· Neural Networks and Inductive Biases

"Look. The models, they just want to learn. You have to understand this. The models, they just want to learn." —Ilya Sutskever

1 Modeling unknown functions

Scientific data is full of relationships we can measure but not write down. A telescope records stellar spectra; the mapping from spectrum to stellar composition is complex. A detector measures particle interactions; extracting physical parameters requires inverting a complicated function. We have (x, y) pairs and want to learn $y = f(x)$, but f has no closed form.

The manifold hypothesis. Scientific data is high-dimensional, but it's generated by low-dimensional processes. A cosmological density field has millions of voxels, but it's determined by just a handful of cosmological parameters (Ω_m, σ_8, H_0, ...) plus initial conditions. A protein structure has thousands of atomic coordinates, but viable configurations are constrained by bond lengths, angles, and evolutionary pressures to a far smaller space.

This is the *manifold hypothesis*: high-dimensional data concentrates near a lower-dimensional manifold embedded in the ambient space. The manifold isn't arbitrary—its geometry reflects the underlying science, and nearby on the manifold correspond to similar physical states.

Why does this help? Learning becomes tractable because we only need to model the manifold, not the full huge-dimensional space. The curse of dimensionality applies to the *intrinsic* dimension (the manifold), not the *ambient* dimension (e.g., the pixel space).

Summary statistics as projections. Scientists have long exploited manifold structure through hand-crafted summary statistics. Consider a cosmological density field. The raw data is a 3D grid of densities—millions of numbers. But much of what we care about is captured by the *power spectrum* $P(k)$: how much structure exists at each spatial scale.

The power spectrum is a projection from the high-dimensional data manifold to a lower-dimensional summary. It preserves certain information (the amplitude of fluctuations at each scale) while discarding other information (the phases—where exactly the structures are located). This is exactly what we want: **compress the data while retaining what matters for the scientific question we're asking**. *Many different data points can map to the same summary.* Infinitely many cosmological fields share the same power spectrum—they differ only in their phases. The summary defines equivalence classes on the manifold.

Why summaries work. Good summaries encode domain knowledge. The power spectrum is translation-invariant by construction—shifting the field doesn't change $P(k)$. The symmetry is built into the summary, so downstream models don't have to learn it.

But summaries have limits. The power spectrum captures Gaussian structure perfectly but misses higher-order correlations. Hand-crafted molecular fingerprints might miss subtle electronic effects. Compressing data is a trade-off!

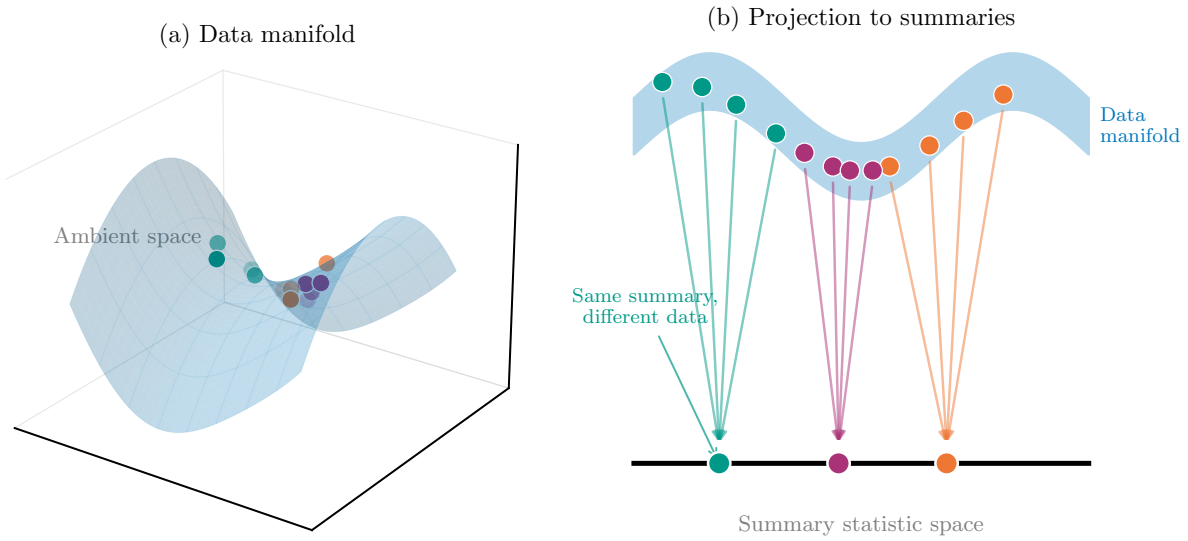


Figure 1. The manifold hypothesis and summary statistics. (a) Scientific data lives on a low-dimensional manifold embedded in high-dimensional ambient space. Points of the same color represent different configurations. (b) Summary statistics project the manifold to a lower-dimensional space. Multiple points on the manifold (same color) map to a single summary—these are equivalence classes of "the same for my purposes."

Data-driven projections. Can we learn summaries without domain expertise? PCA finds the linear projection that preserves the most variance—a data-driven approach that requires no physics knowledge. But PCA assumes the data manifold is flat. It finds the best hyperplane through the data, missing any curvature. When the manifold is nonlinear (as it usually is for complex scientific data), PCA captures only a shadow of the true structure. Other data-driven methods (t-SNE, UMAP, Isomap, ...) try to preserve local distances or neighborhood structure, but they too struggle with complex manifolds and scale poorly to large datasets.

Learning nonlinear projections. Neural networks learn summaries from data. Instead of hand-crafting which aspects of the manifold to preserve, we parameterize a flexible family of projections and let optimization find what works for the task. The architecture encodes structural assumptions (*inductive biases*) that constrain which projections are possible. Without them, we might need impossibly large datasets to discover structure that physics already tells us must be there.

2 Neural networks: composing simple functions

A neural network builds complex functions by composing simple ones. Apply a linear transformation, then a simple nonlinearity, then another linear transformation, and so on:

$$h_1 = \sigma(W_1 x + b_1) \tag{2.1}$$

$$h_2 = \sigma(W_2 h_1 + b_2) \tag{2.2}$$

$$y = W_3 h_2 + b_3 \tag{2.3}$$

Each layer applies weights W_ℓ, biases b_ℓ, and a nonlinear activation σ. This is a *multilayer perceptron* (MLP).

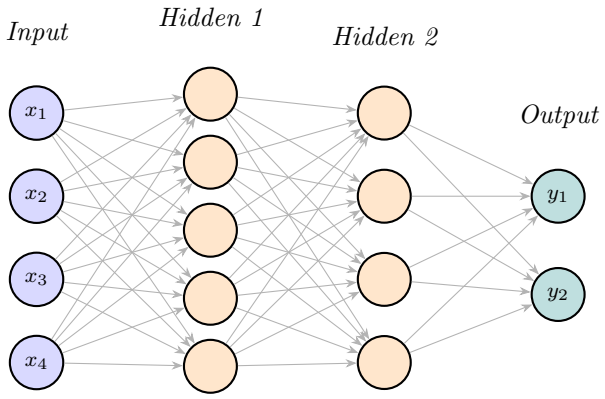


Figure 2. A multilayer perceptron with two hidden layers. Each layer applies a linear transformation followed by a nonlinearity. All neurons in adjacent layers are connected ("fully connected").

Without σ, the network collapses to a single linear map:

$$W_2(W_1 x + b_1) + b_2 = (W_2 W_1)x + (W_2 b_1 + b_2) = W'x + b' \tag{2.4}$$

No matter how many layers, the result is linear. The nonlinearity is what gives depth its power.

What can nonlinearity do? Consider the XOR problem: two classes arranged so no single line can separate them (Figure 3). A linear model fails. But a network with one hidden layer can create a piecewise-linear decision boundary that correctly separates the classes. Adding more layers and neurons produces smoother, more complex boundaries.

Common activations. The most common is ReLU, which is piecewise linear: identity for positive inputs, zero otherwise.

$$\sigma(z) = \max(0, z) \tag{2.5}$$

Universal approximation. With enough neurons, an MLP can approximate any continuous function. This is reassuring but not really of any practical use—it says nothing about how many parameters we need or whether we can find them by training.

3 Training deep networks

Deeper networks can represent more complex functions, but they're harder to train. Gradients must flow backward through many layers; they can vanish (shrink to zero) or explode (grow unboundedly). Two techniques make deep training possible: **residual connections** and **normalization**.

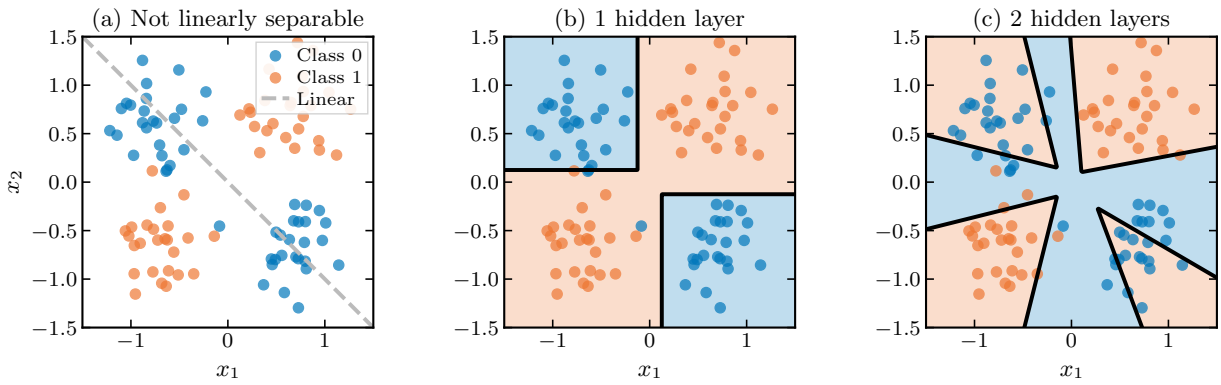


Figure 3. Nonlinearity enables complex decision boundaries. (a) An XOR-like classification problem: no straight line separates the classes. (b) A single hidden layer creates a piecewise-linear boundary. (c) A deeper network produces a smooth boundary that perfectly separates the data.

Residual connections. Instead of learning $h' = F(h)$, learn the *residual* $h' = F(h) + h$. The network learns what to *add* to the input rather than computing the output from scratch.

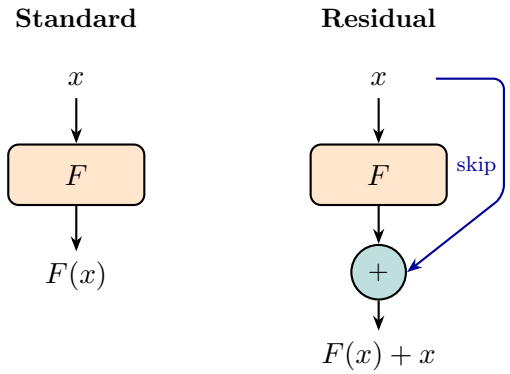


Figure 4. Residual (skip) connections. Left: a standard network computes $F(x)$. Right: a residual block computes $F(x) + x$. The skip connection provides a direct path for gradients, enabling training of very deep networks.

Why does this help? Gradients can flow directly through the skip connection, bypassing the layers entirely. Even if F contributes small gradients, the identity path ensures signal reaches early layers.

Normalization. During training, the distribution of each layer's inputs shifts as earlier layers change. *Batch normalization* fixes this by normalizing each feature across the mini-batch to zero mean and unit variance. *Layer normalization* normalizes across features for each sample independently (standard in Transformers). Both smooth the loss landscape and reduce sensitivity to initialization.

The modern recipe. Nearly all successful deep networks use both:

$$h' = h + F(\text{Norm}(h)) \tag{3.1}$$

First normalize, then apply the learned transformation F, then add the residual. This combination enables training networks with hundreds of layers.

Optimization. We train by minimizing a loss function $\mathcal{L}(\theta)$ over parameters θ. Standard gradient descent computes the gradient over the entire dataset and steps downhill:

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L} \tag{3.2}$$

The problem: gradient descent is deterministic. Where you end up depends entirely on where you start. If you initialize in the wrong "valley" of the loss landscape, you descend to a local minimum and stay there.

Stochastic gradient descent (SGD) fixes this by adding noise. Instead of computing the gradient over all data, we use a random subset—a *mini-batch*—at each step. The gradient estimate is noisy: it points roughly downhill, but not exactly. This noise lets SGD escape shallow local minima. A step might go temporarily uphill, hopping from one valley to another. On average we descend, but the stochasticity explores the landscape. *Adam* and related optimizers adapt the learning rate per-parameter, using running estimates of gradient moments. Adam is often more robust to hyperparameter choices, making it a good default.

Batch size. Mini-batch size controls the noise level. Larger batches give more accurate gradient estimates—less noise, more direct descent. Smaller batches add more noise, helping escape bad minima but making optimization noisier. There's evidence that the noise from small batches acts as implicit regularization, finding flatter minima that generalize better.

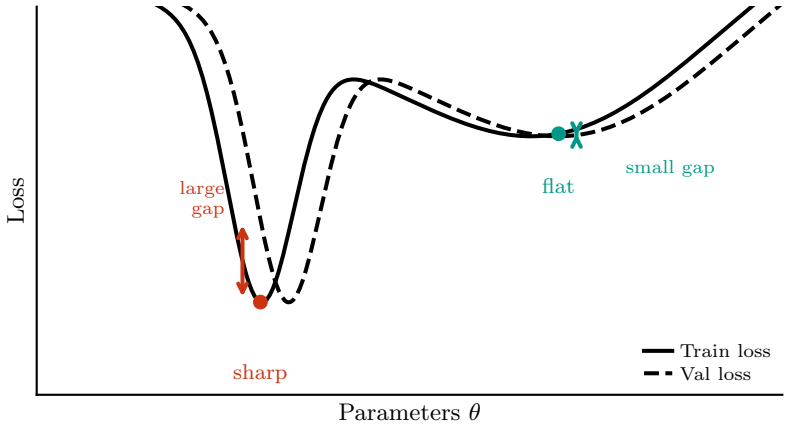


Figure 5. Why flat minima generalize better. The validation loss landscape (dashed) is slightly shifted from training (solid). At a sharp minimum, this shift causes a large generalization gap. At a flat minimum, the same shift produces a much smaller gap—the solution is robust to distribution shift.

4 Autoencoders: learning to compress

Supervised learning requires labels. But labels are expensive, and much scientific data is unlabeled. Can we learn useful representations without supervision?

The autoencoder idea. Train a network to reconstruct its input. This seems trivial—just learn the identity function. The trick is to force the data through a *bottleneck*: a low-dimensional intermediate representation.

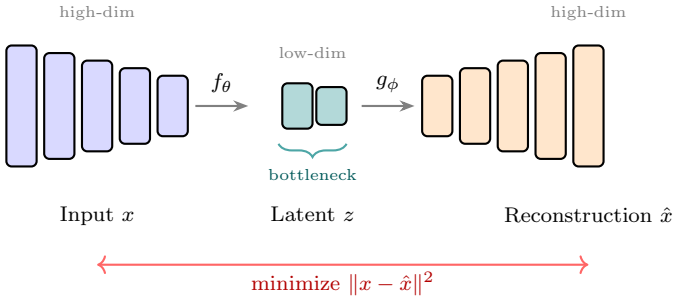


Figure 6. Autoencoder architecture. The encoder f_θ compresses high-dimensional input to a low-dimensional latent representation z. The decoder g_ϕ reconstructs the input from z. The bottleneck forces the network to learn which information matters.

The encoder $f_\theta : \mathbb{R}^D \to \mathbb{R}^d$ maps input x to a low-dimensional latent code $z = f_\theta(x)$. The decoder $g_\phi : \mathbb{R}^d \to \mathbb{R}^D$ reconstructs the input: $\hat{x} = g_\phi(z)$. Train by minimizing reconstruction error:

$$\mathcal{L} = \|x - g_\phi(f_\theta(x))\|^2 \tag{4.1}$$

If $d \ll D$, the network can't memorize inputs—it must learn to compress. The latent space z captures whatever structure is needed for reconstruction.

Connection to the manifold hypothesis. Remember: data lives on a low-dimensional manifold. The autoencoder learns to find it. The encoder projects data onto the manifold (approximately); the decoder maps back to the ambient space. The latent dimension d is a hypothesis about the manifold's intrinsic dimension.

If d is too small, reconstruction suffers—you've lost information. If d is too large, the network might not learn meaningful compression. In practice, you can tune d or let the network learn sparsity through regularization.

The latent space often captures interpretable structure. An autoencoder trained on galaxy images might learn latents corresponding to size, brightness, and morphology—without ever being told these concepts exist. The network discovers the factors of variation that matter for reconstruction.

This makes autoencoders useful for dimensionality reduction—The latent z is a compressed representation, like PCA but nonlinear—and e.g. anomaly detection. Data far from the training manifold reconstructs poorly; high reconstruction error flags anomalies. They can also denoise data by training on corrupted inputs and reconstructing clean outputs.

Limitations and extensions. Standard autoencoders give you a latent space, but it may be irregular—points don't interpolate smoothly, and you can't easily sample new data. Variational autoencoders (VAEs) address this by imposing probabilistic structure on the latent space, enabling smooth interpolation and generation of new samples. We'll return to this when we discuss generative models.

5 Matching architecture to data structure

Fully connected networks treat every input dimension as unrelated. For an image, pixel $(0, 0)$ has no special relationship to pixel $(0, 1)$, even though they're neighbors. The network must learn from scratch that nearby pixels are correlated, that patterns can appear anywhere, that a galaxy in the corner is the same as one in the center.

With enough data, a fully connected network can learn this. But it's inefficient—we're asking the network to rediscover structure we already know. Scientific datasets can be limited in size, and sample efficiency matters—often better to build known structure into the architecture.

The rest of this chapter develops architectures for different data types:

Data type	**Architecture**	**Inductive bias**
Images/grids	CNN	Translation invariance + locality
Sequences	RNN	Temporal order
Sets	Deep Sets	Permutation invariance
Graphs	GNN	Permutation invariance + edges

5.1 Grids → CNNs

Images have two properties that fully connected networks ignore:

- **Translation invariance**: An edge is an edge, regardless of where it appears. A galaxy in the corner should be classified the same as one in the center.
- **Locality**: To understand a pixel, look at its neighbors. Distant pixels are (initially) irrelevant. Global structure emerges from composing local features.

These assumptions reflect the physics of spatial data. The inductive bias is that patterns are local and can appear anywhere.

Deriving convolutions. Start with a fully connected layer:

$$h_{ij} = \sum_{k,l} W_{ijkl}\, x_{kl} \tag{5.1}$$

Weights W_{ijkl} connect input position (k, l) to output position (i, j). For a megapixel image, this requires 10^{12} parameters.

Impose translation invariance: The same pattern should be detected identically everywhere. Weights cannot depend on absolute position—only on the offset. Let $a = k - i$, $b = l - j$:

$$W_{ijkl} = W_{ab} \quad \Rightarrow \quad h_{ij} = \sum_{a,b} W_{ab}\, x_{i+a,j+b} \tag{5.2}$$

This is a *convolution*. The same small set of weights (a *kernel*) slides across the image, computing the same operation at every position.

Impose locality: Only look at nearby pixels. Restrict to a small window:

$$h_{ij} = \sum_{|a|\le k} \sum_{|b|\le k} W_{ab}\, x_{i+a,j+b} \tag{5.3}$$

A 3×3 kernel has 9 parameters. We've gone from 10^{12} to 9—while encoding exactly the structure spatial data has.

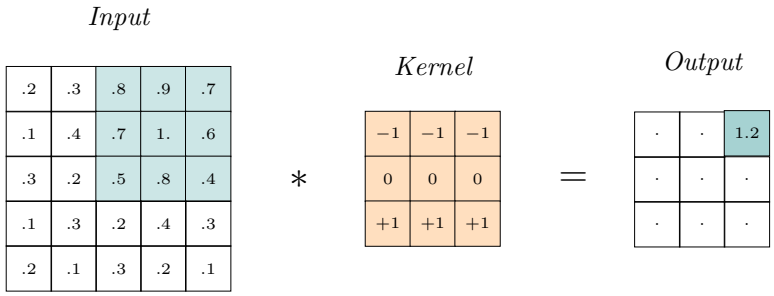


Figure 7. Convolution: a small kernel slides across the image, computing a weighted sum at each position. The same 9 weights are used everywhere (weight sharing). This kernel detects horizontal edges.

What convolutions learn. Each kernel extracts a local pattern. A horizontal edge detector:

$$\begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ +1 & +1 & +1 \end{pmatrix} \tag{5.4}$$

This computes the difference between rows below and above—large where there's a horizontal intensity change. In a CNN, kernels are *learned*. The network discovers which local patterns are useful for the task.

Channels. A single kernel detects one pattern. To detect multiple patterns, use multiple kernels in parallel. Each kernel produces one *output channel*—a feature map detecting that pattern across the image. A layer might have 64 or 256 channels, each detecting a different pattern.

The input can also have multiple channels: RGB images have 3 input channels; scientific images might have spectral bands or multiple observables. Each output channel combines information from all input channels:

$$h_{ij}^{(c)} = \sum_{c'} \sum_{a,b} W_{ab}^{(c,c')} \, x_{i+a,j+b}^{(c')} \tag{5.5}$$

The kernel $W^{(c,c')}$ maps input channel c' to output channel c. For a 3×3 kernel with 64 input and 128 output channels, this is $3 \times 3 \times 64 \times 128 \approx 74{,}000$ parameters—still far fewer than a fully connected layer.

Building hierarchies. Stacking convolutional layers builds a hierarchy of features. Early layers detect simple patterns like edges and gradients; later layers combine these into textures, shapes, and eventually objects. Each layer's *receptive field*—the input region affecting one output—grows with depth. After several layers, each output unit "sees" a large region of the input.

Pooling (taking the max or average over small windows) downsamples spatial dimensions, reducing computation and building robustness to small translations.

CNNs are the standard tool for any data on a grid: images (galaxy classification, medical imaging), spectra (stellar spectra, mass spectra), spatial fields (climate data, cosmological density fields). The same architecture that recognizes cats recognizes spiral galaxies—the inductive bias (translation invariance, locality) matches the data structure.

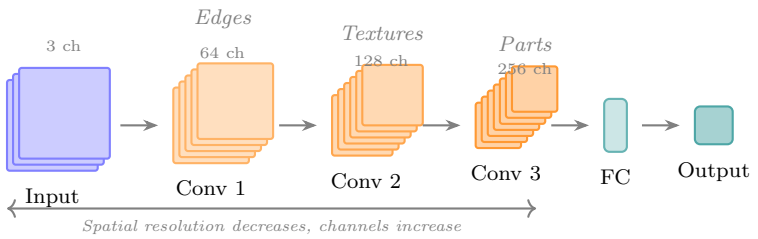


Figure 8. Hierarchical feature learning in CNNs. Early layers detect simple patterns (edges), later layers detect complex structures (objects). Receptive field grows with depth; spatial resolution decreases.

5.2 Sequences → RNNs

Much scientific data is *sequential*: time series, trajectories, genetic sequences. The inductive bias: order matters, and what happens now depends on what came before.

The idea. Process the sequence one element at a time, maintaining a *hidden state* h_t that summarizes what we've seen so far:

$$h_t = \sigma(W_h h_{t-1} + W_x x_t + b) \tag{5.6}$$

This is a **recurrent neural network** (RNN). The same function is applied at every timestep—weight sharing across time, analogous to weight sharing across space in CNNs.

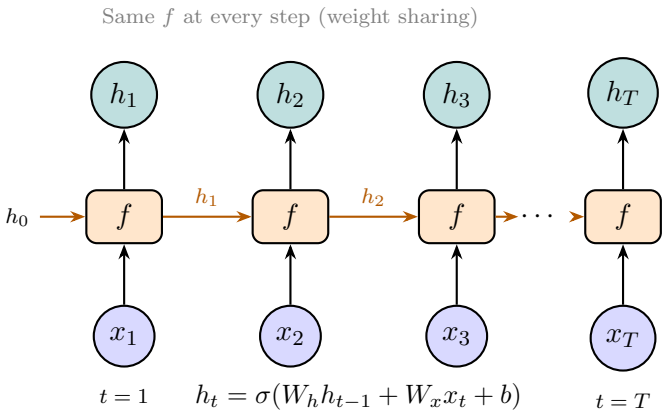


Figure 9. A recurrent neural network unrolled through time. The same function (with shared weights) processes each input x_t and the previous hidden state h_{t-1} to produce the next hidden state h_t. The final state h_T summarizes the entire sequence.

The trouble with RNNs. To learn from long sequences, gradients must flow backward through many timesteps, each multiplying by W_h. If $\|W_h\| < 1$, gradients shrink exponentially (*vanishing gradients*)—the network forgets distant inputs. Gated architectures (LSTMs, GRUs) add learned gates that control information flow, helping gradients propagate. But even with gating, RNNs process sequences *sequentially*—you can't compute h_{100} without first computing h_1 through h_{99}.

Beyond RNNs. Transformers process all positions in parallel using attention (discussed later). They've largely replaced RNNs for most sequence tasks.

5.3 Sets and graphs → Deep Sets and GNNs

Much scientific data has relational structure: a molecule is atoms connected by bonds, a protein is residues along a backbone, a physical system is particles interacting pairwise. These are *graphs*: nodes (entities) and edges (relationships). The structure is irregular—no grid, no canonical ordering.

The key symmetry: **permutation invariance**. Relabeling nodes (e.g., calling atom 1 "atom 2" instead) shouldn't change the output. This is the inductive bias for relational data.

Sets. Before graphs, consider a *set*—elements with no relationships. A point cloud. A bag of features. How do we build a function $f(\{x_1, \ldots, x_n\})$ that is invariant to ordering?

The Deep Sets theorem. Any permutation-invariant function can be written as:

$$f(\{x_1, \ldots, x_n\}) = \rho\left(\sum_{i=1}^{n} \phi(x_i)\right) \tag{5.7}$$

where ϕ processes each element independently and ρ processes the aggregated result. This is the minimal architecture that respects the symmetry of sets.

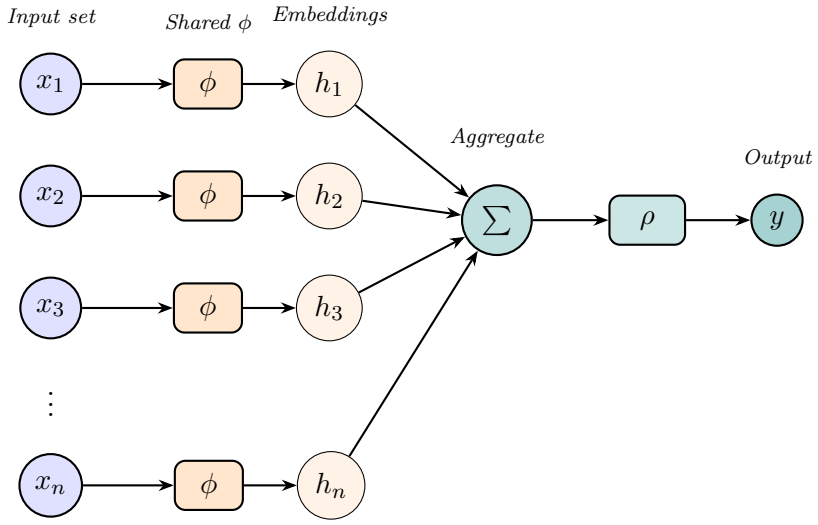


Figure 10. Deep Sets architecture. Each element x_i passes through the same network ϕ (weight sharing). Embeddings are summed (a symmetric operation), then processed by ρ to produce the output. Reordering the inputs doesn't change the sum, so the output is permutation-invariant.

Per-element outputs (equivariance). What if we want outputs for each element, not just the whole set? For example: given particle positions, predict a label for each particle. The architecture:

$$y_i = \psi\left(x_i, \sum_{j=1}^{n} \phi(x_j)\right) \tag{5.8}$$

Each output y_i depends on x_i and a global summary of the set. Permuting the inputs permutes the outputs in the same way—this is permutation *equivariance*.

Deep Sets is the foundation for all permutation-invariant architectures. GNNs extend it by adding edges.

5.3.1 GNNs: adding relational structure

A graph is a set with relationships. We have nodes $\mathcal{V}$ and edges $\mathcal{E}$ connecting them. The key operation in graph neural networks is **message passing**: nodes exchange information with their neighbors.

The intuition. Each node starts knowing only about itself. After one round of message passing, it knows about its immediate neighbors. After two rounds, neighbors-of-neighbors. After k rounds, each node has information about all nodes within k hops.

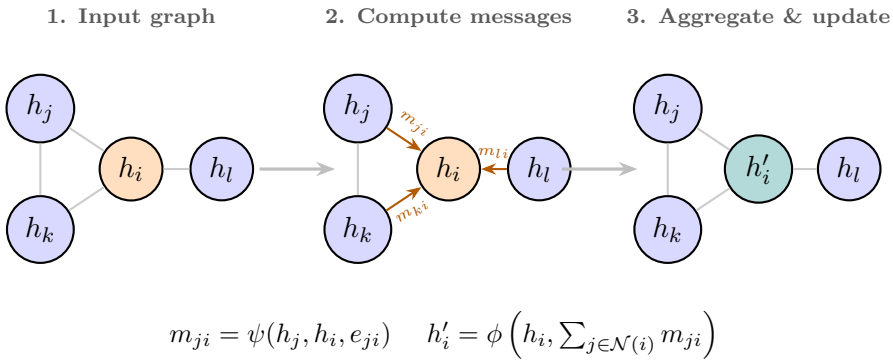


Figure 11. Message passing in three stages. (1) Start with node features on a graph. (2) Each node computes messages from its neighbors using a learned function ψ. (3) Messages are aggregated (summed) and combined with the node's own features to produce updated representations.

A simple message-passing update equation.

$$h_v^{(\ell+1)} = \phi\left(h_v^{(\ell)}, \sum_{u \in \mathcal{N}(v)} \psi(h_u^{(\ell)}, h_v^{(\ell)}, e_{uv})\right) \tag{5.9}$$

Unpacking this a bit:

1. **Message**: For each neighbor u of node v, compute a message $\psi(h_u, h_v, e_{uv})$. The message depends on sender, receiver, and edge features.

2. **Aggregate**: Sum messages over all neighbors. This is where permutation invariance comes from—sum doesn't depend on order.

3. **Update**: Combine aggregated messages with the node's current state using ϕ to get the new representation.

Both ψ and ϕ are learned networks (typically MLPs). The same networks apply to all nodes and edges—*parameter sharing across the graph.*

Hidden dimensions (channels). Like CNNs, GNNs have a notion of channels: the hidden dimension d of the node representations $h_v \in \mathbb{R}^d$. A node's "features" are a d-dimensional vector—you can think of each dimension as detecting a different pattern, just like channels in a CNN. More dimensions mean more expressive power but also more parameters.

Connection to Deep Sets and CNNs. Message passing generalizes both:

- **Deep Sets** is a GNN on a graph with no edges (or a fully-connected graph with uniform edges).
- **CNN** is a GNN on a grid graph where each pixel connects to its spatial neighbors. The kernel weights are the message function.

From nodes to graphs. Message passing produces node representations h_v. For graph-level predictions (is this molecule toxic?), *pool* the nodes:

$$h_{\mathcal{G}} = \sum_{v \in \mathcal{V}} h_v \tag{5.10}$$

This is exactly Deep Sets applied to the final node representations.

Graph attention. Standard message passing treats all neighbors equally (or weights them only by edge features). But different neighbors may have different importance—and the importance may depend on the *content* of the nodes, not just the graph structure. **Attention** learns to weight neighbors by relevance:

$$h'_v = \sum_{u \in \mathcal{N}(v)} \alpha_{uv} \cdot \psi(h_u) \tag{5.11}$$

The attention weights α_{uv} are computed from node features:

$$\alpha_{uv} = \frac{\exp(f(h_u, h_v))}{\sum_{w \in \mathcal{N}(v)} \exp(f(h_w, h_v))} \tag{5.12}$$

The scoring function f (typically a small network or inner product) computes how relevant neighbor u is for node v. The softmax ensures weights sum to 1.

Why attention helps. Consider predicting a molecule's reactivity. A carbon atom may have several neighbors, but not all matter equally—the oxygen in a carbonyl group matters more than a distant methyl. Attention learns these patterns from data.

Attention also provides some (limited) interpretability: the learned weights α_{uv} show which neighbors influenced each node's representation.

6 Attention as a unifying primitive

Attention is a general mechanism that unifies several architectures. The core operation is always the same: compute a weighted combination of inputs. What varies is *which* inputs participate and whether the weights are fixed or content-dependent. Figure 13 illustrates the four main patterns.

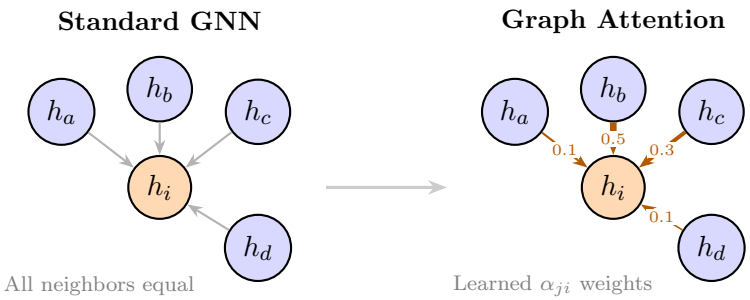


Figure 12. Graph attention learns which neighbors matter. Left: standard GNN weights all neighbors equally. Right: attention weights neighbors by learned relevance scores α_{ji}, allowing the network to focus on the most informative connections.

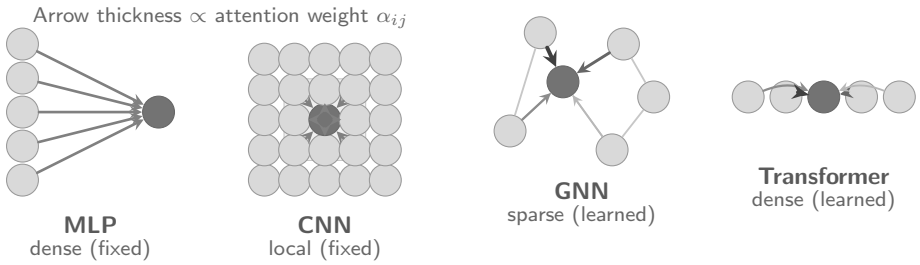


Figure 13. Attention patterns across architectures. Dark nodes are queries; arrows show which inputs contribute. **MLP**: all inputs, fixed weights. **CNN**: local inputs, fixed weights (same kernel everywhere). **GNN**: graph neighbors, learned weights. **Transformer**: all inputs, learned weights. Arrow thickness indicates attention weight.

The general form. Given a query q, keys $k_1, \ldots, k_n$, and values $v_1, \ldots, v_n$:

$$\text{output} = \sum_{i=1}^{n} \alpha_i \cdot v_i, \quad \text{where} \quad \alpha_i = \frac{\exp(q \cdot k_i)}{\sum_j \exp(q \cdot k_j)} \tag{6.1}$$

The query asks "what am I looking for?" The keys say "what do I contain?" The dot product measures relevance. This is *scaled dot-product attention*, the building block of Transformers.

CNNs as fixed local attention. A convolution computes:

$$h_{ij} = \sum_{a,b} W_{ab} \cdot x_{i+a,j+b} \tag{6.2}$$

This is attention where the weights W_{ab} are fixed (learned once, used everywhere) and the neighborhood is local (only nearby pixels). The "attention pattern" is the same at every position—a fixed 3×3 or 5×5 window.

GNNs as sparse attention. Graph attention computes:

$$h_v = \sum_{u \in \mathcal{N}(v)} \alpha_{uv} \cdot \psi(h_u) \tag{6.3}$$

The weights α_{uv} are content-dependent (computed from node features), but attention is restricted to graph neighbors. The graph structure determines *who* can attend to whom; the attention mechanism determines *how much*.

Transformers as dense attention. Self-attention in Transformers:

$$h_i = \sum_{j=1}^{n} \alpha_{ij} \cdot v_j, \quad \alpha_{ij} = \text{softmax}_j(q_i \cdot k_j / \sqrt{d}) \tag{6.4}$$

Every position attends to every other position. No locality constraint, no predefined graph. The attention weights are entirely learned from content. This is maximally flexible but costs $O(n^2)$ computation.

MLPs as structure-free mixing. Where do MLPs fit? A fully connected layer computes:

$$h_i = \sum_{j=1}^{n} W_{ij} \cdot x_j \tag{6.5}$$

Every output depends on every input, with fixed learned weights W_{ij}. This is like dense attention but without content-dependence—the "attention pattern" is baked into the weights at training time and doesn't adapt to the input. MLPs assume no structure: no locality, no graph, no notion of which inputs should interact. Maximum flexibility, but also maximum data requirements.

The tradeoff. These four represent different points on a structure-flexibility spectrum:

	MLP	**Transformer**	**GNN**	**CNN**
Pattern	Dense global	Dense global	Sparse (graph)	Fixed local
Weights	Fixed	Content-dep.	Content-dep.	Fixed
Connectivity	All pairs	All pairs	Graph neighbors	Grid neighbors
Structure assumed	None	Weak	Medium	Strong

Reading left to right: MLPs assume no structure (any input can affect any output). Transformers add content-dependent weights that adapt to each input. GNNs restrict connectivity to graph neighbors. CNNs add the strongest prior: local, translation-invariant patterns.

7 Why overparameterized networks work

Modern neural networks have far more parameters than training examples. A ResNet for ImageNet has 25 million parameters trained on 1.2 million images. Classical statistics says this should overfit catastrophically—with enough parameters, you can memorize the training set and learn nothing generalizable. Yet these networks generalize well. Why?

Traditional learning theory says: more parameters = more ways to fit noise = worse generalization. The optimal model complexity balances fitting the data against overfitting. Add parameters until validation error starts rising. Neural networks violate this. They can *interpolate*—fit the training data perfectly, even with noisy labels—yet still generalize. The test error doesn't explode when you add more parameters past the interpolation threshold. Instead, it often *decreases* (the "double descent" phenomenon).

Why interpolation doesn't necessarily overfit. Among all functions that fit the training data, SGD finds a particular one—and that one tends to be simple.

Think of it geometrically. In high dimensions, the set of parameters that perfectly fit the training data is a large subspace (when you have more parameters than constraints). Within this subspace, there are infinitely many solutions. Some are jagged and complex; others are smooth and simple. SGD, starting from small random weights and taking small steps, tends to find smooth solutions.

Overparameterization smooths the loss landscape. With more parameters, the loss landscape becomes better-behaved. Instead of one narrow valley, there are many paths to low loss, so optimization is less likely to get stuck. Overparameterized networks tend to find flat minima—regions where the loss is low over a large volume of parameter space—which correspond to simpler functions that generalize better. Good solutions aren't isolated points but form connected regions; you can move between different good solutions without crossing high-loss barriers.

Inductive bias from optimization. The architecture constrains *what* the network can represent. But the optimizer constrains *which* of those representable functions it actually finds. SGD with small learning rates, starting from small weights, has an implicit bias toward simpler functions. This "implicit regularization" is as important as the explicit architecture. (We'll come back to this when discussing symmetry-based architectures.)

What this means in practice. Don't fear overparameterization. A network with $10\times$ more parameters than data points may generalize better than a smaller one—if trained properly. The combination of architecture (encoding the right symmetries) and optimization (finding smooth solutions) produces good generalization even in the overparameterized regime.

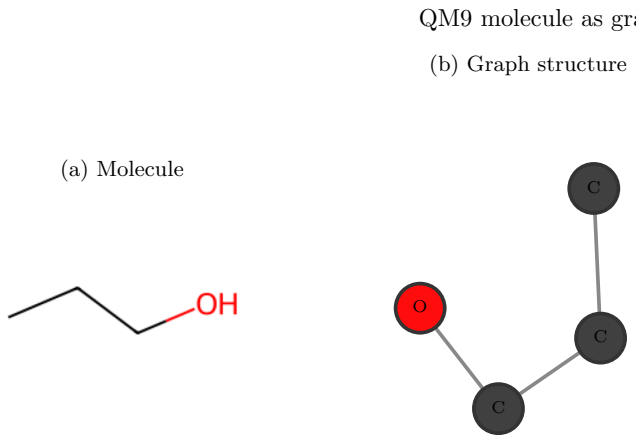
A The QM9 dataset

QM9 is a benchmark dataset of 134,000 small organic molecules with quantum-mechanical properties computed from density functional theory (DFT). Each molecule contains up to 9 heavy atoms (C, N, O, F) plus hydrogens. The dataset provides a concrete example of molecular property prediction: given a molecule's structure, predict its quantum properties without expensive DFT calculations.

Molecules as graphs. A molecule is naturally represented as a graph: atoms are nodes, chemical bonds are edges. This is the input to a GNN. Figure 14 shows the concrete representation.

Node features. Each atom is represented by an 11-dimensional feature vector:

Indices	Feature	Description
0–4	Atom type	One-hot encoding (H, C, N, O, F)
5	Atomic number	Integer (1, 6, 7, 8, 9)
6–8	Hybridization	One-hot (sp, sp^2, sp^3)
9	Aromatic	Binary (0 or 1)
10	Hydrogen count	Number of attached hydrogens



(c) Node features $\mathbf{h}_i \in \mathbf{R}^{11}$

Node	Atom	Z	nH
0	C	6	3
1	C	6	2
2	C	6	2
3	O	8	1

Figure 14. QM9 data representation. (a) A molecule visualized with RDKit. (b) The same molecule as a graph: heavy atoms are nodes, bonds are edges. Hydrogens are implicit—encoded in the nH feature rather than as explicit nodes. (c) Each node has an 11-dimensional feature vector; nH indicates the number of attached hydrogens.

Edge features. Each bond has a 4-dimensional feature vector: a one-hot encoding of bond type (single, double, triple, aromatic). Edges are stored bidirectionally—each bond (i, j) appears twice as directed edges $i \to j$ and $j \to i$.

Target properties. QM9 provides 19 quantum-mechanical properties for each molecule, including:

- **HOMO/LUMO energies**: Frontier orbital energies (eV)
- **HOMO-LUMO gap**: Electronic excitation energy (eV)
- **Dipole moment**: Molecular polarity (Debye)
- **Polarizability**: Response to electric field (Bohr^3)
- **Thermodynamic properties**: Internal energy, enthalpy, free energy, heat capacity

These properties determine a molecule's electronic, optical, and thermodynamic behavior. Predicting them from structure alone—bypassing expensive DFT calculations—is the goal of molecular property prediction.

QM9 is small enough to train on quickly but rich enough to benchmark different architectures.This makes QM9 a good benchmark for comparing how different inductive biases affect learning on the same underlying data.