the sentence because the entire input is known in advance.

The decoder is different. It generates the German sentence one word at a time. Suppose the model has produced

Die schwarze Katze

and is about to generate the next word *schläft*. When computing the representation for the next position, the decoder may attend to the words that already exist — *Die*, *schwarze*, and *Katze* — but it must not look ahead to words that have not yet been generated, such as *auf*, *dem*, or *Teppich*. Allowing this during training would reveal the future and make the task trivial.

To enforce this causal ordering, attention to future positions is masked.

Definition 1.15: Masked Self-Attention

Masked self-attention computes attention within a sequence while preventing positions from attending to future tokens.

$$\mathrm{MaskedAttention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^\top + M}{\sqrt{d_k}}\right) V$$

The mask matrix M is

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

so that token i can attend only to tokens at positions $j \leq i$.

Multiple Heads: Multiple Notions of Relevance

A single set of projections W_Q, W_K, and W_V learns one notion of relevance. At each decoding step, the attention mechanism produces a single distribution over the source sequence — one decision about what information to retrieve. The difficulty is that relevance is not a single concept.

Even within one sentence, generating a word may require answering several different questions simultaneously. When the German decoder generates *schläft*, it may need to determine: *what is the verb?* (to retrieve *sleeps*), *what is the subject?* (to enforce agreement with *cat*), and *what completes the predicate?* (to understand the role of *rug*). These relationships are structurally different. A single attention distribution must place its weight somewhere — it cannot simultaneously focus sharply on all of these positions. The model is therefore forced to average together relationships that should remain distinct.

The Transformer resolves this limitation by running multiple attention mechanisms in parallel. Instead of learning a single notion of relevance, the model learns H different ones, each operating in its own representation subspace. This mechanism is known as *multi-head attention*.

Definition 1.16: Multi-Head Attention (MHA)

Let
$$Q \in \mathbb{R}^{n_q \times d_k}, \quad K \in \mathbb{R}^{n_k \times d_k}, \quad V \in \mathbb{R}^{n_k \times d_v}$$
be matrices of queries, keys, and values obtained from learned projections of token representations.
Multi-head attention partitions the representation space into H subspaces and applies the attention mechanism independently in each subspace.
Let
$$Q^{(h)}, K^{(h)}, V^{(h)}$$
denote the slices of Q, K, V corresponding to head h. Each head computes
$$\text{head}_h = \text{Attention}(Q^{(h)}, K^{(h)}, V^{(h)}).$$
The outputs of all heads are concatenated and projected back to the model dimension:
$$\text{MHA}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_H)\, W^O.$$

Each head learns its own notion of what makes two positions relevant to each other. One head may capture syntactic agreement, another lexical correspondence, another long-range coreference. Because all heads operate on the same representations simultaneously, the model can represent multiple relationships without forcing them into a single attention pattern.

Two important special cases of multi-head attention appear in the Transformer architecture.

Definition 1.17: Multi-Head Self-Attention (MHSA)

Let $X \in \mathbb{R}^{n \times d_{\text{model}}}$ be a sequence of token representations.
Multi-head self-attention runs H attention heads in parallel. For each head h:
$$Q^{(h)} = XW_Q^{(h)}, \qquad K^{(h)} = XW_K^{(h)}, \qquad V^{(h)} = XW_V^{(h)}.$$
Each head computes
$$\text{head}_h = \text{Attention}(Q^{(h)}, K^{(h)}, V^{(h)}).$$
The outputs of all heads are concatenated and projected:
$$\text{MHSA}(X) = \text{Concat}(\text{head}_1, \dots, \text{head}_H)\, W^O.$$

Definition 1.18: Masked Multi-Head Self-Attention (MMHSA)

Let $Y \in \mathbb{R}^{n \times d_{\text{model}}}$ be a sequence of decoder representations.
Masked multi-head self-attention runs H attention heads in parallel. For each head h:

$$Q^{(h)} = YW_Q^{(h)}, \qquad K^{(h)} = YW_K^{(h)}, \qquad V^{(h)} = YW_V^{(h)}.$$

Each head applies the masked attention mechanism defined previously:

$$\text{head}_h = \text{MaskedAttention}(Q^{(h)}, K^{(h)}, V^{(h)}).$$

The outputs of all heads are concatenated and projected:

$$\text{MMHSA}(Y) = \text{Concat}(\text{head}_1, \dots, \text{head}_H)\, W^O.$$

Definition 1.19: Multi-Head Cross-Attention (MHCA)

Let

$$Y \in \mathbb{R}^{n_{\text{dec}} \times d_{\text{model}}}, \qquad X \in \mathbb{R}^{n_{\text{enc}} \times d_{\text{model}}}$$

be decoder and encoder representations.
Multi-head cross-attention runs H attention heads in parallel. For each head h:

$$Q^{(h)} = YW_Q^{(h)}, \qquad K^{(h)} = XW_K^{(h)}, \qquad V^{(h)} = XW_V^{(h)}.$$

Each head computes

$$\text{head}_h = \text{Attention}(Q^{(h)}, K^{(h)}, V^{(h)}).$$

The outputs of all heads are concatenated and projected:

$$\text{MHCA}(Y, X) = \text{Concat}(\text{head}_1, \dots, \text{head}_H)\, W^O.$$

- Each head operates in a reduced-dimensional subspace with $d_k = d_v = d_{\text{model}}/H$.
- Concatenating the H heads restores the original dimensionality $Hd_v = d_{\text{model}}$.
- Because each head works in a smaller space, the total computational cost remains comparable to that of single-head attention.
- The output projection W^O mixes information across heads, combining signals discovered by different heads into a unified representation.

Recurrence was never the goal. Looking back at the full arc: recurrence was not a fundamental principle. It was a mechanism for propagating dependencies step by step because no direct alternative existed. Attention is that alternative — any two positions can interact directly, without a sequential chain between them.

Cross-attention lets the German decoder retrieve information from the English encoder. Self-

attention lets a sequence refine its own representation by allowing its tokens to interact with one another. In the decoder this interaction must respect causal order: when predicting the word at position t, the model must rely only on the words that have already been generated. This is enforced through *masked self-attention*, which blocks attention to future positions.

Both cross-attention and self-attention rely on the same underlying mechanism: queries, keys, and values are computed, compatibility scores determine which positions are relevant, and the resulting information is aggregated.

The pieces are now in place. The Transformer is the architecture that builds entirely around this idea, replacing recurrence with attention throughout and surrounding it with the components needed to make attention work at scale.

1.6.4 Transformer

What the Recurrence Was Providing

Token embeddings carry over unchanged to the Transformer. Given a tokenized sequence $(y_1, y_2, \ldots, y_N)$, the embedding matrix

$$E \in \mathbb{R}^{|\mathcal{V}| \times d_{\text{model}}}$$

maps tokens to dense vectors, producing

$$Y \in \mathbb{R}^{N \times d_{\text{model}}}.$$

What recurrence naturally provided was an implicit encoding of token order. Recurrent models process tokens sequentially, so the representation at position t is built from the representation at $t-1$. Through this chain, the model encodes the order of the sequence via its evolving hidden state.

Self-attention does not have this property. It processes all tokens simultaneously and is therefore permutation-invariant: it has no intrinsic notion of which token came first.

Removing recurrence therefore removes this implicit encoding of order. The Transformer reintroduces this information by adding a **positional encoding** to each token embedding:

$$Y^{(0)} = \mathit{TokenEmbedding} + \mathit{PositionEmbedding}.$$

Each representation now contains both the identity of the token and its position in the sequence. The precise form of the *position embedding*—how positional information is encoded numerically and injected into the model—will be discussed in detail in the chapter on Large Language Models.

Re-engineering the Decoder

In the recurrent architecture, the decoder updated its internal state according to

$$s_t = f(s_{t-1}, \, y_{t-1}, \, c_t).$$

This update combines three distinct roles: incorporating previously generated tokens, retrieving relevant information from the source sequence, and transforming the resulting information into a representation suitable for prediction.

The Transformer preserves these roles but implements them using separate transformations operating on an entire sequence of token representations rather than a single evolving state.

Target history. In the recurrent model, the pair (s_{t-1}, y_{t-1}) summarized the prefix of previously generated tokens. The Transformer replaces this mechanism with *masked multi-head self-attention*. Given decoder representations Y, this transformation allows each token to attend to earlier positions in the sequence:

$$H_{\text{sa}} = \text{Masked Multi-Head Self-Attention}(Y).$$

Each token can therefore gather information from the generated prefix.

Source retrieval. The recurrent decoder obtained source information through the context vector c_t. In the Transformer this role is performed by *multi-head cross-attention*. Decoder representations act as queries, while encoder representations provide keys and values:

$$H_{\text{ca}} = \text{Multi-Head Cross-Attention}(H_{\text{sa}}, X_{\text{enc}}).$$

This transformation retrieves information from the encoded source sequence that is relevant for each decoder position.

Local transformation. Finally, the recurrent update function $f(\cdot)$ transformed the combined information into the next decoder state. In the Transformer, this transformation is implemented by a position-wise feed-forward network:

$$Y' = \text{FFN}(H_{\text{ca}}).$$

Together these computations form a transformation that maps one set of decoder representations into another:

$$Y' = F(Y, X_{\text{enc}}).$$

Rather than evolving a single hidden state over time, the Transformer transforms the entire set of token representations simultaneously.

Repeated Transformation

Applying this transformation once is rarely sufficient. Instead, the computation is repeated, progressively refining the sequence representation:

$$Y^{(0)} \to Y^{(1)} \to Y^{(2)} \to \cdots \to Y^{(L)}.$$

Each stage receives the representation produced by the previous stage and applies another transformation of the same form. The network therefore constructs increasingly structured representations as information propagates through the computation.

Consider *"The black cat sleeps on the rug."* In an early stage, self-attention resolves local relationships: *sleeps* attends to *cat* to identify its subject, while *black* attaches to *cat* as

its modifier. Once these relationships are encoded, later transformations can reason about higher-level structure, such as how the predicate constrains what tokens may follow.

Each step therefore refines the representation constructed by the previous one.

Residual Refinement

Deep stacks of nonlinear transformations are difficult to train because gradients must propagate through many stages of computation. Residual networks addressed this difficulty by introducing skip connections that allow information to pass directly through the network.

Instead of learning a complete transformation $F(x)$, a block learns only how the representation should change:

$$\text{output} = x + F(x).$$

The identity path preserves the current representation while the learned transformation contributes a correction.

Transformers adopt the same principle. Rather than replacing the current representation, each stage predicts a refinement that is added to it:

$$Y^{(l+1)} = Y^{(l)} + \Delta^{(l)}.$$

The term $\Delta^{(l)}$ represents the correction produced during the l-th transformation.

This correction is generated by the three computations introduced earlier. We therefore decompose it into

$$\Delta^{(l)} = \Delta^{(l)}_{\text{sa}} + \Delta^{(l)}_{\text{ca}} + \Delta^{(l)}_{\text{ffn}}.$$

The individual contributions correspond to masked self-attention, cross-attention, and the feed-forward network:

$$\Delta^{(l)}_{\text{sa}} = \text{Masked Multi-Head Self-Attention}(Y^{(l)}),$$

$$\Delta^{(l)}_{\text{ca}} = \text{Multi-Head Cross-Attention}\left(Y^{(l)} + \Delta^{(l)}_{\text{sa}},\, X_{\text{enc}}\right),$$

$$\Delta^{(l)}_{\text{ffn}} = \text{FFN}\left(Y^{(l)} + \Delta^{(l)}_{\text{sa}} + \Delta^{(l)}_{\text{ca}}\right).$$

Unrolling the residual updates across all transformations reveals how the final representation is constructed:

$$Y^{(L)} = Y^{(0)} + \sum_{l=1}^{L} \big(\Delta^{(l)}_{\text{sa}} + \Delta^{(l)}_{\text{ca}} + \Delta^{(l)}_{\text{ffn}}\big).$$

The representation produced by the network is therefore the initial embedding plus every refinement contributed during the computation. Information is not overwritten; it is progressively accumulated.

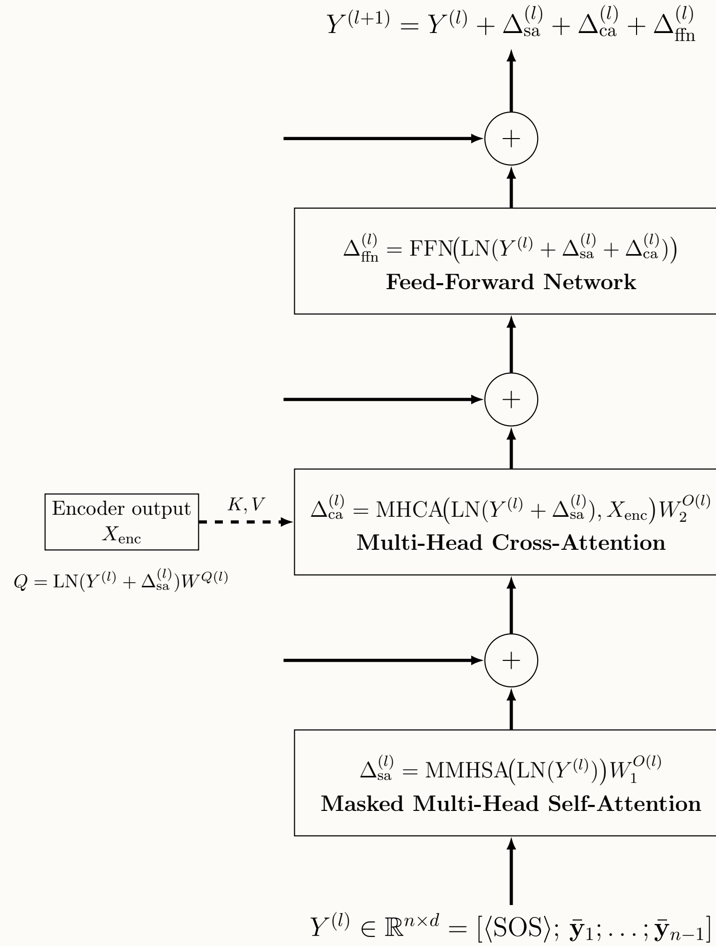

Figure 1.12: One Transformer decoder layer (pre-norm). Input $Y^{(l)}$. Self-attention forms a German context, cross-attention mixes encoder representations X_{enc}, and a feed-forward network refines the representation. Residual connections accumulate the deltas to produce $Y^{(l+1)}$.

From Representations to Predictions

After L layers, each row $y_i^{(L)}$ of $Y^{(L)}$ is a rich contextual representation of token i — the initial embedding plus every correction contributed by the stacked Transformer layers. This vector is not yet a prediction. It is a summary of everything the model knows about position i in context. The question is how to turn that summary into a probability distribution over the next word.

The idea is simple: measure how compatible this representation is with every word in the vocabulary. Each word has an embedding vector in the matrix E, which represents the meaning of that word in the model's embedding space. Training encourages the representation $y_i^{(L)}$ to align with the embedding of the word that should come next, while remaining dissimilar to the embeddings of other words.

This compatibility is computed by projecting the representation into vocabulary space:

$$z_i = W_o\, y_i^{(L)} + b_o,$$

where $z_i \in \mathbb{R}^{|\mathcal{V}|}$ assigns a score (logit) to every word in the vocabulary.
In practice, the output projection W_o is often tied to the input embedding matrix E. The same vectors that encode what words mean when they enter the model are reused to score which word should come next.
These scores are converted into a probability distribution using softmax:

$$p(w \mid y_{<i}) = \frac{\exp(z_i[w])}{\sum_{w'} \exp(z_i[w'])}.$$

Every word receives a probability, and the probabilities sum to one. The word with the highest probability is the model's most confident prediction for the next token at position i.
Because

$$y_i^{(L)} = y_i^{(0)} + \sum_{l=1}^{L} \Delta^{(l)},$$

$$\Delta^{(l)} = \underbrace{\Delta_{\text{sa}}^{(l)}}_{\text{German context}} + \underbrace{\Delta_{\text{ca}}^{(l)}}_{\text{English source context}} + \underbrace{\Delta_{\text{ffn}}^{(l)}}_{\text{representation refinement}}$$

the final prediction reflects the accumulated contributions of all layers. The logits are not produced by the final layer alone; they result from every round of attention and transformation that refined the representation along the way.

Encoder only

Training and Inference

Training objective. The model is trained to predict each word in a sequence given the words that came before it. Given a sentence $(x_1, x_2, \ldots, x_N)$, the model is asked to assign high probability to x_2 given x_1, high probability to x_3 given x_1, x_2, and so on for every position. This left-to-right factorization is the autoregressive property:

$$p_\theta(x_1, \ldots, x_N) = \prod_{i=1}^{N} p_\theta(x_i \mid x_{<i}).$$

The better the model's predictions, the higher this probability. Training maximizes it — equivalently, it minimizes the cross-entropy loss, which penalizes the model for assigning low probability to the word that actually appeared:

$$\mathcal{J}(\theta) = -\sum_{i=1}^{N} \log p_\theta(x_i \mid x_{<i}).$$

Why training can be fully parallel. In the recurrent decoder, training was sequential: computing the hidden state at step t required the hidden state at step $t-1$ to already be

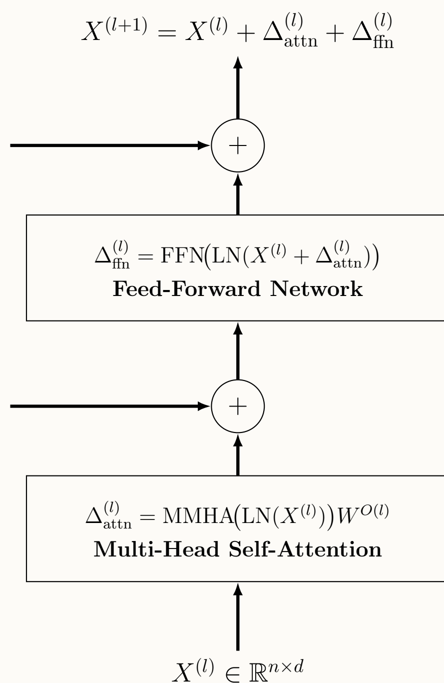

Figure 1.13: One encoder layer of a Transformer. Multi-head self-attention mixes token representations within the input sequence and produces the update $\Delta^{(l)}_{\text{attn}}$. A feed-forward network produces $\Delta^{(l)}_{\text{ffn}}$. Residual connections accumulate these updates to produce $X^{(l+1)}$.

ready. This chain of dependencies serialised the entire forward pass — every position waited for the one before it.

The Transformer breaks this dependency entirely. During training, the full sequence $(x_1, \dots, x_N)$ is fed to the model at once. The causal mask does the work that the sequential chain used to do: it ensures that when computing the representation at position i, attention cannot see any token at position $j > i$. So $x_i^{(L)}$ depends only on $(x_1, \dots, x_{i-1})$ — exactly the prefix it should condition on — but this guarantee comes from a mask applied in parallel, not from running the model step by step.

The result is that $X^{(L)}$ contains all N prefix-conditioned representations at once. Every conditional distribution

$$p_\theta(x_1 \mid \varnothing),\ p_\theta(x_2 \mid x_1),\ \dots,\ p_\theta(x_N \mid x_{<N})$$

is computed in a single forward pass. The loss over the entire sequence is computed and gradients are propagated in one shot. This is what makes training on sequences of thousands of tokens tractable.

Autoregressive inference. At inference time, the situation is different. There is no full sequence to feed in — the model is generating one that does not yet exist. Only a prefix $(x_1, \dots, x_k)$ is available.

The model processes this prefix, and the representation at the final position encodes everything the model knows about what should come next. It computes $p_\theta(x_{k+1} \mid x_{\leq k})$ and selects

a token — by taking the most likely word, or by sampling. That token is appended to the prefix, and the whole process runs again for position $k + 2$.

Unlike training, this is inherently sequential. Each new token depends on the one just generated, so the model must run once per token. The causal mask is still there — it is just that every "future" position is genuinely unknown rather than deliberately hidden.

The full picture. Every component in the Transformer was forced by a concrete problem. Token embeddings convert discrete symbols into vectors. Positional encodings restore the notion of order that recurrence previously provided. Masked self-attention replaces the recurrent target history without requiring sequential processing. Cross-attention retrieves information from the source sequence. The feed-forward network replaces the nonlinear integration once performed by the recurrent update f. Residual connections, borrowed from ResNet, allow these transformations to accumulate across depth without destroying earlier computations. Each component is the answer to a question that arose when the recurrent chain was removed.

The architecture therefore accumulates corrections across depth rather than overwriting state across time. Initial token embeddings enter the first layer and emerge from layer L as rich contextual representations after repeated rounds of communication (attention) and transformation (FFN). Next-token prediction is then a simple linear read-off from that representation stream.

This process of continually transforming representations is reflected in the name **Transformer**. Somewhere in the model's d_{model}-dimensional representation space lie directions and regions corresponding to concepts expressed in language. A token embedding provides only a context-independent starting point in this space. Words like *bank* begin with the same vector regardless of whether the sentence refers to finance or a river.

The role of the Transformer is to transform this initial vector into a contextual representation that reflects the meaning of the word in its specific sentence. Through repeated interactions with surrounding tokens, the representation moves to a region of space that encodes the model's interpretation of the sentence and aligns with directions corresponding to words that are likely to follow.

In this sense, the Transformer does not store meanings for words directly. Instead, it computes meaning dynamically by moving token representations through a high-dimensional space until they reach positions that best explain the sentence and predict what comes next.

decoder only

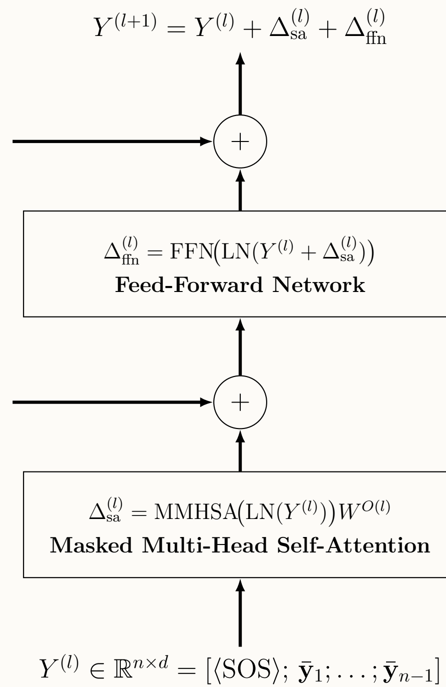

Figure 1.14: One decoder layer of a Transformer used in large language models. Masked multi-head self-attention mixes representations from previous tokens and produces the update $\Delta_{\text{sa}}^{(l)}$. A feed-forward network produces $\Delta_{\text{ffn}}^{(l)}$. Residual connections accumulate these updates to produce $Y^{(l+1)}$.

Worked Example: A Complete Pre-Norm Transformer Block

The preceding sections described the Transformer's components — LayerNorm, multi-head self-attention, residual connections, and the feed-forward network — at the level of equations and diagrams. This section works through a single Pre-Norm block numerically from raw token embeddings to the final output, tracking every matrix computation with concrete numbers.

The Pre-Norm block in one equation. LayerNorm is applied *before* each sub-layer, not after:

$$X_{\text{attn}} = X + \text{MHA}\big(\text{LN}(X)\big), \qquad X_{\text{out}} = X_{\text{attn}} + \text{FFN}\big(\text{LN}(X_{\text{attn}})\big).$$

The raw input X flows unchanged through both skip connections; only the normalized branch enters each sub-layer.

Toy configuration.

Symbol	Meaning	Value
n	sequence length (tokens)	3
d_{model}	embedding dimension	4
h	number of attention heads	2
d_{head}	per-head dimension ($= d_{\text{model}}/h$)	2

Step 1 — Raw input.

$$X = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix} \in \mathbb{R}^{3\times 4}.$$

Rows are tokens w_1, w_2, w_3. X is the *residual stream* — the block only ever adds increments to it; it is never overwritten.

Step 2 — LayerNorm before attention. Applied independently to each row: $\text{LN}(x) = (x - \mu)/\sigma$ where μ, σ are the row mean and standard deviation.
w_1**:** $\mu = 0.5,\ \sigma = 0.5 \ \Rightarrow\ \tilde{x}_1 = [1, -1, 1, -1]$.
w_2**:** $\mu = 0.5,\ \sigma = 0.5 \ \Rightarrow\ \tilde{x}_2 = [-1, 1, 1, -1]$.
w_3**:** $\mu = 0.75,\ \sigma \approx 0.433 \ \Rightarrow\ \tilde{x}_3 \approx [0.58, 0.58, -1.73, 0.58]$.

$$\tilde{X} = \begin{bmatrix} 1 & -1 & 1 & -1 \\ -1 & 1 & 1 & -1 \\ 0.58 & 0.58 & -1.73 & 0.58 \end{bmatrix}.$$

Projections use $\tilde{X}$, skip connections use X. All query/key/value matrices are computed from $\tilde{X}$. The raw X appears *only* in the residual additions below.

Step 3 — Multi-head self-attention. Each head h projects $\tilde{X}$ into 2-dimensional Q/K/V spaces via $W_Q^{(h)}, W_K^{(h)}, W_V^{(h)} \in \mathbb{R}^{4\times 2}$.
Head 1 — projections:

$$W_Q{}^{(1)} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad W_K{}^{(1)} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad W_V{}^{(1)} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}.$$

$$Q^{(1)} = \begin{bmatrix} 2 & -2 \\ 0 & 0 \\ -1.15 & 1.15 \end{bmatrix}, \quad K^{(1)} = \begin{bmatrix} 2 & -1 \\ 0 & 1 \\ -1.15 & -0.58 \end{bmatrix}, \quad V^{(1)} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1.16 & -1.15 \end{bmatrix}.$$

Scores, causal mask ($-\infty$ for $j > i$), and softmax:

$$S^{(1)}_{\text{masked}} = \begin{bmatrix} 6 & -\infty & -\infty \\ 0 & 0 & -\infty \\ -3.45 & 1.15 & 0.66 \end{bmatrix}, \quad A^{(1)} \approx \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 0.5 & 0 \\ 0.006 & 0.61 & 0.38 \end{bmatrix}.$$

Head 2 — projections:

$$W_Q{}^{(2)} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix}, \quad W_K{}^{(2)} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad W_V{}^{(2)} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

$$S^{(2)}_{\text{masked}} = \begin{bmatrix} 0 & -\infty & -\infty \\ -2 & 0 & -\infty \\ 1.15 & -1.15 & -1.33 \end{bmatrix}, \quad A^{(2)} \approx \begin{bmatrix} 1 & 0 & 0 \\ 0.12 & 0.88 & 0 \\ 0.84 & 0.10 & 0.06 \end{bmatrix}.$$

Two heads, two patterns. Head 1 has w_3 attending mostly to w_2 (0.61); Head 2 has w_3 attending mostly to w_1 (0.84). Each head independently extracts a different relational signal from the same sequence — the core motivation for multi-head attention.

Step 4 — Contextual outputs and concatenation. $C^{(h)} = A^{(h)} V^{(h)} \in \mathbb{R}^{3\times 2}$:

$$C^{(1)} \approx \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0.44 & -0.44 \end{bmatrix}, \quad C^{(2)} \approx \begin{bmatrix} 2.00 & -1.00 \\ 0.24 & -0.12 \\ 1.61 & -0.77 \end{bmatrix}.$$

Concatenate heads and apply output projection $W_O = I$:

$$\Delta_{\text{attn}} = [C^{(1)} \mid C^{(2)}] = \begin{bmatrix} 0 & 0 & 2.00 & -1.00 \\ 0 & 0 & 0.24 & -0.12 \\ 0.44 & -0.44 & 1.61 & -0.77 \end{bmatrix} \in \mathbb{R}^{3\times 4}.$$

Step 5 — First residual connection.

$$\begin{aligned} X_{\text{attn}} &= X + \Delta_{\text{attn}} \\ &= \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 2.00 & -1.00 \\ 0 & 0 & 0.24 & -0.12 \\ 0.44 & -0.44 & 1.61 & -0.77 \end{bmatrix} \\ &\approx \begin{bmatrix} 1 & 0 & 3.00 & -1.00 \\ 0 & 1 & 1.24 & -0.12 \\ 1.44 & 0.56 & 1.61 & 0.23 \end{bmatrix} \end{aligned}$$

Step 6 — LayerNorm before FFN.

$$X_{\text{FFN-norm}} = \text{LN}(X_{\text{attn}}) \approx \begin{bmatrix} 0.39 & -0.65 & 1.43 & -1.17 \\ -0.94 & 0.89 & 1.11 & -1.05 \\ 1.68 & -0.15 & -0.69 & -0.84 \end{bmatrix}.$$

Step 7 — Feed-forward network. The FFN $\text{FFN}(x) = W_2\,\sigma(W_1 x + b_1) + b_2$ is applied token-wise to $X_{\text{FFN-norm}}$:

$$\Delta_{\text{FFN}} \approx \begin{bmatrix} 0.22 & -0.08 & 0.31 & -0.06 \\ -0.18 & 0.27 & 0.12 & -0.21 \\ 0.36 & -0.04 & -0.11 & 0.14 \end{bmatrix}.$$

Step 8 — Second residual connection.

$$X_{\text{out}} = X_{\text{attn}} + \Delta_{\text{FFN}} \approx \begin{bmatrix} 1.22 & -0.08 & 3.31 & -1.06 \\ -0.18 & 1.27 & 1.36 & -0.33 \\ 1.80 & 0.52 & 1.50 & 0.37 \end{bmatrix}.$$

The residual stream view. The block is three matrices added together:

$$\boxed{X_{\text{out}} = X + \Delta_{\text{attn}} + \Delta_{\text{FFN}}.}$$

$$\underbrace{\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix}}_{X} + \underbrace{\begin{bmatrix} 0 & 0 & 2.00 & -1.00 \\ 0 & 0 & 0.24 & -0.12 \\ 0.44 & -0.44 & 1.61 & -0.77 \end{bmatrix}}_{\Delta_{\text{attn}}} + \underbrace{\begin{bmatrix} 0.22 & -0.08 & 0.31 & -0.06 \\ -0.18 & 0.27 & 0.12 & -0.21 \\ 0.36 & -0.04 & -0.11 & 0.14 \end{bmatrix}}_{\Delta_{\text{FFN}}} \approx X_{\text{out}}.$$

> **Key takeaway.** A transformer block never *overwrites* representations — it *accumulates* them. Attention writes a context-aware increment Δ_{attn}; the FFN writes a local nonlinear increment Δ_{FFN}. Both gradients flow unimpeded backward through the skip connections, bypassing every nonlinearity — this is why deep transformers are trainable despite their depth.

Encoder–Decoder Transformer for Sequence Transduction

The decoder-only architecture described above is designed for language modelling: given a sequence of tokens, predict the next one. A different class of tasks requires **sequence transduction** — mapping one sequence to another, such as translating English to French. For this, the Transformer adopts an encoder–decoder structure.

The Encoder. The encoder processes the entire source sequence at once. Because the full source is available before decoding begins, there is no need to restrict tokens from attending

to future positions — the causal mask is removed. Each encoder layer applies unmasked self-attention followed by a feed-forward network:

$$X' = X + \mathrm{MHA}(\mathrm{LayerNorm}(X)),$$
$$Y = X' + \mathrm{FFN}(\mathrm{LayerNorm}(X')).$$

After multiple encoder layers, the source sequence is represented as a matrix of contextual vectors:

$$H^{\mathrm{enc}} \in \mathbb{R}^{N \times d_{\mathrm{model}}},$$

where each row is a richly contextualized representation of a source token.

Cross-Attention. The decoder generates the target sequence autoregressively, one token at a time. In addition to masked self-attention over previously generated tokens, each decoder layer contains a **cross-attention** block that connects the decoder to the encoder output.

In cross-attention, queries come from the decoder and keys and values come from the encoder:

$$Q = X^{\mathrm{dec}} W_Q, \qquad K = H^{\mathrm{enc}} W_K, \qquad V = H^{\mathrm{enc}} W_V.$$

Attention is then computed as:

$$\mathrm{CrossAttention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V.$$

No causal mask is applied here — the decoder is free to attend to any encoder position. Each decoder token queries the full source representation and retrieves what is most relevant for predicting the next target token.

Decoder Layer Structure. Each encoder–decoder decoder layer now contains three sub-blocks:

$$X' = X + \mathrm{MaskedMHA}(\mathrm{LayerNorm}(X)),$$
$$X'' = X' + \mathrm{CrossMHA}(\mathrm{LayerNorm}(X'), H^{\mathrm{enc}}),$$
$$Y = X'' + \mathrm{FFN}(\mathrm{LayerNorm}(X'')).$$

The masked self-attention maintains autoregressive generation over the target sequence, cross-attention retrieves relevant source context, and the FFN refines the representation locally.

Summary. In sequence transduction, the encoder builds a full contextual representation of the source without masking, while the decoder generates the target autoregressively, attending to both its own past outputs and the encoder representation through cross-attention. The layer-by-layer refinement described earlier applies to both the encoder and the decoder — the Transformer transforms both sequences into progressively richer representations until prediction becomes easy.

BERT 2018 , Developed by Google in 2018, it is a revolutionary machine learning framework for natural language processing (NLP) that uses a transformer-based, encoder-only architecture to understand the context of words in search queries and text.

Definition 1.20: GPT (Generative Pre-trained Transformer)

GPT is a decoder-only Transformer that uses causal masked self-attention and is trained autoregressively with the objective

$$\max_\theta \sum_t \log p_\theta(x_t \mid x_1, \ldots, x_{t-1}).$$

Advantages over RNNs:

- **Parallel training:** all positions are processed at once, removing the sequential bottleneck during training.
- **Direct long-range access:** any two positions interact through a single attention step, avoiding repeated transformations across time.
- **No compression bottleneck:** instead of storing history in a fixed-size hidden state, attention retrieves relevant information directly.
- **Scalability:** performance improves reliably with more data, parameters, and compute.

Generation remains sequential: token t must be produced before token $t + 1$.

The key conceptual shift from RNNs to attention is a move from *compression* to *retrieval*. An RNN must summarize the entire past into a single hidden state before knowing what future steps will need. Attention postpones this decision: each step directly retrieves whatever past information is most relevant.

Historical Evolution of Ideas Behind the Transformer

The Transformer architecture did not emerge in isolation. It is the result of several lines of research that progressively shaped its components.

- **Recurrent Neural Networks (1990s)** Early sequence models such as RNNs introduced the idea of processing tokens sequentially while accumulating history in a hidden state. This established the need for contextual representations.
- **LSTM and GRU (1997–2014)** Gated recurrent models addressed long-range dependencies and enabled deep stacked sequence models. The practice of stacking multiple layers later reappears in Transformer architectures.
- **Encoder–Decoder Seq2Seq Models (2014)** Neural machine translation introduced the encoder–decoder paradigm, where an input sequence is encoded into representations used by a decoder to generate outputs. The original Transformer preserves this high-level structure.
- **Early Attention Mechanisms (2014–2015)** Attention replaced the fixed-size context vector with dynamically computed weighted combinations of encoder states. This removed the bottleneck of compressing all information into a single vector and directly inspired self-attention.
- **MADE and Autoregressive Masking (2015)** Masked feed-forward networks demonstrated that autoregressive generation can be enforced using structural masking. Transformer decoders adopt the same principle through causal masking in self-attention.

- **Residual Networks (2015)** Skip connections enabled stable training of deep networks. Transformers apply residual connections around both attention and feed-forward blocks.
- **Layer Normalization (2016)** Layer normalization stabilized deep sequence models and became a standard component of Transformer layers.
- **Feed-Forward Attention Models (2016)** Work on attention-based models without recurrence showed that sequence modeling could be parallelized, motivating fully attention-based architectures.
- **Convolutional Seq2Seq Models (2017)** Fully parallel sequence models using convolution demonstrated that recurrence was not strictly necessary, paving the way for the Transformer's non-recurrent design.

Summary. The Transformer can be viewed as a synthesis of earlier ideas: stacked sequence modeling from RNNs, dynamic context via attention, autoregressive masking from feed-forward autoregressive models, residual learning from deep vision networks, and fully parallel computation from convolutional and attention-based sequence models.

1.6.5 Summary: Variable-Length Models

From recurrence to attention. LSTMs and GRUs solved the vanishing gradient problem but introduced a new one: the entire history of a sequence must be compressed into a fixed-size hidden state. This bottleneck is most visible in encoder–decoder models for machine translation, where performance degrades on long sentences because one vector cannot carry everything the decoder needs. Attention was introduced to alleviate this bottleneck: the decoder looks back at all encoder hidden states and forms a weighted sum, placing more weight on positions relevant to the current step. The weights are learned and input-dependent, giving the model soft control over what to retrieve — the same principle that made the LSTM cell state effective.

This idea turns out to be more than a patch. Recurrence was never the goal; it was simply a mechanism for modeling dependencies between positions by passing information forward step by step. Attention provides a different solution: each position can directly access any other, modeling dependencies in a single step rather than propagating them across many. This also removes the sequential constraint inherent in recurrence, allowing all positions to interact in parallel and making training much faster on long sequences.

One issue remains: causal order. During generation, the prediction at position i must depend only on earlier positions. Recurrence enforces this naturally, but attention requires it to be imposed explicitly through masking (similar to the autoregressive masking used in MADE), so that information cannot flow from future to past positions.

The Transformer follows directly from these ideas: replace recurrence with attention as the primary mechanism for modeling dependencies, enforce causal structure through masking, and build the model around parallel, weighted aggregation of information combined with nonlinear transformations.

A Unifying View of Autoregressive Models All of the architectures discussed in this chapter arise from the same objective. Using the chain rule, the joint distribution of a

sequence can be written as

$$p(x_1, \dots, x_N) = \prod_{n=1}^{N} p(x_n \mid x_{<n}).$$

Autoregressive architectures therefore differ primarily in how they construct a representation of the prefix $x_{<n}$ used to predict x_n. NADE accumulates contributions from previous inputs, RNNs repeatedly transform a state representation as new elements arrive, and Transformers compute context dynamically through attention over previous tokens.

Chapter 2

Large Language Models

..

LLMs powerful language cortex– brochas area in brain

1. Foundation → What an LLM is 2. Creation → Pretraining at scale 3. Shaping → Alignment post-training 4. Extension → Adaptation, tools, agents 5. Understanding → Evaluation, reasoning, interpretability

content in cs224n on llms start from there

2.0.1 Mixture of Experts

The Problem: Not Every Input Needs the Same Computation

The attention mechanism we developed asks: given this input, which other positions are relevant? The feedforward layers that follow ask a different question: given this representation, how should it be transformed? In a standard model, every token passes through the same feedforward weights regardless of what the token is, where it came from, or what task is being performed. The model must compress all its capabilities into one set of parameters, and every forward pass pays the full computational cost even when most of that capacity is irrelevant to the input at hand.

This becomes visible at scale. A large language model processes an enormous range of inputs simultaneously: mathematical proofs, legal text, poetry, source code, and sentences in dozens of languages. The knowledge required to handle a differential equation is largely disjoint from the knowledge required to parse a metaphor. Yet both pass through identical weights. The model has no mechanism to route different inputs toward different specialised capabilities — it can only average everything into a single set of parameters and hope the capacity is sufficient.

The Multilingual Case Makes the Problem Concrete

Consider a model trained on both English and French. English and French have different word orders, different morphology, and different ways of encoding grammatical relationships. When the model attends over an English sentence, the projection matrices W_Q, W_K, W_V must produce queries and keys that reflect English syntactic structure. When it attends over a French sentence, the same projection matrices must produce queries and keys that reflect French structure. There is one set of weights trying to be a good projection for two genuinely different geometric problems.

The model partly compensates for this. Token embeddings and hidden states carry language-specific information built up through earlier layers, so the representations entering the attention projections are not identical for French and English tokens — the effective queries and

keys differ even when the weights do not. But the projection matrices themselves are still being pulled in multiple directions. A single W_Q must simultaneously encode what a good English query looks like and what a good French query looks like. As more languages are added, this tension compounds. Each new language makes additional demands on shared parameters that were already serving other purposes. This is sometimes called the *curse of multilinguality*: adding more languages to a fixed-capacity model degrades performance on each individual language, because the shared weights cannot fully serve all of them at once.

The root cause is the same as the dense feedforward problem: everything shares everything, with no mechanism to route different inputs toward the computation they actually need.

The Fix: Learned Routing to Specialised Experts

The solution is to replace the single feedforward layer with a collection of E smaller feedforward networks — **experts** — and a lightweight **router** that examines each token and decides which experts to activate. For any given token, only a small subset $k \ll E$ of experts process it. The rest are bypassed entirely.

Formally, let $f_1, \ldots, f_E$ denote the expert networks and let $r(x) \in \mathbb{R}^E$ denote the router's output for token x. The router produces a score for each expert, selects the top-k experts by score, normalises their weights via softmax, and computes the output as a weighted sum:

$$\text{MoE}(x) = \sum_{i \in \text{top-}k} r_i(x)\, f_i(x).$$

The router is a learned linear layer followed by a softmax — small enough that its cost is negligible compared to the experts themselves. It is not told which expert should handle which language or which domain. It discovers a useful division of labour entirely from the training signal.

What emerges is that different experts tend to specialise. In the multilingual setting, the router learns to activate different experts for different languages — French input follows one computational pathway, English input follows another, code follows a third. Each expert can develop representations tuned to the inputs it sees most, without being pulled toward the demands of other languages. The curse of multilinguality is directly addressed: languages are no longer forced to share every parameter.

Capacity Without Proportional Cost

The key trade-off MoE offers is a decoupling of parameter count from computational cost. A dense model with N active parameters uses all N on every token. A MoE model might have $E \cdot N$ total parameters but activate only N worth on any given token — the same inference cost, with access to a much larger total capacity. The model sees far more parameters over training, but no single forward pass is proportionally more expensive.

This is not free. Several failure modes must be managed:

Expert collapse. Without intervention, the router tends to favour a small number of experts early in training — those experts receive more gradient signal, improve faster, and become even more favoured. The remaining experts are starved and never specialise. The model effectively becomes dense again, but with most of its capacity wasted. The standard

fix is an auxiliary load-balancing loss that penalises uneven expert utilisation, encouraging the router to distribute tokens more evenly.

Communication cost. In distributed training, different experts typically live on different devices. Routing a token to an expert on another device requires communication across the network. This all-to-all communication can become a bottleneck at scale, partially offsetting the computational savings.

Routing instability. The router and the experts co-evolve during training. A change in the router changes which tokens each expert sees, which changes what each expert learns, which changes what the router should do. This feedback loop can be unstable, and stabilising it requires careful initialisation and training schedules.

MoE and Multi-Head Attention Are Complementary

It is worth being precise about how MoE relates to multi-head attention, since both involve multiple parallel components.

Multi-head attention runs all heads on the same input, with all heads always active. The specialisation is in *what relationship each head tracks* within a single input — one head for syntactic agreement, another for coreference, another for positional proximity. Every token sees every head, every time.

MoE routes different inputs to different experts, with only a subset of experts active for any token. The specialisation is in *what kind of input each expert handles* — different languages, different domains, different styles of reasoning. Different tokens see different experts.

The two mechanisms decompose the problem along orthogonal axes. Multi-head attention handles the diversity of *relationships within an input*. MoE handles the diversity of *inputs themselves*. In practice, large models use both: multi-head attention in every layer for relational reasoning, with MoE replacing the dense feedforward layers to give different inputs access to different computational pathways.

positional encoder- rope new attention variants

Chapter 3

Latent Variable Models

variational inference and vae

In the previous chapter we built autoregressive models: generative models that decompose the joint distribution over observed variables as a product of conditionals,

$$p(\mathbf{x}) = \prod_{i=1}^{d} p(x_i \mid x_1, \dots, x_{i-1}). \tag{3.1}$$

These models are elegant. Each conditional is easy to train by maximum likelihood, the chain rule guarantees that (3.1) is a valid joint distribution for any choice of conditionals, and with a neural network parameterising each factor the model is highly expressive. For language modelling in particular, autoregressive models have proven extraordinarily powerful.

Yet working with them reveals three persistent difficulties, and noticing them together points toward a different class of models entirely.

Problem 1: sampling is slow. To draw a single sample from an autoregressive model, we must evaluate the network d times in sequence—once per dimension, each conditioned on all previously sampled values. For a 256×256 image this means 65,536 sequential forward passes. There is no way to parallelise them: the value of x_i is not known until x_{i-1} has been sampled. This is not an implementation detail. It is a structural consequence of the factorisation (3.1): the model generates data one piece at a time, with no compressed summary of what kind of thing it is generating.

Problem 2: there is no representation. An autoregressive model has no internal code that summarises an input. Given an image $\mathbf{x}$, there is nowhere to look to find out what the image is *about*—no compact vector that captures its identity, style, or content. This matters for any task downstream of generation: interpolating between two images, transferring style, doing retrieval, or feeding a compressed representation to a downstream classifier. Autoregressive models simply have no mechanism for this. They model $p(\mathbf{x})$ without ever forming an internal representation of $\mathbf{x}$.

Problem 3: inference is one-directional. Given a trained autoregressive model, we can evaluate $p(\mathbf{x})$ for any $\mathbf{x}$ and we can sample new $\mathbf{x}$. But we cannot answer the question: *given this observation, what caused it?* There is no encoder, no inverse map, no way to ask what underlying factors explain a given datapoint. The model knows how to generate data but not how to reason about data it has already seen.

The deeper issue: the wrong causal structure. These three problems are not unrelated. They all trace back to the same modelling assumption, which is worth examining carefully.

When we write $p(\mathbf{x}) = \prod_i p(x_i \mid x_1, \ldots, x_{i-1})$, we are modelling each observed dimension as if it were *caused by the preceding observed dimensions*. In a sentence, the next word is caused by the words before it. This works well for language: words really are sequentially dependent in a way that respects the left-to-right order.

But consider an image of a face. The pixel in the top-left corner is not *caused by* having no pixels before it; it is caused by the person's bone structure, the lighting in the room, and the camera angle. The pixel in the top-right corner is not caused by the top-left pixel; it is caused by the same underlying factors. The observed pixels are not causes of each other. They are *joint effects of shared hidden causes.*

This is a different generative story:

1. Some unobserved variable $\mathbf{z}$—the hidden cause—is drawn from a prior: $\mathbf{z} \sim p(\mathbf{z})$.
2. All the observed variables $\mathbf{x}$ are generated jointly from $\mathbf{z}$: $\mathbf{x} \sim p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z})$.

A model with this structure—a *latent variable model*—directly addresses all three problems above. Because $\mathbf{z}$ is a compact summary of what is being generated, sampling can in principle be done in two steps regardless of the dimensionality of $\mathbf{x}$: first sample $\mathbf{z}$, then decode. Because $\mathbf{z}$ is a representation of $\mathbf{x}$, we have an internal code we can use for downstream tasks. And because the model has an explicit generative structure, we can ask about the posterior $p(\mathbf{z} \mid \mathbf{x})$—what hidden causes are consistent with this observation?—and use it for reasoning and inference.

The question that drives the rest of this chapter is: how do we learn such a model? The latent variables are never observed, so we cannot train on them directly. The marginal likelihood $p_{\boldsymbol{\theta}}(\mathbf{x}) = \int p(\mathbf{z})\, p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z})\, d\mathbf{z}$ requires integrating over all possible hidden causes—an integral that turns out to be intractable for the expressive models we need. Each section of this chapter is a principled response to that difficulty.

We begin with *shallow latent variable models*—factor analysis, Gaussian mixtures, and hidden Markov models—where the decoder is simple enough that the integral can be solved exactly. These models are tractable and interpretable, but they hit a hard capacity wall: a linear or discrete decoder cannot capture the structure of images, audio, or text. Replacing the decoder with a neural network breaks the tractability. *Variational inference* restores it approximately. *Black-box variational inference* makes the approximation runnable. The *reparameterisation trick* makes it efficient. *Amortised inference* makes it scalable. The result is the *variational autoencoder* (VAE)—the canonical deep latent variable model. We close with *normalizing flows*, which show that the approximation can be avoided entirely if we are willing to accept a different architectural constraint, and with a contrast between the two philosophies that sharpens intuition about both.

3.1 Representation: Directed Latent Variable Models

We begin by formalising the generative story sketched above. The unobserved variable $\mathbf{z}$ is drawn from a prior, and then the observation $\mathbf{x}$ is produced by a process that depends on $\mathbf{z}$. Writing this down as a probability model is straightforward. The harder questions—how to *learn* such a model from data, and how to *invert* it to recover $\mathbf{z}$ from a given $\mathbf{x}$—are what the

rest of the chapter addresses. Both surface a shared computational obstacle, and resolving it will require a sequence of principled refinements.

3.1.1 The Graphical Model

Consider a *directed, latent variable model* over two sets of random variables: a latent variable $\mathbf{z}$ and an observed variable $\mathbf{x}$ (see Figure 3.1).

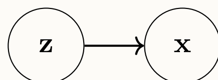

Figure 3.1: Graphical model for a directed, latent variable model.

The joint distribution factorises as

$$p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}) \;=\; p_{\boldsymbol{\theta}}(\mathbf{z})\, p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z}). \tag{3.2}$$

This factorisation encodes a *generative process*:

1. Sample the latent variable: $\mathbf{z} \sim p_{\boldsymbol{\theta}}(\mathbf{z})$.
2. Sample the observation: $\mathbf{x} \sim p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z})$.

If one adopts the view that the latent variables $\mathbf{z}$ encode semantically meaningful information about an observation $\mathbf{x}$, the generative process can be interpreted as first producing high-level semantic structure before generating the observable data itself.

This perspective motivates richer latent-variable models. For example, hierarchical generative models introduce multiple layers of latent variables,

$$p(\mathbf{x}, \mathbf{z}_1, \ldots, \mathbf{z}_m) = p(\mathbf{x} \mid \mathbf{z}_1) \prod_{i=1}^{m-1} p(\mathbf{z}_i \mid \mathbf{z}_{i+1})\, p(\mathbf{z}_m),$$

where increasingly abstract information about $\mathbf{x}$ is generated at higher levels and progressively refined through successive layers. Other structures arise in temporal models such as Hidden Markov Models, where latent states evolve over time before producing observations.

3.1.2 Hypothesis Class

Before we can learn or do inference, we need to be precise about what family of models we are considering. Choosing this family is the first design decision of the modelling process.

To formalise this modelling framework, we consider a family of distributions over latent variables, denoted $\mathcal{P}_z$, where each $p(\mathbf{z}) \in \mathcal{P}_z$ specifies a prior distribution over $\mathbf{z}$. We also consider a family of conditional distributions $\mathcal{P}_{x|z}$, where each $p_\theta(\mathbf{x} \mid \mathbf{z}) \in \mathcal{P}_{x|z}$ defines a distribution over $\mathbf{x}$ conditioned on $\mathbf{z}$.

The resulting hypothesis class of generative models is

$$\mathcal{P}_{x,z} = \left\{ p(\mathbf{x}, \mathbf{z}) \mid p(\mathbf{z}) \in \mathcal{P}_z,\ p_\theta(\mathbf{x} \mid \mathbf{z}) \in \mathcal{P}_{x|z} \right\}.$$

Given a dataset $\mathcal{D} = \{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}$ of i.i.d. samples, two questions immediately arise—and answering them will occupy the rest of this chapter:

- **Learning:** how do we select a model $p(\mathbf{x}, \mathbf{z}) \in \mathcal{P}_{x,z}$ that best explains the observed data, when the latent variables are never observed?
- **Posterior inference:** given a trained model $p(\mathbf{x}, \mathbf{z})$ and a new observation $\mathbf{x}$, how do we compute—or approximate—the distribution over the latent variables that produced it,

$$p(\mathbf{z} \mid \mathbf{x})\,?$$

Both problems are easy to state and hard to solve. The next section takes on the first: learning $\boldsymbol{\theta}$ from data. It will immediately surface a wall.

3.2 Shallow Latent Variable Models

Before confronting the general intractability problem, it is worth seeing what LVMs can do when the model is simple enough that learning and inference are tractable in closed form. These *shallow* models—so called because the mapping from $\mathbf{z}$ to $\mathbf{x}$ is a single, simple function—are the historical starting point of the field.

3.2.1 Factor Analysis and Probabilistic PCA

Consider a dataset of d-dimensional observations that, despite their high dimensionality, seem to cluster near a low-dimensional surface. Gene expression data, for instance, may have tens of thousands of measured genes per sample, yet most variation can be explained by a handful of biological processes. Factor analysis makes this intuition precise: it posits that each observation is generated by a small number of unobserved *factors* $\mathbf{z} \in \mathbb{R}^k$ ($k \ll d$) mixed together by a linear map, plus observation noise.

The generative model is

$$p(\mathbf{z}) = \mathcal{N}(\mathbf{z};\, \mathbf{0},\, \mathbf{I}), \tag{3.3}$$

$$p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z}) = \mathcal{N}(\mathbf{x};\, \mathbf{W}\mathbf{z} + \boldsymbol{\mu},\, \boldsymbol{\Psi}), \tag{3.4}$$

where $\mathbf{W} \in \mathbb{R}^{d \times k}$ is the *factor loading matrix*, $\boldsymbol{\mu} \in \mathbb{R}^d$ is a mean offset, and $\boldsymbol{\Psi} = \operatorname{diag}(\psi_1, \ldots, \psi_d)$ is a diagonal noise covariance. The parameters are $\boldsymbol{\theta} = \{\mathbf{W}, \boldsymbol{\mu}, \boldsymbol{\Psi}\}$.

Because both the prior and the likelihood are Gaussian, and the likelihood is *linear* in $\mathbf{z}$, the model is conjugate. Every quantity of interest has a closed form.

Marginal distribution. Integrating out $\mathbf{z}$ gives

$$p_{\boldsymbol{\theta}}(\mathbf{x}) = \mathcal{N}(\mathbf{x};\, \boldsymbol{\mu},\, \mathbf{W}\mathbf{W}^\top + \boldsymbol{\Psi}). \tag{3.5}$$

The $d \times d$ covariance $\mathbf{W}\mathbf{W}^\top + \boldsymbol{\Psi}$ decomposes into a low-rank signal term and a diagonal noise term—a structure that makes it both interpretable and efficient to compute with.

Posterior distribution. Given an observation $\mathbf{x}$, the posterior over the latent factors is also Gaussian:

$$p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x}) = \mathcal{N}\Big(\mathbf{z};\, \mathbf{M}\mathbf{W}^\top\boldsymbol{\Psi}^{-1}(\mathbf{x} - \boldsymbol{\mu}),\, \mathbf{M}\Big), \tag{3.6}$$

where $\mathbf{M} = \big(\mathbf{I} + \mathbf{W}^\top\boldsymbol{\Psi}^{-1}\mathbf{W}\big)^{-1}$ is a $k \times k$ matrix. Computing this requires only a $k \times k$ inversion—cheap when k is small.

Learning via EM. Because the posterior is tractable, EM can be applied in closed form.

- **E-step.** For each datapoint $\mathbf{x}^{(n)}$, compute the posterior mean $\mathbb{E}[\mathbf{z}^{(n)} \mid \mathbf{x}^{(n)}]$ and second moment $\mathbb{E}[\mathbf{z}^{(n)}(\mathbf{z}^{(n)})^\top \mid \mathbf{x}^{(n)}]$ using (3.6).
- **M-step.** Update $\mathbf{W}$, $\boldsymbol{\mu}$, and $\boldsymbol{\Psi}$ by maximising the expected complete-data log-likelihood; all updates have closed-form solutions involving the posterior moments.

Each EM iteration is guaranteed to increase the marginal log-likelihood $\sum_n \log p_{\boldsymbol{\theta}}(\mathbf{x}^{(n)})$.

Probabilistic PCA. A special case arises when the noise covariance is isotropic: $\boldsymbol{\Psi} = \sigma^2\mathbf{I}$. This is *probabilistic PCA* (PPCA). As $\sigma^2 \to 0$, the posterior mean $\mathbb{E}[\mathbf{z} \mid \mathbf{x}]$ converges to the standard PCA projection, and the columns of $\mathbf{W}$ converge to the leading principal components. PPCA thus provides a probabilistic interpretation of PCA, complete with a likelihood function and a principled way to handle missing data.

Limitation: the linear ceiling. The elegance of factor analysis comes entirely from the linear decoder $\mathbf{x} = \mathbf{W}\mathbf{z} + \boldsymbol{\mu} + \boldsymbol{\varepsilon}$. The marginal (3.5) is always Gaussian, regardless of the data. Real observations—images, audio, text—live on highly nonlinear, non-Gaussian manifolds. Reconstructing a face from its factors produces a weighted sum of *eigenfaces*: a blurry, globally averaged image that cannot represent sharp edges, cast shadows, or fine texture. To capture such structure, the decoder must become nonlinear. But as soon as it does, the conjugacy that made everything tractable breaks down.

3.2.2 Gaussian Mixture Models

Factor analysis assumes a single connected component: the data distribution is unimodal. Many real datasets are not. A collection of handwritten digits, for instance, contains ten visually distinct clusters that no single Gaussian can span. The remedy is to let the latent variable $\mathbf{z}$ be *discrete*—a cluster assignment—so that each observation is explained by one of K separate Gaussian components.

The generative model is

$$p(\mathbf{z} = k) = \pi_k, \qquad \sum_{k=1}^{K} \pi_k = 1, \quad \pi_k \geq 0, \tag{3.7}$$

$$p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z} = k) = \mathcal{N}(\mathbf{x};\ \boldsymbol{\mu}_k,\ \boldsymbol{\Sigma}_k), \tag{3.8}$$

with parameters $\boldsymbol{\theta} = \{\pi_k, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\}_{k=1}^K$. The marginal distribution is a *Gaussian mixture*,

$$p_{\boldsymbol{\theta}}(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x};\ \boldsymbol{\mu}_k,\ \boldsymbol{\Sigma}_k). \tag{3.9}$$

Posterior: responsibilities. The posterior $p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x})$ is the distribution over cluster assignments given the observation. By Bayes' rule,

$$r_k(\mathbf{x}) \ \equiv \ p_{\boldsymbol{\theta}}(\mathbf{z} = k \mid \mathbf{x}) = \frac{\pi_k \mathcal{N}(\mathbf{x};\ \boldsymbol{\mu}_k,\ \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x};\ \boldsymbol{\mu}_j,\ \boldsymbol{\Sigma}_j)}. \tag{3.10}$$

The value $r_k(\mathbf{x})$ is called the *responsibility* of component k for datapoint $\mathbf{x}$. The denominator is just $p_{\boldsymbol{\theta}}(\mathbf{x})$, a finite sum over K terms—tractable for any reasonable K.

Learning via EM. The EM algorithm for a GMM alternates between two closed-form steps.

- **E-step.** For every datapoint $\mathbf{x}^{(n)}$ and every component k, compute the responsibility $r_{nk} = r_k(\mathbf{x}^{(n)})$ using (3.10). Let $N_k = \sum_n r_{nk}$ denote the effective number of points assigned to component k.
- **M-step.** Update the parameters by maximising the expected complete-data log-likelihood:

$$\pi_k^{\text{new}} = \frac{N_k}{N}, \tag{3.11}$$

$$\boldsymbol{\mu}_k^{\text{new}} = \frac{1}{N_k} \sum_{n=1}^{N} r_{nk}\, \mathbf{x}^{(n)}, \tag{3.12}$$

$$\boldsymbol{\Sigma}_k^{\text{new}} = \frac{1}{N_k} \sum_{n=1}^{N} r_{nk} \left(\mathbf{x}^{(n)} - \boldsymbol{\mu}_k^{\text{new}}\right) \left(\mathbf{x}^{(n)} - \boldsymbol{\mu}_k^{\text{new}}\right)^{\top}. \tag{3.13}$$

  Each formula is a weighted empirical mean or covariance, with the responsibilities as weights.

Each full E–M cycle is guaranteed to increase the marginal log-likelihood $\sum_n \log p_{\boldsymbol{\theta}}(\mathbf{x}^{(n)})$.

EM as exact coordinate ascent on the ELBO. This point is worth pausing on because it directly foreshadows the general variational framework. The ELBO for a single datapoint is

$$\mathcal{L}(\boldsymbol{\theta}, q; \mathbf{x}) = \mathbb{E}_{q(\mathbf{z})}\left[\log \frac{p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})}{q(\mathbf{z})}\right] = \log p_{\boldsymbol{\theta}}(\mathbf{x}) - D_{\mathrm{KL}}(q(\mathbf{z}) \,\|\, p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x})). \tag{3.14}$$

The E-step of EM sets $q(\mathbf{z}) = p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x})$, which drives $D_{\mathrm{KL}}(q \,\|\, p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}))$ to exactly zero. The ELBO then equals the marginal log-likelihood. The M-step maximises this tight bound over $\boldsymbol{\theta}$. EM is therefore coordinate ascent on the ELBO in the special case where the E-step can be solved exactly. When the posterior is intractable—as it will be once we use a neural-network decoder—this exact E-step is no longer available, and we will need to settle for an *approximate* one. That approximation is variational inference.

Limitation: finite discrete structure. Mixture models are more flexible than factor analysis—they can represent multimodal distributions—but they impose their own constraints. The number of components K must be chosen in advance. Each component is still a simple Gaussian. Most fundamentally, the latent variable is *discrete*: two datapoints that belong to the same component are modelled as exchangeable, with no notion of a continuous latent space between them. A model of faces should be able to interpolate smoothly between identities, ages, and expressions. A mixture model cannot.

3.2.3 Hidden Markov Models

Factor analysis and Gaussian mixture models both treat datapoints as independently and identically distributed. Much real data is sequential: a phoneme in speech depends on the preceding phoneme; a word in a sentence depends on its context; a frame of video depends

on the frames before it. The *hidden Markov model* (HMM) extends the LVM framework to sequences by letting the latent variable $\mathbf{z}_t$ evolve over time according to a Markov chain.

The generative model for a sequence $(\mathbf{x}_1, \ldots, \mathbf{x}_T)$ with corresponding hidden states $(\mathbf{z}_1, \ldots, \mathbf{z}_T)$, each taking values in $\{1, \ldots, K\}$, is

$$p(\mathbf{z}_1) = \text{Categorical}(\boldsymbol{\pi}_0), \tag{3.15}$$

$$p(\mathbf{z}_t \mid \mathbf{z}_{t-1}) = \text{Categorical}(\mathbf{A}_{\mathbf{z}_{t-1}}), \tag{3.16}$$

$$p_{\boldsymbol{\theta}}(\mathbf{x}_t \mid \mathbf{z}_t) = p(\mathbf{x}_t;\ \boldsymbol{\phi}_{\mathbf{z}_t}), \tag{3.17}$$

where $\boldsymbol{\pi}_0$ is the initial state distribution, $\mathbf{A} \in \mathbb{R}^{K \times K}$ is the *transition matrix* with $A_{jk} = p(\mathbf{z}_t = k \mid \mathbf{z}_{t-1} = j)$, and $p(\mathbf{x}_t;\ \boldsymbol{\phi}_k)$ is the *emission distribution* for state k (often a Gaussian or categorical). The joint distribution factorises as

$$p_{\boldsymbol{\theta}}(\mathbf{x}_{1:T}, \mathbf{z}_{1:T}) = p(\mathbf{z}_1) \prod_{t=2}^{T} p(\mathbf{z}_t \mid \mathbf{z}_{t-1}) \prod_{t=1}^{T} p_{\boldsymbol{\theta}}(\mathbf{x}_t \mid \mathbf{z}_t). \tag{3.18}$$

Inference via dynamic programming. Naively computing the posterior $p(\mathbf{z}_{1:T} \mid \mathbf{x}_{1:T})$ requires summing over all K^T state sequences—exponential in T. Dynamic programming reduces this to $\mathcal{O}(TK^2)$ via the *forward–backward algorithm.*

Define the *forward variable* $\alpha_t(k) = p(\mathbf{x}_{1:t}, \mathbf{z}_t = k)$ and the *backward variable* $\beta_t(k) = p(\mathbf{x}_{t+1:T} \mid \mathbf{z}_t = k)$. These satisfy the recursions

$$\alpha_t(k) = p(\mathbf{x}_t \mid \mathbf{z}_t = k) \sum_{j=1}^{K} A_{jk}\, \alpha_{t-1}(j), \tag{3.19}$$

$$\beta_t(k) = \sum_{j=1}^{K} A_{kj}\, p(\mathbf{x}_{t+1} \mid \mathbf{z}_{t+1} = j)\, \beta_{t+1}(j). \tag{3.20}$$

The marginal posterior at time t is then

$$\gamma_t(k) \equiv p(\mathbf{z}_t = k \mid \mathbf{x}_{1:T}) \ \propto\ \alpha_t(k)\, \beta_t(k), \tag{3.21}$$

and the pairwise posterior $\xi_t(j,k) = p(\mathbf{z}_{t-1} = j, \mathbf{z}_t = k \mid \mathbf{x}_{1:T})$ is obtained similarly. The marginal likelihood $p(\mathbf{x}_{1:T})$ is recovered from the forward variables as $\sum_k \alpha_T(k)$.

Learning via Baum–Welch. The Baum–Welch algorithm is EM for HMMs. The E-step runs the forward–backward algorithm to compute $\gamma_t(k)$ and $\xi_t(j,k)$ for all t, j, k. The M-step updates the parameters in closed form:

$$\pi_{0,k}^{\text{new}} = \gamma_1(k), \tag{3.22}$$

$$A_{jk}^{\text{new}} = \frac{\sum_{t=2}^{T} \xi_t(j,k)}{\sum_{t=2}^{T} \gamma_{t-1}(j)}, \tag{3.23}$$

$$\boldsymbol{\phi}_k^{\text{new}} = \arg\max_{\boldsymbol{\phi}} \sum_{t=1}^{T} \gamma_t(k)\ \log p(\mathbf{x}_t;\ \boldsymbol{\phi}). \tag{3.24}$$

For Gaussian emissions the last step also has a closed form.

Limitation: discrete states and simple emissions. The HMM extends LVMs to sequences, but the latent state remains discrete. A video sequence in $\mathbb{R}^d$ with $d = 10^6$ (one megapixel per frame) cannot be described by a small discrete alphabet of states: the number of states needed to tile the observation space grows exponentially in d. Furthermore, the emission distribution $p(\mathbf{x}_t \mid \mathbf{z}_t)$ is still simple—usually a Gaussian or a categorical. Modelling complex, structured observations (images, waveforms, natural language) within a single Gaussian emission is hopeless for the same reason as in factor analysis. We again hit the same wall.

3.2.4 The Capacity Wall

Looking across the three models above, a common structure emerges. In every case, tractability rests on two pillars: the prior $p(\mathbf{z})$ is simple (isotropic Gaussian, categorical, or Markov chain), and the decoder $p(\mathbf{x} \mid \mathbf{z})$ is simple (linear Gaussian, Gaussian component, or categorical emission). Together, these choices make the posterior $p(\mathbf{z} \mid \mathbf{x})$ available in closed form, which is what allows EM to run exactly.

The same simplicity that buys tractability imposes a hard limit on what the model can represent.

- In factor analysis, the marginal $p_{\boldsymbol{\theta}}(\mathbf{x})$ is always a single Gaussian regardless of $\boldsymbol{\theta}$. Real image distributions are highly non-Gaussian: heavy-tailed, multimodal, and concentrated near complex nonlinear manifolds. No choice of $\mathbf{W}$ and $\boldsymbol{\Psi}$ can fix this.
- In GMMs, $p_{\boldsymbol{\theta}}(\mathbf{x})$ is a mixture of Gaussians. In principle, any continuous distribution can be approximated arbitrarily well by a mixture of Gaussians—but the required number of components grows exponentially in the data dimension d. For even modest images ($d = 784$ for MNIST) this is completely impractical.
- In HMMs, the latent variable is discrete, so interpolation between states is not meaningful. A continuous latent space—essential for smooth generation and structured representation—cannot be represented at all.

The obvious fix. The decoder $p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z})$ is the bottleneck. Replace it with a deep neural network $f_{\boldsymbol{\theta}} : \mathbb{R}^k \to \mathbb{R}^d$:

$$p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z}) = p(\mathbf{x};\ f_{\boldsymbol{\theta}}(\mathbf{z})), \tag{3.25}$$

where $f_{\boldsymbol{\theta}}(\mathbf{z})$ outputs the parameters of an observation distribution (e.g. the mean of a Gaussian, or the logits of a Bernoulli). With a sufficiently deep network, $f_{\boldsymbol{\theta}}$ is a universal approximator: in principle, the model can capture any continuous distribution over $\mathbf{x}$.

The new problem. This fix immediately destroys the property that made shallow LVMs tractable. With a nonlinear decoder, the posterior

$$p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x}) \ \propto\ p(\mathbf{z})\, p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z}) \tag{3.26}$$

no longer has a closed form: $p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z})$ is no longer a Gaussian in $\mathbf{z}$, so the prior and likelihood are no longer conjugate. The EM E-step—which required evaluating this posterior exactly—fails. We cannot compute responsibilities, forward-backward variables, or posterior moments.

We are at a decision point. Shallow models are tractable but inexpressive. Deep models are expressive but intractable. The rest of this chapter is the story of how to recover tractability

without sacrificing expressiveness. The key idea—variational inference—is to replace the intractable exact posterior with a tractable *approximation*, and to turn the quality of that approximation into an optimisation objective.

3.3 Learning Directed Latent Variable Models

3.3.1 Maximum Marginal Likelihood

The natural objective for learning a generative model is to maximise how well the model explains the observed data. Since the latent variables $\mathbf{z}$ are never observed, we must integrate them out: the model's fit to a datapoint $\mathbf{x}$ is its *marginal likelihood* $p_{\boldsymbol{\theta}}(\mathbf{x})$, obtained by averaging over all possible latent configurations that could have produced it.

One principled way to measure how well $p_{\boldsymbol{\theta}}$ fits the data is to minimise the Kullback–Leibler (KL) divergence between the empirical data distribution $q_{\mathcal{D}}(\mathbf{x})$ and the model's marginal distribution $p_{\boldsymbol{\theta}}(\mathbf{x})$:

$$\min_{\boldsymbol{\theta}} \; D_{\mathrm{KL}}(q_{\mathcal{D}}(\mathbf{x}) \,\|\, p_{\boldsymbol{\theta}}(\mathbf{x}))\,. \tag{3.27}$$

As we have seen in earlier chapters, minimising this KL divergence is equivalent to maximising the *marginal log-likelihood* over the dataset:

$$\max_{\boldsymbol{\theta}} \sum_{\mathbf{x}\in\mathcal{D}} \log p_{\boldsymbol{\theta}}(\mathbf{x}), \qquad p_{\boldsymbol{\theta}}(\mathbf{x}) = \int p_{\boldsymbol{\theta}}(\mathbf{z})\, p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z})\, d\mathbf{z}. \tag{3.28}$$

3.3.2 Intractability

There is an immediate obstacle. The integral in (3.28) requires summing the likelihood $p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z})$ over *every possible latent configuration*—an exponentially large or continuous space. For the expressive models we care about, this is computationally intractable. A naive Monte Carlo estimate,

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}) \;\approx\; \log \frac{1}{K} \sum_{k=1}^{K} p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z}^{(k)}), \qquad \mathbf{z}^{(k)} \sim p_{\boldsymbol{\theta}}(\mathbf{z}), \tag{3.29}$$

is possible in principle, but in practice the gradient estimates of this estimator exhibit very high variance, making direct optimisation unreliable.

Furthermore, evaluating $p_{\boldsymbol{\theta}}(\mathbf{x})$ is at least as hard as evaluating the posterior, since by Bayes' rule

$$p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x}) = \frac{p_{\boldsymbol{\theta}}(\mathbf{z})\, p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z})}{p_{\boldsymbol{\theta}}(\mathbf{x})}, \tag{3.30}$$

so intractability of the marginal immediately implies intractability of the posterior. We are stuck: we cannot optimise the model without evaluating the marginal, and we cannot evaluate the marginal without solving an intractable integral. We need a different strategy.

3.4 Variational Inference and the ELBO

The strategy is to *turn the inference problem into an optimisation problem.* Instead of computing the true posterior $p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x})$ exactly, we ask: what is the best approximation to it within some tractable family of distributions? And can we optimise that approximation in a

way that also drives the model parameters $\boldsymbol{\theta}$ in the right direction? If yes, intractability is no longer a dead end—it becomes a design choice.

3.4.1 Variational Families

We need a family of distributions over $\mathbf{z}$ that is simple enough to optimise, yet flexible enough to be a useful stand-in for the true posterior. We call such a family a *variational family*

$$\mathcal{Q} = \{q_{\boldsymbol{\lambda}}(\mathbf{z} \mid \mathbf{x})\}.$$

In classical variational inference, a separate variational distribution is introduced for each datapoint. For a datapoint $\mathbf{x}_i$, the approximate posterior is written

$$q_{\boldsymbol{\lambda}_i}(\mathbf{z} \mid \mathbf{x}_i),$$

where $\boldsymbol{\lambda}_i$ denotes the variational parameters associated with that datapoint.
Each member of the variational family therefore approximates the intractable posterior $p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x}_i)$ and is specified by tunable variational parameters $\boldsymbol{\lambda}_i$.
For example, if the variational family is Gaussian, the parameters $\boldsymbol{\lambda}_i$ may consist of the mean and covariance defining the distribution $q_{\boldsymbol{\lambda}_i}(\mathbf{z} \mid \mathbf{x}_i)$.
The question then becomes: how do we choose $\boldsymbol{\lambda}_i$ to make $q_{\boldsymbol{\lambda}_i}(\mathbf{z} \mid \mathbf{x}_i)$ as close as possible to the true posterior $p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x}_i)$, and what does "close" mean in this context?
The answer to "close" is KL divergence. Before deriving the ELBO formally, it is helpful to build intuition for why KL divergence is the right measure of distance between distributions. The following information-theoretic quantities make this natural.

3.4.2 Information-Theoretic Bridge to the ELBO

Surprise

Surprise measures how *unexpected* a particular outcome is.

$$\text{Surprise}(x) = -\log P(x)$$

- If $P(x)$ is **small**, the event is **surprising** (large value).
- If $P(x)$ is **large**, the event is **unsurprising** (small value).

Entropy (Expected Surprise)

Entropy is the *average surprise* of a random variable under its own distribution P:

$$H(P) = \mathbb{E}_{x\sim P}[-\log P(x)]$$

For discrete variables this becomes

$$H(P) = -\sum_x P(x)\log P(x)$$

- It measures the **uncertainty** of a distribution.
- High entropy $\Rightarrow$ unpredictable distribution.
- Low entropy $\Rightarrow$ predictable distribution.

Cross-Entropy

Now suppose the *true* distribution is P, but our model predicts Q.

Cross-entropy is the expected surprise when the world follows P but we evaluate outcomes using the model Q:

$$H(P, Q) = \mathbb{E}_{x \sim P}[-\log Q(x)]$$

For discrete variables,

$$H(P, Q) = -\sum_{x} P(x) \log Q(x)$$

Intuitively, it answers the question:

"If reality follows P, how surprised am I if I believe Q?"

KL Divergence

Even if our model were perfect, we would still experience some surprise due to the inherent randomness of the world. That unavoidable surprise is the entropy $H(P)$.

However, if our model Q does not match the true distribution P, the total surprise increases to the cross-entropy $H(P, Q)$.

The **extra, avoidable surprise** caused by using the wrong model is the **Kullback–Leibler divergence**:

$$D_{\mathrm{KL}}(P \,\|\, Q) = H(P, Q) - H(P)$$

It can also be written directly as

$$D_{\mathrm{KL}}(P \,\|\, Q) = \sum_{x} P(x) \log \frac{P(x)}{Q(x)}$$

KL divergence measures how different two probability distributions are. It quantifies the additional surprise incurred when the model Q is used to approximate the true distribution P.

In our setting, P is the intractable true posterior $p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x})$ and Q is our variational approximation $q_{\boldsymbol{\lambda}}(\mathbf{z} \mid \mathbf{x})$. The goal of variational inference is to choose the parameters $\boldsymbol{\lambda}$ so that the approximate posterior is as close as possible to the true posterior.

Here $\boldsymbol{\lambda}$ denotes the parameters of the variational distribution. For example, if the variational family is Gaussian, then $\boldsymbol{\lambda}$ may consist of the mean and covariance parameters that define $q_{\boldsymbol{\lambda}}(\mathbf{z} \mid \mathbf{x})$.

Minimising $D_{\mathrm{KL}}(q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x}) \,\|\, p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}))$ is therefore exactly the right objective. The remaining question is: can we minimise this KL divergence without ever evaluating the intractable $p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x})$? The answer is yes, and the ELBO provides the mechanism.

3.4.3 Derivation of the Evidence Lower Bound

We begin from the marginal likelihood of a datapoint $\mathbf{x}$:

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}) = \log \int p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})\, d\mathbf{z}$$

Since the posterior $p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$ is generally intractable, we introduce a variational distribution $q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})$. Multiplying and dividing inside the integral by $q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})$ does not change the expression:

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}) = \log \int q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x}) \frac{p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})}{q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})}\, d\mathbf{z}$$

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}) = \log \mathbb{E}_{q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})} \left[\frac{p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})}{q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})} \right]$$

Applying Jensen's inequality to move the logarithm inside the expectation gives

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}) \geq \mathbb{E}_{q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})} \left[\log \frac{p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})}{q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})} \right]$$

$$p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}) = p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})\, p_{\boldsymbol{\theta}}(\mathbf{z})$$

$$\mathbb{E}_{q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})} \left[\log \frac{p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})\, p_{\boldsymbol{\theta}}(\mathbf{z})}{q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})} \right]$$

$$= \mathbb{E}_{q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})} [\log p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z}) + \log p_{\boldsymbol{\theta}}(\mathbf{z}) - \log q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})]$$

$$= \mathbb{E}_{q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})} \big[\log p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})\big] + \mathbb{E}_{q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})} \big[\log p_{\boldsymbol{\theta}}(\mathbf{z}) - \log q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})\big]$$

$$D_{\mathrm{KL}}(q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x}) \,\|\, p_{\boldsymbol{\theta}}(\mathbf{z})) = \mathbb{E}_{q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})} \left[\log \frac{q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})}{p_{\boldsymbol{\theta}}(\mathbf{z})} \right]$$

$$\mathbb{E}_{q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})} \big[\log p_{\boldsymbol{\theta}}(\mathbf{z}) - \log q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})\big] = -D_{\mathrm{KL}}(q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x}) \,\|\, p_{\boldsymbol{\theta}}(\mathbf{z}))$$

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}) \geq \mathbb{E}_{q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})} \big[\log p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})\big] - D_{\mathrm{KL}}(q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x}) \,\|\, p_{\boldsymbol{\theta}}(\mathbf{z}))$$

The quantity on the right-hand side is called the **Evidence Lower Bound (ELBO)**:

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathbf{x}) = \mathbb{E}_{q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})} \big[\log p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})\big] - D_{\mathrm{KL}}(q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x}) \,\|\, p_{\boldsymbol{\theta}}(\mathbf{z}))$$

Notice what happened: the intractable posterior $p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x})$ has disappeared from the objective. We are left with two quantities we can evaluate: the reconstruction likelihood and the KL divergence to the prior, both computable when $q_{\boldsymbol{\lambda}}$ and $p_{\boldsymbol{\theta}}$ are simple distributions. Maximising the ELBO pushes the model to explain the data well while keeping the approximate posterior close to the prior. But how tight is this bound as a surrogate for the true log-likelihood?

3.4.4 Tightness of the Bound

The Jensen-based derivation establishes the bound but does not tell us how loose it is, or when we can trust it. An exact decomposition makes this transparent. Since $\log p_{\boldsymbol{\theta}}(\mathbf{x})$ does not depend on $\mathbf{z}$, we have

$$\begin{aligned}
\log p_{\boldsymbol{\theta}}(\mathbf{x}) &= \mathbb{E}_{q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})}[\log p_{\boldsymbol{\theta}}(\mathbf{x})] && (3.31)\\
&= \mathbb{E}_{q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})}\left[\log \frac{p_{\boldsymbol{\theta}}(\mathbf{x},\mathbf{z})}{p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})}\right] && (3.32)\\
&= \mathbb{E}_{q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})}\left[\log \frac{p_{\boldsymbol{\theta}}(\mathbf{x},\mathbf{z})}{q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})}\cdot\frac{q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})}{p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})}\right] && (3.33)\\
&= \underbrace{\mathbb{E}_{q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})}\left[\log \frac{p_{\boldsymbol{\theta}}(\mathbf{x},\mathbf{z})}{q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})}\right]}_{=\mathcal{L}(\boldsymbol{\theta},\boldsymbol{\lambda};\mathbf{x})} + \underbrace{\mathbb{E}_{q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})}\left[\log \frac{q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})}{p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})}\right]}_{=D_{\mathrm{KL}}(q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})\,\|\,p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}))}. && (3.34)
\end{aligned}$$

This equality reveals an important interpretation of the KL divergence $D_{\mathrm{KL}}(q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})\,\|\,p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}))$. Since the KL divergence is always non-negative, it immediately follows that

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}) \geq \mathcal{L}(\boldsymbol{\theta},\boldsymbol{\lambda};\mathbf{x}),$$

recovering the lower bound derived earlier.

More importantly, the KL divergence plays two roles simultaneously. First, by definition it measures how different the variational distribution $q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})$ is from the true posterior $p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$. Second, it exactly quantifies the gap between the ELBO and the marginal log-likelihood:

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}) - \mathcal{L}(\boldsymbol{\theta},\boldsymbol{\lambda};\mathbf{x}) = D_{\mathrm{KL}}(q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})\,\|\,p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}))\,.$$

The closer $q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})$ is to the true posterior, the smaller this divergence becomes and the tighter the bound. In the ideal case where $q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x}) = p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$, the KL divergence vanishes and the ELBO becomes equal to $\log p_{\boldsymbol{\theta}}(\mathbf{x})$.

Starting from the definition of the ELBO

$$\mathcal{L}(\boldsymbol{\theta},\boldsymbol{\lambda};\mathbf{x}) = \log p_{\boldsymbol{\theta}}(\mathbf{x}) - \mathbb{E}_{q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})}\left[\log \frac{q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})}{p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})}\right],$$

we expand the logarithm,

$$= \log p_{\boldsymbol{\theta}}(\mathbf{x}) - \mathbb{E}_{q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})}\left[\log q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x}) - \log p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})\right].$$

Now apply Bayes' rule,

$$p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}) = \frac{p_{\boldsymbol{\theta}}(\mathbf{x},\mathbf{z})}{p_{\boldsymbol{\theta}}(\mathbf{x})},$$

so that

$$\log p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}) = \log p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}) - \log p_{\boldsymbol{\theta}}(\mathbf{x}).$$

Substituting this expression gives

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathbf{x}) = \log p_{\boldsymbol{\theta}}(\mathbf{x}) - \mathbb{E}_{q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})}\left[\log q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x}) - \log p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}) + \log p_{\boldsymbol{\theta}}(\mathbf{x})\right].$$

The terms involving $\log p_{\boldsymbol{\theta}}(\mathbf{x})$ cancel, leaving

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathbf{x}) = \mathbb{E}_{q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})}\left[\log p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}) - \log q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})\right].$$

This is the alternative form of the ELBO that we derived earlier.

So optimising the ELBO jointly over $\boldsymbol{\theta}$ and $\boldsymbol{\lambda}$ does two things at once: it improves the generative model, and it reduces the approximation gap. This is the theoretical foundation. The practical question is how to actually compute the gradients and run the optimisation.

3.4.5 Learning via ELBO Maximisation

The ELBO formulation above motivates the following joint optimisation

$$\max_{\boldsymbol{\theta}, \boldsymbol{\lambda}} \sum_{\mathbf{x} \in \mathcal{D}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\lambda};\, \mathbf{x}). \tag{3.35}$$

Maximising over $\boldsymbol{\lambda}$ tightens the bound by bringing $q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})$ closer to the true posterior; maximising over $\boldsymbol{\theta}$ improves the generative model itself.

The ELBO admits a tractable unbiased Monte Carlo estimator

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\lambda};\, \mathbf{x}) \;\approx\; \frac{1}{K} \sum_{k=1}^{K} \log \frac{p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}^{(k)})}{q_{\boldsymbol{\lambda}}(\mathbf{z}^{(k)}|\mathbf{x})}, \qquad \mathbf{z}^{(k)} \sim q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x}), \tag{3.36}$$

so long as it is easy to sample from $q_{\boldsymbol{\lambda}}$ and evaluate its density. The ELBO is now computable. The next question is how to differentiate through it with respect to $\boldsymbol{\lambda}$, since the sampling distribution itself depends on $\boldsymbol{\lambda}$.

3.4.6 The ELBO as Two KL Divergences

The Jensen-based derivation of the ELBO in Section 3.4.3 shows that the bound exists and is tight when $q = p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x})$. But it does not fully explain *why* the ELBO is the right objective to maximise. A second derivation, which works at the level of the full dataset rather than a single datapoint, makes this clear. It also reveals that maximising the ELBO simultaneously achieves two goals that might seem independent.

Setup. The ultimate learning objective is to fit the model distribution $p_{\boldsymbol{\theta}}(\mathbf{x})$ to the empirical data distribution $q_{\mathcal{D}}(\mathbf{x})$. As established in Section 3.3.1, this is equivalent to maximising the marginal log-likelihood, or equivalently to minimising

$$D_{\mathrm{KL}}(q_{\mathcal{D}}(\mathbf{x}) \,\|\, p_{\boldsymbol{\theta}}(\mathbf{x})). \tag{3.37}$$

This KL involves $p_{\boldsymbol{\theta}}(\mathbf{x}) = \int p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})\, d\mathbf{z}$, which is intractable. We cannot optimise (3.37) directly.

Now introduce a variational distribution $q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})$ and define the *joint variational distribution* over both observed and latent variables:

$$q_{\boldsymbol{\phi}}(\mathbf{x}, \mathbf{z}) \;=\; q_{\mathcal{D}}(\mathbf{x})\, q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x}). \tag{3.38}$$

This is a distribution we can evaluate and sample from: sample $\mathbf{x}$ from the dataset, then sample $\mathbf{z}$ from the encoder.

The decomposition. Compute the KL divergence between this joint variational distribution and the model's joint distribution $p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})$:

$$\begin{aligned} D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(\mathbf{x}, \mathbf{z}) \,\|\, p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})) &= \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{x}, \mathbf{z})}\left[\log \frac{q_{\boldsymbol{\phi}}(\mathbf{x}, \mathbf{z})}{p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})}\right] \\ &= \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{x}, \mathbf{z})}\left[\log \frac{q_{\mathcal{D}}(\mathbf{x})\, q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})}{p_{\boldsymbol{\theta}}(\mathbf{x})\, p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x})}\right] \\ &= D_{\mathrm{KL}}(q_{\mathcal{D}}(\mathbf{x}) \,\|\, p_{\boldsymbol{\theta}}(\mathbf{x})) + \mathbb{E}_{q_{\mathcal{D}}(\mathbf{x})}[D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x}) \,\|\, p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x}))]\,. \end{aligned} \tag{3.39}$$

Since $D_{\mathrm{KL}}(\cdot \,\|\, \cdot) \geq 0$, every term in (3.39) is non-negative. Rearranging and using the fact that $D_{\mathrm{KL}}(q_{\mathcal{D}} \,\|\, p_{\boldsymbol{\theta}})$ differs from $-\sum_{\mathbf{x}} \log p_{\boldsymbol{\theta}}(\mathbf{x})$ only by the constant data entropy, we obtain

$$-\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}) \;=\; D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(\mathbf{x}, \mathbf{z}) \,\|\, p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}))\,. \tag{3.40}$$

Two goals, one objective. Equation (3.39) decomposes the negative ELBO into two KL divergences:

$$-\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}) = \underbrace{D_{\mathrm{KL}}(q_{\mathcal{D}}(\mathbf{x}) \,\|\, p_{\boldsymbol{\theta}}(\mathbf{x}))}_{\substack{\text{model fit:} \\ \text{how well does } p_\theta \text{ explain the data?}}} + \underbrace{\mathbb{E}_{\mathbf{x}}[D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) \,\|\, p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}))]}_{\substack{\text{inference quality:} \\ \text{how close is } q_\phi \text{ to the true posterior?}}}\,. \tag{3.41}$$

Maximising the ELBO over both $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ simultaneously *minimises the model fit KL* (making $p_{\boldsymbol{\theta}}$ match the data) and *minimises the inference KL* (making $q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})$ match the true posterior). These are exactly the two goals we set out with—the learning problem and the posterior inference problem—and both are served by a single objective.

This is the deeper reason the ELBO is the right thing to optimise. It is not merely a lower bound that happened to be convenient; it is precisely the quantity whose minimisation aligns the model with the data and the approximate posterior with the truth.

Remark 3.1. *When the variational family is rich enough that $q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x}) = p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x})$ for all $\mathbf{x}$, the inference KL vanishes and maximising the ELBO becomes equivalent to maximum marginal likelihood. EM on a GMM is exactly this case: the E-step sets $q = p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x})$ exactly, collapsing the bound.*

3.5 Black-Box Variational Inference

We have a tractable objective. Now we need a scalable algorithm to optimise it. *Black-Box Variational Inference* (BBVI) provides this: a general-purpose EM-style procedure that

requires only the ability to sample from $q_{\boldsymbol{\lambda}}$ and evaluate log-probabilities. It brings us close to a fully practical algorithm—but it leaves two problems unsolved that will force us to go further.

3.5.1 The BBVI Algorithm

Black-Box Variational Inference (BBVI) is a general-purpose Expectation–Maximisation-like algorithm for optimising the ELBO with first-order stochastic gradient methods. For each mini-batch $\mathcal{M} \subseteq \mathcal{D}$, BBVI performs two alternating steps.

Step 1: Variational update (E-step). For each datapoint $\mathbf{x} \in \mathcal{M}$, optimise the variational parameters by iteratively applying

$$\boldsymbol{\lambda} \leftarrow \boldsymbol{\lambda} + \eta \, \widetilde{\nabla}_{\boldsymbol{\lambda}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathbf{x}), \tag{3.42}$$

where η is a step size and $\widetilde{\nabla}_{\boldsymbol{\lambda}}$ denotes an unbiased estimator of the ELBO gradient. This step seeks the best approximation to $p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x})$ within the variational family.

Step 2: Model update (M-step). Perform a single gradient step on the model parameters using the mini-batch:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \, \widetilde{\nabla}_{\boldsymbol{\theta}} \sum_{\mathbf{x} \in \mathcal{M}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathbf{x}). \tag{3.43}$$

3.5.2 Gradient Estimation

Gradients with respect to $\boldsymbol{\theta}$ are straightforward to estimate via Monte Carlo by pushing $\nabla_{\boldsymbol{\theta}}$ through the expectation operator. The gradient with respect to $\boldsymbol{\lambda}$ is more subtle because the expectation is taken under $q_{\boldsymbol{\lambda}}(\mathbf{z})$, which itself depends on $\boldsymbol{\lambda}$. The *log-derivative trick* (REINFORCE) gives

$$\nabla_{\boldsymbol{\lambda}} \mathbb{E}_{q_{\boldsymbol{\lambda}}(\mathbf{z})}\big[f(\mathbf{z})\big] = \mathbb{E}_{q_{\boldsymbol{\lambda}}(\mathbf{z})}\Big[f(\mathbf{z}) \, \nabla_{\boldsymbol{\lambda}} \log q_{\boldsymbol{\lambda}}(\mathbf{z})\Big]. \tag{3.44}$$

Although unbiased, this estimator is well-known to suffer from high variance in practice.

3.5.3 REINFORCE Trick proof (Discrete z)

$$\nabla_\lambda \mathbb{E}_{z \sim q_\lambda(z)}\left[\log \frac{p_\theta(x,z)}{q_\lambda(z)}\right] = \nabla_\lambda \int q_\lambda(z) \log \frac{p_\theta(x,z)}{q_\lambda(z)} \, dz$$

$$= \nabla_\lambda \int q_\lambda(z) \big(\log p_\theta(x,z) - \log q_\lambda(z)\big) dz$$

$$= \nabla_\lambda \int q_\lambda(z) \log p_\theta(x,z) \, dz - \nabla_\lambda \int q_\lambda(z) \log q_\lambda(z) \, dz$$

$$= A - B$$

Term A

$$A = \nabla_\lambda \int q_\lambda(z) \log p_\theta(x, z)\, dz$$

$$= \int \nabla_\lambda \big(q_\lambda(z) \log p_\theta(x, z)\big) dz$$

$$= \int \left(\nabla_\lambda q_\lambda(z) \log p_\theta(x, z) + q_\lambda(z) \nabla_\lambda \log p_\theta(x, z)\right) dz$$

Since $p_\theta(x, z)$ does not depend on λ,

$$\nabla_\lambda \log p_\theta(x, z) = 0$$

$$= \int \nabla_\lambda q_\lambda(z) \log p_\theta(x, z)\, dz$$

Using the identity

$$\nabla_\lambda q_\lambda(z) = q_\lambda(z) \nabla_\lambda \log q_\lambda(z)$$

$$= \int q_\lambda(z) \nabla_\lambda \log q_\lambda(z) \log p_\theta(x, z)\, dz$$

$$= \mathbb{E}_{z \sim q_\lambda(z)} \left[\nabla_\lambda \log q_\lambda(z) \log p_\theta(x, z)\right]$$

Term B

$$B = \nabla_\lambda \int q_\lambda(z) \log q_\lambda(z)\, dz$$

$$= \int \nabla_\lambda \big(q_\lambda(z) \log q_\lambda(z)\big) dz$$

$$= \int \left(\nabla_\lambda q_\lambda(z) \log q_\lambda(z) + q_\lambda(z) \nabla_\lambda \log q_\lambda(z)\right) dz$$

$$\nabla_\lambda q_\lambda(z) = q_\lambda(z) \nabla_\lambda \log q_\lambda(z)$$

$$= \int q_\lambda(z) \nabla_\lambda \log q_\lambda(z) \log q_\lambda(z)\, dz + \int \nabla_\lambda q_\lambda(z)\, dz$$

$$\int \nabla_\lambda q_\lambda(z)\, dz = \nabla_\lambda \int q_\lambda(z)\, dz = \nabla_\lambda 1 = 0$$

Therefore,

$$B = \int q_\lambda(z) \nabla_\lambda \log q_\lambda(z) \log q_\lambda(z)\, dz$$

$$= \mathbb{E}_{z \sim q_\lambda(z)} \left[\nabla_\lambda \log q_\lambda(z) \log q_\lambda(z)\right]$$

Combining $A - B$

$$A - B = \mathbb{E}_{z\sim q_\lambda(z)}\left[\nabla_\lambda \log q_\lambda(z) \log p_\theta(x,z)\right] - \mathbb{E}_{z\sim q_\lambda(z)}\left[\nabla_\lambda \log q_\lambda(z) \log q_\lambda(z)\right]$$

$$= \mathbb{E}_{z\sim q_\lambda(z)}\left[\nabla_\lambda \log q_\lambda(z) \left(\log p_\theta(x,z) - \log q_\lambda(z)\right)\right]$$

$$= \mathbb{E}_{z\sim q_\lambda(z)}\left[\nabla_\lambda \log q_\lambda(z) \log \frac{p_\theta(x,z)}{q_\lambda(z)}\right]$$

Monte Carlo Estimator

$$\approx \frac{1}{K}\sum_{i=1}^{K} \log \frac{p_\theta(x,z^{(i)})}{q_\lambda(z^{(i)})}\, \nabla_\lambda \log q_\lambda(z^{(i)}) \quad z^{(i)} \sim q_\lambda(z)$$

BBVI works, but it has a practical weakness: the REINFORCE gradient estimator is unbiased yet notoriously high-variance, requiring many samples or sophisticated control variates to converge reliably. For high-dimensional latent spaces this is a serious bottleneck. We need a fundamentally different way to differentiate through a sampling step.

3.6 The Reparameterisation Trick

The root cause of REINFORCE's high variance is that the sampling operation $\mathbf{z} \sim q_{\boldsymbol{\lambda}}(\mathbf{z})$ sits *inside* the expectation, blocking the gradient from flowing through it directly. The reparameterisation trick eliminates this blockage by separating the randomness from the parameters.

3.6.1 Change of Variables

Introduce a fixed auxiliary distribution $p(\boldsymbol{\varepsilon})$ and a differentiable, invertible function $g_{\boldsymbol{\lambda}}$ such that

$$\mathbf{z} \;=\; g_{\boldsymbol{\lambda}}(\boldsymbol{\varepsilon},\, \mathbf{x}), \qquad \boldsymbol{\varepsilon} \sim p(\boldsymbol{\varepsilon}) \tag{3.45}$$

is equivalent in distribution to $\mathbf{z} \sim q_{\boldsymbol{\lambda}}(\mathbf{z} \mid \mathbf{x})$.

3.6.2 Low-Variance Gradient Estimator

By the Law of the Unconscious Statistician, the expectation under $q_{\boldsymbol{\lambda}}(\mathbf{z} \mid \mathbf{x})$ can be rewritten as an expectation under $p(\boldsymbol{\varepsilon})$:

$$\mathbb{E}_{q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})}\big[f(\mathbf{z})\big] \;=\; \mathbb{E}_{p(\boldsymbol{\varepsilon})}\big[f(g_{\boldsymbol{\lambda}}(\boldsymbol{\varepsilon},\mathbf{x}))\big]. \tag{3.46}$$

Because $p(\boldsymbol{\varepsilon})$ does not depend on $\boldsymbol{\lambda}$, we may now interchange the gradient and expectation operators:

$$\nabla_{\boldsymbol{\lambda}}\, \mathbb{E}_{q_{\boldsymbol{\lambda}}(\mathbf{z}|\mathbf{x})}\big[f(\mathbf{z})\big] \;=\; \mathbb{E}_{p(\boldsymbol{\varepsilon})}\big[\nabla_{\boldsymbol{\lambda}}\, f(g_{\boldsymbol{\lambda}}(\boldsymbol{\varepsilon},\mathbf{x}))\big] \;\approx\; \nabla_{\boldsymbol{\lambda}}\, f(g_{\boldsymbol{\lambda}}(\boldsymbol{\varepsilon},\mathbf{x})), \quad \boldsymbol{\varepsilon} \sim p(\boldsymbol{\varepsilon}). \tag{3.47}$$

Crucially, $g_{\boldsymbol{\lambda}}$ is differentiable, so the gradient can be computed via standard backpropagation. Empirically this estimator exhibits much lower variance than (3.44).

Remark 3.2. *For a diagonal Gaussian* $q_{\boldsymbol{\lambda}}(\mathbf{z} \mid \mathbf{x}) = \mathcal{N}(\mathbf{z};\, \boldsymbol{\mu},\, \mathrm{diag}(\boldsymbol{\sigma}^2))$ *with* $\boldsymbol{\lambda} = (\boldsymbol{\mu}, \boldsymbol{\sigma})$*, the reparameterisation is*

$$\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\varepsilon}, \qquad \boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \tag{3.48}$$

where $\odot$ *denotes element-wise multiplication.*

We now have a low-variance, backpropagation-compatible gradient estimator for the ELBO. But we have not yet committed to a specific form for the two distributions in the model: the likelihood $p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z})$ and the approximate posterior $q_{\boldsymbol{\lambda}}(\mathbf{z} \mid \mathbf{x})$. Choosing these well is what turns the abstract framework into a concrete, scalable model.

3.7 Deep Latent Variable Models

The shallow LVMs of Section 3.2 achieved tractability by restricting the decoder $p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z})$ to be a simple, fixed-form distribution: linear Gaussian in factor analysis, a mixture component in GMMs, a categorical in HMMs. This restriction is precisely what made the posterior conjugate and EM exact. It is also, as we saw, what limits these models to distributions far simpler than the images, audio, and text we actually want to model.

The defining move of a *deep latent variable model* is to parameterise the decoder with a neural network. Instead of $p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z}) = \mathcal{N}(\mathbf{x};\, \mathbf{W}\mathbf{z} + \boldsymbol{\mu}, \boldsymbol{\Psi})$, we write

$$p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z}) = p(\mathbf{x};\, f_{\boldsymbol{\theta}}(\mathbf{z}))\,, \tag{3.49}$$

where $f_{\boldsymbol{\theta}}$ is a deep neural network that maps a latent code $\mathbf{z}$ to the parameters of the observation distribution. For continuous observations, $f_{\boldsymbol{\theta}}(\mathbf{z})$ might output the mean of a Gaussian; for binary images, the logits of a Bernoulli. Because neural networks are universal approximators, this single change—swapping the linear map for a neural network—gives the model the capacity to represent essentially any data distribution.

The prior $p(\mathbf{z})$ is usually kept simple. An isotropic Gaussian $p(\mathbf{z}) = \mathcal{N}(\mathbf{0}, \mathbf{I})$ is the standard choice: it is easy to sample from, has no learnable parameters, and encourages the encoder (defined shortly) to learn a structured latent space rather than one that memorises the training data.

With this change, the intractability discussed in Section 3.3.2 is no longer hypothetical—it is the regime we are operating in. The posterior $p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x}) \propto p(\mathbf{z})\, p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z})$ has no closed form because $p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z})$ is not conjugate to $p(\mathbf{z})$ when $f_{\boldsymbol{\theta}}$ is nonlinear. The sections that follow show how variational inference, combined with the reparameterisation trick and amortised inference, restores a practical and scalable learning algorithm.

3.8 Parameterising Distributions with Neural Networks

Now that we can differentiate through a sampling step, we are free to make the distributions in our model as expressive as we like—as long as they remain differentiable. Neural networks are the natural choice. This section specifies the decoder and encoder that parameterise the generative model and its variational approximation.

3.8.1 Prior Distribution

A common choice for the prior over the latent variable is the isotropic Gaussian:

$$p(\mathbf{z}) \;=\; \mathcal{N}(\mathbf{z};\, \mathbf{0},\, \mathbf{I}). \tag{3.50}$$

Because this is a fixed distribution, it has no learnable parameters. An alternative sometimes used in practice is a mixture of Gaussians with trainable mean and covariance parameters.

3.8.2 Decoder: Parameterising $p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z})$

The conditional distribution is constructed by choosing a distribution family over $\mathbf{x}$ and specifying a *mapping function* $f_{\boldsymbol{\theta}}$ from the latent space to the parameters of that distribution. In other words,

$$p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z}) \;=\; p(\mathbf{x} \mid f_{\boldsymbol{\theta}}(\mathbf{z}))\,. \tag{3.51}$$

The function $f_{\boldsymbol{\theta}}$ is called the *decoding function* (or *decoder*) since it maps a latent code $\mathbf{z}$ to the parameters of a distribution over observed variables. In practice, $f_{\boldsymbol{\theta}}$ is parameterised by a deep neural network.

For a Gaussian observation model one writes

$$p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z}) \;=\; \mathcal{N}(\mathbf{x};\; \boldsymbol{\mu}_{\boldsymbol{\theta}}(\mathbf{z}),\; \boldsymbol{\Sigma}_{\boldsymbol{\theta}}(\mathbf{z}))\,, \tag{3.52}$$

where $\boldsymbol{\mu}_{\boldsymbol{\theta}}$ and $\boldsymbol{\Sigma}_{\boldsymbol{\theta}}$ are neural networks that map $\mathbf{z}$ to the mean and covariance of the Gaussian.

3.8.3 Encoder: Parameterising the Variational Family

The variational distribution must be chosen so that the reparameterisation trick is applicable. Many continuous distributions in the location-scale family admit such a reparameterisation. In practice, a popular choice is a factorised Gaussian:

$$q_{\boldsymbol{\lambda}}(\mathbf{z} \mid \mathbf{x}) \;=\; \mathcal{N}(\mathbf{z};\; \boldsymbol{\mu}_{\boldsymbol{\lambda}}(\mathbf{x}),\; \boldsymbol{\Sigma}_{\boldsymbol{\lambda}}(\mathbf{x}))\,, \tag{3.53}$$

where the covariance $\boldsymbol{\Sigma}_{\boldsymbol{\lambda}}(\mathbf{x})$ is typically restricted to a diagonal matrix.

Notice the notation: $\boldsymbol{\mu}_{\boldsymbol{\lambda}}(\mathbf{x})$ appears to be a function of $\mathbf{x}$, parameterised by $\boldsymbol{\lambda}$. But so far we have said nothing about *what kind of function* $\boldsymbol{\lambda}$ defines. In classical variational inference, $\boldsymbol{\lambda}$ is not a function at all—it is just a pair of numbers $(\boldsymbol{\mu}, \boldsymbol{\sigma})$ stored independently for each datapoint. Recognising this is the key to the final step.

3.9 Amortised Variational Inference

Pause and count the parameters. With a million training images, BBVI with local variational parameters stores a separate $(\boldsymbol{\mu}_i, \boldsymbol{\sigma}_i)$ for every image—and re-optimises each one every time the model parameters $\boldsymbol{\theta}$ change. This inner loop is a second major bottleneck. There is a cleaner solution: instead of re-solving inference from scratch for every datapoint, *learn* the mapping from observations to variational parameters once, and reuse it everywhere.

3.9.1 Classical vs. Amortised Inference: The Key Distinction

To understand what amortised inference changes, it is worth being precise about what classical variational inference does. When we write $q_{\boldsymbol{\lambda}}(\mathbf{z} \mid \mathbf{x})$, the variational parameters $\boldsymbol{\lambda}$ are *free parameters that are optimised directly*—they have no functional dependence on $\mathbf{x}$ other than through the optimisation. For a dataset of N datapoints $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}\}$, classical VI therefore introduces a *separate* set of local parameters for every datapoint:

$$q_{\boldsymbol{\lambda}_i}(\mathbf{z}_i \mid \mathbf{x}^{(i)}) \;=\; \mathcal{N}\big(\mathbf{z}_i;\; \boldsymbol{\mu}_i,\, \mathrm{diag}(\boldsymbol{\sigma}_i^2)\big), \qquad i = 1, \ldots, N. \tag{3.54}$$

The overall optimisation problem is then

$$\max_{\boldsymbol{\theta},\, \{\boldsymbol{\lambda}_i\}_{i=1}^N} \sum_{i=1}^{N} \mathcal{L}\Big(\boldsymbol{\theta}, \boldsymbol{\lambda}_i;\, \mathbf{x}^{(i)}\Big)\,, \tag{3.55}$$

where $\boldsymbol{\mu}_i$ and $\boldsymbol{\sigma}_i$ are *directly optimised numbers*—not outputs of any function of $\mathbf{x}^{(i)}$. For a dataset of one million points, this means storing and optimising one million pairs of variational parameters. These are called *local variational parameters*.

Amortised inference replaces this per-datapoint parameterisation entirely. Rather than assigning a separate $(\boldsymbol{\mu}_i, \boldsymbol{\sigma}_i)$ to each datapoint and optimising each pair in isolation, we instead *learn a function* that predicts the appropriate variational parameters from the input:

$$\boldsymbol{\mu}_i \;\approx\; \boldsymbol{\mu}_{\boldsymbol{\phi}}(\mathbf{x}^{(i)}), \qquad \boldsymbol{\sigma}_i \;\approx\; \boldsymbol{\sigma}_{\boldsymbol{\phi}}(\mathbf{x}^{(i)}), \tag{3.56}$$

where $\boldsymbol{\mu}_{\boldsymbol{\phi}}$ and $\boldsymbol{\sigma}_{\boldsymbol{\phi}}$ are neural networks with shared weights $\boldsymbol{\phi}$. The encoder becomes a learned inference rule rather than a lookup table: a single function that generalises across all datapoints.

Approach	**How $\boldsymbol{\mu}_i, \boldsymbol{\sigma}_i$ are obtained**	**Parameters**
Classical VI	Directly optimised per datapoint	N local parameter pairs
Amortised VI	Predicted by a shared encoder network	$\boldsymbol{\phi}$ (network weights)

3.9.2 Learning the Inference Mapping

The key observation is that the per-datapoint optimisation in BBVI is solving the same kind of problem every time: given some $\mathbf{x}$, find the Gaussian over $\mathbf{z}$ that best explains it. If this mapping from observations to variational parameters were smooth and consistent—which we expect for natural data—then it should be *learnable*. We need not solve it from scratch for each new $\mathbf{x}$; we can train a function that does it in a single forward pass.

Concretely, define the optimal local parameters for a datapoint as

$$\boldsymbol{\lambda}^*(\mathbf{x}) \;=\; \arg\max_{\boldsymbol{\lambda}} \; \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\lambda};\, \mathbf{x}). \tag{3.57}$$

Rather than computing this optimisation at every training step, we introduce an *encoding function* $f_{\boldsymbol{\phi}} : \mathcal{X} \to \Lambda$ (the space of variational parameters), parameterised by $\boldsymbol{\phi}$, and train it on the objective

$$\max_{\boldsymbol{\phi}} \sum_{\mathbf{x} \in \mathcal{D}} \mathcal{L}(\boldsymbol{\theta},\, f_{\boldsymbol{\phi}}(\mathbf{x});\, \mathbf{x})\,. \tag{3.58}$$

Interpreting $f_{\boldsymbol{\phi}}$ as defining the conditional distribution $q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})$, we can rewrite the combined optimisation as

$$\max_{\boldsymbol{\theta},\boldsymbol{\phi}} \sum_{\mathbf{x}\in\mathcal{D}} \mathcal{L}(\boldsymbol{\theta},\boldsymbol{\phi};\,\mathbf{x}), \qquad \mathcal{L}(\boldsymbol{\theta},\boldsymbol{\phi};\,\mathbf{x}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\log \frac{p_{\boldsymbol{\theta}}(\mathbf{x},\mathbf{z})}{q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})}\right]. \tag{3.59}$$

Rather than running Step 1 as a subroutine, we instead interleave the updates of $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$: for each mini-batch $\mathcal{M}$, we perform the single joint update

$$\boldsymbol{\phi} \leftarrow \boldsymbol{\phi} + \eta\,\widetilde{\nabla}_{\boldsymbol{\phi}} \sum_{\mathbf{x}\in\mathcal{M}} \mathcal{L}(\boldsymbol{\theta},\boldsymbol{\phi};\,\mathbf{x}), \tag{3.60}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta\,\widetilde{\nabla}_{\boldsymbol{\theta}} \sum_{\mathbf{x}\in\mathcal{M}} \mathcal{L}(\boldsymbol{\theta},\boldsymbol{\phi};\,\mathbf{x}). \tag{3.61}$$

By sharing a single set of parameters $\boldsymbol{\phi}$ across all datapoints, this procedure *amortises* the cost of inference over the entire dataset. The inference cost is paid once, in learning $\boldsymbol{\phi}$, rather than repeatedly, for every datapoint. With this final piece in place, we have everything we need.

3.10 Variational Autoencoders

The general framework developed in the preceding sections applies to any directed latent variable model with any tractable variational family. One particularly influential instantiation is obtained by choosing deep neural networks to parameterise both the generative model and the inference network—the result is the **variational autoencoder** (VAE). This section shows concretely what the framework looks like when all its components are specified.

3.10.1 Architecture

When $f_{\boldsymbol{\phi}}$ is a deep neural network—the natural choice given the expressive power we need—the resulting model is the **Variational Autoencoder** (VAE), introduced by Kingma and Welling (2014) and Rezende *et al.* (2014).

The VAE consists of two coupled neural networks:

- **Encoder** (inference model / recognition network): a neural network that takes $\mathbf{x}$ as input and outputs the parameters of the approximate posterior,

$$(\boldsymbol{\mu}_{\boldsymbol{\phi}}(\mathbf{x}),\, \log \boldsymbol{\sigma}_{\boldsymbol{\phi}}(\mathbf{x})) \;=\; \mathrm{EncoderNet}_{\boldsymbol{\phi}}(\mathbf{x}), \qquad q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x}) = \mathcal{N}\big(\mathbf{z};\, \boldsymbol{\mu}_{\boldsymbol{\phi}}(\mathbf{x}),\, \mathrm{diag}(\boldsymbol{\sigma}^2_{\boldsymbol{\phi}}(\mathbf{x}))\big). \tag{3.62}$$

- **Decoder** (generative model): a neural network that maps a latent sample $\mathbf{z}$ back to the parameters of the distribution over $\mathbf{x}$,

$$p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z}) \;=\; p(\mathbf{x};\, \mathrm{DecoderNet}_{\boldsymbol{\theta}}(\mathbf{z})). \tag{3.63}$$

3.10.2 Training Objective

Substituting the Gaussian encoder (3.62) into (3.59), the per-datapoint ELBO becomes

$$\mathcal{L}(\boldsymbol{\theta},\boldsymbol{\phi};\,\mathbf{x}) \;=\; \underbrace{\mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\big[\log p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z})\big]}_{\text{reconstruction term}} \;-\; \underbrace{D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x}) \,\|\, p(\mathbf{z}))}_{\text{regularisation term}}. \tag{3.64}$$

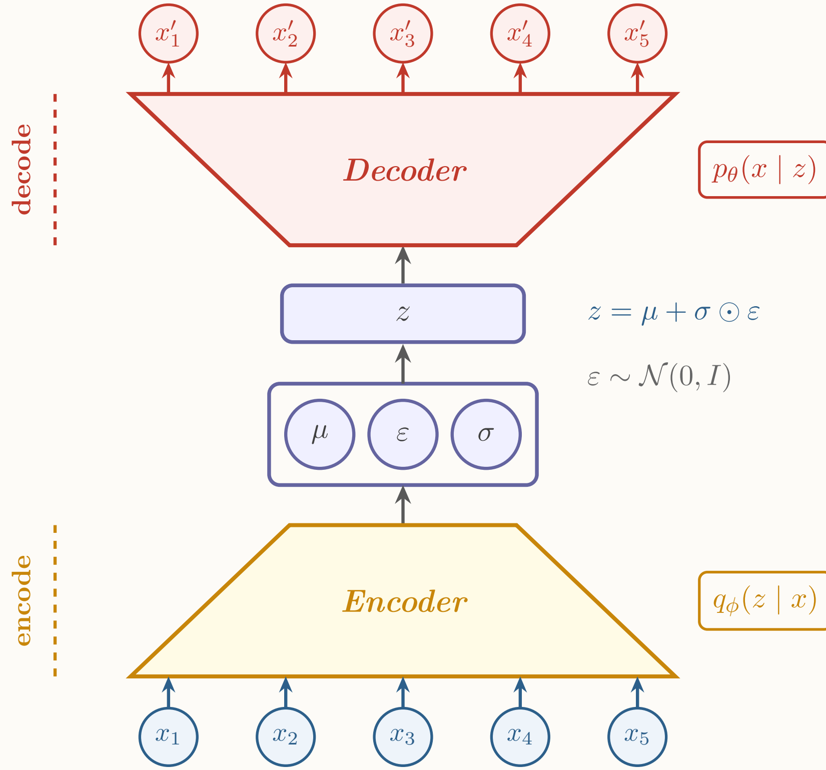

Figure 3.2: The variational autoencoder. The encoder $q_\phi(z \mid x)$ maps the input to a Gaussian over the latent space, producing mean μ and standard deviation σ. Combined with noise $\varepsilon \sim \mathcal{N}(0, I)$, the reparameterisation $z = \mu + \sigma \odot \varepsilon$ produces the latent code z, keeping gradients differentiable through μ and σ. The decoder $p_\theta(x \mid z)$ maps z back to the observation space; outputs x'_i are fresh samples from that distribution.

The **reconstruction term** encourages the decoder to reproduce $\mathbf{x}$ faithfully from the latent code. The **regularisation term** penalises the approximate posterior for deviating from the prior, acting as a regulariser on the latent space.

For a Gaussian encoder and standard Gaussian prior $p(\mathbf{z}) = \mathcal{N}(\mathbf{0}, \mathbf{I})$, the KL term has the closed form

$$D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x}) \,\|\, p(\mathbf{z})) \;=\; -\frac{1}{2} \sum_{j=1}^{d} \left(1 + \log \sigma^2_{\boldsymbol{\phi},j}(\mathbf{x}) - \mu^2_{\boldsymbol{\phi},j}(\mathbf{x}) - \sigma^2_{\boldsymbol{\phi},j}(\mathbf{x})\right). \tag{3.65}$$

3.10.3 Training via the Reparameterised ELBO

Applying the reparameterisation trick to the Gaussian encoder:

$$\boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \tag{3.66}$$

$$\mathbf{z} = \boldsymbol{\mu}_{\boldsymbol{\phi}}(\mathbf{x}) + \boldsymbol{\sigma}_{\boldsymbol{\phi}}(\mathbf{x}) \odot \boldsymbol{\varepsilon}, \tag{3.67}$$

a single-sample Monte Carlo estimator of the ELBO gradient is

$$\widetilde{\mathcal{L}}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}, \boldsymbol{\varepsilon}) \;=\; \log p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z}) + \log p(\mathbf{z}) - \log q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x}). \tag{3.68}$$

This expression is differentiable with respect to both $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$, enabling end-to-end training via mini-batch stochastic gradient ascent (the AEVB algorithm of Kingma and Welling, 2014).

Computation of $\log q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})$ Evaluation of the ELBO requires computing the encoder density $\log q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})$ for a given input $\mathbf{x}$ and latent variable $\mathbf{z}$. In the reparameterised formulation, the latent variable is expressed as a deterministic transformation of an auxiliary noise variable:

$$\mathbf{z} = g_{\boldsymbol{\phi}}(\mathbf{x}, \boldsymbol{\epsilon}), \qquad \boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon}), \tag{3.69}$$

where $p(\boldsymbol{\epsilon})$ is typically chosen to be a standard normal distribution.

If the transformation $g_{\boldsymbol{\phi}}$ is invertible with respect to $\boldsymbol{\epsilon}$, the density of $\mathbf{z}$ is obtained via the change-of-variables formula:

$$\log q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x}) = \log p(\boldsymbol{\epsilon}) - \log \left| \det \frac{\partial \mathbf{z}}{\partial \boldsymbol{\epsilon}} \right|. \tag{3.70}$$

The Jacobian matrix $\frac{\partial \mathbf{z}}{\partial \boldsymbol{\epsilon}}$ contains all first-order partial derivatives of the transformation and captures how local volumes are stretched or compressed. The determinant of this matrix therefore measures the change in volume induced by the mapping from $\boldsymbol{\epsilon}$ to $\mathbf{z}$, and the log-determinant adjusts the density accordingly.

In the case of a standard VAE, the encoder is defined as a diagonal Gaussian with reparameterisation

$$z_j = \mu_{\boldsymbol{\phi},j}(\mathbf{x}) + \sigma_{\boldsymbol{\phi},j}(\mathbf{x})\, \epsilon_j, \qquad j = 1, \ldots, d. \tag{3.71}$$

Each latent dimension z_j depends only on the corresponding noise variable ϵ_j, and not on any ϵ_k for $k \neq j$. Consequently, all off-diagonal partial derivatives $\frac{\partial z_i}{\partial \epsilon_j}$ vanish for $i \neq j$, while the diagonal entries are given by $\frac{\partial z_j}{\partial \epsilon_j} = \sigma_{\boldsymbol{\phi},j}(\mathbf{x})$. The Jacobian matrix is therefore diagonal:

$$\frac{\partial \mathbf{z}}{\partial \boldsymbol{\epsilon}} = \operatorname{diag}\big(\sigma_{\boldsymbol{\phi},1}(\mathbf{x}), \ldots, \sigma_{\boldsymbol{\phi},d}(\mathbf{x})\big). \tag{3.72}$$

The determinant of a diagonal matrix is the product of its diagonal entries, yielding

$$\det \frac{\partial \mathbf{z}}{\partial \boldsymbol{\epsilon}} = \prod_{j=1}^{d} \sigma_{\boldsymbol{\phi},j}(\mathbf{x}), \tag{3.73}$$

and therefore the log-determinant simplifies to

$$\log \left| \det \frac{\partial \mathbf{z}}{\partial \boldsymbol{\epsilon}} \right| = \sum_{j=1}^{d} \log \sigma_{\boldsymbol{\phi},j}(\mathbf{x}). \tag{3.74}$$

Substituting this into the change-of-variables expression shows that $\log q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})$ can be written entirely in terms of the base density $p(\boldsymbol{\epsilon})$ and the encoder outputs $\boldsymbol{\mu}_{\boldsymbol{\phi}}(\mathbf{x})$ and $\boldsymbol{\sigma}_{\boldsymbol{\phi}}(\mathbf{x})$. This recovers the familiar Gaussian log-density and, in practice, allows the encoder term (and the corresponding KL divergence in the ELBO) to be computed in closed form without explicitly evaluating Jacobian determinants.

3.11 Challenges and Limitations of the VAE

The VAE is a principled and scalable model, but it has known failure modes that a careful practitioner will encounter. Understanding them is not merely academic: each challenge points toward a concrete fix, and two of those fixes—the importance-weighted bound and normalizing flows inside q—are important results in their own right.

3.11.1 Posterior Collapse

The most practically significant failure mode of the VAE is *posterior collapse*: the encoder ignores the input and the approximate posterior degenerates to the prior,

$$q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x}) \approx p(\mathbf{z}) \quad \text{for all } \mathbf{x} \in \mathcal{D}. \tag{3.75}$$

When this happens the KL regularisation term in the ELBO vanishes ($D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) \,\|\, p(\mathbf{z})) \approx 0$), and the decoder learns to ignore $\mathbf{z}$ entirely, operating as an unconditional model. The latent space becomes useless for representation learning or generation despite a numerically high ELBO.

Why it happens. The ELBO objective rewards the model for both reconstructing $\mathbf{x}$ well *and* keeping $q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})$ close to the prior. With a sufficiently expressive decoder—in particular, an autoregressive decoder such as PixelCNN or a Transformer—the model can explain $\mathbf{x}$ without using $\mathbf{z}$ at all, satisfying both terms simultaneously. The optimiser finds this degenerate solution whenever the decoder is powerful enough to reach it.

Fixes. Several practical remedies exist.

KL annealing starts training with the KL coefficient set to zero (pure reconstruction) and gradually increases it to one over the first few epochs. This forces the encoder to first learn to use $\mathbf{z}$ for reconstruction before the regulariser kicks in.

Free bits (Kingma et al., 2016) replaces the KL term with a hinge loss:

$$\mathcal{L}_{\lambda} = \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}[\log p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z})] - \sum_{j=1}^{k} \max(\lambda,\ D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(z_j \mid \mathbf{x}) \,\|\, p(z_j))) , \tag{3.76}$$

where z_j denotes the j-th latent dimension. Each dimension is guaranteed at least λ nats of information by construction—collapse is prevented by design.

3.11.2 The Expressiveness of the Variational Family

The standard VAE encoder outputs a diagonal Gaussian posterior:

$$q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x}) = \mathcal{N}\big(\mathbf{z};\ \boldsymbol{\mu}_{\boldsymbol{\phi}}(\mathbf{x}),\ \mathrm{diag}\big(\boldsymbol{\sigma}^2_{\boldsymbol{\phi}}(\mathbf{x})\big)\big) . \tag{3.77}$$

This choice is convenient—the KL divergence to the standard Gaussian prior has a closed form—but it imposes a structural constraint that the true posterior $p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x})$ need not satisfy. If the true posterior is multimodal, or has strong correlations between dimensions, no diagonal Gaussian can approximate it well. The inference KL $D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) \,\|\, p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}))$ remains bounded

away from zero not because of insufficient training, but because the family (3.77) simply cannot contain the true posterior. This means the ELBO gap

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}) - \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}) = D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) \,\|\, p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})) > 0 \tag{3.78}$$

cannot be closed by training, and the model learns a suboptimal $\boldsymbol{\theta}$ as a result.

The fix is to make q more flexible. The next section, on normalizing flows, shows how flows applied *inside* the variational posterior directly address this problem.

3.11.3 The ELBO is Not the Likelihood

A subtler issue: we optimise the ELBO, but what we care about is $\log p_{\boldsymbol{\theta}}(\mathbf{x})$. The two differ by the inference gap, so a model that achieves a higher ELBO does not necessarily have a higher likelihood (the gap may have increased too). Comparing two VAEs by their ELBO values is only valid if their inference gaps are similar.

A principled remedy is the *importance-weighted ELBO* (IWAE; Burda et al., 2016). Draw K independent samples $\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(K)} \sim q_{\boldsymbol{\phi}}(\mathbf{z} \mid \mathbf{x})$ and define

$$\mathcal{L}_K(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}) = \mathbb{E}_{\mathbf{z}^{(1)},\ldots,\mathbf{z}^{(K)}}\left[\log \frac{1}{K}\sum_{k=1}^{K} \frac{p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}^{(k)})}{q_{\boldsymbol{\phi}}(\mathbf{z}^{(k)} \mid \mathbf{x})}\right]. \tag{3.79}$$

The bound satisfies $\mathcal{L}_K \geq \mathcal{L}_{K-1} \geq \cdots \geq \mathcal{L}_1 = \mathcal{L}$, and as $K \to \infty$, $\mathcal{L}_K \to \log p_{\boldsymbol{\theta}}(\mathbf{x})$. Training on $\mathcal{L}_K$ therefore gives a tighter surrogate for the true objective, and for evaluation it provides a principled upper bound on the negative log-likelihood.

The cost is that gradient estimates for $\boldsymbol{\phi}$ from the IWAE objective have higher variance than those from the ELBO, and there is a known tension: using a very large K at training time can actually *weaken* the encoder because the variance reduction from importance sampling replaces the information gradient from the encoder. In practice, K is kept moderate (4–16) and the IWAE is used primarily for evaluation.

3.12 Normalizing Flows

The VAE approximates the marginal likelihood with the ELBO because the true likelihood $\log p_{\boldsymbol{\theta}}(\mathbf{x})$ requires integrating out $\mathbf{z}$—an integral that is intractable with a neural-network decoder. This raises a natural question: is the approximation necessary? Can we design a generative model where $\log p_{\boldsymbol{\theta}}(\mathbf{x})$ is computable exactly?

The answer is yes, provided we are willing to accept a different architectural constraint. *Normalizing flows* construct the model distribution by pushing a simple base distribution through a sequence of invertible transformations. The change-of-variables formula then gives the exact log-likelihood—no bound, no approximate posterior.

3.12.1 The Change-of-Variables Formula

Start with a simple *base distribution* $p_{\mathbf{z}}(\mathbf{z})$, typically $\mathcal{N}(\mathbf{0}, \mathbf{I})$, and define $\mathbf{x} = f_{\boldsymbol{\theta}}(\mathbf{z})$ where $f_{\boldsymbol{\theta}} : \mathbb{R}^d \to \mathbb{R}^d$ is a differentiable, invertible function. Because $f_{\boldsymbol{\theta}}$ is a bijection, the induced distribution over $\mathbf{x}$ is well-defined. The change-of-variables formula gives its log-density ex-

actly:

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}) = \log p_{\mathbf{z}}\big(f_{\boldsymbol{\theta}}^{-1}(\mathbf{x})\big) + \log \left|\det \mathbf{J}_{f_{\boldsymbol{\theta}}^{-1}}(\mathbf{x})\right|, \tag{3.80}$$

where $\mathbf{J}_{f_{\boldsymbol{\theta}}^{-1}}(\mathbf{x}) = \partial f_{\boldsymbol{\theta}}^{-1}(\mathbf{x})/\partial \mathbf{x}$ is the Jacobian of the inverse map. The second term accounts for how $f_{\boldsymbol{\theta}}$ stretches or compresses volume: a transformation that concentrates probability mass in a small region must have a large Jacobian determinant to compensate.

Crucially, no integral appears. Given $\mathbf{x}$, we invert the flow to get $\mathbf{z} = f_{\boldsymbol{\theta}}^{-1}(\mathbf{x})$, evaluate $p_{\mathbf{z}}(\mathbf{z})$, and compute the log-determinant. The result is the *exact* log-likelihood. Training by maximum likelihood is therefore straightforward: maximise $\sum_n \log p_{\boldsymbol{\theta}}(\mathbf{x}^{(n)})$ directly, with no variational approximation in sight.

3.12.2 Composing Flows

A single invertible transformation may not be expressive enough to map a simple Gaussian to a complex data distribution. The standard remedy is to compose L transformations:

$$\mathbf{x} = f_L \circ f_{L-1} \circ \cdots \circ f_1(\mathbf{z}), \qquad \mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z}). \tag{3.81}$$

Let $\mathbf{z}_0 = \mathbf{z}$ and $\mathbf{z}_l = f_l(\mathbf{z}_{l-1})$, so $\mathbf{x} = \mathbf{z}_L$. The log-likelihood accumulates the log-determinant at each layer:

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}) = \log p_{\mathbf{z}}(\mathbf{z}_0) + \sum_{l=1}^{L} \log \left|\det \mathbf{J}_{f_l^{-1}}(\mathbf{z}_l)\right|. \tag{3.82}$$

Each layer is trained end-to-end by maximising this exact log-likelihood. Increasing L increases expressiveness at the cost of more computation.

Sampling. Sampling is also exact and efficient: draw $\mathbf{z} \sim p_{\mathbf{z}}$, then apply $f_1, f_2, \ldots, f_L$ in sequence. No Markov chain, no rejection, no variational approximation. This is a significant practical advantage over the VAE, where the decoder introduces an additional source of stochasticity.

3.12.3 Architectural Constraints: Making the Jacobian Cheap

The bottleneck in (3.82) is the log-determinant $\log|\det \mathbf{J}_{f_l^{-1}}|$. For a general $d \times d$ Jacobian, LU decomposition costs $\mathcal{O}(d^3)$: infeasible for images where d may be tens of thousands. Normalizing flows are therefore defined by their answer to the question: *how do we parameterise f_l so that $\log|\det \mathbf{J}_{f_l^{-1}}|$ is cheap to compute?*

Two main families have emerged.

Coupling layers (NICE, RealNVP, Glow). Partition the dimensions into two halves: $\mathbf{z} = (\mathbf{z}_A, \mathbf{z}_B)$. Define the transformation

$$\mathbf{z}'_A = \mathbf{z}_A, \tag{3.83}$$

$$\mathbf{z}'_B = \mathbf{z}_B \odot \exp(s(\mathbf{z}_A)) + t(\mathbf{z}_A), \tag{3.84}$$

where $s, t : \mathbb{R}^{d/2} \to \mathbb{R}^{d/2}$ are arbitrary neural networks (the *scale* and *translate* networks) and $\odot$ denotes element-wise multiplication. Because $\mathbf{z}_A$ is copied unchanged, the Jacobian of this

transformation is lower-triangular, and its determinant is simply the product of the diagonal entries:

$$\log |\det \mathbf{J}| = \sum_i s_i(\mathbf{z}_A), \tag{3.85}$$

computable in $\mathcal{O}(d)$. The inverse is also efficient: $\mathbf{z}_B = (\mathbf{z}'_B - t(\mathbf{z}_A)) \odot \exp(-s(\mathbf{z}_A))$. Stacking coupling layers with alternating partitions ensures all dimensions are eventually transformed. *Glow* (Kingma and Dhariwal, 2018) adds 1×1 invertible convolutions between coupling layers to allow flexible permutations of the channels.

Autoregressive flows (MAF, IAF). In an autoregressive flow, each output dimension depends only on the preceding dimensions:

$$z'_i = z_i \cdot \sigma(z_{1:i-1}) + \mu(z_{1:i-1}), \tag{3.86}$$

where σ and μ are neural networks. The Jacobian is lower-triangular by construction, so its determinant is again the product of the diagonal—in this case the values $\sigma(z_{1:i-1})$.

The two autoregressive directions yield different trade-offs. In a *Masked Autoregressive Flow* (MAF), the forward pass $\mathbf{z} \to \mathbf{z}'$ requires d sequential evaluations (each z'_i depends on earlier $z'_{1:i-1}$ values), making density evaluation *fast* but sampling *slow*. An *Inverse Autoregressive Flow* (IAF) inverts this structure—sampling is fast and parallelisable but density evaluation is slow.

3.12.4 Flows Inside the Variational Posterior

Flows are not limited to use as standalone generative models. They provide a direct fix to the expressiveness problem identified in Section 3.11.2: if the diagonal Gaussian $q_\phi(\mathbf{z} \mid \mathbf{x})$ is too restrictive, apply a normalizing flow to make it richer.

The procedure (Kingma et al., 2016) is:

1. Sample $\mathbf{z}_0 \sim q_0(\mathbf{z} \mid \mathbf{x}) = \mathcal{N}(\boldsymbol{\mu}_\phi(\mathbf{x}), \operatorname{diag}(\boldsymbol{\sigma}^2_\phi(\mathbf{x})))$ using the reparameterisation trick.
2. Apply L invertible flow transformations: $\mathbf{z}_l = f_l(\mathbf{z}_{l-1})$, $l = 1, \ldots, L$.
3. Use $\mathbf{z}_L$ as the latent sample.

The log-density of $\mathbf{z}_L$ under the flow posterior is obtained exactly by the change-of-variables formula:

$$\log q_L(\mathbf{z}_L \mid \mathbf{x}) = \log q_0(\mathbf{z}_0 \mid \mathbf{x}) - \sum_{l=1}^{L} \log \left|\det \mathbf{J}_{f_l}(\mathbf{z}_{l-1})\right|. \tag{3.87}$$

Substituting (3.87) into the ELBO gives a valid lower bound that can be optimised by stochastic gradient descent. The richer posterior q_L can now match multimodal or correlated true posteriors that the simple Gaussian cannot, tightening the ELBO gap.

Because IAF transformations are fast to *sample* (the bottleneck at training time), IAF is the natural choice for the flow inside q. The resulting model is often called a *VAE with IAF*.

3.12.5 VAE vs. Normalizing Flows: Two Philosophies

Having developed both the VAE and normalizing flows, it is instructive to place them side by side.

	VAE	**Normalizing flow**
Likelihood	Approximate (ELBO lower bound)	Exact
Decoder	Unconstrained neural network	Must be invertible
Encoder	Learned approximate posterior q_ϕ	None (invert the flow)
Sampling	Sample $\mathbf{z}$, decode	Apply $f_1, \ldots, f_L$ to $\mathbf{z}$
Posterior access	Via encoder in one forward pass	By inversion (may be slow)
Latent space	Structured by KL regularisation	Determined by flow geometry
Best suited for	Representation learning, semi-supervised tasks	Density estimation, exact likelihood

The VAE trades exact inference for architectural freedom and a learned encoder. The encoder is not a side effect—it is a primary output of training, giving us a fast approximate posterior for every datapoint. This makes the VAE natural for downstream tasks such as semi-supervised learning, anomaly detection, and structured generation where we need to *read off* $\mathbf{z}$ from $\mathbf{x}$ quickly.

Normalizing flows trade architectural freedom for an exact likelihood. Every layer must be invertible with a tractable Jacobian determinant, which is a significant constraint on what neural-network architectures are permissible. The payoff is that we can evaluate $\log p_{\boldsymbol{\theta}}(\mathbf{x})$ exactly and compare models rigorously by their test likelihood.

Both families learn a continuous, structured latent space and can generate high-quality samples. Both can be improved by increasing model depth. And, as flows-inside-q shows, the two approaches are not mutually exclusive: the VAE's approximate posterior can itself be a normalizing flow.

Forward pointer. Both the VAE and normalizing flows define their generative model explicitly via a parameterised likelihood $p_{\boldsymbol{\theta}}(\mathbf{x} \mid \mathbf{z})$ or $p_{\boldsymbol{\theta}}(\mathbf{x})$. The next chapter explores a third approach—*score matching and diffusion models*—which sidesteps the likelihood entirely by learning the score function $\nabla_{\mathbf{x}} \log p(\mathbf{x})$. Notably, the VAE reappears as the compression backbone of *latent diffusion models* (Rombach et al., 2022), where the diffusion process operates in the VAE's latent space rather than pixel space, combining the representational efficiency of the VAE with the generation quality of diffusion.

3.13 Summary

Each section of this chapter answered a question left open by the previous one.

We started with **shallow latent variable models**—factor analysis, Gaussian mixtures, and hidden Markov models—where the decoder is simple enough that learning and inference are tractable in closed form via EM. These models are elegant but hit a hard capacity wall: a linear or categorical decoder cannot capture the complex structure of real data.

The natural fix was to replace the shallow decoder with a neural network, giving us **deep latent variable models**. Doing so broke the tractability of inference: the posterior $p_{\boldsymbol{\theta}}(\mathbf{z} \mid \mathbf{x})$ no longer has a closed form, and EM's E-step fails.

Variational inference resolved the intractability by turning inference into optimisation: introduce $q_{\boldsymbol{\lambda}}(\mathbf{z} \mid \mathbf{x})$ and derive the ELBO, a tractable lower bound whose gap to the true log-likelihood equals $D_{\mathrm{KL}}(q_{\boldsymbol{\lambda}} \,\|\, p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}))$. The two-KL view shows that maximising the ELBO simultaneously fits the model to data and drives the approximate posterior toward the true one—a single objective, two goals.

BBVI made the optimisation runnable via stochastic gradients. The **reparameterisation trick** replaced REINFORCE's high-variance estimates with low-variance, backpropagation-compatible gradients. **Amortised inference** eliminated the inner optimisation loop by replacing per-datapoint variational parameters with a shared encoder network.

The **VAE** is the result of combining all of these: neural decoder, neural encoder, reparameterised amortised ELBO. Every component was forced by a concrete problem; none was introduced from outside. The VAE's encoder is also the compression backbone of latent diffusion models such as Stable Diffusion—see the diffusion chapter for how score matching builds on top of it.

The VAE has known **challenges**: posterior collapse (strong decoders ignore $\mathbf{z}$), limited expressiveness of the diagonal Gaussian posterior, and the ELBO gap. Each challenge points toward an active research direction. The IWAE provides a tighter bound; flows inside q address posterior expressiveness.

Normalizing flows offer a different philosophy entirely: instead of approximating the likelihood, they compute it exactly by constructing $p(\mathbf{x})$ via an invertible change of variables. Architectural constraints (coupling layers, autoregressive structure) make the required Jacobian determinants tractable. VAE and flows are complementary—the VAE trades exactness for a learnable encoder and flexible decoder; flows trade architectural freedom for an exact likelihood.

The next chapter turns to a third philosophy—**score matching and diffusion models**—which sidesteps the likelihood entirely and instead learns to generate data by modelling the gradient of the log-density.

3.14 Problems

Question 3.1. *1. Consider that you are supposed to optimize the negative log-likelihood of the data sampled from an unknown distribution p_{data} in a VAE setup. Answer the following.*

(a) Construct the objective function in terms of the divergence between the aggregated posterior

$$q_\phi(z) = \int q_\phi(z|x) p_{data}(x)\, dx$$

and the latent prior $p_\theta(z)$ together with the conditional likelihood $p_\theta(x|z)$.

(b) Show that the optimal value of the ELBO is equal to the negative of the data entropy.

(c) Suppose p_{data} is non-Gaussian. Show that it is impossible to reach the optimum value derived above for the ELBO if a Gaussian latent prior $p_\theta(z)$ is assumed.

(d) Suppose we want to compute the gradient

$$\nabla_\theta \mathbb{E}_{p(z;\theta)}[f(z)]$$

where $f(z)$ is an arbitrary function. Show that this is equivalent to

$$\mathbb{E}_{p(\epsilon)}\left[\nabla_\theta f(g(\epsilon,\theta))\right]$$

where $p(\epsilon)$ is an arbitrary distribution and $g(\epsilon,\theta)$ maps $p(\epsilon)$ to $p(z)$. Taking $p(z)$ to be an exponential distribution with $\lambda = 2$ and $p(\epsilon) \sim U[0,1]$, find $g(\epsilon)$.

3.15 Solutions

Solution.

(a)

The ELBO for a datapoint x is

$$\log p_\theta(x) \geq \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{\mathrm{KL}}(q_\phi(z|x) \| p_\theta(z)).$$

Taking expectation over the data distribution gives

$$\mathbb{E}_{p_{\mathrm{data}}(x)}\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - \mathbb{E}_{p_{\mathrm{data}}(x)}D_{\mathrm{KL}}(q_\phi(z|x) \| p_\theta(z)).$$

Define the aggregated posterior

$$q_\phi(z) = \int q_\phi(z|x)p_{\mathrm{data}}(x)\,dx.$$

Introducing the mutual information

$$I_q(x;z) = \mathbb{E}_{p_{\mathrm{data}}(x)}D_{\mathrm{KL}}(q_\phi(z|x) \| q_\phi(z)),$$

we obtain

$$\mathbb{E}_{p_{\mathrm{data}}(x)}D_{\mathrm{KL}}(q_\phi(z|x) \| p_\theta(z)) = D_{\mathrm{KL}}(q_\phi(z) \| p_\theta(z)) + I_q(x;z).$$

Substituting gives

$$\mathcal{L} = \mathbb{E}_{p_{\mathrm{data}}(x)}\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{\mathrm{KL}}(q_\phi(z) \| p_\theta(z)) - I_q(x;z).$$

Ignoring the mutual information term yields

$$\mathbb{E}_{p_{\mathrm{data}}(x)}\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{\mathrm{KL}}(q_\phi(z) \| p_\theta(z)).$$

(b)

At the optimum

$$p_\theta(x) = p_{\mathrm{data}}(x).$$

Therefore

$$\max \mathrm{ELBO} = \mathbb{E}_{p_{\text{data}}(x)}[\log p_{\text{data}}(x)].$$

The entropy of the data distribution is

$$H(p_{\text{data}}) = -\mathbb{E}_{p_{\text{data}}(x)}[\log p_{\text{data}}(x)].$$

Hence

$$\max \mathrm{ELBO} = -H(p_{\text{data}}).$$

(c)
Optimality requires

$$q_\phi(z) = p_\theta(z).$$

If the prior is Gaussian

$$p_\theta(z) = \mathcal{N}(0, I),$$

but

$$q_\phi(z) = \int q_\phi(z|x) p_{\text{data}}(x) dx$$

is generally a mixture distribution induced by the data. For non-Gaussian data this distribution is not necessarily Gaussian. Hence

$$q_\phi(z) \neq \mathcal{N}(0, I).$$

Therefore the optimal ELBO cannot be reached.

(d)
Let

$$z = g(\epsilon, \theta), \qquad \epsilon \sim p(\epsilon).$$

Then

$$\mathbb{E}_{p(z;\theta)}[f(z)] = \mathbb{E}_{p(\epsilon)}[f(g(\epsilon, \theta))].$$

Taking the gradient,

$$\nabla_\theta \mathbb{E}_{p(z;\theta)}[f(z)] = \mathbb{E}_{p(\epsilon)}[\nabla_\theta f(g(\epsilon, \theta))].$$

For an exponential distribution

$$p(z) = \lambda e^{-\lambda z}, \quad z \geq 0.$$

The CDF is

$$F(z) = 1 - e^{-\lambda z}.$$

Let $\epsilon \sim U[0, 1]$,

$$\epsilon = 1 - e^{-\lambda z}.$$

Solving for z,

$$z = -\frac{1}{\lambda} \log(1 - \epsilon).$$

For $\lambda = 2$,

$$g(\epsilon) = -\frac{1}{2} \log(1 - \epsilon).$$