Mathematical Foundations of Modern AI

A 5-day workshop

Yaé Ulrich Gaba

2026

Contents

Preface

Modern AI moves fast, but the ideas that actually power today's systems are stable and accessible. The same handful of mathematical structures — vector spaces, gradients, probability distributions, inductive bias, generalization — show up in every model we deploy, from a logistic regression to a diffusion model.

This 5-day workshop builds that mathematical intuition end-to-end. We start from refreshers in linear algebra and calculus, build toward the structures behind multilayer perceptrons, autoencoders, and diffusion models, and finish with the discipline of evaluating what we have built. The goal is not to turn participants into deep-learning specialists. It is to give them the mental structures to evaluate a method, talk to a technical team, frame a business problem as a learning problem, and decide when — and when not — to deploy an AI tool.

Who this is for. Engineers, product managers, analysts, and technical team leads who make decisions about AI. Practitioners who work alongside ML teams and want a real grasp of the foundations. Researchers from other disciplines who want to move from talking about AI to using it.

What you will be able to do after the workshop.

- Read a description of a neural network architecture and understand what each piece is doing mathematically.
- Reason about why a model is trained the way it is — and when a training choice signals a problem.
- Translate a real-world problem into the structured language of a learning problem.
- Identify the inductive bias built into a model and judge whether it matches the data.
- Recognize the standard failure modes (overfitting, distribution shift, calibration loss) and the standard remedies.
- Decide when AI is the right tool, when a simpler method is enough, and when more data or a different framing is the real fix.

Prerequisites. Comfortable with high-school and first-year undergraduate mathematics (vectors, functions, derivatives, basic probability). Some Python familiarity helps for the labs but is not required for the lectures.

Format. Five days. Each day pairs a lecture with a hands-on Jupyter notebook lab. All code runs on Google Colab (free tier with GPU) or any local Python 3.10+ environment with PyTorch and NumPy.

Yaé Ulrich Gaba
Cotonou, 2026

Chapter 1

Linear algebra and parametric models

> *"A neural network is a stack of linear maps with non-linear interruptions. The mathematics has been understood for a century; the engineering surprise is that it works at scale."*

A modern AI model is, at its core, a function. It takes an input vector $\boldsymbol{x} \in \mathbb{R}^d$ — pixels of an image, embeddings of a sentence, features of a customer — and returns an output: a class label, a probability, another vector. The vocabulary of that function is linear algebra. Vector spaces describe where the data lives. Linear maps move data between spaces. The non-linear ingredients (ReLU, softmax) are punctuation. Almost everything else about a network — depth, width, embedding dimension, latent space, attention heads — is a choice of how to compose linear maps.

This chapter sets up the linear-algebraic spine of the workshop. We will not prove the spectral theorem. We will name the structures, give the geometric intuition, and write the small amount of code needed to see them in action. Day 2 will introduce gradients to make these models trainable; Day 3 will add probability to make them generative. But the structures are all here.

1.1 Vector spaces, the short version

A *vector space* over $\mathbb{R}$ is a set V where you can add elements and multiply them by real numbers, and the usual rules hold (associativity, distributivity, a zero element). Every concrete example we care about in this workshop is some flavor of $\mathbb{R}^d$: ordered tuples of real numbers.

Definition 1.1 (The standard example)**.** *$\mathbb{R}^d$ is the set of d-tuples $\boldsymbol{x} = (x_1, \ldots, x_d)$ with addition and scalar multiplication done component-wise.*

Example 1.1. *A grayscale image of 28×28 pixels is an element of $\mathbb{R}^{784}$ once we flatten it. A sentence embedding from a small language model is an element of $\mathbb{R}^{768}$. A user's row in a recommender system's feature table is an element of $\mathbb{R}^{50}$ or so. These all look very different in the world but identical to the mathematics — they are points in a vector space, and the model treats them as such.*

The *dimension* of a vector space is the number of independent directions you can move in. $\mathbb{R}^{784}$ has 784 of them, one per pixel. In practice, the data does not fill the space: nearly all $\mathbb{R}^{784}$ vectors look like random noise, and almost none of them look like a digit. This observation — that real data lives on a much lower-dimensional subset — is the *manifold hypothesis*, and it is the reason embeddings work.

Intuition

A high-dimensional vector space is mostly empty. Modern AI exploits this: even when the ambient space has 10^4 dimensions, the structure we care about (the set of plausible images, the set of meaningful sentences) lives on a tiny, curved subset. Learning that subset is a large part of what training does.

1.2 Linear maps and matrices

A *linear map* $f : \mathbb{R}^d \to \mathbb{R}^k$ is a function that respects addition and scalar multiplication:

$$f(\boldsymbol{x} + \boldsymbol{y}) = f(\boldsymbol{x}) + f(\boldsymbol{y}), \qquad f(\alpha \boldsymbol{x}) = \alpha f(\boldsymbol{x}).$$

Every such map is given by a matrix $W \in \mathbb{R}^{k \times d}$: $f(\boldsymbol{x}) = W\boldsymbol{x}$. Pick a basis and the map becomes a list of numbers.

Definition 1.2 (Affine map)**.** *An affine map is a linear map plus a constant offset: $\boldsymbol{x} \mapsto W\boldsymbol{x} + \boldsymbol{b}$, where $\boldsymbol{b} \in \mathbb{R}^k$ is the bias. Every layer of a standard neural network is an affine map followed by a non-linearity.*

The matrix W does three things at once: it scales, it rotates, and it projects onto a (possibly lower-dimensional) subspace. The singular value decomposition makes this precise — any W can be written $U\Sigma V^\top$ where U and V are rotations and Σ is a non-negative diagonal scaling. We will not need the SVD machinery in detail, but the picture is worth holding:

Intuition

Every linear layer in a neural network rotates the input space, stretches some directions and shrinks others, and possibly throws some directions away (those with singular value zero). Stacking layers stacks these operations. The non-linearities are what prevent the stack from collapsing into a single linear map.

1.3 From the perceptron to the MLP

The simplest neural network is a single neuron, the *perceptron* (Rosenblatt, 1958):

$$y = \sigma(\boldsymbol{w}^\top \boldsymbol{x} + b)$$

where $\boldsymbol{w} \in \mathbb{R}^d$ is a vector of weights, $b \in \mathbb{R}$ is a bias, and σ is a non-linear function. With $\sigma(z) = 1/(1 + e^{-z})$ (the logistic sigmoid), this is exactly logistic regression dressed up in different vocabulary.

Proposition 1.1 (Why a single layer is not enough)**.** *A single affine layer can only separate input space by a hyperplane. The XOR function — output is 1 if exactly one of two binary inputs is 1 — cannot be expressed by any single perceptron. This is the original 1969 critique of Minsky and Papert that briefly killed the field.*

The fix is composition. A *multilayer perceptron* (MLP) stacks affine maps with non-linearities between them. A two-layer MLP from $\mathbb{R}^d$ to $\mathbb{R}^k$ is:

$$\boldsymbol{h} = \sigma(W_1\boldsymbol{x} + \boldsymbol{b}_1), \qquad \boldsymbol{y} = W_2\boldsymbol{h} + \boldsymbol{b}_2$$

where $W_1 \in \mathbb{R}^{m\times d}$, $W_2 \in \mathbb{R}^{k\times m}$, and m is the width of the hidden layer. The non-linearity σ is applied component-wise. Modern networks use $\sigma(z) = \max(0, z)$, the rectified linear unit (ReLU), almost everywhere.

Note

The universal approximation theorem (Cybenko, 1989; Hornik, 1991) says that even a one-hidden-layer MLP with enough width can approximate any continuous function on a compact set to arbitrary accuracy. This is a statement about *existence*, not about *learnability* or efficiency. Modern deep networks are not deep because shallow networks cannot represent the function. They are deep because depth makes the right functions easier to find by gradient descent.

1.3.1 Counting parameters

Take an MLP with input dimension $d = 784$, one hidden layer of width $m = 256$, and output dimension $k = 10$. The parameter count is:

$$\underbrace{784 \cdot 256 + 256}_{\text{first layer: } W_1,\boldsymbol{b}_1} + \underbrace{256 \cdot 10 + 10}_{\text{second layer: } W_2,\boldsymbol{b}_2} = 203\,530.$$

A small MNIST classifier already has a couple hundred thousand parameters. GPT-3 has 175 billion. The numbers grow fast, but the structure does not change — every layer is still an affine map followed by a non-linearity.

1.3.2 The MLP in code

```python
import torch
import torch.nn as nn

class TwoLayerMLP(nn.Module):
    def __init__(self, d_in: int, d_hidden: int, d_out: int):
        super().__init__()
        self.fc1 = nn.Linear(d_in, d_hidden)   # affine map: W1 x + b1
        self.fc2 = nn.Linear(d_hidden, d_out)  # affine map: W2 h + b2

    def forward(self, x: torch.Tensor) -> torch.Tensor:
```

```
        h = torch.relu(self.fc1(x))            # non-linearity
        y = self.fc2(h)
        return y

model = TwoLayerMLP(d_in=784, d_hidden=256, d_out=10)
n_params = sum(p.numel() for p in model.parameters())
print(f"Number of parameters: {n_params:,}")
```

The `nn.Linear` layer holds exactly the W and $\boldsymbol{b}$ from the formulas above. Everything else in this workshop is variations on this theme: more layers, different non-linearities, weight-sharing (CNNs), structured connections (GNNs), attention (Transformers). The base unit is the affine map.

Exercise

A residual block in a ResNet has the form $\boldsymbol{y} = \boldsymbol{x} + F(\boldsymbol{x})$, where F is a small sub-network. Argue informally why this should make training easier than the bare composition $\boldsymbol{y} = F(\boldsymbol{x})$. (Hint: what happens if F outputs zero? What does that imply for the gradient?)

1.4 Embeddings: vectors as representations

Up to this point, we have treated the input $\boldsymbol{x}$ as if it were already a vector. But the real input is often something else: a word, a user ID, a discrete category. Turning it into a vector is the first decision the model makes.

Definition 1.3 (Embedding)**.** *An embedding is a function from a discrete set V (vocabulary, set of users, set of categories) into a vector space $\mathbb{R}^d$. Concretely it is a matrix $E \in \mathbb{R}^{|V| \times d}$: each row is the vector representation of one item.*

The embedding matrix is a parameter of the model. It is learned. After training, similar items end up close in the embedding space, and the geometry of that space carries the meaning the model has discovered.

Example 1.2. *Word2Vec (Mikolov et al., 2013) trains an embedding matrix for English words so that the cosine similarity of two word vectors tracks their semantic similarity. The famous geometric regularity $\boldsymbol{e}_{king} - \boldsymbol{e}_{man} + \boldsymbol{e}_{woman} \approx \boldsymbol{e}_{queen}$ is an empirical observation about the resulting space, not an objective the model was trained on. The training objective was much simpler: predict a word from its context.*

In modern language models, the input embedding is the first layer (a lookup in an embedding matrix), and the rest of the network operates on those vectors. The trick is that the same vector space is doing double duty: it is the input space *and* the model's working memory.

Caution

"Embedding" is overloaded in machine learning. It can mean a learned input representation (Word2Vec, an LLM's token embedding), a hidden activation inside a network ("the embedding of this image is the output of the second-to-last layer"), or a separately trained representation used as a feature in a downstream model (sentence-transformers). All three are vectors. The differences are about how they are produced and what they are used for.

1.5 Autoencoders and the geometry of latent space

An *autoencoder* learns a compressed representation of its input by trying to reproduce that input from the compression. It has two parts:

$$\underbrace{\boldsymbol{z} = f_{\text{enc}}(\boldsymbol{x})}_{\text{encoder, } \mathbb{R}^d \to \mathbb{R}^k} \qquad \underbrace{\hat{\boldsymbol{x}} = f_{\text{dec}}(\boldsymbol{z})}_{\text{decoder, } \mathbb{R}^k \to \mathbb{R}^d}$$

with $k < d$. The training objective is reconstruction: minimize $\|\boldsymbol{x} - \hat{\boldsymbol{x}}\|^2$ over a dataset. Because the bottleneck $\boldsymbol{z}$ is lower-dimensional than $\boldsymbol{x}$, the autoencoder cannot memorize the input; it has to find structure.

Intuition

PCA is a linear autoencoder. If f_{enc} and f_{dec} are forced to be linear maps and we minimize squared reconstruction error, the optimal solution projects onto the top-k principal components. Non-linear autoencoders generalize this idea: they learn a curved k-dimensional surface inside $\mathbb{R}^d$ that fits the data.

The bottleneck space $\mathbb{R}^k$ is called the *latent space*. It is where the model's compressed understanding of the data lives. Two questions are usually worth asking about any latent space:

- **What does each dimension mean?** Often nothing on its own — the axes are arbitrary — but directions in the space often correspond to semantic factors (lighting, pose, identity for a face dataset; topic, sentiment, tense for a text dataset).
- **How does it transport between nearby points?** If small movements in $\boldsymbol{z}$ produce small, plausible changes in $\hat{\boldsymbol{x}}$, the model has learned the right manifold. If small movements produce jumps or nonsense, it has not.

Day 3 will return to autoencoders in their probabilistic form (variational autoencoders, VAEs) as one of the bridges to generative modelling. The autoencoder structure — encoder, latent, decoder — is also exactly the structure of a diffusion model in disguise.

1.6 What you have seen

Every parametric model in this workshop is built from three pieces: vectors (the data and intermediate representations), affine maps (the layers, with their weights and biases), and non-linearities (the activations that prevent collapse). Modern architectures — CNNs, RNNs, Transformers, GNNs, diffusion models — are variations on which affine maps are tied together and which non-linearities are inserted where. The names are different. The mathematics is the same.

What we have not yet done is make any of this *learn*. So far the weights W and biases $\boldsymbol{b}$ are just numbers; the model is a function. Day 2 introduces the gradient, the loss function, and the optimization machinery that turns this function into something that can be fit to data. The structure stays the same. Training is what fills it in.

Lab

Open the Day 1 notebook (`Day1_LinearAlgebra_ParametricModels.ipynb`). You will:

1. Implement a two-layer MLP by hand in NumPy — writing the matrix multiplications and ReLU explicitly — and verify the parameter count matches what we computed above.
2. Train the same MLP in PyTorch on MNIST, getting roughly 97% test accuracy in under a minute.
3. Extract the activations of the hidden layer for the test set, project them to 2D with PCA, and color the points by their true digit class. You will see clusters — the network has learned a representation in which similar digits live near each other.
4. Train a small autoencoder on MNIST with a 2-dimensional latent space. Visualize the latent space directly and walk along straight lines in it: you will see digits morph smoothly into other digits.

By the end of the lab you will have built the machinery that the rest of the workshop assumes: a working MLP, a working autoencoder, and the habit of inspecting what a network has learned by looking at its internal representations.

Chapter 2

Calculus and optimization

> *"Training a neural network is the chain rule, applied carefully, many times per second."*

At the end of Day 1 we had a model: an MLP, a stack of affine maps and ReLUs, with weights W and biases $\boldsymbol{b}$ that were just random numbers. The model was a function. It did nothing useful. To make it useful we need to adjust those numbers so the model's outputs match the data — and we need to do that adjustment automatically, because there are too many numbers to set by hand.

This chapter is about that adjustment. The mathematics is calculus: derivatives say how a function's output changes when you nudge its inputs, and gradients generalize that idea to functions of many variables. The algorithm is gradient descent: take a small step in the direction that decreases the loss. The engineering trick that makes it tractable for million-parameter networks is the chain rule, applied recursively — backpropagation. By the end of the day you will have implemented backpropagation by hand, in NumPy, for the MLP we built yesterday.

2.1 The loss function

Before we can optimize anything we have to say what we are optimizing. We need a single real number that measures *how wrong* the model is on a dataset. That number is the *loss*.

Definition 2.1 (Loss function)**.** *Given a model f_θ with parameters θ and a dataset $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$, a loss function $\mathcal{L}(\theta)$ is a real-valued function of θ that is small when the model fits the data well and large when it does not. Training is the optimization problem* $\min_\theta \mathcal{L}(\theta)$.

Two losses cover most of this workshop:

Mean squared error (regression). For real-valued targets:

$$\mathcal{L}_{\text{MSE}}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \| f_\theta(\boldsymbol{x}_i) - y_i \|^2.$$

This is the squared distance between prediction and target, averaged over the dataset.

Cross-entropy (classification). For class labels $y_i \in \{1, \dots, K\}$, the model outputs a probability distribution $f_\theta(\boldsymbol{x}_i) \in \Delta^{K-1}$ over classes (via softmax), and:

$$\mathcal{L}_{\text{CE}}(\theta) = -\frac{1}{N} \sum_{i=1}^{N} \log f_\theta(\boldsymbol{x}_i)_{y_i}.$$

This is the negative log-likelihood under the model's predicted distribution. Day 3 explains why *this* loss and not some other.

Intuition

The loss is a landscape. The parameters θ are coordinates. Training is the act of finding a low point. The landscape has millions of dimensions (one per parameter) but is otherwise just a function $\mathcal{L} : \mathbb{R}^P \to \mathbb{R}$. Everything in this chapter is about how to walk downhill on that landscape efficiently.

2.2 Derivatives and gradients

For a scalar function $f : \mathbb{R} \to \mathbb{R}$, the derivative $f'(x)$ is the rate of change at x: if we nudge x by ϵ, the output changes by approximately $f'(x) \cdot \epsilon$. The gradient generalizes this to functions of many variables.

Definition 2.2 (Gradient)**.** *For a function $f : \mathbb{R}^d \to \mathbb{R}$, the gradient at $\boldsymbol{x}$ is the vector of partial derivatives:*

$$\nabla f(\boldsymbol{x}) = \left(\frac{\partial f}{\partial x_1}(\boldsymbol{x}),\ \frac{\partial f}{\partial x_2}(\boldsymbol{x}),\ \dots,\ \frac{\partial f}{\partial x_d}(\boldsymbol{x}) \right).$$

The gradient points in the direction of steepest ascent*. The opposite direction, $-\nabla f(\boldsymbol{x})$, is steepest descent.*

This is the key geometric fact behind every optimization algorithm in this workshop: to decrease a function, follow its negative gradient.

Example 2.1. *Let $f(x_1, x_2) = x_1^2 + 3x_2^2$. The gradient is $\nabla f = (2x_1, 6x_2)$. At the point $(1, 1)$ the gradient is $(2, 6)$. The function increases most rapidly in the direction $(2, 6)/\|(2, 6)\|$ and decreases most rapidly in the opposite direction. Both axes point outward from the minimum at $(0, 0)$, but the x_2 direction is steeper, because the function curves more steeply along x_2.*

2.3 Gradient descent

The simplest optimization algorithm is the one suggested by the previous section: start somewhere, compute the gradient, step in the negative-gradient direction, repeat.

Definition 2.3 (Gradient descent)**.** *Given a learning rate $\eta > 0$ and an initial point θ_0, gradient descent produces a sequence*

$$\theta_{t+1} = \theta_t - \eta \, \nabla \mathcal{L}(\theta_t).$$

The learning rate η controls how big a step we take. If η is too small, training is slow. If η is too large, the steps overshoot and the loss oscillates or diverges.

Caution

The learning rate is the single most consequential hyperparameter in deep learning. "My model is not training" is, in practice, almost always "my learning rate is wrong." The lab today includes a learning-rate sweep so you can see this directly.

2.4 The chain rule and backpropagation

For an MLP with two layers, the loss is a composition of functions:

$$\mathcal{L}(\theta) = \ell\big(f_2(f_1(\boldsymbol{x}))\big)$$

where f_1 is the first affine-plus-ReLU layer, f_2 is the second, and ℓ is the loss applied to the output. To do gradient descent on $\mathcal{L}$ we need the gradient with respect to every parameter in f_1 and f_2.

The chain rule from single-variable calculus generalizes to vector-valued functions. If $\boldsymbol{y} = g(\boldsymbol{u})$ and $\boldsymbol{u} = h(\boldsymbol{x})$, then

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{u}} \cdot \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{x}}$$

where the partial derivatives are Jacobian matrices and the product is matrix multiplication. Applied repeatedly to the composition above, this lets us compute $\partial \mathcal{L}/\partial \theta$ for every parameter.

Intuition

The forward pass computes the loss from inputs to scalar output. The backward pass computes derivatives in the opposite direction: it starts from the scalar loss (whose derivative with respect to itself is 1) and propagates partial derivatives back through every layer, multiplying Jacobians as it goes. The total work of the backward pass is roughly the same as the forward pass — a factor of 2 to 3, depending on the network. This is why training is feasible at all.

2.4.1 Backprop for a 2-layer MLP, by hand

For the two-layer MLP from Day 1:

$$\begin{aligned} \boldsymbol{h}_{\text{pre}} &= W_1 \boldsymbol{x} + \boldsymbol{b}_1 \\ \boldsymbol{h} &= \text{ReLU}(\boldsymbol{h}_{\text{pre}}) \\ \boldsymbol{y} &= W_2 \boldsymbol{h} + \boldsymbol{b}_2 \end{aligned}$$

with cross-entropy loss after a softmax, the gradients have a closed form. Let $\boldsymbol{p} = \mathrm{softmax}(\boldsymbol{y})$ and $\boldsymbol{y}^*$ be the one-hot target. Then:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \boldsymbol{y}} &= \boldsymbol{p} - \boldsymbol{y}^* && \text{(softmax-CE collapses)} \\
\frac{\partial \mathcal{L}}{\partial W_2} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{y}} \cdot \boldsymbol{h}^\top, \quad \frac{\partial \mathcal{L}}{\partial \boldsymbol{b}_2} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{y}} \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}} &= W_2^\top \cdot \frac{\partial \mathcal{L}}{\partial \boldsymbol{y}} \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_{\text{pre}}} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}} \odot \mathbf{1}[\boldsymbol{h}_{\text{pre}} > 0] && \text{(ReLU gradient)} \\
\frac{\partial \mathcal{L}}{\partial W_1} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_{\text{pre}}} \cdot \boldsymbol{x}^\top, \quad \frac{\partial \mathcal{L}}{\partial \boldsymbol{b}_1} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_{\text{pre}}}.
\end{aligned}$$

Five lines. That is backpropagation for a two-layer MLP. The lab implements these formulas in NumPy and checks them against PyTorch's autograd to the bit.

2.4.2 Why we never write this code in practice

For a two-layer MLP we could derive these formulas. For a fifty-layer ResNet, or for a Transformer with cross-attention, the algebra is hopeless. Modern frameworks (PyTorch, JAX, TensorFlow) record the computational graph of the forward pass and replay it backward, applying the chain rule at each node. Every primitive operation (matmul, ReLU, softmax) has a hand-coded backward; the framework composes them automatically. This is what *differentiable programming* means.

> **Note**
>
> You will never write the backward pass for a production model. But you should understand what it is, because almost every training failure — vanishing gradients, exploding gradients, dead ReLUs, gradient checkpointing trade-offs — is a story about backpropagation. "My loss is NaN" is a story about backpropagation.

2.5 Stochastic gradient descent

The loss is a sum over the dataset:

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \ell_i(\theta).$$

Computing $\nabla \mathcal{L}$ exactly requires a full pass over the dataset. For MNIST that is 60 000 examples per step. For ImageNet, 1.2 million. For an LLM pretrained on 10^{12} tokens, the number is meaningless.

Definition 2.4 (Stochastic gradient descent, SGD)**.** *At each step, sample a small batch $B \subset \{1, \dots, N\}$ and update with the batch gradient:*

$$\theta_{t+1} = \theta_t - \eta \cdot \frac{1}{|B|} \sum_{i \in B} \nabla \ell_i(\theta_t).$$

The batch gradient is a noisy but unbiased estimate of the full gradient.

Two things make SGD work in practice:

1. **Computational.** A batch of 128 fits in memory, runs in milliseconds, and gives a usable step. Full-batch gradient descent would be a hundred times slower per step for the same compute budget.
2. **Statistical.** The noise in the gradient estimate acts as implicit regularization, helping the optimizer escape narrow minima. This is part of why neural networks generalize at all (Hardt et al., 2016; Keskar et al., 2017).

2.6 Adam: momentum and adaptive step sizes

Plain SGD is the simplest optimizer, but rarely the best one in practice. Adam (Kingma & Ba, 2015) adds two ingredients:

Momentum. Instead of taking a step in the direction of the current gradient, take a step in the direction of an exponentially-weighted average of recent gradients. This smooths out noise and accelerates progress along consistent directions.

Per-parameter learning rates. Each parameter gets its own effective learning rate, based on a running estimate of the magnitude of *its* recent gradients. Parameters with consistently small gradients take larger steps; parameters with consistently large gradients take smaller steps.

The update rule, with $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$:

$$\begin{aligned} \boldsymbol{m}_t &= \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\nabla\mathcal{L}(\theta_t) && \text{(momentum)} \\ \boldsymbol{v}_t &= \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)(\nabla\mathcal{L}(\theta_t))^2 && \text{(squared-gradient running mean)} \\ \theta_{t+1} &= \theta_t - \eta \cdot \frac{\hat{\boldsymbol{m}}_t}{\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon} \end{aligned}$$

where $\hat{\boldsymbol{m}}, \hat{\boldsymbol{v}}$ are bias-corrected versions of $\boldsymbol{m}, \boldsymbol{v}$. The square root and division are element-wise.

Adam is the default optimizer for almost every modern network. SGD with momentum still beats it for some computer-vision benchmarks (Wilson et al., 2017), but Adam's tolerance of bad hyperparameter choices is what makes it the practical default.

2.7 Reading a learning curve

A learning curve plots loss (or accuracy) against training step. It is the diagnostic instrument of deep learning. Four characteristic shapes:

- **Healthy:** train loss decreases smoothly, validation loss decreases too and tracks within a small gap. The model is learning, and what it learns generalizes.
- **Overfitting:** train loss decreases, validation loss decreases for a while and then starts to climb. The model is memorizing training examples instead of learning the underlying pattern. Day 5 returns to this.
- **Underfitting:** both losses plateau high. The model lacks capacity, the optimizer is stuck, or the learning rate is wrong.
- **Diverging:** loss spikes upward, often to infinity. The learning rate is too large, or the loss is being computed on a non-differentiable region (log of zero, division by zero in attention).

Intuition

Reading the learning curve is the first thing to do when training does not work. Most pathologies have a characteristic shape, and the curve tells you which one you are looking at within seconds. Today's lab generates each of these shapes deliberately so you can recognize them next time.

2.8 What you have seen

Training a neural network is gradient descent on a high-dimensional loss landscape, made tractable by the chain rule (backpropagation) and made practical by stochastic mini-batches and adaptive optimizers like Adam. The mathematics is calculus; the engineering is automatic differentiation. Every modern framework gives you the second for free, but only the first lets you reason about what is going wrong when something does.

Day 3 adds the missing ingredient: probability. We have been speaking of "predicting" outputs and "matching" targets, but generative models do something stranger — they learn an entire probability distribution and sample from it. That requires us to take the loss function seriously as a likelihood.

Lab

Open the Day 2 notebook (`Day2_Calculus_Optimization.ipynb`). You will:

1. Compute the gradient of a small function by hand, then verify against PyTorch's autograd.
2. Add a manual backward pass to the NumPy MLP from Day 1, then check every gradient against PyTorch's autograd to the last decimal — the standard gradient-check trick used to debug new layers.
3. Train the manual MLP on MNIST with vanilla SGD and your own training loop. No frameworks.

4. Replace the manual training loop with PyTorch + Adam and observe the difference.
5. Sweep the learning rate over $\{10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}\}$ and plot the resulting learning curves on a single axis. You will see all four characteristic shapes from above.

By the end of the lab you will have written and verified a backward pass, trained a network from first principles, and built the habit of looking at the learning curve *before* blaming the model.

Chapter 3

Probability and generative modelling

> *"A generative model is a probability distribution you can sample from. Everything else — the loss function, the architecture, the training algorithm — is in service of learning that distribution from data."*

Days 1 and 2 built a function: an MLP that maps inputs to outputs, with weights trained by gradient descent on a loss. The setup was deterministic. A given input produced a given output. That picture is enough for classification and regression, but it does not capture what modern generative models do. A diffusion model does not predict a single output from an input; it samples one image (or many different images) from a learned distribution. A language model does not predict one next token; it gives you a distribution over the vocabulary, and the sampler draws from it.

To talk about that, we need probability. This chapter is the bridge: it reintroduces the loss function from Day 2 as a likelihood, develops the variational autoencoder as the simplest non-trivial generative model, and ends with the denoising diffusion picture that underlies essentially every state-of-the-art image and audio generator built in the last five years.

3.1 Distributions, densities, and what we are trying to learn

A probability distribution assigns weight to every possible outcome. For a finite set, a distribution is a list of non-negative numbers summing to 1. For a continuous space like $\mathbb{R}^d$, it is a density function $p(\boldsymbol{x}) \geq 0$ whose integral over the whole space is 1.

Definition 3.1 (Density)**.** *A probability density $p : \mathbb{R}^d \to \mathbb{R}_{\geq 0}$ satisfies $\int p(\boldsymbol{x})\, d\boldsymbol{x} = 1$. For any region $A \subseteq \mathbb{R}^d$, the probability that a sample falls in A is $\int_A p(\boldsymbol{x})\, d\boldsymbol{x}$.*

The data we care about — images, texts, molecules — comes from some unknown distribution p_{data}. Generative modelling means: estimate p_{data} from samples, well enough to generate new samples that look like they came from it.

Intuition

For MNIST, p_{data} is a density on $\mathbb{R}^{784}$. Almost every point in $\mathbb{R}^{784}$ is a noise image with density essentially zero. Real digits live on a tiny, curved subset (the manifold from Day 1), where the density is enormous. A generative model is something that has internalized the shape of that high-density region.

3.2 Likelihood: a loss with a meaning

Suppose we have a parametric family of densities $\{p_\theta : \theta \in \Theta\}$. Given samples $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$ drawn from p_{data}, the *likelihood* is the probability density that the model assigns to the data:

$$\mathcal{L}_{\text{lik}}(\theta) = \prod_{i=1}^{N} p_\theta(\boldsymbol{x}_i).$$

Maximum likelihood estimation (MLE) picks the θ that makes the data look most plausible under p_θ. In practice we work with the negative log-likelihood, because it turns the product into a sum and gives us something to minimize:

$$-\log \mathcal{L}_{\text{lik}}(\theta) = -\sum_{i=1}^{N} \log p_\theta(\boldsymbol{x}_i).$$

Proposition 3.1 (Cross-entropy is negative log-likelihood)**.** *For a classifier with softmax output $p_\theta(y \mid \boldsymbol{x})$, the cross-entropy loss from Day 2,*

$$\mathcal{L}_{CE}(\theta) = -\frac{1}{N} \sum_{i=1}^{N} \log p_\theta(y_i \mid \boldsymbol{x}_i),$$

is exactly $-\frac{1}{N} \log \mathcal{L}_{lik}(\theta)$. Training a classifier with cross-entropy is maximum likelihood on the conditional distribution $p_\theta(y \mid \boldsymbol{x})$.

The classifier from Day 2 is already a generative model — of labels conditional on inputs. To generate *inputs* we need a model of the input distribution itself. That is what the rest of this chapter is about.

3.3 KL divergence: distance between distributions

To compare two distributions we use the Kullback–Leibler divergence:

$$D_{\text{KL}}(q \,\|\, p) = \int q(\boldsymbol{x}) \log \frac{q(\boldsymbol{x})}{p(\boldsymbol{x})} \, d\boldsymbol{x} = \mathbb{E}_{\boldsymbol{x} \sim q}[\log q(\boldsymbol{x}) - \log p(\boldsymbol{x})] \,.$$

KL is non-negative, zero if and only if $p = q$, and asymmetric. It is not a metric — $D_{\text{KL}}(p \,\|\, q) \neq D_{\text{KL}}(q \,\|\, p)$ in general — but it is the natural object when one of the distributions is the data and the other is the model.

Proposition 3.2 (MLE minimizes KL to the data)**.** *Up to a constant that does not depend on θ, the negative log-likelihood equals $D_{\mathrm{KL}}(p_{data} \,\|\, p_\theta)$. Maximum likelihood is approximate KL minimization between the data distribution and the model distribution.*

This is the cleanest statement of what training a generative model is doing: making the model distribution close to the data distribution in the KL sense. The rest of the chapter is two stories about *how* to do that minimization when p_θ is a neural network.

3.4 Variational autoencoders

Recall the autoencoder from Day 1: an encoder $f_{\mathrm{enc}} : \mathbb{R}^d \to \mathbb{R}^k$ and a decoder $f_{\mathrm{dec}} : \mathbb{R}^k \to \mathbb{R}^d$, with $k < d$, trained to minimize $\|\boldsymbol{x} - f_{\mathrm{dec}}(f_{\mathrm{enc}}(\boldsymbol{x}))\|^2$. The autoencoder learns a latent space, but it has no notion of probability — you cannot sample from it. The variational autoencoder (VAE, Kingma & Welling, 2014) is the probabilistic cousin.

3.4.1 The model

The VAE assumes data is generated as follows:

1. Sample a latent vector $\boldsymbol{z}$ from a simple prior, typically $p(\boldsymbol{z}) = \mathcal{N}(\mathbf{0}, I)$.
2. Pass $\boldsymbol{z}$ through a neural network decoder f_{dec} to get parameters of a distribution $p_\theta(\boldsymbol{x} \mid \boldsymbol{z})$ over data (e.g., a Gaussian with mean $f_{\mathrm{dec}}(\boldsymbol{z})$).
3. Sample $\boldsymbol{x}$ from $p_\theta(\boldsymbol{x} \mid \boldsymbol{z})$.

The model distribution is the marginal $p_\theta(\boldsymbol{x}) = \int p_\theta(\boldsymbol{x} \mid \boldsymbol{z})\, p(\boldsymbol{z})\, d\boldsymbol{z}$. This integral is intractable for a neural-network decoder, so we cannot evaluate $\log p_\theta(\boldsymbol{x})$ directly. The VAE introduces a second neural network — an *encoder* $q_\phi(\boldsymbol{z} \mid \boldsymbol{x})$, typically Gaussian with mean and variance output by f_{enc} — and optimizes a lower bound on the log-likelihood.

3.4.2 The Evidence Lower BOund (ELBO)

For any choice of q_ϕ,

$$\log p_\theta(\boldsymbol{x}) \geq \underbrace{\mathbb{E}_{q_\phi(\boldsymbol{z}|\boldsymbol{x})}[\log p_\theta(\boldsymbol{x} \mid \boldsymbol{z})]}_{\text{reconstruction term}} - \underbrace{D_{\mathrm{KL}}(q_\phi(\boldsymbol{z} \mid \boldsymbol{x}) \,\|\, p(\boldsymbol{z}))}_{\text{regularizer to the prior}}.$$

This is the ELBO. Maximizing the ELBO with respect to θ and ϕ jointly is the training objective. The first term wants the decoder to reconstruct $\boldsymbol{x}$ well from a $\boldsymbol{z}$ drawn from the encoder. The second term wants the encoder's posterior to look like the prior.

Intuition

The VAE is an autoencoder with two changes. First, the encoder outputs a distribution, not a single $\boldsymbol{z}$ — you sample to get the latent. Second, an extra loss term pulls the encoder distributions toward the standard Gaussian prior. Together these changes make the latent space *generative*: samples from the prior, decoded, look like data, because the encoder has been pushed to use the same region of latent space the prior covers.

3.4.3 The reparameterization trick

The expectation $\mathbb{E}_{q_\phi(\boldsymbol{z}|\boldsymbol{x})}[\cdot]$ is taken over a random $\boldsymbol{z}$, and the encoder produces the parameters of that randomness. We cannot directly differentiate through random sampling. The trick (Kingma & Welling, 2014): write $\boldsymbol{z} = \boldsymbol{\mu}_\phi(\boldsymbol{x}) + \boldsymbol{\sigma}_\phi(\boldsymbol{x}) \odot \boldsymbol{\epsilon}$ with $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, I)$. Now the randomness is in $\boldsymbol{\epsilon}$, which has no parameters; $\boldsymbol{\mu}_\phi$ and $\boldsymbol{\sigma}_\phi$ are deterministic and differentiable. Backprop flows through.

The lab today trains a small VAE on MNIST and visualizes both the latent space and what happens when you decode a smooth path through it.

3.5 Denoising diffusion

Diffusion models (Sohl-Dickstein et al., 2015; Ho et al., 2020) take a different route to the same destination. Instead of trying to learn the data distribution directly, they learn to *reverse* a fixed noising process.

3.5.1 The forward process

Start from a real image $\boldsymbol{x}_0$. Define a sequence of corrupted versions $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T$ by adding a small amount of Gaussian noise at each step:

$$\boldsymbol{x}_t = \sqrt{1-\beta_t}\,\boldsymbol{x}_{t-1} + \sqrt{\beta_t}\,\boldsymbol{\epsilon}_t, \qquad \boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, I).$$

The variances β_t are small and chosen so that by $t = T$ (often $T = 1000$), $\boldsymbol{x}_T$ is essentially pure Gaussian noise. This forward process has no parameters; it is just a noise schedule.

A useful closed-form consequence: for any t, we can write $\boldsymbol{x}_t$ directly in terms of $\boldsymbol{x}_0$ and a single noise vector:

$$\boldsymbol{x}_t = \sqrt{\bar{\alpha}_t}\,\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\,\boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, I),$$

where $\bar{\alpha}_t = \prod_{s=1}^{t}(1-\beta_s)$. This is what makes training efficient — no need to simulate the chain step by step.

3.5.2 The reverse process

We train a neural network $\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)$ that takes a noisy image and a timestep and predicts the noise that was added. The training loss is plain MSE:

$$\mathcal{L}_{\text{diff}}(\theta) = \mathbb{E}_{\boldsymbol{x}_0, t, \boldsymbol{\epsilon}}\left[\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)\|^2\right]$$

where $\boldsymbol{x}_t$ is the noisy version of $\boldsymbol{x}_0$ at timestep t using the closed-form formula above. Once the network can predict the noise, we can subtract it: given a noisy $\boldsymbol{x}_t$, recover an estimate of $\boldsymbol{x}_0$ (or, more carefully, of $\boldsymbol{x}_{t-1}$).

Note

The training objective is just MSE on noise prediction. There is no log-likelihood, no KL divergence in the loss, no variational bound to negotiate. Diffusion's empirical advantage over VAEs is largely that this loss is much better behaved than the ELBO. The deep reason it works comes from the connection to score matching (Hyvärinen, 2005; Song & Ermon, 2019): the noise prediction is, up to a known scaling, an estimate of the gradient of the log-density of the data distribution at the noised point.

3.5.3 Sampling

To generate a new image, start from pure noise $\boldsymbol{x}_T \sim \mathcal{N}(\mathbf{0}, I)$ and run the reverse process: at each step, use the network to predict the noise, then take one step closer to a clean image. After T steps you have a sample from the model's approximation to p_{data}. Modern samplers (DDIM, Karras et al.) reduce the number of required steps from thousands to tens, at little loss of quality.

Intuition

A trained diffusion model is a denoiser that knows what data looks like at every noise level. Sampling is just denoising in reverse, starting from pure noise and ending at something the model considers a valid sample. The lab today trains this picture from scratch on a 2D spiral so you can watch the noise become a spiral in real time.

3.6 Why probability changes the picture

The deterministic networks of Days 1 and 2 were function approximators. The probabilistic networks of Day 3 are distribution approximators. The difference is consequential:

- A classifier tells you the probability of each label. A diffusion model tells you the probability of each possible image.
- A classifier is queried once per input. A generative model is sampled — it produces something different each time you call it.

- A classifier's failure is wrong predictions. A generative model's failure is producing samples that the data distribution would never produce, or failing to cover regions that it does.

The mathematical objects (loss, gradient, optimizer) are the same. The interpretation is different, and the design choices that follow — what to model, what to condition on, how to sample — are different.

3.7 What you have seen

The cross-entropy loss from Day 2 is negative log-likelihood. Maximum likelihood is approximate KL minimization between data and model. The VAE optimizes a tractable lower bound on the log-likelihood and uses a reparameterized sampling trick to backprop through randomness. Diffusion models bypass the likelihood entirely, training a denoiser by plain MSE and sampling by reversing the noising process.

These three ideas — likelihood, ELBO, denoising score — cover essentially every generative model used in production: language models (next-token likelihood), Stable Diffusion (denoising), image VAEs (the perceptual compression stages of latent diffusion). Day 4 returns to a question we have ducked: when the data has structure (graphs, sequences, images on a regular grid), what architectures encode that structure into the model?

Lab

Open the Day 3 notebook (`Day3_Probability_GenerativeModels.ipynb`). You will:

1. Sample from a Gaussian, fit one to data, and compute KL between two Gaussians by hand and by Monte Carlo — so you see why the closed-form matters.
2. Train a small VAE on MNIST. Visualize the 2D latent space, sample from it, and decode a path between two latents to watch one digit morph into another.
3. Implement a denoising diffusion model from scratch on a 2D spiral dataset. Visualize the forward noising process and the reverse generation process step by step.
4. Train a small diffusion model on MNIST, sample a few digits, and inspect the intermediate noise levels.

By the end of the lab you will have built two generative models from scratch and trained both successfully on real data, using only NumPy, PyTorch, and the math from this chapter.

Chapter 4

Modelling for AI

> *"The architecture you choose is the prior you place on which functions are easy to learn. Everything else is hyperparameter tuning on top of that prior."*

Days 1 to 3 built the machinery: vectors, gradients, likelihoods. Every model has been an MLP — the universal-approximation default, capable in principle of representing anything but capable in practice of representing very little before its parameter count explodes. Real systems use CNNs, GNNs, Transformers, U-Nets, attention with relative positions. These are not arbitrary engineering choices. Each architecture is a *prior*: a statement about what kinds of functions are plausible, baked into the model before any training data is seen.

This chapter is about how that prior is encoded. The general name is *inductive bias*. The specific technical vocabulary is *invariance* and *equivariance*. The practical question is: given a real problem, how do you choose — or design — an architecture whose biases match the structure of your data?

4.1 Inductive bias: the prior an architecture encodes

Two models can have the same expressive power — the same set of functions they can in principle represent — and yet learn very differently from the same data. The difference is how easily each can find a good function by gradient descent. That difference is the inductive bias.

Definition 4.1 (Inductive bias)**.** *The set of assumptions, encoded in the architecture and the loss, that determine which functions the model prefers in the absence of data. Equivalently: the prior over functions induced by training the model.*

The universal approximation theorem from Day 1 says a one-hidden-layer MLP can in principle approximate any continuous function. It says nothing about how many parameters that takes or how much data is needed to find the right function. For an image-classification problem with 10^9 pixel arrangements, a sufficient-width MLP exists — you will not have enough data or time to find it. A CNN, with the right inductive bias, finds it.

Intuition

Inductive bias is the difference between *can represent* and *can learn from realistic data*. The architecture is the part of the model that you, the designer, are allowed to say: "I know something about this problem. Here is the structure."

4.2 Constraints and regularization

The first kind of bias is the one we introduced last week without naming it: regularization. The unconstrained training objective is just the loss on the training data. Adding a penalty changes the problem:

$$\min_{\theta} \; \mathcal{L}(\theta) + \lambda \, R(\theta).$$

R encodes what kinds of θ we prefer in the absence of data — typically simple, small-norm, or sparse parameters. λ controls how strongly that preference dominates.

L^2 regularization (weight decay). $R(\theta) = \|\theta\|^2$. Pulls weights toward zero. Equivalent, under a Bayesian interpretation, to a Gaussian prior on the weights.

L^1 regularization. $R(\theta) = \|\theta\|_1$. Pulls weights toward zero and tends to make them exactly zero. Useful when sparsity is genuinely meaningful (feature selection, compressed sensing).

Dropout (Srivastava et al., 2014). At each training step, randomly set a fraction of hidden activations to zero. The trained model behaves as an ensemble. Not a penalty in the loss, but plays the same role in practice.

Regularization is the softest form of inductive bias: it changes the loss landscape, it does not change the model. The next two sections describe biases that change the model itself.

4.3 Invariance and equivariance

The second kind of bias is structural. It says: *the function we want to learn has a symmetry, and the architecture should respect that symmetry.*

Definition 4.2 (Invariance and equivariance)**.** *A function f is* invariant *under a transformation g if $f(g(\boldsymbol{x})) = f(\boldsymbol{x})$ for every $\boldsymbol{x}$. It is* equivariant *if $f(g(\boldsymbol{x})) = g(f(\boldsymbol{x}))$: applying the transformation to the input is the same as applying it to the output.*

Classification tasks are typically invariant under transformations that preserve identity: an image of a cat is still a cat after a shift, a sentence is still "positive sentiment" after switching to synonyms, a molecule still has the same activity after a rotation. The function we want to learn should respect that invariance, and an architecture that bakes it in does not have to learn it from data.

Many intermediate layers are equivariant rather than invariant: a feature map should shift when its input shifts, so that a later layer can pool the shifted features into something invariant. The standard recipe is a stack of equivariant layers ending in an invariant pooling step.

4.4 CNNs: translation equivariance for grids

Image data lives on a regular grid. A cat shifted ten pixels right is still a cat. If we want the architecture to know that, we need translation equivariance.

Proposition 4.1 (Convolution is translation equivariant)**.** *Let T_d denote translation by d pixels and K a convolutional filter. Then $K \star (T_d \boldsymbol{x}) = T_d(K \star \boldsymbol{x})$ for every $\boldsymbol{x}$ and every d. The convolution operation commutes with translation; therefore any function built only of convolutions and pointwise non-linearities is translation equivariant.*

A convolutional layer with $k \times k$ filters has $k \cdot k \cdot C_{\text{in}} \cdot C_{\text{out}}$ parameters, regardless of the image size. An MLP layer mapping the same image to the same output would have $H \cdot W \cdot C_{\text{in}} \cdot H \cdot W \cdot C_{\text{out}}$ parameters — four orders of magnitude more for a typical image. The convolution is dramatically more parameter-efficient *because* it ties the same weights across all spatial positions, which is exactly what translation equivariance demands.

Note

Convolutional networks were not designed by tuning hyperparameters until something worked on ImageNet. They were designed by Yann LeCun in the 1980s starting from translation equivariance and weight-sharing as principles. The success of CNNs on images is a confirmation that the right inductive bias is worth more than scale of data alone — although since 2020, Transformers with enough data and enough scale have shown that even very weak inductive biases can be overcome by sufficient training (Dosovitskiy et al., 2021).

4.5 GNNs: permutation equivariance for graphs

Graph data has a different symmetry. A graph is a set of nodes connected by edges. Relabeling the nodes ($1 \to 7, 2 \to 3, \ldots$) gives the same graph as a mathematical object. A function over graphs should treat any two relabelings as equivalent.

Definition 4.3 (Permutation equivariance for node features)**.** *Let P be a permutation matrix acting on the nodes of a graph. A function f over node features is permutation equivariant if $f(P\boldsymbol{X}, PAP^\top) = Pf(\boldsymbol{X}, A)$, where $\boldsymbol{X}$ is the node feature matrix and A is the adjacency matrix.*

Message-passing graph neural networks (Gilmer et al., 2017) achieve this by computing each node's new representation as an aggregation of its neighbors' current representations,

using only permutation-invariant aggregations (sum, mean, max). The update for node v at layer $\ell + 1$ is:

$$\boldsymbol{h}_v^{(\ell+1)} = \phi\left(\boldsymbol{h}_v^{(\ell)}, \bigoplus_{u \in \mathcal{N}(v)} \psi(\boldsymbol{h}_v^{(\ell)}, \boldsymbol{h}_u^{(\ell)})\right)$$

where $\bigoplus$ is a permutation-invariant aggregator (most commonly sum or mean), and ϕ, ψ are small neural networks (typically MLPs) with learned parameters.

For graph-level outputs (classifying the whole graph), one final permutation-invariant pooling step is added: sum or mean the node representations together.

4.6 Attention and Transformers

A Transformer (Vaswani et al., 2017) treats the input as an unordered set of tokens and computes pairwise interactions through attention. The attention operation is itself permutation equivariant — positional encoding is added precisely because we usually *want* the model to know the order, and pure attention would be order-blind.

Definition 4.4 (Self-attention)**.** *Given a sequence of vectors* $\boldsymbol{X} = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n]$*, self-attention computes:*

$$\boldsymbol{Q} = \boldsymbol{X}W_Q, \quad \boldsymbol{K} = \boldsymbol{X}W_K, \quad \boldsymbol{V} = \boldsymbol{X}W_V, \quad \boldsymbol{A} = \mathrm{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d_k}}\right), \quad \boldsymbol{Y} = \boldsymbol{A}\boldsymbol{V}.$$

$\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$ *are linear projections of the input.* $\boldsymbol{A}$ *is an* $n \times n$ *matrix of pairwise weights. Each output is a weighted average of all inputs, where the weights depend on content.*

Intuition

Convolution aggregates information from spatial neighbors with fixed weights. Attention aggregates information from *all* positions with weights that depend on the content. A convolutional layer asks "what is here?" Attention asks "what here is similar to what I am looking for, and where is it?" The first is structural, the second is associative. Both turn out to be useful, and the modern picture (Vision Transformers, hybrid architectures, MoE) is largely about combining them.

4.7 Choosing an architecture: the matching exercise

Given a real-world problem, the design move is to identify the symmetries of the data and pick an architecture whose inductive biases match them:

- **Regular grid with translation symmetry** (images, audio spectrograms, dense sensor arrays) → CNN.
- **Sequence with order matters** (text, time series, audio waveforms) → RNN historically, Transformer in practice now.

- **Graph or set** (molecules, social networks, point clouds, particle systems) → GNN, DeepSets, point-cloud networks.
- **Set with translation/rotation equivariance** (3D molecules, physical systems with conservation laws) → equivariant networks (SE(3)-Transformers, EGNN).
- **Heterogeneous tabular features** (rows in a database) → MLP or gradient-boosted trees. The deep-learning prior buys you little here.

Where there is no obvious symmetry, an MLP is the safe default, and trees (XGBoost, LightGBM) frequently beat any neural network. The right inductive bias is doing real work; the wrong one is just an arbitrary constraint that slows you down.

4.8 Translating a real problem into a learning problem

Architecture choice is one of four interlocking decisions you make when framing a learning problem:

1. **Inputs.** What information does the model actually see? Raw, or featurized? At what granularity (pixel, token, transaction, day)?
2. **Targets.** What is the model trying to predict? A class? A real number? A distribution? A sequence?
3. **Loss.** How do we measure "wrong"? Cross-entropy, MSE, contrastive, custom?
4. **Evaluation metric.** How do we declare success? Often *not* the same as the loss — accuracy is not differentiable, but it is what the user actually cares about.

When a project fails, the architecture is usually not the problem. The problem is usually that one of these four was wrong. The famous example: a model trained to predict pneumonia from chest X-rays that learned to predict the hospital's chest-X-ray machine instead, because that information leaked into the input. Wrong inputs. Different fix from "add more layers."

Intuition

The architecture is the smallest of the four decisions. Inputs, targets, loss, and metric determine whether the problem is even well-posed; the architecture only determines whether a well-posed problem can be solved efficiently. A long career in ML is mostly about getting the framing right — not about picking the right number of layers.

4.9 What you have seen

Every neural network is a function approximator with a prior baked in — the inductive bias. CNNs encode translation equivariance for grid data; GNNs encode permutation equivariance for graphs; Transformers replace structural priors with content-based weighted aggregation. Choosing an architecture is choosing a prior, and that choice matters far more than the size of the network. Surrounding that choice is the larger framing problem: inputs, targets, loss, and evaluation metric all have to line up before any architecture can do useful work.

Day 5, the final chapter, asks the question we have been deferring all along: how do we know the model is actually learning the right thing, and not memorizing the dataset or exploiting an artifact?

Lab

Open the Day 4 notebook (`Day4_Modeling_InductiveBias.ipynb`). You will:

1. Train an MLP and a CNN with comparable parameter counts on MNIST, then evaluate both on a shifted version of the test set. You will see the CNN's translation equivariance pay off when train and test no longer match exactly.
2. Implement a tiny message-passing GNN by hand and apply it to a synthetic node-classification problem. Compare to an MLP that ignores the graph structure and uses only node features.
3. Compute self-attention from scratch on a small toy sequence. Visualize the attention weights as a heatmap and see how query/key/value projections produce content-based selection.

By the end of the lab you will have direct evidence for the central claim of this chapter — that the right inductive bias is doing work even before any training begins.

Chapter 5

Evaluating AI models

> *"A model with 99% test accuracy is not done. A model is done when you can defend, in front of a stakeholder, both the cases where it works and the cases where it fails."*

We have a trained model. It is a stack of affine maps and non-linearities (Day 1), fitted to data by gradient descent on a likelihood-derived loss (Days 2 and 3), with an architecture chosen to match the structure of the data (Day 4). The training loss is small. The test loss is small. Is the model done?

The honest answer is: usually no. Evaluation is the part of machine learning that gets the least attention in courses and the most attention in real projects. Every production failure of an ML system in the last decade — a self-driving car that did not recognize a white truck against a bright sky, a hiring tool that systematically downranked women, a medical model that learned to predict the hospital instead of the disease — is a story about evaluation gone wrong. This chapter is about how to do evaluation right, what the standard failure modes look like, and how to decide whether a model should ship or go back to the drawing board.

5.1 What test accuracy actually measures

A test accuracy is one number. It measures the fraction of correct predictions on a single held-out set, drawn from the same distribution as the training set, evaluated with the same metric used during model selection. Each of these qualifications matters:

- **One number.** A single scalar collapses everything you might care about into one summary. Where does the model fail? On which classes? Under which inputs? At which thresholds? Test accuracy says nothing.
- **Single held-out set.** The fluctuation across different held-out sets is real, especially for small test sets. A reported 92% can be 87% next week.
- **Same distribution.** Test data drawn from the same distribution as training data tells you about *interpolation*, not about generalization to the cases you care about in deployment.

- **Same metric used during model selection.** If you picked the model that maximized test accuracy across many tries, the test set has effectively become a training set, and you have overfit it.

Intuition

Test accuracy is a necessary statistic, not a sufficient one. It tells you the model fits the test set. It does not tell you the model has learned the underlying task. Distinguishing these two is essentially what this chapter is about.

5.2 The bias-variance decomposition

For a regression problem with squared-error loss and a fixed test point $\boldsymbol{x}_0$, the expected loss of a model $\hat{f}(\boldsymbol{x}_0)$ trained on a random training set decomposes as:

$$\mathbb{E}\big[(y_0 - \hat{f}(\boldsymbol{x}_0))^2\big] = \underbrace{(f(\boldsymbol{x}_0) - \mathbb{E}[\hat{f}(\boldsymbol{x}_0)])^2}_{\text{bias}^2} + \underbrace{\mathrm{Var}(\hat{f}(\boldsymbol{x}_0))}_{\text{variance}} + \underbrace{\sigma^2}_{\text{irreducible noise}}$$

where f is the unknown true function and σ^2 is the variance of the label noise. The expectation is over both the random training set and the random test label.

- **Bias** is how wrong the model is, on average, across many training runs. A model too simple to capture the truth has high bias — underfitting.
- **Variance** is how much the model fluctuates from one training run to another. A model too flexible relative to the dataset size has high variance — overfitting.
- **Irreducible noise** is what no model can reduce, because the label itself is noisy.

The classical picture says: as model capacity grows, bias decreases and variance increases. The sweet spot is somewhere in the middle. The lab today produces this picture explicitly by fitting polynomials of varying degree to a small dataset.

Note

For large neural networks the classical picture breaks down. The *double descent* phenomenon (Belkin et al., 2019) shows test error first rising and then falling again as model capacity passes the dataset size. Modern deep learning operates in a regime where extra capacity does not consistently hurt generalization. This is empirical, not yet fully understood theoretically, and is one of the open questions of the field.

5.3 Splits: training, validation, test, and what each is for

To estimate generalization without contaminating the model selection process, the standard discipline is three disjoint datasets:

- **Training set** ($\sim$ 60–80%). The data the model sees and fits.
- **Validation set** ($\sim$ 10–20%). The data used to choose between candidate models, hyperparameters, architectures. Held out from training, but the modeler interacts with it heavily.
- **Test set** ($\sim$ 10–20%). The data used to estimate final generalization. Touched once, after all decisions have been made.

Caution

The test set is contaminated the moment you make any decision based on its performance. "We tried 50 architectures and picked the one with the best test accuracy" has effectively made the test set a validation set. The reported test accuracy is then biased upward, often substantially. The fix is discipline — a true held-out set the modeler does not look at until the very end — and, where possible, an external test set the team cannot iterate on.

For small datasets, k-fold cross-validation rotates the validation role through different splits so that every example serves as validation once. This gives a better estimate of generalization at the cost of k times more training. For datasets above a few tens of thousands of examples, a single train/val/test split is usually enough.

5.4 Calibration: confident is not the same as right

A classifier returns a probability per class. "90%" should mean that across all predictions made with 90% confidence, 90% are correct. *Calibration* is whether that is true. Modern deep networks tend to be *overconfident*: they say 99% when they should say 80%.

Definition 5.1 (Reliability diagram)**.** *Group predictions into bins by their confidence (say, 0.0–0.1, 0.1–0.2, …). For each bin, plot (mean confidence in the bin, fraction correct in the bin). A perfectly calibrated model lies on the diagonal.*

The Expected Calibration Error (ECE) is the average distance from the diagonal, weighted by bin population. It is a single number that summarizes how miscalibrated the model is.

Intuition

Calibration matters when the downstream decision depends on the probability, not just the argmax. A medical triage system that flags "70% probability of sepsis" had better mean it — the physician will treat differently than for "95% probability." A search ranker can be wildly miscalibrated and nobody notices, because only the ordering matters. Decide whether your application needs calibrated probabilities, and measure if so.

Temperature scaling (Guo et al., 2017) is the simplest fix. After training, divide the pre-softmax logits by a scalar temperature T, chosen on a validation set to minimize negative log-likelihood. One scalar. Fixes most modern-network calibration problems with no retraining.

5.5 Distribution shift: when the test set is not the deployment set

Test accuracy estimates performance on data *from the same distribution as training data.* Deployed models see data from a different distribution — because the world changes, because the data collection process is different, because users behave differently than the dataset suggested.

Three common shifts:

- **Covariate shift.** $p_{\text{deploy}}(\boldsymbol{x}) \neq p_{\text{train}}(\boldsymbol{x})$, but $p(y \mid \boldsymbol{x})$ is unchanged. The kinds of inputs change; the labeling rule does not. Camera-quality changes between training and deployment are a common example.
- **Label shift.** $p_{\text{deploy}}(y) \neq p_{\text{train}}(y)$, but $p(\boldsymbol{x} \mid y)$ is unchanged. Class prevalence changes; the appearance of each class does not. A model trained on a balanced dataset and deployed on imbalanced data fails this way.
- **Concept shift.** $p(y \mid \boldsymbol{x})$ itself changes. The labeling rule has changed. Recommender systems suffer this constantly — what users *want* this year is not what they wanted last year.

The lab today produces a covariate-shift experiment by rotating the test set and measuring the drop in accuracy. The drop is dramatic for the MLP from Day 4, less so for the CNN. This is exactly the inductive-bias story from yesterday, told from the evaluation side.

5.6 Adversarial examples and out-of-distribution detection

Some failures are more pointed than gradual distribution shift. Adversarial examples (Szegedy et al., 2014; Goodfellow et al., 2015) are inputs constructed specifically to fool the model — imperceptibly small perturbations that flip a confident correct prediction to a confident wrong one. They reveal that the model has learned a function that agrees with the data on natural inputs but diverges from it elsewhere.

Out-of-distribution (OOD) inputs are not adversarial, just unfamiliar: a model trained on cats and dogs being asked to classify a fish. A well-behaved model should signal "I do not know." Most modern networks do the opposite: they confidently assign one of the trained classes.

Intuition

A model trained on a closed set of classes has no representation of "other." Detecting that an input is OOD is a separate problem, typically tackled by examining the softmax confidence (low confidence sometimes means OOD), the energy of the logits, or distance in feature space to the nearest training example. The lab today shows the failure mode using a softmax-confidence detector on Fashion-MNIST inputs to

an MNIST-trained classifier.

5.7 The decision framework: ship, collect, or redesign

Evaluation outputs a decision. There are three:

1. **Ship.** Test performance is acceptable on the relevant population, calibration is adequate for the downstream use, and the failure modes you have measured are tolerable. Deploy with monitoring.

2. **Collect more data.** The model has the right structure but not enough examples. Cross-validation curves are still rising; test loss is plateauing for the right reason. Add labeled data and continue.

3. **Redesign.** The model is wrong about something structural — the architecture mismatches the task, the loss does not capture what users want, the targets are not what was supposed to be learned. No amount of data fixes this. Go back to the framing.

The two failure modes of this decision: *shipping when you should redesign* (the model looks fine on the test set but is solving the wrong problem) and *collecting when you should ship* (perfectionism delays a model that would already be useful). Real evaluation discipline is about distinguishing these cases, and the only way to do it is to know your model's failure modes well enough that you can defend the decision to a skeptical stakeholder.

5.8 What we have built across the five days

The workshop began with vectors. A model in modern AI is a function from one vector space to another, built out of affine maps and non-linearities (Day 1). It learns by gradient descent on a loss function that summarizes how wrong it is; the chain rule makes this computable for million-parameter networks (Day 2). When the goal is generation rather than prediction, the loss becomes a likelihood and the model becomes a distribution we can sample from — the unifying picture behind VAEs, language models, and diffusion (Day 3). The architecture encodes a prior on which functions are easy to learn; CNNs, GNNs, and Transformers are three different priors for three different kinds of data (Day 4). And evaluation is the discipline that tells us whether all of the above is actually doing the right thing, or merely something that looks right on a test set (Day 5).

The same five ideas — vectors and linear maps, gradient-based learning, likelihood, inductive bias, evaluation discipline — power every model used in production today, from a logistic regression to GPT-class systems. The implementations get bigger; the structures do not change. What you have built across these five days is the vocabulary to read a paper, evaluate a vendor, audit a deployed system, and decide when an AI tool is the right answer to the problem in front of you.

Lab

Open the Day 5 notebook (`Day5_Evaluation_Generalization.ipynb`). You will:

1. Fit polynomials of degree 1, 3, 9, 15 to the same small dataset, repeated over many random training sets. Plot bias and variance separately to see the decomposition in action.
2. Train a classifier on MNIST and produce its reliability diagram. Compute the Expected Calibration Error. Apply temperature scaling on a validation split and re-measure.
3. Take a CNN trained on MNIST and evaluate it on rotated MNIST. Measure the accuracy drop as the rotation angle grows.
4. Use the same CNN's softmax confidence as an out-of-distribution detector on Fashion-MNIST. Plot the distribution of confidences for in-distribution vs OOD inputs and see how separable (or not) they are.

By the end of the lab you will have seen, with your own numbers, each of the failure modes named in this chapter, and you will have the tools to detect them in your own models.

Appendix A: Environment setup

Option 1: Google Colab (recommended)

Go to https://colab.research.google.com. Select **Runtime > Change runtime type > T4 GPU** for the diffusion-model labs in Day 3 and Day 4. All libraries can be installed in notebook cells with `pip install`.

Option 2: Local installation

Python 3.10 or later. For Day 3's diffusion lab and Day 4's graph-learning lab, an NVIDIA GPU with 6+ GB VRAM accelerates training, but every lab can run on CPU in under 15 minutes.

Core libraries

```
pip install torch torchvision numpy matplotlib scikit-learn
pip install jupyter notebook
```

Appendix B: Reading list

- Goodfellow, Bengio, Courville, *Deep Learning* (MIT Press, free online).
- Bishop & Bishop, *Deep Learning: Foundations and Concepts* (Springer, 2024).
- Murphy, *Probabilistic Machine Learning* (MIT Press, free online).
- Strang, *Linear Algebra and Learning from Data* (Wellesley-Cambridge, 2019).
- Ho, Jain, Abbeel, "Denoising Diffusion Probabilistic Models", NeurIPS 2020, https://arxiv.org/abs/2006.11239.
- Vaswani et al., "Attention Is All You Need", NeurIPS 2017, https://arxiv.org/abs/1706.03762.