Inductive Biases in Neural Networks

A Built-from-Scratch Tour

Alexander Towell

2026

Contents

Part I

Foundations

Before a network can carry a useful bias, it has to represent functions at all. Part I builds that floor. Chapter 1 starts from function approximation with no constraints, shows why a single linear unit cannot separate XOR, and recovers the multilayer perceptron as the smallest fix. Chapter 2 introduces the first inductive bias proper: the output head. A loss is a prior over the response distribution, and the same network body becomes a classifier, a count model, or a regressor depending only on the head bolted onto it. The two axes that organize the rest of the book, architecture and output head, both make their first appearance here.

Chapter 1

Foundations: Function Approximation and the Multilayer Perceptron

Throughout this chapter (and the chapters that follow) we are doing **supervised learning**: we have labeled (x, y) pairs and learn f such that $f(x) \approx y$. The labels can come from humans (classical supervised) or from the data's own structure (self-supervised, as in language modeling). Both are mathematically the same. What differs across sections is the *interpretation* of the output: the choice of output activation (link function) and matching loss.

The network produces raw values. The output's meaning is the loss's job.

1.1 What is a neural network?

A neural network is a parameterized function $f_\theta : \mathbb{R}^n \to \mathbb{R}^m$. It takes a real-valued input vector and produces a real-valued output vector. That is the whole structural commitment. What the output *means* is a separate decision, made by the loss.

The network produces raw values; the loss interprets those values as a prediction by composing them with an **output activation** (also called a **link function** in the language of generalized linear models). The choice of link function and matching loss together specify what kind of thing the network is predicting.

The simplest case is **regression**: the raw output *is* the prediction. The link function is the identity, and the loss is **mean squared error**:

$$L = \frac{1}{2}\sum_i (\hat{y}_i - y_i)^2, \qquad \frac{\partial L}{\partial \hat{y}_i} = \hat{y}_i - y_i. \tag{1.1}$$

The gradient is just the residual. This pattern, $\hat{y} - y$, recurs throughout the chapter. It is the gradient of every canonical loss with respect to its raw output, for reasons that turn out to be a theorem about exponential families and canonical links, not an accident (Chapter 2 proves it).

The `MSELoss` class encodes this directly:

```
class MSELoss(Loss):
    def value(self, logits, y):
        return 0.5 * sum((zi - yi) ** 2 for zi, yi in zip(logits, y))

    def grad(self, logits, y):
        return [zi - yi for zi, yi in zip(logits, y)]
```

```
    def probs(self, logits):
        # Identity link: the prediction IS the raw output.
        return list(logits)
```

Note the $\frac{1}{2}$ convention in `value`: it makes the gradient exactly $\hat{y} - y$ with no leading constant, matching the pattern of the other losses.

A concrete demonstration: fit $y = \sin(x) + \varepsilon$ on $x \in [-\pi, \pi]$ with a small MLP, identity output, and MSE loss (`examples/regression.py`). After 2000 epochs of mini-batch SGD the network learns the curve. This architecture is the same MLP we build up to in Section 1.3; only the output head and loss differ.

With regression in hand as the unspecialized base case, the rest of the chapter is a tour through two questions. First, what *expressive limits* does the architecture itself impose? That is Sections 1.2 and 1.3: a single linear unit cannot represent all continuous functions; the multilayer perceptron can. Second, what *output-side specializations* let the same architecture solve concrete prediction problems beyond regression? That is Sections 1.4 and 1.5: classification by sigmoid for binary outcomes, softmax for categorical.

1.2 The single linear unit and its limits

The smallest non-trivial network is one `Linear` layer mapping inputs to outputs:

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

This is a **single-layer perceptron** (SLP): an affine map with no non-linearity. Trained with MSE it learns the least-squares fit. Trained with a sigmoid head and BCE loss (Section 1.4), it is logistic regression. The function class either way is *linear* in the input.

The SLP can fit any linearly separable target. It cannot fit anything else.

The canonical demonstration is XOR on $\{0,1\}^2$:

$$\text{XOR}(0,0) = 0, \quad \text{XOR}(0,1) = 1, \quad \text{XOR}(1,0) = 1, \quad \text{XOR}(1,1) = 0.$$

Training a single `Linear` layer with `SigmoidBCE` on these four points for 500 epochs (batch size 4, seed 42):

```
slp = Network([Linear(2, 1)], loss=SigmoidBCE())
slp.fit(xor_X, xor_Y, epochs=500, lr=0.5, batch_size=4)
```

Every input converges to $p = 0.5000$. The model literally cannot do better. To see why, write down what fitting all four points would require. With $\hat{y} = \sigma(w_1 x_1 + w_2 x_2 + b)$ and targets $\{0, 1, 1, 0\}$, the BCE loss is minimized when each $\hat{y}$ matches its target. Since sigmoid is monotone, the four constraints reduce to four inequalities:

$$b < 0, \quad w_1 + b > 0, \quad w_2 + b > 0, \quad w_1 + w_2 + b < 0.$$

Adding the second and third gives $w_1 + w_2 + 2b > 0$, so $w_1 + w_2 + b > -b > 0$. But the fourth requires $w_1 + w_2 + b < 0$. Contradiction. No choice of (w_1, w_2, b) fits all four points simultaneously.

Geometrically: a linear classifier draws a single hyperplane in the input space (a line in $\mathbb{R}^2$). XOR's positive class $\{(0,1), (1,0)\}$ and negative class $\{(0,0), (1,1)\}$ lie at diagonal corners of the unit square. No line separates them, as the left panel of Figure 1.1 shows.

This was Minsky and Papert (1969)'s argument in 1969, which stalled the perceptron research program for a decade and a half. The resolution is one of the cleanest examples of how architecture and expressivity interact. To escape linearity, add a non-linearity: the subject of Section 1.3.

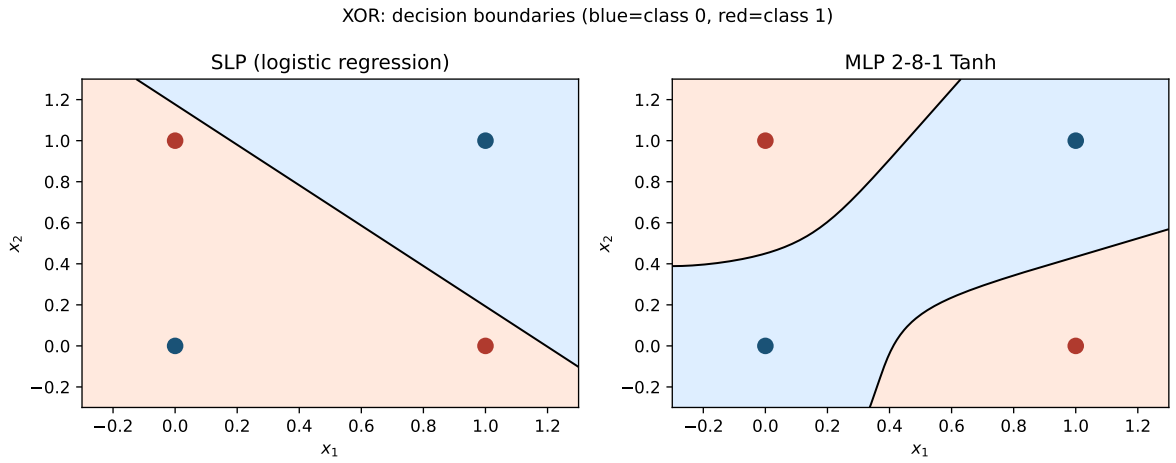

Figure 1.1: XOR decision boundaries on the unit square (blue points are class 0, red are class 1). *Left*: a single linear unit (logistic regression) draws only one straight boundary, collapses to $p \approx 0.5$ everywhere, and cannot separate the diagonal classes. *Right*: a 2-8-1 `Tanh` MLP (Section 1.3) bends the boundary and separates them. Same output head, different architecture.

1.3 The multilayer perceptron, and what depth and non-linearity buy

The multilayer perceptron (MLP) stacks `Linear` layers with a non-linear **activation** between them (`Tanh` or `ReLU`). Without the non-linearity, stacked linear maps collapse: two affine maps composed are again affine, and no number of stacked `Linear` layers without non-linearities buys anything over one. The fix from Section 1.2 is therefore not to add more linear layers; it is to add a single non-linear layer between them.

```
mlp = Network([Linear(2, 8), Tanh(),
               Linear(8, 1)], loss=SigmoidBCE())
mlp.fit(xor_X, xor_Y, epochs=4000, lr=0.5, batch_size=4)
```

Training this 2-8-1 network on XOR (seed 42) reaches a final loss below 0.001: the MLP fits XOR exactly. Mechanically, the hidden layer constructs intermediate features that *are* linearly separable. One natural decomposition for XOR is

$$\mathrm{XOR}(x_1, x_2) = \mathrm{OR}(x_1, x_2) - \mathrm{AND}(x_1, x_2),$$

so a hidden unit detecting OR and another detecting AND give the output layer a linearly separable problem. The training run discovers something equivalent.

The general fact is the **universal approximation theorem** (Cybenko 1989; Hornik, Stinchcombe, and White 1989): a feedforward network with one hidden layer of sufficient width and any non-polynomial activation can approximate any continuous function on a compact set to arbitrary precision. *Depth is not required for expressivity.* One hidden layer suffices. What deeper networks buy is *efficient* representation: many natural functions that one wide hidden layer can approximate at exponential cost are representable at polynomial cost with a few stacked hidden layers. That is the empirical case for depth; the theoretical case for hidden layers at all is XOR.

Backpropagation as the local chain rule

The training rule for an MLP is **backpropagation**. The trick is to make the chain rule *local*: each layer implements `forward` (input to output) and `backward` (gradient with respect to the output, to gradient with respect to the input), plus `parameters`, which exposes its weights and their gradients. A layer never sees the rest of the network. That contract is the entire base class:

```
class Layer:
    def forward(self, x):
        raise NotImplementedError

    def backward(self, g):
        """Given g = dL/d(output), return dL/d(input).
        Layers with parameters also accumulate their parameter grads."""
        raise NotImplementedError

    def parameters(self):
        """A list of (values, grads) pairs: each a parameter vector
        and its same-length gradient vector."""
        return []
```

`Linear` ($\mathbf{y} = W\mathbf{x} + \mathbf{b}$) is the only layer with parameters. Its `backward` writes out three gradients:

$$\frac{\partial L}{\partial W_{ij}} = g_i x_j, \qquad \frac{\partial L}{\partial b_i} = g_i, \qquad \frac{\partial L}{\partial x_j} = \sum_i W_{ij}\, g_i. \tag{1.2}$$

That is exactly what the code says. Note the `+=`: parameter gradients *accumulate*, so a mini-batch can sum per-example contributions before the optimizer steps.

```
class Linear(Layer):
    def forward(self, x):
        self.x = x
        return [dot(w, x) + b for w, b in zip(self.weights, self.bias)]

    def backward(self, g):
        for i, gi in enumerate(g):
            for j, xj in enumerate(self.x):
                self.dweights[i][j] += gi * xj # dL/dW_ij = g_i x_j
            self.dbias[i] += gi # dL/db_i = g_i
        return [sum(self.weights[i][j] * g[i] for i in range(len(g)))
                for j in range(len(self.x))] # dL/dx_j = sum_i W_ij g_i
```

There is no matrix type. A vector is a `list[float]`, `dot` is the only named helper, and the outer product and transposed mat-vec above are written inline as comprehensions. The two activations are smaller still, each matching the derivatives one-to-one ($1 - \tanh^2$ for `Tanh`, the positive-part mask for `ReLU`):

```
class Tanh(Layer):
    def forward(self, x):
        self.out = [math.tanh(xi) for xi in x]
        return self.out
    def backward(self, g):
        return [gi * (1.0 - oi * oi) for gi, oi in zip(g, self.out)]

class ReLU(Layer):
```

```
    def forward(self, x):
        self.positive = [xi > 0.0 for xi in x]
        return [xi if xi > 0.0 else 0.0 for xi in x]
    def backward(self, g):
        return [gi if p else 0.0 for gi, p in zip(g, self.positive)]
```

The `Network` threads these together. `forward` runs the layers in order and caches the logits; `backward` seeds the gradient from the loss and pushes it back through the layers in reverse:

```
class Network:
    def forward(self, x):
        for layer in self.layers:
            x = layer.forward(x)
        self.logits = x
        return x

    def backward(self, y):
        g = self.loss.grad(self.logits, y) # the familiar p - y
        for layer in reversed(self.layers):
            g = layer.backward(g)
        return g
```

The optimizer is written *once*, generically, on top of `parameters()`. There is no per-layer `step` and no `Optimizer` class. `zero_grad` clears every gradient; `step` applies the *mean* gradient over a batch of n by dividing by n, which is the counterpart to the accumulating `+=` in `backward`:

```
    def zero_grad(self):
        for _values, grads in self.parameters():
            for k in range(len(grads)):
                grads[k] = 0.0

    def step(self, lr, n):
        for values, grads in self.parameters():
            for k in range(len(values)):
                values[k] -= (lr / n) * grads[k]
```

The key realization: the MLP introduced **no new math** beyond what a single linear unit needed. The chain rule applied locally, composed across layers, gives gradients for any depth. Depth is just more composition of the same local steps, and the optimizer never has to know how many layers there are. Section 1.9 verifies all this numerically.

1.4 Logistic regression

A logistic-regression model scores an input $\mathbf{x}$ with a weighted sum, the **logit**:

$$z = \mathbf{w} \cdot \mathbf{x} + b.$$

Here $\mathbf{w}$ and b are a single weight vector and a scalar bias, the one-output specialization of the $W\mathbf{x} + \mathbf{b}$ from Section 1.2. It squashes that score into a probability with the **sigmoid**:

$$p = \sigma(z) = \frac{1}{1 + e^{-z}}.$$

Here p is the model's estimate of $P(y = 1 \mid \mathbf{x})$. The loss that penalizes disagreement between p and a binary label $y \in \{0, 1\}$ is **binary cross-entropy** (BCE):

$$L = -\big[y \log p + (1 - y) \log(1 - p)\big].$$

Differentiating with respect to the logit z gives a clean result:

$$\frac{\partial L}{\partial z} = p - y. \tag{1.3}$$

Prediction minus target. Every weight gradient follows by the chain rule, $\partial L/\partial w_j = (p - y)\, x_j$, and we descend: $\mathbf{w} \leftarrow \mathbf{w} - \alpha\, \partial L/\partial \mathbf{w}$.

In code, logistic regression is one `Linear` layer with a `SigmoidBCE` loss:

```
net = Network([Linear(n_features, 1)], loss=SigmoidBCE())
```

The loss owns the output activation. The `Network` emits the raw logit; `SigmoidBCE` turns it into a probability and a gradient. Its `grad` is the $p - y$ of Equation (1.3); its `value` computes binary cross-entropy straight from the logit in a numerically stable form (discussed in Section 1.8), never by first forming p and taking a log:

```
class SigmoidBCE(Loss):
    def value(self, logits, y):
        z = logits[0]
        # max(z,0) - z*y + log1p(exp(-|z|))
        return max(z, 0.0) - z * y + math.log1p(math.exp(-abs(z)))

    def grad(self, logits, y):
        return [sigmoid(logits[0]) - y] # p - y

    def probs(self, logits):
        return [sigmoid(logits[0])]
```

This is structurally the same SLP from Section 1.2, with a probability interpretation. The SLP's limits still apply: logistic regression can solve any linearly separable binary classification but cannot solve XOR. To solve XOR with sigmoid output, swap the body for the MLP from Section 1.3:

```
net = Network([Linear(2, 8), Tanh(),
               Linear(8, 1)], loss=SigmoidBCE())
```

The output head (sigmoid + BCE) and the architecture (linear vs. MLP) are independent choices, a theme that becomes the load-bearing principle of Section 1.7.

1.5 Multi-class regression via softmax

With K classes the model produces K logits, one weighted sum per class. The logit vector $\mathbf{z}$ becomes a probability distribution through the **softmax**:

$$p_i = \frac{e^{z_i}}{\sum_j e^{z_j}}.$$

The matching loss is **categorical cross-entropy**, $L = -\log p_y$, where $y \in \{0, \ldots, K - 1\}$ is the index of the correct class. Its gradient with respect to the logits is, again:

$$\frac{\partial L}{\partial z_i} = p_i - \mathbf{1}[i = y]. \tag{1.4}$$

That is $\mathbf{p}-\mathbf{y}$ with $\mathbf{y}$ read as a one-hot vector. The same expression as the binary case (Equation (1.3)). That is no coincidence: binary logistic regression *is* two-class softmax with one logit pinned to zero (softmax depends only on logit differences, so one logit may be fixed freely). The binary sigmoid is the $K = 2$ softmax in disguise.

In code, softmax regression is one `Linear` layer with a categorical head:

```
net = Network([Linear(n_features, K)], loss=SoftmaxCrossEntropy())
```

The head mirrors `SigmoidBCE` exactly, one level up. `value` uses the stable $\text{logsumexp}(\mathbf{z}) - z_y$ form of $-\log \text{softmax}(\mathbf{z})_y$; `grad` is $\mathbf{p}$ with 1 subtracted at the true class:

```
class SoftmaxCrossEntropy(Loss):
    def value(self, logits, y):
        # Stable: logsumexp(z) - z[y]
        return logsumexp(logits) - logits[y]

    def grad(self, logits, y):
        p = softmax(logits)
        p[y] -= 1.0
        return p

    def probs(self, logits):
        return softmax(logits)
```

On the 8×8 UCI optical-recognition digits (1 437 training, 360 test, 10 classes, stratified 80/20 split, seed 42), this one-layer softmax model reaches **94.4%** test accuracy after 30 epochs of mini-batch SGD. Linear classifiers on this dataset are unexpectedly competitive, which is itself a hint about the data: the 4×4 block-counting that produced these 8×8 features has already done most of the work.

Swap the body for an MLP:

```
net = Network([Linear(64, 32), Tanh(),
               Linear(32, 10)], loss=SoftmaxCrossEntropy())
```

This 64-32-10 MLP reaches **97.2%** test accuracy after 50 epochs. The MLP fits the training set to **99.3%** and generalizes somewhat worse; it roughly halves the linear model's test error, from 5.6% to 2.8%, a real gain but a smaller one than its far greater expressivity might promise. Why is the gain not larger, given how much more expressive the MLP is than logistic regression? The answer is in Section 1.7.

1.6 Logits are log-probabilities, up to a constant

Take the log of the softmax from Section 1.5:

$$\log p_i = z_i - \log \sum_j e^{z_j}.$$

The logits are *not* the log-probabilities. They are the log-probabilities plus a shared additive term $\log \sum_j e^{z_j}$, the same for every class i. The logits are **unnormalized log-probabilities**.

That constant cannot be recovered from $\mathbf{p}$. Softmax is unchanged by adding any constant c to every logit: e^{z_i+c} has a factor e^c on top and bottom, and it cancels. Training therefore can never pin the constant down. What training *does* pin down is logit **differences**:

$$z_i - z_j = \log p_i - \log p_j = \log \frac{p_i}{p_j}, \tag{1.5}$$

the log-odds between classes. The constant is **gauge**: free, unobservable, with no effect on the predicted distribution.

One sharp way to see that logits are not raw log-probabilities: a probability is at most 1, so a log-probability is at most 0. But nothing stops a network's logits from being positive. Put a ReLU just before the output, and every logit is at least 0. They still describe a perfectly good distribution, because the additive constant absorbs the offset. Logits live on a different scale than log-probabilities; only their differences are meaningful.

This gauge freedom returns in Section 1.8 as the key to numerical stability: since softmax does not care about a shared shift, we can spend that freedom on keeping the exponentials in a safe range.

1.7 Inductive biases

The MLP on digits is essentially saturated at 97.2% test accuracy (seed 42, 80/20 split), even though it is plenty expressive: it fits the training set to 99.3%. A model that fits the training data almost perfectly yet plateaus below it on held-out data is not capacity-limited. The bottleneck is the **prior** the architecture and the loss commit to.

Inductive bias enters at two distinct points in every supervised model:

1. **The output head**: a choice of link function and matching loss. Identity plus MSE assumes Gaussian targets; sigmoid plus BCE assumes Bernoulli; softmax plus cross-entropy assumes Categorical. The link function is a prior about the data-generating process on the output side. Chapter 2 takes this axis up in depth.

2. **The architecture**: a choice of how the network computes features. An MLP assumes every input feature is independent of every other and learns their relationships from data. A convolutional network assumes nearby pixels matter together and the same detector should fire anywhere. A recurrent network assumes the same computation runs at every step of a sequence. A Transformer assumes positions interact pairwise through attention. Part II of this book walks through each of these priors.

Both axes are inductive biases. Both improve sample efficiency when matched to the data and hurt when mismatched. The two are independent: any architecture can pair with any output head, as long as the head's input dimension matches the architecture's output.

For the 64 pixels of a digit, the MLP architecture says every feature is independent of every other. That is a strong claim, and it happens to be wrong. Pixel $(3, 4)$ and pixel $(3, 5)$ are neighbors; the MLP treats them as completely unrelated and has to relearn their relationship from 1 437 examples. Pixel $(3, 4)$ and pixel $(7, 0)$ are unrelated, but the MLP cannot tell that apart from $(3, 4)$ and $(3, 5)$ either.

Different architectures commit to different invariances:

- A **convolutional network** (CNN) commits to **locality** and **translation equivariance**: nearby pixels matter together, and a feature detector that fires at one position should fire at any position. The right prior for image data.

- An **RNN** commits to **time-translation equivariance**: the same computation runs at every step of a sequence, and the hidden state summarizes the past. The right prior for sequential data.

- A **Transformer** commits to **permutation equivariance** over positions, with positional encoding added back to recover a sequence prior. Pairwise interactions are explicit through attention rather than recurrence.

These are bets. A CNN beats an MLP on images *because* its prior matches the data. On tabular data with no spatial structure, a CNN loses to an MLP, because the spatial prior is wrong. The right prior is empirical, not universal.

There is a data-side counterpart. **Data augmentation** expresses the same prior from the data side instead of the architecture side. Random rotations of training images approximate the prior "the answer should not depend on rotation" through the loss: a rotation-equivariant architecture and rotation-augmented data are two routes to the same prior, and they often compose.

A concrete illustration for digits: a single 3×3 convolutional filter applied across the 8×8 image with 16 output channels has **160 parameters**. The MLP we trained has **2 410 parameters**. The CNN ends up beating the MLP with one-fifteenth the parameters, *because* it has committed to the right prior and does not have to learn translation invariance from scratch for every position. That is the whole story of inductive bias in one number. (The exact counts are computed in the paired notebook, not asserted.)

When you do not know the structure, an MLP is the safe bet on the architecture side and identity plus MSE is the safe bet on the output side: both commit to nothing, and pay for that with data efficiency.

The next chapter takes up the output-head axis in full: how each output head corresponds to a distributional assumption about the target, why the $p - y$ gradient recurs, and how the two axes compose. The architecture axis runs through Part II. A closing chapter on reinforcement learning asks how much of this inductive-bias frame survives the shift from supervised learning to learning from scalar reward.

1.8 Numerical stability is the gauge freedom, reused

e^z overflows for moderately large z. The fix for softmax: subtract the largest logit first,

$$\mathrm{softmax}(\mathbf{z}) = \mathrm{softmax}\left(\mathbf{z} - \max_i z_i\right).$$

This is *exactly* the additive-constant freedom from Section 1.6: since only logit differences carry information (the log-odds of Equation (1.5)), a shared shift of the logits changes nothing, and we spend that freedom to keep the exponentials in a safe range. Stability here is not a new idea bolted on. It is the gauge symmetry put to work.

Cross-entropy is computed as $\mathrm{logsumexp}(\mathbf{z}) - z_y$ rather than $-\log \mathrm{softmax}(\mathbf{z})_y$, which would risk $\log 0$. The sigmoid is evaluated with a sign branch; binary cross-entropy is computed straight from the logit, both to avoid overflow and $\log 0$. These are the primitives the loss classes call:

```
def sigmoid(z):
    if z >= 0.0:
        return 1.0 / (1.0 + math.exp(-z)) # safe: exp(-z), z >= 0
    ez = math.exp(z) # safe: exp(z), z < 0
    return ez / (1.0 + ez)

def logsumexp(zs):
    m = max(zs)
    return m + math.log(sum(math.exp(z - m) for z in zs))
```

```
def softmax(zs):
    m = max(zs)
    exps = [math.exp(z - m) for z in zs]
    total = sum(exps)
    return [e / total for e in exps]
```

In every case the exponent argument is held at or below 0, so e^z stays in $[0, 1]$ and never overflows. The max-subtraction in `softmax` and `logsumexp` is the Section 1.6 gauge freedom; the sign branch in `sigmoid` is the same trick for the two-class case.

1.9 Trust, but verify: numerical gradients

Hand-derived gradients are easy to get subtly wrong. There is a slow but foolproof check: the definition of a derivative itself. For any parameter θ,

$$\frac{\partial L}{\partial \theta} \approx \frac{L(\theta + \varepsilon) - L(\theta - \varepsilon)}{2\varepsilon}.$$

`gradient_check` computes this central difference for every parameter and compares it to the analytical gradient from `backward`. It perturbs each parameter in place, twice, and reads off the loss:

```
def gradient_check(net, x, y, eps=1e-5):
    net.zero_grad()
    net.forward(x)
    net.backward(y)
    worst = 0.0
    for values, grads in net.parameters():
        for k in range(len(values)):
            original = values[k]
            values[k] = original + eps
            loss_plus = net.loss_value(x, y)
            values[k] = original - eps
            loss_minus = net.loss_value(x, y)
            values[k] = original
            numerical = (loss_plus - loss_minus) / (2.0 * eps)
            analytical = grads[k]
            denom = max(abs(numerical) + abs(analytical), 1e-12)
            worst = max(worst, abs(numerical - analytical) / denom)
    return worst
```

This is too slow to train with: one forward pass per parameter. But it is the ground truth that keeps the fast method honest. It works for *any* layer and loss because it only ever touches `parameters()`, `forward`, and `backward`. Running it on four configurations (seed 42):

- Logistic regression (SigmoidBCE): worst relative error $\approx 6 \times 10^{-12}$
- Softmax regression (SoftmaxCE): $\approx 1 \times 10^{-10}$
- Tanh MLP (SoftmaxCE): $\approx 7 \times 10^{-9}$
- ReLU MLP (SigmoidBCE): $\approx 1 \times 10^{-9}$

All residuals are near machine precision. One caveat: `ReLU` has a kink at 0 and no derivative there. A finite difference that straddles the kink will disagree with the analytical sub-gradient. With random initial weights the probability of a pre-activation landing exactly on the kink is effectively zero, so the check passes in practice. The tolerance is anchored on a `Tanh` network, which is smooth everywhere; the 10^{-9} figure for the ReLU network is real, not a relaxed threshold masking a bug.

1.10 Closing note: from per-layer to per-operation

Backpropagation here is per-*layer*: each layer carries a local `backward`. But nothing forced the unit to be a layer. If every scalar *operation* (every $+$, every $\times$, every tanh) carried its own local backward, the same reverse pass would compute gradients through an arbitrary expression, with no hand-derived layer gradients at all. That is **automatic differentiation**, and it is what every modern framework's autograd engine does (Baydin et al. 2018). It is the same idea as this library, at a finer grain. Building one is the natural next step, and Chapter 2 returns to this template when framing its own natural extensions.

Bibliographic Notes

The algebraic impossibility argument for XOR is from Minsky and Papert (1969). Their book demonstrated that single-layer perceptrons cannot represent parity or connected figures, which redirected neural network research toward multi-layer architectures for the following decade.

Universal approximation for shallow networks with sigmoidal activations was proved independently by Cybenko (1989) and Hornik, Stinchcombe, and White (1989). Both results apply to a single hidden layer of sufficient width; they say nothing about the efficiency of depth. The empirical case for depth over width is a separate (and still active) story.

Backpropagation in the form the machine-learning community uses it was presented by Rumelhart, Hinton, and Williams (1986), though the underlying chain-rule recursion was known earlier in control theory and numerical analysis.

The UCI optical-recognition dataset (Alpaydin and Kaynak 1998) is the 8×8 image version derived from the handwritten-digits recordings at the National Institute of Standards and Technology. It ships with `sklearn.datasets.load_digits` as a 1 797-sample subset of the full UCI release.

For the full story of automatic differentiation as a discipline distinct from symbolic and numerical differentiation, the survey by Baydin et al. (2018) is the standard reference. It places reverse-mode AD (which is what backpropagation is) in the context of the broader field and covers both the forward and reverse accumulation modes with worked examples.

Goodfellow, Bengio, and Courville (2016) covers the material of this chapter at book length, with particular depth on regularization, optimization, and the practical engineering of deep networks.

The intuition-first style of this book, hand-deriving each gradient and checking it against real computation, follows the tradition of MacKay (2003).

Chapter 2

Output Heads as Inductive Bias

Chapter 1 closed by naming two distinct axes at which the prior enters a supervised model: the **architecture** (how the network computes features) and the **output head** (what the raw output *means*). This chapter takes up the output-head axis, and the central claim is one sentence:

> **Every supervised neural network is a maximum-likelihood estimator under an assumed output distribution. The link function and the matching loss together specify that distribution, and the choice of distribution is itself an inductive bias about the data-generating process.**

Chapter 1 quietly committed to three such distributions (Gaussian, Bernoulli, Categorical) without naming them as choices. This chapter names them, then extends the catalogue to counts (Poisson), to outputs with heteroscedastic noise (Gaussian with predicted variance), and to continuous proportions (sigmoid for bounded targets). Each is a different bet about what kind of object y is.

The architecture axis, the second axis Chapter 1 introduced, continues in Chapter 3 on convolutional networks and through the rest of Part II.

2.1 The unifying frame

Every supervised loss in Chapter 1 had the same shape:

$$L(\theta) = -\frac{1}{N}\sum_{n=1}^{N} \log p_\theta(y_n \mid x_n).$$

Minimizing the loss *is* maximum likelihood under an assumed conditional distribution $p_\theta(y \mid x)$. The network parameterizes that distribution; gradient descent moves the parameters toward higher likelihood of the observed data.

The output head is the bridge. The network produces a raw vector $z = f_\theta(x)$. A **link function** maps z into the natural parameter (or the mean, or a probability vector) of the assumed distribution. The **loss** is the negative log-likelihood of y under that distribution. Chapter 1's organizing principle, *the network produces logits, the loss interprets them*, is exactly this: the link function and the assumed distribution together *are* the interpretation.

That is the whole story. The rest of this chapter is the unpacking.

2.2 The catalogue

Five canonical pairings, each a member of the exponential family with its **canonical link** (the link that makes the gradient come out cleanest, as Section 2.3 will prove):

Output type	Link	Loss	Assumed distribution
Real-valued	identity	MSE	Gaussian (fixed variance)
Binary	logit (sigmoid)	BCE	Bernoulli
Categorical	softmax	cross-entropy	Categorical
Count	log	Poisson NLL	Poisson
Positive continuous	reciprocal	Gamma NLL	Gamma

Table 2.1: Five canonical head pairings. Each row is a different prior about what kind of object y is. The first three are what Chapter 1 already built; the rest of this chapter extends the table.

Each row is a different commitment about what kind of object y is. Real-valued and unbounded? Gaussian. A 0 or 1? Bernoulli. One of K mutually exclusive classes? Categorical. A non-negative integer event count? Poisson.

The link maps the network's unconstrained raw output $z \in \mathbb{R}$ into a valid parameter for the distribution: a real mean for Gaussian, a probability in $[0, 1]$ for Bernoulli, a probability vector on the simplex for Categorical, a positive rate for Poisson.

This chapter develops Poisson in full and points to Beta and MDN as natural extensions. The Gamma row follows the same recipe (positive-support reciprocal link, Gamma NLL) and is a further natural extension; it is included in the catalogue for completeness but not developed here.

2.3 The canonical-link theorem

Chapter 1 observed that both BCE and softmax cross-entropy have gradient $\partial L/\partial z = \mathbf{p} - \mathbf{y}$ with respect to the logits: the section on logistic regression in Section 1.4 derived $p - y$ for the binary case, and Section 1.5 derived $\mathbf{p} - \mathbf{y}$ for the multiclass case. Chapter 1 called that pattern no coincidence but did not prove the general statement. Here it is.

Write an exponential family in natural form:

$$p(y \mid \eta) = h(y) \exp\big(\eta \cdot T(y) - A(\eta)\big),$$

where η is the **natural parameter**, $T(y)$ is the **sufficient statistic**, and $A(\eta) = \log \int h(y) \exp(\eta \cdot T(y))\, dy$ is the **log-partition function**. For Bernoulli, $\eta = \log(p/(1-p))$ is the logit; for Categorical, $\eta_i = \log p_i$ up to a shared shift; for Gaussian (fixed variance), $\eta = \mu/\sigma^2$; for Poisson, $\eta = \log \lambda$.

A foundational identity in exponential families: differentiating the normalization condition $\int p(y \mid \eta)\, dy = 1$ with respect to η yields

$$A'(\eta) = \mathbb{E}[T(y)].$$

The derivative of the log-partition function equals the expected sufficient statistic. Call this $\hat{p}$, the model's prediction in the relevant sense: for Bernoulli, $\hat{p}$ is the success probability; for Categorical, the probability vector; for Gaussian, the mean; for Poisson, the rate.

Now suppose the network's output z *is* the natural parameter, $\eta = z$. This is the **canonical link** choice. The negative log-likelihood is

$$-\log p(y \mid z) = -\log h(y) - z \cdot T(y) + A(z),$$

and its gradient with respect to z is

$$\frac{\partial}{\partial z}[-\log p(y \mid z)] = -T(y) + A'(z) = \hat{p} - y, \tag{2.1}$$

reading y as $T(y)$ (the sufficient statistic; for Categorical, the one-hot of y).

Theorem 2.1. *For any exponential-family distribution with its canonical link, the gradient of the NLL with respect to the network output is the expected sufficient statistic minus the observed.*

Proof. Differentiate $-\log p(y \mid z)$ directly: $\partial/\partial z[-z \cdot T(y) + A(z)] = -T(y) + A'(z)$. By the exponential-family identity, $A'(z) = \hat{p}$. Reading $T(y) = y$ gives $\hat{p} - y$. □

The residual form is not an algebraic accident. The gradient is geometrically the gap between the model's expected value and what was observed; the form of the response variable does not enter.

Chapter 1's pattern is one instance. MSE (identity plus Gaussian) is another: $\partial L/\partial z = z - y$, which is $\hat{p} - y$ with $\hat{p} = z$. This is the same residual Chapter 1 wrote as $\hat{y} - y$ in Equation (1.1): the prediction $\hat{y}$ there is the Gaussian mean, which is exactly the expected sufficient statistic $\hat{p}$ here. The Poisson pairing in Section 2.5 will be a third.

2.4 Chapter 1's three pairings, named

Brief recaps; the derivations are in Chapter 1.

Identity plus Gaussian plus MSE. Assumed distribution: $y \mid x \sim \mathcal{N}(\mu(x), \sigma^2)$ with σ^2 fixed and absorbed into the constant. Link: identity (the network output z is μ directly). NLL (dropping constants): $\frac{1}{2}(y - z)^2$, mean squared error. Gradient: $z - y$. Inductive bias: errors are symmetric, additive, and Gaussian. If the true distribution is heavy-tailed or skewed, MSE pays for that with biased estimates.

Logit plus Bernoulli plus BCE. Assumed distribution: $y \mid x \sim \text{Bernoulli}(p(x))$, $y \in \{0, 1\}$. Link: logit, $\eta = \log(p/(1-p)) = z$; the inverse is sigmoid, $p = \sigma(z)$. NLL: $-y \log p - (1-y)\log(1-p)$, computed stably from z. Gradient: $p - y$. Inductive bias: the response is binary; a probabilistic prediction is well calibrated when training converges.

Softmax plus Categorical plus cross-entropy. Assumed distribution: $y \mid x \sim \text{Cat}(\mathbf{p}(x))$, $y \in \{0, \ldots, K-1\}$. Link: softmax, $\mathbf{p} = \text{softmax}(\mathbf{z})$. NLL: $-\log p_y = \text{logsumexp}(\mathbf{z}) - z_y$ in the stable form. Gradient: $p_i - \mathbf{1}[i = y]$, i.e. $\mathbf{p} - \mathbf{y}$ with $\mathbf{y}$ the one-hot. Inductive bias: the classes are mutually exclusive and exhaustive; the model cannot represent two classes at once.

Three different objects ($y \in \mathbb{R}$, $y \in \{0, 1\}$, $y \in \{0, \ldots, K-1\}$), three different distributions, three different links. One pattern: $\hat{p} - y$.

2.5 Count data: log link and Poisson NLL

The first genuinely new pairing.

Assumed distribution: $y \mid x \sim \text{Poisson}(\lambda(x))$, $y \in \{0, 1, 2, \ldots\}$, non-negative integers. The PMF is $p(y \mid \lambda) = \lambda^y e^{-\lambda}/y!$.

Link: logarithm, $\eta = \log \lambda = z$, so $\lambda = e^z$. The exponential guarantees $\lambda > 0$ for any real z, without any clamping or projection.

NLL:

$$-\log p(y \mid \lambda) = \lambda - y \log \lambda + \log y! = e^z - yz + \log y!$$

The $\log y!$ term does not depend on z and drops from the gradient. For the loss value during optimization we omit it as well (it shifts the loss by a y-dependent constant but does not affect the optimum). The comparative NLL values reported in the experiment likewise omit $\log y!$; the constant cancels in model-to-model comparisons on the same test set, so the values are valid for relative comparison but should not be compared against literature benchmarks that include it.

Gradient:

$$\frac{\partial L}{\partial z} = e^z - y = \lambda - y = \hat{p} - y.$$

A third instance of Theorem 2.1, with no new derivation needed.

In `scratchnn`:

```
class PoissonNLLLoss(Loss):
    """Log-link Poisson negative log-likelihood. Expects a single logit.

    The raw output z is interpreted as a log-rate. The predicted rate is
    lambda = exp(z), guaranteed positive. Target y is a count (typically a
    non-negative integer, but any real is accepted). The constant log(y!)
    is omitted; it does not depend on z and does not affect optimization.

    Canonical-link form: dL/dz = exp(z) - y = lambda - y.
    """

    def value(self, logits, y):
        z = logits[0]
        return math.exp(z) - z * y

    def grad(self, logits, y):
        z = logits[0]
        return [math.exp(z) - y]

    def probs(self, logits):
        # Inverse of log link: the predicted rate is exp(z).
        return [math.exp(logits[0])]
```

Ten lines; same hand-derived backward as the other losses. The gradient check in Section 1.9 passes at 1.5×10^{-10} for this loss on a Tanh MLP.

Worked experiment. A 1-D regression where the true rate is non-monotonic: $\lambda(x) = \max(0.1,\, 2 + 5\sin(\pi x))$ for $x \in [0, 2]$. Targets are Poisson counts. Two identical bodies ($1 \to 16 \to 1$, Tanh hidden), one with identity plus MSE and one with log plus Poisson NLL (data seed 0, model seed 0, 80 epochs). See Figure 2.1.

Findings, honestly. On this benign synthetic dataset both models fit the conditional mean reasonably well. Neither predicts negative rates on a fine grid, because the Tanh activations keep predictions in a sensible range.

- The MSE model's minimum predicted rate on a 201-point grid is 0.152; the Poisson model's minimum is 0.139. Both have zero negative predictions here. The Poisson head cannot predict negative rates by construction; the MSE head has no such guarantee.
- The MSE model will predict negative rates if the data are less benign (sparser counts, more extreme non-monotonicity, or a different random seed). The Poisson model cannot, ever.
- On a count dataset where many observations are zero (rare events), the MSE model genuinely

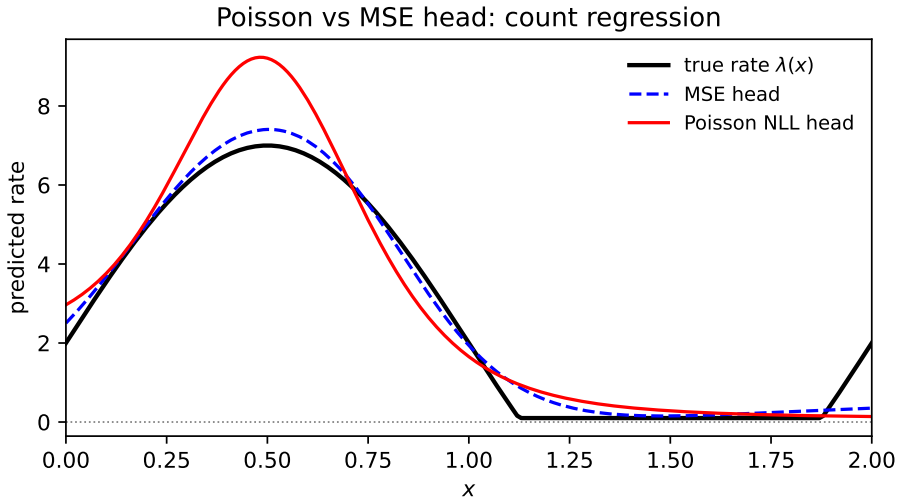

Figure 2.1: Poisson versus MSE head on count regression. True rate $\lambda(x)$ (black), MSE prediction (blue dashed), Poisson NLL prediction (red). Both heads track the mean; neither predicts negative rates on a fine grid. The MSE model's minimum predicted rate is 0.152 and the Poisson model's is 0.139. The Poisson head cannot predict negative rates by construction; the MSE head happens not to here, but carries no such guarantee.

fails: it pays MSE penalty for being slightly positive on average, and its predictions cross zero. The Poisson head never does.

The Poisson NLL also accounts for the Poisson property $\mathrm{Var}(y) = \mathbb{E}(y) = \lambda$. MSE assumes constant variance, which is wrong for counts where small means have small variance and large means have large variance. On a benign dataset this matters less; on real count data it can matter a great deal.

Use the matching head as structural insurance even when, as here, the constraint was not violated on this dataset. The guarantee holds regardless of the data; the MSE model's accidental non-negativity does not.

2.6 Uncertainty: heteroscedastic Gaussian NLL

Chapter 1's MSE loss assumed Gaussian errors with *constant* variance, absorbed into the loss constant. That is fine when the noise really is constant. When the noise is input-dependent the model has no way to report it. A point prediction does not say "I am confident here" or "I am unsure here." The user has to find that out the hard way.

Heteroscedastic Gaussian regression fixes this with a two-output head. The network outputs (z_μ, z_s) per example, and the loss interprets them as the parameters of a per-input Gaussian:

$$\mu = z_\mu, \qquad \sigma = \mathrm{softplus}(z_s).$$

Softplus, $\mathrm{softplus}(z) = \log(1 + e^z)$, is the smooth analogue of the positive part. It guarantees $\sigma > 0$ for any $z_s \in \mathbb{R}$, without clamping; its derivative is the logistic sigmoid, $\mathrm{softplus}'(z) = \sigma_{\mathrm{logistic}}(z)$.

NLL (dropping the $\frac{1}{2}\log(2\pi)$ constant):

$$L = \frac{(y-\mu)^2}{2\sigma^2} + \log\sigma.$$

The first term is the familiar squared error, rescaled by the predicted precision $1/\sigma^2$. A confident prediction (small σ) gets a *larger* MSE-like gradient. The second term prevents the model from inflating σ to infinity to wash out the first term.

Gradients, hand-derived:

$$\frac{\partial L}{\partial \mu} = \frac{\mu - y}{\sigma^2}.$$

The MSE gradient, scaled by predicted precision.

$$\frac{\partial L}{\partial \sigma} = -\frac{(y-\mu)^2}{\sigma^3} + \frac{1}{\sigma}.$$

Zero when $(y-\mu)^2 = \sigma^2$: the maximum-likelihood point, where the model sets σ equal to the realized residual.

Chaining through softplus to get the gradient with respect to the network output z_s:

$$\frac{\partial L}{\partial z_s} = \frac{\partial L}{\partial \sigma} \cdot \sigma_{\text{logistic}}(z_s).$$

In `scratchnn`:

```
def softplus(z):
    """Softplus: log(1 + exp(z)). The smooth analogue of the positive part.

    Evaluated as max(z, 0) + log1p(exp(-|z|)) to avoid overflow for large |z|.
    Its derivative is the logistic sigmoid, sigmoid(z).
    """
    return max(z, 0.0) + math.log1p(math.exp(-abs(z)))


class GaussianNLLLoss(Loss):
    """Heteroscedastic Gaussian NLL. Expects two logits, (z_mu, z_s).

    The first logit is the mean mu = z_mu (identity link on the mean). The
    second logit z_s is mapped through softplus to a positive scale,
    sigma = softplus(z_s), enforcing positivity without clamping. Target y
    is a single real number. The constant 0.5 * log(2*pi) is omitted.

    Loss: 0.5 * (y - mu)**2 / sigma**2 + log(sigma).
    Gradients are hand-derived; the chain through softplus uses
    d(softplus(z_s))/dz_s = sigmoid(z_s).
    """

    def value(self, logits, y):
        mu = logits[0]
        s = softplus(logits[1])
        return 0.5 * (y - mu) ** 2 / (s * s) + math.log(s)

    def grad(self, logits, y):
        mu = logits[0]
        z_s = logits[1]
        s = softplus(z_s)
        dmu = (mu - y) / (s * s)
        ds = -((y - mu) ** 2) / (s ** 3) + 1.0 / s
        return [dmu, ds * sigmoid(z_s)]
```

```
def probs(self, logits):
    # Report (mu, sigma) for the predictive Gaussian.
    return [logits[0], softplus(logits[1])]
```

The `softplus` helper is listed above in full; `GaussianNLLLoss` has fifteen lines excluding docstring. Same `Loss` interface as the other heads; the only novelty is that the head produces *two* numbers per example.

Worked experiment. A 1-D function with input-dependent noise: $y = \sin(x) + \varepsilon$ where $\varepsilon \sim \mathcal{N}(0, \sigma(x)^2)$ and $\sigma(x) = |x|/3 + 0.1$ over $x \in [0, 6]$. Noise is small on the left, growing to about $\sigma = 2$ on the right. Two networks of identical body ($1 \to 16 \to 1$ for MSE, $1 \to 16 \to 2$ for heteroscedastic), data seed 0, model seed 0, 300 epochs. See Figure 2.2.

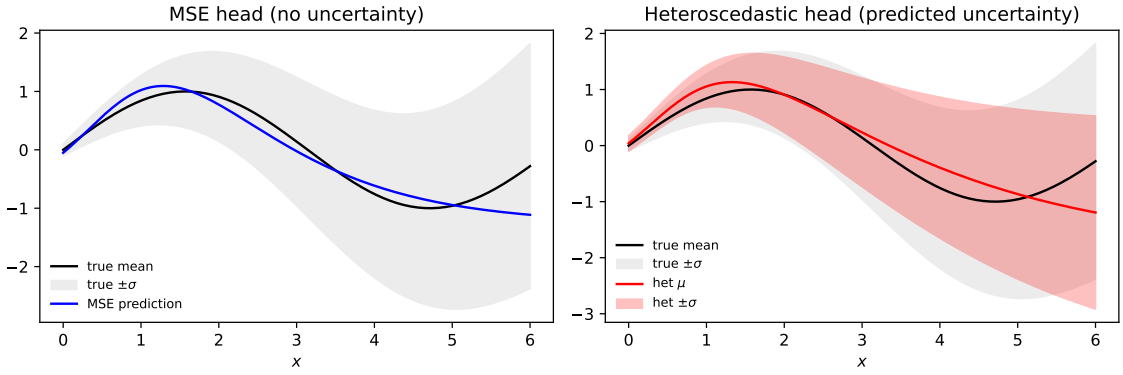

Figure 2.2: MSE head versus heteroscedastic Gaussian head. Left: MSE prediction (blue) against the true mean (black) with the true $\pm\sigma$ band (grey). The MSE model predicts only a mean. Right: heteroscedastic head, which predicts both a mean (red) and a per-input uncertainty band (pink). The predicted band tracks the true band closely, widening on the noisy right side.

Findings:

x	true μ	true σ	MSE pred	het μ	het σ
0.5	0.479	0.267	0.568	0.635	0.234
1.5	0.997	0.600	1.059	1.114	0.525
2.5	0.598	0.933	0.371	0.584	0.821
3.5	−0.351	1.267	−0.356	−0.098	1.115
4.5	−0.978	1.600	−0.806	−0.650	1.392
5.5	−0.706	1.933	−1.044	−1.046	1.625

Table 2.2: Predictions at six query points. The heteroscedastic model recovers the input-dependent σ within 10–16% across the range. The MSE model has no σ at all.

The heteroscedastic model tracks the true σ within 10–16% across the range (worst case at $x = 5.5$: predicted 1.625 versus true 1.933). The MSE model has no σ at all; it only predicts the mean.

To compare on equal footing: fit a single global σ for the MSE model post-hoc (the RMS training residual), then compute Gaussian NLL on the test set under each model's predicted distribution.

The heteroscedastic head improves test NLL by 0.24 nats per sample. That is the calibration

Model	Test Gaussian NLL
MSE model + global σ	0.745
Heteroscedastic head	0.509

Table 2.3: Test Gaussian NLL on 200 held-out samples (lower is better). Both values omit the $\frac{1}{2}\log(2\pi)$ constant, which cancels in the comparison; they are valid for relative ranking but should not be compared to benchmarks that include the constant. Numbers regenerated from the paired notebook with seed 0.

premium: predicting variance separately for each x is worth a quarter of a nat over the homoscedastic assumption on this dataset. The qualitative gain is bigger than the numerical one. The MSE model plus global σ is overconfident on the noisy right side and underconfident on the clean left side; the heteroscedastic head is correct on both.

This is the cleanest demonstration in this chapter that the choice of head buys something tangible (calibrated, input-dependent uncertainty) that the standard head cannot provide.

2.7 Link and likelihood are independent choices

The catalogue in Section 2.2 paired each likelihood with its canonical link. That is the *natural* pairing, the one that makes the gradient collapse to $\hat{p} - y$ and the optimization land cleanly. It is not the *only* pairing.

Suppose the target y is a continuous proportion: $y \in [0, 1]$, something like "fraction of pixels lit," "vote share," or "relative humidity." Not a class label, a *number*. We want the prediction $\hat{p} \in (0, 1)$ as well.

The natural link to enforce $\hat{p} \in (0, 1)$ is the sigmoid: $\hat{p} = \sigma(z)$. We can pair sigmoid with two different likelihoods:

Sigmoid plus MSE on the sigmoid output. Loss: $L = \frac{1}{2}(y - \sigma(z))^2$. Implicit likelihood: Gaussian noise in proportion space. Gradient: $(\sigma(z) - y) \cdot \sigma'(z)$, the chain rule. This works fine in the bulk of the domain but saturates near $z \to \pm\infty$ (where $\sigma'(z) \to 0$ and gradient flow stalls), making it hard for the model to push predictions very near the boundaries.

Sigmoid plus Beta NLL. Predict the mean $\mu = \sigma(z_\mu)$ and a "precision" $\nu = \text{softplus}(z_\nu)$, set $\alpha = \mu\nu$, $\beta = (1 - \mu)\nu$, and use the Beta NLL,

$$L = -(\alpha - 1)\log y - (\beta - 1)\log(1 - y) + \log B(\alpha, \beta).$$

This treats y as drawn from a Beta distribution. It is far more sensitive than the MSE choice near the boundaries (where Beta log-density is heavy-tailed in the right way for proportion data) and predicts a full posterior over $[0, 1]$ instead of a point.

Same link (sigmoid). Different likelihoods (Gaussian on the sigmoid output versus Beta directly). The choice changes what is assumed about the noise distribution, how the loss treats extreme values, and whether the head is one output or two.

This breaks the catalogue's apparent one-to-one. The link function is not uniquely determined by the assumed distribution. There is a canonical pairing, the one the theorem gives, but it is one choice among many. The skill is recognizing the choice.

Beta NLL is not in `scratchnn`, but it is a natural extension: it requires only `math.lgamma`, which is in the standard library. The components (sigmoid, softplus, and the Beta log-density) all follow the same `Loss` interface as the losses already built. The pedagogical point of this section is that link and likelihood are independent, not that every likelihood needs to be in the library.

2.8 Beyond unimodal: a pointer to mixture density networks

All of the heads above assumed $p(y \mid x)$ is unimodal. When it is not, a unimodal head collapses: a robot arm with two valid joint configurations for the same end-effector pose, an inverse problem with multiple solutions, the conditional next-frame distribution for a stochastic video model. A Gaussian head will predict the mean of the modes, which may lie in a region of *zero density* under the true distribution.

A **mixture density network (MDN)** outputs the parameters of a K-component Gaussian mixture: K means, K scales, and K mixture weights (the latter through a softmax head). The NLL is

$$-\log \sum_{k=1}^{K} \pi_k(x)\, \mathcal{N}(y \mid \mu_k(x), \sigma_k(x)^2).$$

Standard regression collapses to the midpoint between two modes; the MDN captures each mode with its own component, weighted by its own probability.

The MDN is not in `scratchnn`. It is a natural extension, framed the same way Section 1.10 frames automatic differentiation: the idea is the same framework (output head plus matching NLL), the implementation composes pieces the library already has (softplus for the scales, softmax for the weights, logsumexp for the stable mixture log-density), plus the hand-derived gradient through the mixture. Building it is the next thing to build, not a new concept.

2.9 Inductive bias, the parallel axis

Section 1.7 in Chapter 1 named the two axes at which prior knowledge enters a supervised model and deferred the full synthesis. The preceding eight sections have developed the output-head axis in detail. The two axes are *independent and composable*: any architecture pairs with any output head, provided the head's input dimension matches the architecture's output. The composability is literal: the `Network` class takes a list of layers and a loss; the layers are the body (any architecture), the loss is the head (any output distribution). Nothing in the body knows about the head, and nothing in the head knows about the body.

The composability matrix. Four concrete examples that cross body type with head type:

- **CNN body, Poisson head**: pixel-wise photon-counting imaging. The CNN imposes locality and translation equivariance; the Poisson head encodes that pixel counts are non-negative integers with variance equal to mean.

- **Transformer body, Categorical head over a vocabulary**: a language model. The Transformer imposes pairwise content-addressable attention; the Categorical head encodes that the next token is one of K mutually exclusive choices.

- **MLP body, heteroscedastic Gaussian head**: regression with calibrated, input-dependent uncertainty. The MLP makes no spatial or sequential assumption; the Gaussian head encodes that the response is real-valued with per-input noise.

- **CNN body, Bernoulli head per pixel**: binary segmentation. The CNN imposes spatial structure; the Bernoulli head encodes that each pixel is independently foreground or background.

In each case the architecture is a prior about the *function* (how features compose) and the head is a prior about the *distribution* (what y is). They multiply: a matched head buys sample efficiency *on top of* a matched body, not instead of it.

What a matched head gives you:

- **Sample efficiency.** Constraints encoded by the link (non-negativity via log, boundedness via sigmoid, simplex membership via softmax) are free. The network does not spend parameters on relearning them from data.
- **Calibrated uncertainty.** A two-output Gaussian head reports per-input σ. A Categorical head reports a probability vector. Both are usable by downstream decisions without a post-processing step.
- **Honesty about the data.** A Poisson head says "I am modeling counts." That is a statement to the reader of the code, as much as a structural constraint on the optimizer.

What a mismatched head costs you:

- Wasted capacity: the network must discover, from data, constraints the link could have given for free (non-negativity, boundedness, normalization).
- Pathological predictions in low-density regions: negative counts under MSE, point estimates between modes under a unimodal head, miscalibrated confidence under a saturated softmax.
- Misleading uncertainty: MSE gives no uncertainty at all; a poorly calibrated softmax is a known failure mode in deployed classifiers.

This shape is identical to the architecture story in Section 1.7. Architectural priors and output-head priors are parallel commitments, each evaluable on the same questions: does the prior match the data? does it improve sample efficiency when matched? does it hurt when mismatched?

The two axes interact but do not conflate. Getting both right is a multiplication, not an addition: a language model is not just a good architecture (Transformer) or just a good head (Categorical), it is the combination. The architecture encodes how context matters; the head encodes what the prediction is. Neither is optional.

2.10 Library additions

`src/scratchnn/neural_net.py` contains the following additions relevant to this chapter. All fit the `Loss` interface (`value`, `grad`, `probs`); all are standard-library only; all have a `gradient_check` case.

- **`softplus(z)`** (helper): $\log(1+e^z)$, evaluated as $\max(z,0)+\log(1+e^{-|z|})$ to avoid overflow. Used inside `GaussianNLLLoss` to map a real-valued network output to a positive scale parameter. Derivative is the sigmoid.
- **`MSELoss`**: identity link, mean squared error. Single-output regression. The $\frac{1}{2}$ convention makes the gradient $z - y$, matching the canonical $\hat{p} - y$ pattern of the other losses.
- **`PoissonNLLLoss`** (ten lines): single logit, log link. $L = e^z - yz$ (omitting $\log y!$). $\partial L/\partial z = e^z - y$. `probs` returns e^z.

- `GaussianNLLLoss` (fifteen lines): two logits, $\mu = z_\mu$, $\sigma = \text{softplus}(z_s)$. $L = \frac{(y-\mu)^2}{2\sigma^2} + \log\sigma$. Gradient derived in Section 2.6. `probs` returns (μ, σ).

Gradient-check residuals on a Tanh MLP (seed 42): MSE 9.7×10^{-10}, Poisson 1.5×10^{-10}, Gaussian 1.1×10^{-10}. All well below the 10^{-4} tolerance.

Not in the library: Beta NLL and the MDN. Both are natural extensions (see Sections 2.7 and 2.8), not missing infrastructure. Beta NLL needs only `math.lgamma`; the MDN composes existing primitives. They are the next things to build, not gaps.

2.11 Handoff

This chapter took up the output-head axis of inductive bias in full. The unifying frame: every supervised network is a maximum-likelihood estimator under an assumed output distribution. The canonical-link theorem (Theorem 2.1): with the canonical link, the gradient of the NLL is always $\hat{p} - y$. Three review pairings from Chapter 1 (Gaussian, Bernoulli, Categorical), now named as members of one family. Two new worked heads: Poisson for count data, and the heteroscedastic Gaussian for calibrated, input-dependent uncertainty. One example of link/likelihood independence (sigmoid plus Beta). A pointer to mixture density networks. And the parallel-axes synthesis showing how the architecture axis and the output-head axis compose.

The architecture axis continues in the next chapter, on convolutional networks: the prior that nearby inputs matter together and the same feature detector should fire anywhere. After CNNs, the following chapters cover fixed-context language models, recurrent networks, and the Transformer, each encoding a different structural prior. Every architectural choice in those chapters composes freely with any of the output heads in this chapter: a language model is a Transformer body with a Categorical head over a vocabulary; pixel-wise depth estimation is a CNN body with a Gaussian head; photon-counting imaging is a CNN body with a Poisson head. The two axes multiply.

The book closes with a chapter on reinforcement learning, where the training signal shifts from a per-example label to a scalar reward over whole trajectories. The heads-as-bias frame still applies there (a policy head is a softmax over actions; a value head is identity plus MSE on returns), but the supervised-learning frame, the frame this chapter closes, no longer holds. That shift is the genuinely different third learning paradigm.

Bibliographic Notes

The exponential-family view of loss functions and the canonical-link connection between the link function and the gradient structure originate in the theory of generalized linear models (GLMs). The seminal paper is Nelder and Wedderburn (1972); the standard book-length treatment is McCullagh and Nelder (1989). Both are foundational references in statistics; the neural-network community largely rediscovered the same ideas under different names.

Heteroscedastic Gaussian regression with a two-output network, predicting mean and variance jointly, is due to Nix and Weigend (1994). The formulation in this chapter (predicting log-variance rather than variance directly, to enforce positivity) is a minor variant standard in modern practice.

Mixture density networks are introduced in Bishop (1994) and treated at length in Bishop's later textbook (Bishop 2006, chapter 5). The implementation using softmax weights, softplus scales, and logsumexp for numerical stability is the standard form. Goodfellow, Bengio, and Courville (2016) covers MDNs in the context of structured output distributions.

For the general treatment of exponential families, see any standard probability textbook; MacKay (2003) is an accessible entry point that also connects to information theory.

Part II

Architectural Inductive Biases

If the output head is a prior over the response, the architecture is a prior over how features compose. Part II takes the major families one at a time and names the structural assumption each one bakes in. Convolutional networks assume locality and translation equivariance (Chapter 3). Fixed-context language models assume a bounded window suffices (Chapter 4). Recurrent networks assume time-translation equivariance and squeeze the past through a state bottleneck (Chapter 5). Transformers assume nothing about which positions matter and instead learn a content-addressable lookup, on the fly, position by position (Chapter 6). Each chapter states the prior, builds a model small enough to test it, and checks whether the assumption holds.

Chapter 3

Convolutional Networks: Locality and Translation Equivariance

Chapter 1 named two axes at which a prior enters a supervised model: the **output head**, which fixes what the raw output means, and the **architecture**, which fixes how the network computes its features. Part I followed the output-head axis. Part II follows the other one, and this chapter takes its first non-trivial step.

The architecture is a prior over how features compose. A multilayer perceptron makes the weakest possible such commitment: every input coordinate may interact with every other on equal footing, and which interactions matter is left entirely to the data. That generality is a strength on tabular features with no inherent geometry. On an image it is a waste. Pixels are arranged in a plane, neighbors mean something, and a digit is the same digit a few pixels to the left. An MLP knows none of this; it has to relearn the plane from scratch every time.

A convolutional network is what we build when we refuse to throw that structure away. It bakes two assumptions into the operator itself rather than leaving them to be learned. **Locality**: a unit reads a small neighborhood, not the whole input. **Translation equivariance**: the same unit is applied at every position, so a feature detected in one place is detected everywhere. The whole chapter is the unpacking of that one move, in math, in code, and in a falsification experiment. It closes by marking a boundary: the convolution is the last large layer this book writes in pure Python.

3.1 One unit, reused everywhere

The multilayer perceptron of Chapter 1 classifies the 8×8 UCI digits at about 96% test accuracy. That looks reasonable until you ask what the model believes about its input. The MLP treats each of the 64 pixels as an independent feature. Pixel $(3, 4)$ and pixel $(3, 5)$, which are neighbors in the image, are no more related to it than pixel $(3, 4)$ and pixel $(7, 0)$, which are at opposite ends. The MLP has no concept of adjacency. It learns relationships between pixels from data, exactly as it would learn relationships between any other tabular features.

A convolutional unit refuses to start from that blank slate. Recall that a `Linear` layer is one or more weight vectors $\mathbf{w}$ paired with biases b, computing a logit $z = \mathbf{w} \cdot \mathbf{x} + b$ (Section 1.3). A convolutional unit is the same dot product, restricted to a small **kernel** of size $k \times k$ and applied at

every position of a 2-D input:

$$z_{r,c} = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} W_{i,j}\, x_{r+i,\, c+j} + b.$$

The kernel W is shared across all output positions (r, c). There is *one* W, not one per position. That single restriction is the whole definition of a convolution. The small window is the **locality** prior: a unit looks at a neighborhood, never the entire input at once.

Everything else is bookkeeping, and there is little of it:

- **Borders.** When the kernel runs off the edge of the input, we either drop those positions (no padding, the choice this chapter uses) or pad the input with zeros to preserve the spatial size.
- **Stride.** Sliding by one pixel between applications is the default used here; a larger stride downsamples the output.
- **Channels.** A real image has several input channels (RGB has three), and a layer has several output channels too, one per independent kernel run in parallel. With C_{in} input channels and C_{out} output channels, a kernel has shape (C_{in}, k, k), dotting through every input channel at each spatial position, and the layer stacks C_{out} such kernels.

For single-channel 8×8 digits with $k = 3$, no padding, stride one, and four output channels, the layer produces an output of shape $(4, 6, 6)$: four feature maps, each 6×6 because the kernel cannot fit at the last two positions in either dimension. The forward pass that computes it is shown in Section 3.4.

3.2 Weight sharing is the translation prior

Reusing W at every position is what makes the layer **translation equivariant**. Stated cleanly: if x' is x shifted by $(\Delta r, \Delta c)$, then the layer's output on x' is its output on x shifted by the same $(\Delta r, \Delta c)$, modulo whatever falls off the borders. A detector that fires on a stroke in one corner fires on the same stroke anywhere.

This is not a property the network *learns*. It is a property of the *operator*: true at initialization, true after training, true for every kernel. The reason is the sharing itself. Shift the input by one pixel and every windowed dot product is now computed at a position also shifted by one, so the output simply slides. The MLP has no such guarantee. Each of its output units carries its own weight vector tied to specific coordinates of the input, so translating the input scrambles which weights see which pixels.

Weight sharing and translation equivariance are therefore the same fact, read once from the parameter side and once from the function-symmetry side. The architecture commits to a symmetry of the data by reusing parameters across the positions that symmetry relates.

3.3 The parameter count tells the story

The compactness of weight sharing shows up immediately in the parameter count, and the count is the cleanest single statement of the prior.

The MLP for the 8×8 digits has shape $64 \to 32 \to 10$ (Section 1.3). Its parameter count is

$$64 \cdot 32 + 32 + 32 \cdot 10 + 10 = 2410.$$

A small convolutional network for the same task uses one conv layer of four 3×3 kernels followed by a fully connected head:

```
Conv2D(in_channels=1, out_channels=4, kernel_size=3)
ReLU()
Linear(144, 10)
```

The conv layer has $4 \cdot (1 \cdot 9) + 4 = 40$ parameters. With no padding and stride one its output is $4 \times 6 \times 6 = 144$ units, so the head is `Linear(144, 10)` with $144 \cdot 10 + 10 = 1450$ parameters. The total is $40 + 1450 = 1490$, about 38% smaller than the MLP. Almost all of the model's spatial capacity sits in the 40 parameters of the conv kernel; the head is a thin reader on top of the feature maps.

The comparison is sharper if we ask what it would cost to replace the conv layer with a `Linear` layer producing the same 144-dimensional output. That layer is `Linear(64, 144)`, with $64 \cdot 144 + 144 = 9360$ parameters. The convolution accomplishes the same input-to-output shape transformation with 40 parameters instead of 9360:

$$\frac{9360}{40} = 234. \tag{3.1}$$

That factor of 234 is the explicit price of *not* committing to locality and translation equivariance. The forward cost is modest in proportion: four output channels, one input channel, a 3×3 kernel, over 6×6 output positions is $4 \cdot 1 \cdot 9 \cdot 36 = 1296$ multiply-adds per example. The prior buys a $234\times$ reduction in parameters at the cost of a fixed, transparent amount of arithmetic. Both numbers are regenerated by the chapter's paired notebook.

3.4 Forward pass, in code

The `Conv2D` layer in `scratchnn` holds `kernels` of shape $(C_{\text{out}}, C_{\text{in}}, k, k)$ and a `bias` of shape $(C_{\text{out}},)$. Inputs and outputs are flat `list[float]`, indexed as `c * H * W + r * W + col` for channel `c`, row `r`, column `col`. The flat layout keeps the rest of the library working unchanged: to `ReLU`, `Tanh`, and `Linear`, the conv output is just a flat vector of length $C_{\text{out}} \cdot H' \cdot W'$, with the 2-D shape folded into the indexing rather than carried as a nested list.

The forward pass is five nested loops, with no NumPy:

```
for c_out in range(C_out):
    kernel = kernels[c_out]
    b = bias[c_out]
    for r in range(out_h):
        for col in range(out_w):
            z = b
            for c_in in range(C_in):
                for i in range(k):
                    for j in range(k):
                        z += kernel[c_in, i, j] * x[c_in, r + i, col + j]
            out[c_out, r, col] = z
```

Five loops looks unflattering, and it is exactly as expensive as the math says: $C_{\text{out}} \cdot C_{\text{in}} \cdot k^2$ multiply-adds per output spatial position, times $H' \cdot W'$ positions. For the digit network of Section 3.3 that is $4 \cdot 1 \cdot 9 \cdot 36 = 1296$ multiply-adds per forward pass. The nesting makes the expense visible in the code, which is the pedagogical point: nothing is hidden inside a tensor call. The input `x` is cached for the backward pass, the same pattern `Linear` uses in Section 1.3.

This is the largest layer this book writes in pure Python. The five-loop form is honest about its cost, and at this scale the cost is affordable.

3.5 Backward pass, by careful chain rule

Let $g_{c_{\text{out}},r,c} = \partial L / \partial z_{c_{\text{out}},r,c}$ be the gradient flowing into the output feature map. Three quantities follow: gradients with respect to the kernel weights, the bias, and the input.

Kernel gradient. Each weight $W_{c_{\text{out}},c_{\text{in}},i,j}$ is read at *every* output position (r, c) in the forward pass. By the chain rule its gradient sums over those positions:

$$\frac{\partial L}{\partial W_{c_{\text{out}},c_{\text{in}},i,j}} = \sum_{r,c} g_{c_{\text{out}},r,c}\, x_{c_{\text{in}},\, r+i,\, c+j}. \tag{3.2}$$

This is the `Linear` formula $g_i\, x_j$ from Section 1.3, summed over the spatial axes. The reuse of W in the forward pass is exactly what causes its gradient to accumulate in the backward pass.

Bias gradient. The bias adds to every output position on its feature map equally, so its gradient is the sum of the incoming gradients there:

$$\frac{\partial L}{\partial b_{c_{\text{out}}}} = \sum_{r,c} g_{c_{\text{out}},r,c}.$$

Input gradient. The pixel $x_{c_{\text{in}},r',c'}$ was read by the output positions (r, c) with $r + i = r'$ and $c + j = c'$, that is $(r, c) = (r' - i, c' - j)$. Summing over all kernels and the valid positions where the pixel was used,

$$\frac{\partial L}{\partial x_{c_{\text{in}},r',c'}} = \sum_{c_{\text{out}}} \sum_{i,j} W_{c_{\text{out}},c_{\text{in}},i,j}\, g_{c_{\text{out}},\, r'-i,\, c'-j},$$

with out-of-range terms dropped. This is the spatial transpose of the forward pass; in convolutional-network jargon it is a *transposed convolution*.

All three fall out of one pass. Here is the actual `Conv2D.backward` from `scratchnn`. The offset variables are flat-index bookkeeping (the 2-D shape folded into a `list[float]`, channel-major then row-major); the three lines that carry the math are the `+=` accumulations, into the bias, into the shared kernel, and into the input:

```
def backward(self, g):
    k, in_w, in_hw = self.k, self.in_w, self.in_h * self.in_w
    out_w, out_hw = self.out_w, self.out_h * self.out_w
    dx = [0.0] * len(self.x)
    for c_out in range(self.out_channels):
        kernel, dkernel = self.kernels[c_out], self.dkernels[c_out]
        for r in range(self.out_h):
            for col in range(self.out_w):
                g_rc = g[c_out * out_hw + r * out_w + col]
                self.dbias[c_out] += g_rc # dL/db
                for c_in in range(self.in_channels):
                    ox = c_in * in_hw + r * in_w
                    ok = c_in * k * k
                    for i in range(k):
                        for j in range(k):
                            xidx = ox + i * in_w + col + j
                            kidx = ok + i * k + j
```

```
                        dkernel[kidx] += g_rc * self.x[xidx] # dL/dW
                        dx[xidx] += g_rc * kernel[kidx] # dL/dx
    return dx
```

The observation that matters: every accumulation in `Conv2D` is the *same* `+=` pattern the library already uses for mini-batch gradients in `Linear` (Section 1.3). The mini-batch sums per-example gradients; the convolution sums per-position gradients. Because one kernel weight is read at every output position in `forward`, the line `dkernel[kidx] += ...` fires at every position in `backward`. Weight sharing and gradient accumulation are one fact read forward and backward: the same operator, at a finer grain.

The `Layer` contract from Section 1.3 carries over unchanged. `Conv2D` implements `forward`, `backward`, and `parameters`, and the generic optimizer, written once, never learns that a convolution exists. A numerical gradient check confirms the hand-derived backward: the worst relative error is on the order of 10^{-9} across small random inputs, well under the standard 10^{-4} tolerance, regenerated by the chapter's paired notebook.

3.6 Training on 8x8 digits

We train the convolutional network against a multilayer-perceptron baseline on UCI optdigits (3823 training, 1797 test, ten classes), pixel values normalized to $[0, 1]$, mini-batch stochastic gradient descent. This is the fuller optdigits release; Chapter 1 used the smaller sklearn subset of the same collection (1437 train, 360 test), which is why the multilayer perceptron's test accuracy here, 96.1%, sits a touch below the 97.2% it reached there. The two models in this chapter use the same hyperparameters, 40 epochs at learning rate 0.1 with batch size 32, so the CNN-versus-MLP comparison is on equal footing. Both finish in a few minutes of pure Python.

Model	Architecture	Parameters	Test accuracy
MLP	$64 \to 32 \to 10$	2410	96.1%
CNN	`Conv2D(1, 4, k=3)` $\to$ `ReLU` $\to$ `Linear(144, 10)`	1490	95.4%

The convolutional network is about 38% smaller in parameter count and lands within a percentage point of the MLP. It does not dominate. There are two readings, and they have very different consequences:

1. The convolutional prior is wrong for this data, so locality does not help.

2. The prior is right, but the data has already had locality baked into its preprocessing, so asserting locality again gains little.

The second reading is the correct one. The 8×8 UCI digits are not raw images. They were produced from the original 32×32 NIST bitmaps by counting the "on" pixels in each 4×4 block. That block-counting step is itself a fixed, translation-invariant local feature extractor: every 8×8 pixel is already a small local aggregation over a contiguous 4×4 patch. By the time the convolution sees the data, much of the spatial work is done, which is why the absolute gap is small. On raw 28×28 MNIST or on actual photographs the gap between an MLP and a CNN is far larger, a few percentage points to tens, precisely because the spatial structure has not been pre-summarized away. To decide between the two readings on this dataset, rather than argue from the gap, we run an experiment.

3.7 The permuted-pixel control

Apply a single fixed random permutation to the 64 input pixels, then train and test both models on the permuted version. For a human the result is unreadable: the digits are scrambled past recognition. For the MLP every pixel position was interchangeable to begin with, so a relabeling of the coordinates changes nothing it can see, and its accuracy should be essentially unchanged. For the convolutional network the locality assumption is now *actively wrong*: neighbors in the permuted input are no longer neighbors in the original digit, so a 3×3 kernel reads a meaningless window of unrelated pixels. The same fixed permutation is applied to both models, under the same training regime as before.

Model	Pixels	Test accuracy	Drop from standard
MLP (64-32-10)	standard	96.1%	
MLP (64-32-10)	permuted	95.7%	−0.45 pp
CNN $(1, 4, k{=}3)$	standard	95.4%	
CNN $(1, 4, k{=}3)$	permuted	93.9%	−1.50 pp

The MLP drops by about 0.45 percentage points. The CNN drops by about 1.50, more than three times as much (Figure 3.1). That gap is the falsification test. If the convolution were ignoring its locality prior, treating the restricted receptive field as a bottleneck it works around rather than a constraint it exploits, it would drop by the same tiny amount as the MLP. It does not. The convolution is genuinely using spatial adjacency, and we can tell because removing the adjacency costs it more.

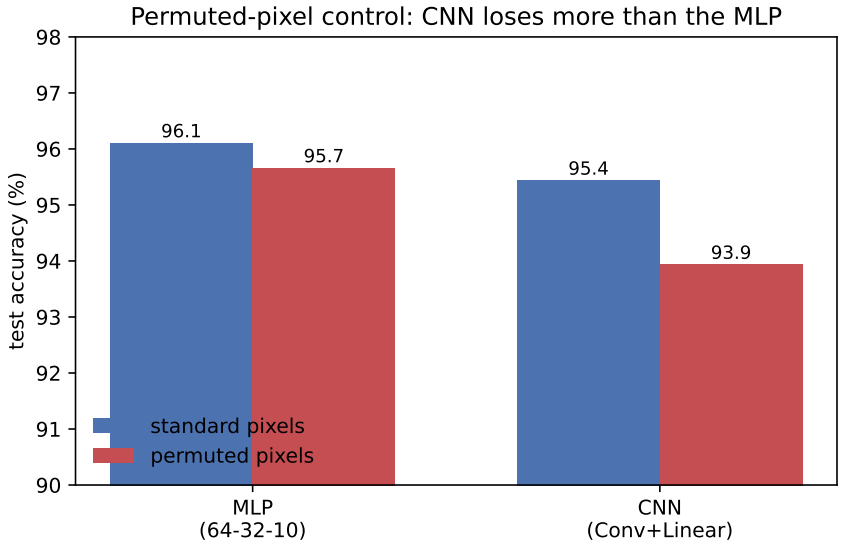

Figure 3.1: The permuted-pixel control. Test accuracy on standard pixels (blue) and after a single fixed random permutation of the 64 inputs (red), for the MLP and the CNN. The MLP, which treats pixel positions as interchangeable, barely moves (−0.45 pp). The CNN, whose kernels assume adjacent pixels are related, loses more than three times as much (−1.50 pp). The direction of the gap is the signature of an inductive bias doing real work; its small absolute size reflects how much spatial structure the 4×4 block-counting preprocessing has already removed.

This is the experimental signature of an inductive bias in operation. The *direction* of the gap, the CNN worse off under permutation, confirms the prior is load-bearing. The *magnitude*, small in absolute terms because the dataset was pre-pooled, reflects how much spatial structure survived into the features. The two findings together say more than the headline "CNN matches the MLP with fewer parameters" can on its own: the parameter efficiency is not an artifact of a thin head, it comes from the conv kernel actually exploiting adjacency, and when adjacency is destroyed the efficiency goes with it.

3.8 Inductive bias, named

The convolutional network commits to two priors. Made explicit:

- **Locality.** A unit reads a $k \times k$ window, not the whole image. Long-range interactions emerge only by stacking conv layers, each output cell's receptive field growing with depth.
- **Translation equivariance.** The same weights apply at every position. Shift the input, shift the output.

What it gives up is equally concrete:

- **Free interaction between distant pixels.** An MLP can wire the upper-left corner directly to the lower-right with a single weight. One conv layer cannot represent that at all; deeper stacks can, but only indirectly, through receptive-field growth.
- **Position-specific filters.** If the absolute location of the digit carries information, the convolution has to compensate elsewhere, or be given extra channels, or be helped by centering the input before it arrives.

This is the trade, and it is the same shape of trade Section 1.7 described. An MLP is the most generic supervised model: every input may interact with every other on equal footing. A CNN asserts that the useful interactions on an image are local and translation-equivariant, and bakes that assertion into the weights through structural reuse. Whether the assertion pays is empirical, not a matter of taste. The CNN matches the MLP with fewer parameters here, where the spatial structure was already half-extracted by preprocessing, and it pulls clearly ahead on data where that structure is intact. On genuinely tabular features, where there is no spatial adjacency to exploit, the same prior is a lie and the CNN loses. The right prior is the one that matches the data, and the permuted-pixel control is how we checked that it does.

This chapter took the architecture axis on its first non-trivial commitment, to a specific group of symmetries, the 2-D translations. The next chapter makes the same kind of move on a different data type. Sequences, whether text or time series, carry their own notion of locality, the recent past mattering more than the distant past, and their own translation symmetry, a shift in time rather than in space. A fixed-context language model, and after it a recurrent network, will commit to those priors by reusing weights across time steps instead of across spatial positions. Different data, the same lesson: structural weight sharing is how a network commits to an invariance, and committing to the right invariance is how it learns more from less.

3.9 What stayed pure Python, and what will not

The five-nested-loop `Conv2D` worked here because the problem is small: an 8×8 single-channel input, a 3×3 kernel, four output channels. The forward pass is about 1300 multiply-adds per

example and the backward pass roughly the same, so 3823 examples over 40 epochs finish in a few minutes of interpreted Python.

This is the last large layer this book writes in pure Python. For 28×28 MNIST with several conv layers and tens of channels each, the inner loop count grows by two orders of magnitude, and the five nested loops stop being merely unflattering and become intractable. At that point a vectorized implementation, first in NumPy and eventually with batched tensor operations in a framework, is genuine relief rather than premature optimization. The math does not change and the indices do not change; only the implementation moves from interpreted loops to compiled tensor calls. The chapter on the Transformer will reach for NumPy for exactly this reason, and will say so plainly when it arrives. Pure Python has carried the whole library to here; it does not have to carry it everywhere.

One natural extension is worth a line, in the spirit of Chapter 1's closing note on automatic differentiation. The convolution produces feature maps whose positions track the input, which is equivariance. A classifier usually wants the opposite, invariance: the digit is a seven wherever it sits. Averaging each feature map down to a single number, a global average pool, collapses an equivariant stack into a translation-invariant reader, discarding where the activation was while keeping that it occurred. The library includes such a layer; stacking it after the convolution is the natural next step toward a deeper invariant classifier, and it needs no new machinery, only the `Layer` contract already in hand.

Bibliographic Notes

The idea of a shift-invariant network built from locally connected units with shared weights goes back to the neocognitron of Fukushima (1980), which already framed pattern recognition as something a network should do "unaffected by shift in position." The modern form, a convolutional network trained end to end by backpropagation, is due to LeCun, Boser, et al. (1989), who applied it to handwritten zip-code recognition. The fuller treatment, including the LeNet architecture and the gradient-based learning framework that became the template for the field, is LeCun, Bottou, et al. (1998).

The dataset used here is the UCI optical-recognition-of-handwritten-digits collection (Alpaydin and Kaynak 1998), the same 8×8 optdigits collection as the Part I experiments, here in its fuller train/test release rather than the smaller sklearn subset used in Chapter 1; its 8×8 images are the 4×4 block-counted reduction of the original 32×32 NIST bitmaps, which is why the spatial structure is already partly summarized before any convolution sees it. For a textbook account of convolutional networks, including pooling, multi-layer stacks, and the receptive-field growth this chapter only points at, see the convolutional-networks chapter of Goodfellow, Bengio, and Courville (2016).

Chapter 4

Fixed-Context Language Models

The convolutional network of Chapter 3 committed to a prior over a plane: pixels have neighbors, and a feature is the same feature a few pixels over. Sequences ask the same question of a line. The recent past matters more than the distant past, and a pattern is the same pattern a few steps later. This chapter takes the architecture axis onto sequences with the simplest model that earns the name *neural language model*: look at the last N tokens, embed each one through a shared lookup table, concatenate the embeddings, and feed the result through a small multilayer perceptron that outputs a distribution over the next token. No recurrence, no attention, no machinery beyond the embedding lookup and an Chapter 1 MLP. This is the architecture Bengio and collaborators introduced in *A Neural Probabilistic Language Model* in 2003, and it was the dominant neural language model for several years before recurrent networks and then Transformers displaced it.

Its prior is a single clean assumption: a bounded window suffices. The model conditions on exactly the last N tokens and forgets everything older by construction. That is the Markov assumption of order N, made architectural rather than estimated from counts. The bet buys three things. Training is one MLP backward pass per example with no unrolling through time. Every window-target pair is independent, so there are no sequential dependencies to serialize. And there is no long product of Jacobians to shrink a gradient toward zero. What the bet gives up is the ability to use anything past N tokens back. The next chapter loosens exactly that constraint by carrying a hidden state instead of a fixed window; this chapter is the fixed-window baseline that makes the comparison legible when it arrives.

The plan is the usual one for Part II. State the prior, build the smallest model that isolates it, read its real code, train it on a corpus small enough to watch, and locate it on the map of architectural priors. Everything here is pure-Python `scratchnn`: the model is a plain `Network`, and the experiment runs in the same training loop as every model before it.

4.1 The architecture

The model has three parts, and each one is a building block already in hand or close to it.

First, an **embedding table** E of shape $V \times d$, where V is the vocabulary size and d is the embedding dimension. Each token id c indexes a row E_c, a learned dense vector of length d. The rows are *shared across positions*: the embedding of a token is the same vector whether the token appears first or last in the context window.

Second, a **concatenation step**. Given a context of N token ids $c_{t-N+1}, \ldots, c_t$, look up the N embeddings and concatenate them into a single vector $\mathbf{x}$ of length Nd. This vector carries two kinds of information at once: *what* tokens are present, through the embeddings, and *where* each one sits

in the window, through its offset in the concatenation.

Third, a **small multilayer perceptron** that maps $\mathbf{x}$ to logits over the vocabulary,

$$\mathbf{h} = \tanh(W_1\mathbf{x} + \mathbf{b}_1), \qquad \mathbf{z} = W_2\mathbf{h} + \mathbf{b}_2,$$

exactly the head from Chapter 1. A softmax of $\mathbf{z}$ is the predicted distribution over the next token, and `SoftmaxCrossEntropy` is the loss. The network emits logits; the loss interprets them, the same division of labor the whole book has kept.

That is the entire model. In `scratchnn` it is a plain `Network` with no custom training loop:

```
net = Network([
    EmbedConcat(vocab_size, embed_dim, context_size),
    Linear(context_size * embed_dim, hidden_size),
    Tanh(),
    Linear(hidden_size, vocab_size),
], SoftmaxCrossEntropy())
```

The only new layer is `EmbedConcat`, which wraps the embedding lookup and the concatenation. Its forward takes a `list[int]` of length `context_size` and returns a flat `list[float]` of length `context_size * embed_dim`. From the downstream layers' point of view the input is just a vector, so `Linear`, `Tanh`, and `SoftmaxCrossEntropy` compose without change. Training is `net.fit(X, Y, ...)` where each `X[i]` is a context (a `list[int]`) and each `Y[i]` is the next token id.

4.2 The Embedding layer

An `Embedding` maps a discrete token id to a dense vector. It is mathematically a `Linear` layer applied to a one-hot input, but it skips the multiply-by-zero by indexing the row directly. The forward returns a copy of the relevant row and records the id; the backward pops the id and accumulates the upstream gradient into that row. Here is the real layer from `scratchnn`, with the constructor trimmed:

```
class Embedding(Layer):
    def forward(self, token_id):
        self.cache.append(token_id)
        return list(self.weights[token_id])

    def backward(self, grad):
        token_id = self.cache.pop()
        row = self.dweights[token_id]
        for k in range(len(grad)):
            row[k] += grad[k]
        return None # discrete input, no upstream gradient
```

The lookup view and the one-hot view are the same computation. Real frameworks (PyTorch's `nn.Embedding`, for one) implement the lookup form for efficiency; the math does not change. The weight table is exactly the matrix a `Linear` layer would hold for one-hot inputs.

Because the same `Embedding` is called once per context position in a single forward pass, the layer keeps an internal LIFO cache of which ids were used. The `backward` pops the cache and accumulates each gradient into the matching row, the same `+=` accumulation pattern `Linear` uses in Section 1.3, here routed by token id rather than by neuron.

`EmbedConcat` is the thin wrapper that turns N token ids into the one flat vector the MLP head wants. Its forward embeds each id and concatenates; its backward slices the incoming gradient into

N chunks of length d and routes each back in reverse, so the pops line up with the pushes:

```
class EmbedConcat(Layer):
    def forward(self, ids): # ids: list[int] of length N
        self.embed.reset_cache()
        out = []
        for token_id in ids:
            out.extend(self.embed.forward(token_id))
        return out # flat list, length N * d

    def backward(self, g):
        d = self.embed_dim
        # Backward in reverse to match the embedding's LIFO cache.
        for i in reversed(range(self.context_len)):
            self.embed.backward(g[i * d:(i + 1) * d])
        return None
```

The concatenation is the whole positional prior in one line of indexing: token i's embedding always lands in slots $[id, (i+1)d)$, so the MLP head downstream can learn position-specific weights over those slots. The `parameters()` method delegates to the wrapped `Embedding`, so the table is the only place the rows live, and the generic optimizer from Chapter 1 steps them without knowing an embedding exists.

4.3 The inductive biases

The model makes three architectural commitments, each one a deliberate bet about how language works.

Markov of order N. The next token depends only on the previous N tokens. Anything older is forgotten by construction, not by a learned decay. This is the central prior, and it cuts both ways. Choose N too small and the model cannot represent agreement that spans a longer distance; choose N too large and the head's first weight matrix grows linearly in N, costing parameters and inviting overfitting. The window is a hard architectural knob, set before training and fixed.

Shared embeddings. Every token has one embedding vector, independent of where in the window it appears. This is weight sharing across the position axis, applied only to the embedding layer. The same lookup runs at slot 1, slot 2, and so on through slot N. It is the same move the convolutional network of Chapter 3 made in space, here made on the identity of a token: a token means the same thing wherever it sits, so it should be embedded the same way wherever it sits.

Position-sensitive head. The concatenation step is the choice that gives the model word order. By placing position 1's embedding in slots $1, \ldots, d$ of $\mathbf{x}$, position 2's in slots $d+1, \ldots, 2d$, and so on, the MLP head learns *position-specific* features over the window. It can distinguish "the cat sat" from "sat the cat" because each token's embedding occupies a different slice of the input, and the first weight matrix can weight those slices differently.

Put together, this is a *hybrid* prior, and the hybrid is the point. The embedding layer is position-equivariant: the same token gets the same vector regardless of position. The head is position-sensitive: it reads each slot with its own weights. Bengio gets token-identity sharing for free, one embedding per token, while still using word order through the head. The two halves pull in opposite directions on the position axis, and the architecture holds both at once.

This sits inside a larger design space of choices on the same axis, which Section 4.6 returns to once the experiment has run: a 1-D convolution that would share the head's weights across positions too, an averaging step that would discard order entirely, an attention mechanism that would learn

which positions matter rather than hardwiring it. Bengio's choice, concatenate and feed an MLP, is the simplest non-trivial point in that space. It hardwires positional sensitivity into the architecture with no convolution and no attention machinery.

4.4 Training

Training needs nothing new. There is no backpropagation through time and no recurrent unrolling. Each forward pass reads one N-token window and predicts one target token; the backward pass runs from the loss back through the MLP head, then through `EmbedConcat`, which splits the gradient on `x` into N chunks of size d and routes each back through the embedding in LIFO order. The output head is the categorical one from Section 1.5: a softmax over the V vocabulary logits, cross-entropy against the true next token. So the whole training call is one line,

```
net.fit(X, Y, epochs=4, lr=0.02, batch_size=1)
```

and the `Network` loop from Chapter 1 handles the rest. Each training example is independent of every other: there is no inter-example state to carry forward, no `reset_cache()` calls threaded through the loop, no bookkeeping beyond what `fit` already does. The training script is as short as logistic regression's was.

A gradient check confirms the new layer composes correctly. Running the standard `gradient_check` on a small `Network` built on `EmbedConcat` and an MLP head matches central finite differences to a worst relative error on the order of 10^{-9}, far under the standard 10^{-4} tolerance, regenerated by the chapter's paired notebook (`tests/test_gradients.py` carries the same check). With the math verified, the only question left is empirical: how well does an eight-character window predict English?

4.5 The experiment: char-level Alice

The corpus is the first 30,000 characters of *Alice's Adventures in Wonderland*, a 75-character vocabulary at the character level. The same corpus and vocabulary feed the next chapter's recurrent network, so the two models are compared on identical data. The configuration is small on purpose:

- context $N = 8$ characters (each example is eight ids in, one id out),
- embedding dimension $d = 16$,
- hidden layer $h = 64$ units,
- SGD at learning rate 0.02, batch size 1,
- 4 epochs over the corpus.

Sliding the window over the corpus gives 29,992 training examples. The parameter count is

$$75 \cdot 16 + 8 \cdot 16 \cdot 64 + 64 + 64 \cdot 75 + 75 \;=\; 14{,}331,$$

the embedding table, the first `Linear` and its bias, and the second `Linear` and its bias.

A model that predicts the next character uniformly at random scores $\log 75 \approx 4.32$ nats per character, the entropy of the uniform distribution over the vocabulary. That is the baseline to beat. Over the four epochs the mean per-character loss falls

$$4.32 \;\to\; 2.57 \;\to\; 2.22 \;\to\; 2.11 \;\to\; 2.04$$

nats per character, ending at 2.04 (perplexity $e^{2.04} \approx 7.67$). Figure 4.1 plots the trajectory. Most of the gain arrives in the first epoch, where the model learns character frequencies and the commonest short patterns; the later epochs sharpen word boundaries and local agreement.

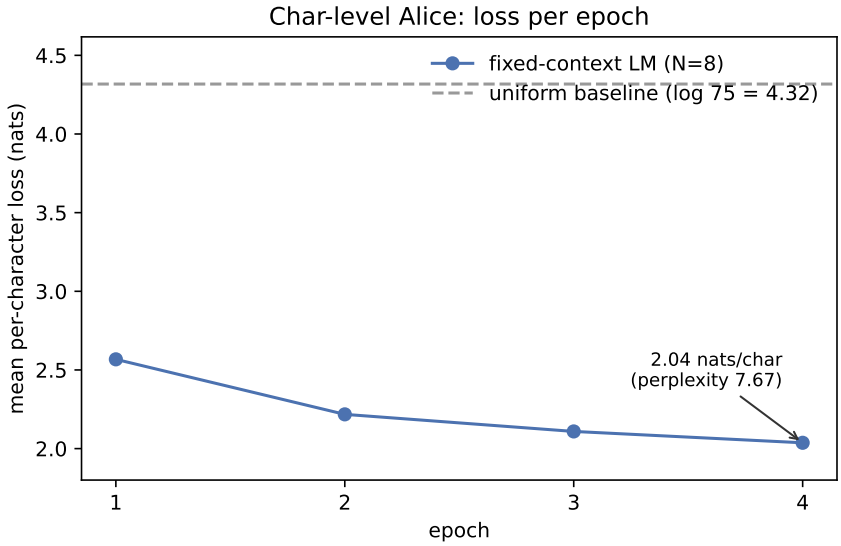

Figure 4.1: Char-level Alice: mean per-character loss per epoch for the fixed-context language model ($N = 8$, $d = 16$, $h = 64$). The dashed line is the uniform-random baseline, $\log 75 \approx 4.32$ nats per character. In four epochs over 29,992 eight-character windows the loss falls to 2.04 nats per character (perplexity 7.67), which is 2.94 bits per character. The steep first epoch is the model learning character frequencies and the commonest short patterns; the flatter tail is local agreement being sharpened.

What the samples look like. Sampling from the model at each epoch, seeded with `"alice was "` and drawn at temperature 0.8, shows the trajectory directly (the paired notebook stores the full samples). After one epoch the output is the right letters in the wrong order, with a few real words surfacing: spacing and common short tokens like `the` and `to` appear, but most of the stream is plausible-looking nonsense. By the fourth epoch the fragments are English-shaped, with recognizable words (`the`, `was`, `and`, `her`, `of`) and occasional plausible phrases, though punctuation and quotation balance stay hopeless and there is no coherence beyond a few characters. Character names other than `Alice` do not reliably appear, which is exactly what an eight-character window predicts: a name introduced hundreds of characters earlier is long gone from the model's input.

The compression reading. The per-character cross-entropy in nats converts directly to bits per character, $2.04/\ln 2 \approx 2.94$ bits per character. That number is the model's compression rate on this corpus, and the identity is not an analogy. Cross-entropy in bits per character *is* the expected code length per character under a Shannon-optimal code built from the model's predictive distribution p_θ. A model that predicts better assigns higher probability to what actually comes next, which is the same thing as encoding it in fewer bits. This is the chapter's north star, the thread that runs from prediction to compression and back: learning a good language model and finding a good code for the data are one problem viewed from two sides, and the bibliographic notes trace the lineage from Shannon's entropy of English to the minimum-description-length and Solomonoff view of induction. The Transformer chapter returns to this same bits-per-character measure when it asks what a better architecture buys.

4.6 What the bounded window gives up

The prior that makes this model simple is also its ceiling. Conditioning on exactly the last N tokens means the model cannot use anything older, ever. If predicting the token at position t depends on a token at position $t - 100$, an eight-character window has already forgotten it. For char-level Alice this rarely bites, because character prediction leans heavily on local structure: morphology, word boundaries, common bigrams and trigrams all live well inside eight characters. But the limit is real and it is structural, not a matter of more training. There are two natural ways to loosen it, and they point at the next two chapters.

A recurrent state. Instead of a fixed window, carry a hidden state forward and update it one token at a time. The state is a summary of the entire past, not a slice of it, so in principle nothing is forgotten by construction. That is the recurrent network, the subject of the next chapter. It trades the fixed window for an unbounded-in-principle memory, and it pays for the trade in ways the fixed-context model does not: training must run sequentially through time, and the gradient has to travel back through a long product of Jacobians that tends to shrink it toward zero. The next chapter builds the recurrent cell on the same corpus and the same vocabulary used here, precisely so the two priors can be set against each other directly; the head-to-head comparison of the bounded window and the recurrent state belongs there, where both models are on the table.

A convolution in time. A 1-D convolution with kernel size N takes the other fork on the position axis. Where this model gives each position its own slice of the head's first weight matrix, a convolution shares one learned context detector across all positions and slides it along the sequence, exactly the weight sharing the spatial convolution of Chapter 3 used, transposed from space to time. The trade is parameter efficiency against position specificity. The convolution has fewer parameters, one kernel applied many times, and it is the natural choice when the same pattern can appear at any offset, as in motif detection or edge detection in a 1-D signal. Bengio's concatenation has more parameters, one set per position-times-hidden-unit pair, but it can learn that the immediately preceding token matters more than the one $N - 1$ back, or that agreement runs between two particular slots. For language, where position carries real information and the nearest token is usually the most informative, paying for position specificity is a reasonable bet.

The three options form one progression along a single question: how much of the past does the architecture let a prediction reach, and how is that reach paid for? The fixed window reaches exactly N tokens, cheaply and in parallel. The convolution reaches a fixed receptive field while sharing weights across offsets. The recurrent state reaches arbitrarily far in principle, at the cost of sequential training and a fragile gradient. Each chapter in this part takes one more step along that line.

4.7 Where this sits in the inductive-bias map

Two axes organize the architectural priors for sequences: how the model handles position, and how far back its memory reaches. The fixed-context model occupies a specific corner of that grid, and naming the neighbors makes the corner legible.

Architecture	Position handling	Memory horizon
MLP	no sharing; flat input	full input dimension
1-D CNN	weight sharing across positions	fixed receptive field
Fixed-context LM	shared embedding, position-sensitive MLP	**fixed window** N
Recurrent network	weight sharing across time; state carries	unbounded in principle
Transformer	permutation-equivariant plus positional code	full sequence

The bottom two rows are forward references: the recurrent network is the next chapter, and the Transformer is the chapter after it. Read top to bottom, the table is the same loosening of the memory constraint the previous section described. The fixed-context model sits between the 1-D convolution and the recurrent network: position-sensitive at the head level, where the convolution is not, and capped at a hard memory cutoff of N tokens, where the recurrent network is not. It has no direct interaction between positions beyond what the head learns, which is where the Transformer, with its explicit pairwise attention, will go further still.

It is the architecture for problems where the relevant context is small *and you know how small*. When that assumption holds, it is fast, simple, and easy to train, with none of the sequential-training or gradient-stability costs the unbounded-memory architectures carry. When the assumption fails, when the prediction genuinely needs a token from far outside the window, no amount of training inside the window can recover it, and the unbounded-context architectures become necessary.

4.8 What comes next: a state, then a lookup

The fixed-context model already shows the appeal of dropping recurrence: no sequential dependency between examples, no gradient threaded back through time, just an MLP over a window. Its limit is equally clear: the window is a hard wall, and inside it the positional structure is hardwired by concatenation order rather than learned. The next two chapters take the two ways forward. The recurrent network keeps a running state so the reach is no longer bounded by a fixed N. The Transformer keeps the fixed-context model's no-recurrence design but replaces the hardwired, position-by-position head with attention, letting every position read every other with weights learned from the data, and then adds a positional encoding back in to recover the order that pure attention throws away. That is the destination this part is walking toward: from a window, to a state, to a learned content-addressable lookup over the whole sequence. The recurrent network comes first.

Bibliographic Notes

The fixed-context neural language model is due to Bengio, Ducharme, et al. (2003), *A Neural Probabilistic Language Model*, which introduced the embed-concatenate-and-feed-an-MLP design and the idea of learning distributed word representations jointly with the predictor. It was the dominant neural language model for several years, until recurrent networks and then Transformers displaced it; the next two chapters follow that displacement.

The compression reading of this chapter rests on a long lineage. Shannon (1951), *Prediction and Entropy of Printed English*, established that the per-symbol entropy of a language is its incompressible core and estimated it for English from human prediction experiments, fixing the bits-per-character measure this chapter reports. The identity between predicting well and compressing well is made into a principle of inference by the minimum-description-length and algorithmic-probability tradition; Hutter (2005) gives the modern treatment, in which the best predictor is the one that yields the shortest code for the data, and induction itself is recast as compression. That is the Solomonoff

north star the chapter points at, and the Transformer chapter returns to the same bits-per-character yardstick.

For a textbook account of neural language models and the embedding layer, including the word-level setting this chapter sets aside in favor of characters, see the relevant chapters of Goodfellow, Bengio, and Courville (2016).

Chapter 5

Recurrent Networks: Time-Translation Equivariance

A recurrent network is a multilayer perceptron with a single prior wired into its weights: the same computation runs at every step of a sequence, and a hidden state carries a summary of the past forward from one step to the next. The first move is the convolutional network's move from Chapter 3, weight sharing across positions, with one substitution. "Position" is now an index in time rather than in space.

Chapter 3 shared one kernel across spatial positions and got translation equivariance for images. This chapter makes the same move for sequences. The math, the implementation, and even the `+=` accumulation pattern in the backward pass all transfer directly. Two things change. The axis the sharing runs along is time instead of space, and the recurrence adds a wrinkle the spatial version did not have: the layer's output at one step is also its input at the next, so the chain of computations forms one long sequential dependency. That dependency is what lets the state reach back in principle to any earlier input, and it is also what makes a recurrent network hard to train at long range. By the end of the chapter we will have earned the right to name vanishing gradients as the price of the prior.

The plan is the usual one for Part II. State the prior, build the smallest model that isolates it, read its real code, derive the gradient by hand, train it on a corpus small enough to watch, and locate it on the map of architectural priors. The corpus is the same char-level *Alice* Chapter 4 used, and the chapter closes by setting the recurrent network's unbounded-in-principle memory against that chapter's bounded window, the head-to-head comparison the fixed-context chapter deferred to here. Everything is pure-Python `scratchnn`: a stateful `RNNCell`, a `Linear` head, and `SoftmaxCrossEntropy`, trained by a hand-derived backpropagation through time.

5.1 The prior

A sequence is an ordered series of inputs $x_1, x_2, \ldots, x_T$. The contents might be characters in a string, tokens in a sentence, ticks of a time series, frames of a video, or pixels of an image read out in raster order. The structure that makes it a sequence is that time matters: x_t sits closer to x_{t+1} than to x_{t+5}, and a computation that is useful at one step should be useful at any step.

Two assumptions turn that structure into a prior, and they have the same shape as the convolutional network's. The first is locality in time: the prediction at step t leans most heavily on the recent past, not on every token from the start of the sequence with equal weight. The second is time-translation equivariance: shift the input sequence by Δt and the output shifts by the same Δt.

The same computation applies at every step, so the model's weights do not depend on the absolute timestamp. Neither assumption is free; each is a bet that the data respects a symmetry, and the next section makes the bet concrete.

A multilayer perceptron applied to a flattened sequence has neither prior. It treats x_t at every t as an independent feature, with its own slab of weights for each position. Given enough data it can still learn the right function, but it learns each position from scratch and does not carry what it learned at one offset over to another. A pattern seen only early in training at one position is a pattern it has never seen at any other.

There are two ways to wire the temporal priors into an architecture, and they part on a single question: how much of the past does a prediction get to see. Chapter 4 took the first answer. Look at a fixed window of the last N tokens, share the embedding across positions, and feed a small head. That is a bounded window, and everything older than N is gone by construction. The recurrent network takes the second answer. Rather than a fixed slice of the past, it carries a hidden state vector h_t that summarizes the past, and it updates that summary one step at a time. The next state h_{t+1} is computed from the current state h_t and the current input x_{t+1}, using the same weights at every step. Because the state is a running summary rather than a window, the model can in principle let a prediction depend on an input from arbitrarily far back. In practice that reach has a hard limit, which a later section in this chapter spends its whole length on.

5.2 The cell

A vanilla recurrent cell is one equation:

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

where W_{xh} is the input-to-hidden weight matrix, W_{hh} is the recurrent hidden-to-hidden matrix, b_h is the bias, and tanh is applied componentwise to the pre-activation vector. The cell's output is h_t itself. A separate `Linear` layer projects h_t to logits, or to whatever shape the task downstream wants, the same division of labor every chapter has kept: the body produces a representation, and a head interprets it.

The property that makes this a recurrent network and not just a stack of distinct layers is that W_{xh}, W_{hh}, and b_h do not depend on t. One set of weights computes the state update at every step. This is weight sharing along the time axis, the exact analogue of the convolutional network's weight sharing across spatial positions in Chapter 3. There the same kernel slid across an image; here the same update slides along a sequence.

In code the cell extends the `Layer` contract from Chapter 1. A stateless layer (`Linear`, `Tanh`, `ReLU`) has `forward(x)` returning an output and `backward(g)` returning the gradient on its input. A stateful layer extends both signatures: `forward(x, state)` takes the previous state and returns `(output, new_state)`, and `backward(g_out, dstate_next)` takes a gradient on the new state and returns `(dL/dx, dL/dstate_prev)`. For a vanilla cell the new state equals the output, both are h_t, so the forward returns the same vector twice, as `(h_new, h_new)`. The two protocols share `parameters()`, so the generic `step()` and `zero_grad()` from Chapter 1 need no change. This is the same split between a stateful module and a stateless function that production frameworks navigate.

Parameter storage matches `Linear` exactly. The cell holds W_{xh} and W_{hh} as lists of rows, one row per hidden unit, alongside the bias vector, so `parameters()` yields one (values, gradients) pair per row and the optimizer steps them without knowing a recurrence exists. Here is the real forward, with the constructor summarized in a comment:

```
class RNNCell(Layer):
    def __init__(self, input_size, hidden_size):
        # W_xh, W_hh, b_h and their gradient accumulators. Xavier-style
        # init. Stored as list[list[float]], one row per hidden unit, like
        # Linear, so parameters() yields one (values, grads) pair per row.
        # A LIFO cache holds (x, h_prev, h_new) per forward, popped in
        # backward.
        ...

    def forward(self, x, state=None):
        H = self.hidden_size
        if state is None:
            state = [0.0] * H
        h_new = [0.0] * H
        for i in range(H): # one row per hidden unit
            z = self.b_h[i]
            for j in range(self.input_size):
                z += self.W_xh[i][j] * x[j] # W_xh @ x
            for j in range(H):
                z += self.W_hh[i][j] * state[j] # W_hh @ h_prev
            h_new[i] = math.tanh(z)
        self.cache.append((x, state, h_new)) # LIFO, popped in backward
        return h_new, h_new

    def backward(self, dh_out, dstate_next=None):
        ... # the BPTT step, shown in full in a later section
        return dx, dh_prev
```

The forward is two nested loops, the same matrix-vector shape as `Linear`, with the second loop adding the recurrent contribution $W_{hh}\, h_{t-1}$ on top of the input contribution $W_{xh}\, x_t$. The only genuinely new line is the cache push. The cell saves the inputs it will need to differentiate, because the backward pass that undoes this step runs much later, after the whole sequence has been read. That deferred, accumulated backward is the subject of the next section.

5.3 Time-translation equivariance

The argument from Chapter 3 transfers without change. Replace shift in space with shift in time, and every line still holds.

Suppose we shift the input sequence by Δt, so the new sequence is $x'_t = x_{t+\Delta t}$. Starting from the same initial state, the new sequence of hidden states is $h'_t = h_{t+\Delta t}$. The cell applies one computation at every step, so shifting the input shifts the output by the same amount. This is a property of the operator, not of any particular weights: it holds at initialization, it holds after training, and it holds for every weight setting in between. Weight sharing across time is time-translation equivariance, in the same sense that weight sharing across space was translation equivariance for images.

(Strictly, the equivariance holds when the initial state h_0 is the same. In practice we reset $h_0 = 0$ between sequences, which breaks the symmetry at sequence boundaries. The within-sequence computation is fully time-translation-equivariant, and that is what the inductive-bias claim rests on.)

Equivariance is exactly the property the permuted-pixel control of Section 3.7 probed, now along the time axis. There the test destroyed spatial structure by applying one fixed permutation to the pixels of every image. The convolutional network, which assumes neighbors are neighbors, lost

more accuracy than the multilayer perceptron, which assumes nothing about layout, because the convolutional network was actually using the locality its weights bake in. The analogue here is a multilayer perceptron on a flattened sequence. It would learn the phrase "the cat" at positions one through three and then fail to recognize the same phrase at positions five through seven, because it holds separate weights for each position and the two occurrences share none. The recurrent cell recognizes the phrase wherever it falls, because the same weights read every position. That is the prior earning its keep, and it is the same prior the permuted-pixel control of Chapter 3 confirmed for space.

5.4 Unrolling and backprop through time

Training a recurrent network means computing the gradient of the loss with respect to W_{xh}, W_{hh}, and b_h for an unrolled sequence of length T. The standard technique is backpropagation through time. Unroll the cell T times, treat the result as a deep feedforward network with T layers that happen to share one set of weights, and apply the usual chain rule.

The forward pass processes $x_1, \dots, x_T$ in order, threading the state $h_0 \to h_1 \to \cdots \to h_T$. At each step an output layer, here a `Linear` projection to vocabulary logits followed by `SoftmaxCrossEntropy`, contributes a per-step loss L_t. The total loss is the sum over steps:

$$L = \sum_{t=1}^{T} L_t.$$

The backward pass is where the recurrence shows up. The gradient on h_t arrives from two places:

$$\frac{\partial L}{\partial h_t} = \underbrace{\frac{\partial L_t}{\partial h_t}}_{\text{from } L_t \text{ directly}} + \underbrace{\frac{\partial L}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial h_t}}_{\text{from the future, through } h_{t+1}}.$$

The first term is the gradient that the output projection at step t sends back. The second is everything downstream of step t, funneled through the single channel the state provides, h_t feeding h_{t+1}. In code this is one extra argument on the cell's `backward`. On top of `dh_out` from the output projection, the cell receives `dstate_next`, the gradient on h_t that the previous backward call computed, the call for step $t+1$. The two are added before the tanh derivative is applied:

```
def backward(self, dh_out, dstate_next=None):
    x, h_prev, h_new = self.cache.pop() # LIFO: this timestep
    H = self.hidden_size
    if dstate_next is None:
        dstate_next = [0.0] * H
    # Two gradient paths into h_t: this step's output, and the future.
    dh_total = [dh_out[i] + dstate_next[i] for i in range(H)]
    # Through tanh: d/dz tanh(z) = 1 - h^2.
    da = [dh_total[i] * (1.0 - h_new[i] * h_new[i]) for i in range(H)]
    # Accumulate weight grads (the += across timesteps), as in Linear.
    for i in range(H):
        for j in range(self.input_size):
            self.dW_xh[i][j] += da[i] * x[j]
        for j in range(H):
            self.dW_hh[i][j] += da[i] * h_prev[j]
        self.db_h[i] += da[i]
    # Backprop to this step's input and to the previous state.
```

```
dx = [sum(self.W_xh[i][j] * da[i] for i in range(H))
      for j in range(self.input_size)]
dh_prev = [sum(self.W_hh[i][j] * da[i] for i in range(H))
           for j in range(H)]
return dx, dh_prev
```

The single line `dh_total = dh_out + dstate_next` is the whole of backpropagation through time: the gradient at step t is what came back from this step's output plus what came back from the future through h_{t+1}.

The weight gradients accumulate across all T timesteps with `+=`. This is the same accumulation the library has used since the `Linear` layer of Section 1.3, in a third flavor. Per-example gradients sum within a mini-batch. Per-position gradients sum within a single convolutional forward, because the kernel is reused at every spatial position. Per-timestep gradients sum within one recurrent unroll, because the recurrent matrix is reused at every step. The same operation runs at a finer grain each time: across examples, then across positions, then across time.

Two pieces of bookkeeping make the unroll trainable. The first is gradient clipping. After all T backward steps, the global L2 norm of the gradient is capped at 5, the standard mitigation for the exploding-gradient half of the problem the next section names. The second is the mean-gradient step. The gradients summed over the T steps are divided by T before the update, which turns the sum back into a mean and keeps the effective learning rate sane regardless of how long the unroll was. This is exactly the convention `Network.step(lr, n)` uses for a mini-batch, with T playing the role of the batch size.

A gradient check confirms the derivation. Unroll the cell for $T = 3$ with a small `Linear` output projection and `SoftmaxCrossEntropy`, build the analytic gradient with one forward and backward sweep, and compare every parameter against a central finite difference. The worst relative error lands on the order of 10^{-9}, far under the 10^{-4} tolerance the book has used since Chapter 1. The chapter's paired notebook regenerates this, and `scratchnn`'s `test_gradients.py` carries the same check.

5.5 Vanishing gradients

Backpropagation through time is correct, and it is fragile. Each step of the backward pass multiplies the gradient on the state by a Jacobian:

$$\frac{\partial h_{t+1}}{\partial h_t} = \operatorname{diag}\!\left(1 - h_{t+1}^2\right) W_{hh}. \tag{5.1}$$

The diagonal factor is the tanh derivative, componentwise; the matrix factor is the recurrent weight matrix. To send a gradient back k steps, the backward pass multiplies k of these Jacobians together, so the magnitude of the gradient on an early step is governed by the product. That product is in turn governed by the spectral properties of W_{hh}, scaled by the tanh derivative, which lies in $(0, 1]$ and is almost always strictly below 1.

The product of many matrices of roughly constant size grows or shrinks geometrically. If the spectral radius of the effective Jacobian sits below 1, the gradient on early timesteps decays exponentially in k, and those parameters receive essentially no signal from losses many steps later. If it sits above 1, the gradient blows up, and one bad sequence can throw the weights off entirely. This is the vanishing-gradient problem, with exploding gradients as its twin, and it is the reason a vanilla recurrent network cannot easily learn dependencies more than a few dozen steps long. The architecture is the right shape for sequences in principle. The learning dynamics make long-range information transfer hard in practice.

This is the price of the prior, and the experiment later in this chapter shows it from both sides. Numerically, the paired notebook pushes a unit gradient back through the recurrence alone and watches its norm fall by several orders of magnitude over a few dozen steps, a direct measurement of Equation (5.1) compounding. Qualitatively, the generated text drifts over long spans even after the local structure is correct: spelling and short words come out right while coherence across a sentence wanders. That drift is not a separate defect. It is the same limit, the state failing to carry information far enough, showing in the output rather than in the gradient.

Two mitigations are standard, and the chapter mentions them without building them. Gated cells, the long short-term memory and the gated recurrent unit, replace the simple tanh recurrence with an additive update. The long short-term memory keeps a cell state whose Jacobian is $\mathrm{diag}(f_t)$ for a learned forget gate f_t with entries in $[0, 1]$; when the gate sits near 1, the gradient flows through unattenuated, a gradient highway across many steps. This library implements the vanilla `RNNCell` only, and the extension is mechanical: more matrix multiplies, one per gate and one for the cell-state update, the same `+=` accumulation in the backward pass, and no new conceptual machinery. Gradient clipping, already in the training loop, handles the exploding half. The point worth keeping is that this one Jacobian-product issue is what motivated the entire gated-recurrent literature, and eventually the architecture the next chapter turns to.

5.6 The experiment: char-level Alice, and the comparison with Bengio

We train a vanilla `RNNCell` and a `Linear` output projection on the first 30,000 characters of *Alice's Adventures in Wonderland*, the same corpus and the same 75-character vocabulary Chapter 4 used. That is the point of reusing it: the recurrent network and the fixed-context model are trained on identical data, so the comparison at the end is fair. The model reads one character at a time, and its job at each step is to predict the next character from the state that summarizes everything read so far.

The configuration is small on purpose. The cell is `RNNCell(input_size=75, hidden_size=64)`, the head is `Linear(64, 75)`, and the loss is `SoftmaxCrossEntropy`. Sequences are unrolled 32 steps for backpropagation through time, the learning rate is 0.5 against the mean per-timestep gradient, the global gradient norm is clipped to 5, and training runs 15 epochs over the 937 non-overlapping chunks the corpus cuts into. The parameter count is the input-to-hidden matrix, the recurrent matrix, the hidden bias, and the output projection with its bias:

$$64 \cdot 75 + 64 \cdot 64 + 64 + 75 \cdot 64 + 75 = 13{,}835.$$

A model that guesses the next character uniformly at random scores $\log 75 \approx 4.32$ nats per character, the entropy of the uniform distribution over the vocabulary. That is the baseline to beat. Over the 15 epochs the mean per-character loss descends steadily,

$$4.32 \;\to\; 3.09 \;\to\; 2.40 \;\to\; 2.17 \;\to\; \cdots \;\to\; 2.05$$

nats per character (the values shown are the start, then epochs one, five, and ten, then the end), finishing at 2.05 nats per character, a perplexity of $e^{2.05} \approx 7.79$. Figure 5.1 plots the full trajectory. The descent is smooth and still gently sloping at epoch 15: unlike the fixed-context model, which did most of its work in the first epoch, the recurrent network improves gradually across all 15, because each chunk is one sequential SGD step rather than thousands of independent window updates.

What the samples look like. The clearest view of what the model learns is to generate from it. At each epoch boundary we seed the state with `"Alice "` and roll forward 200 characters at

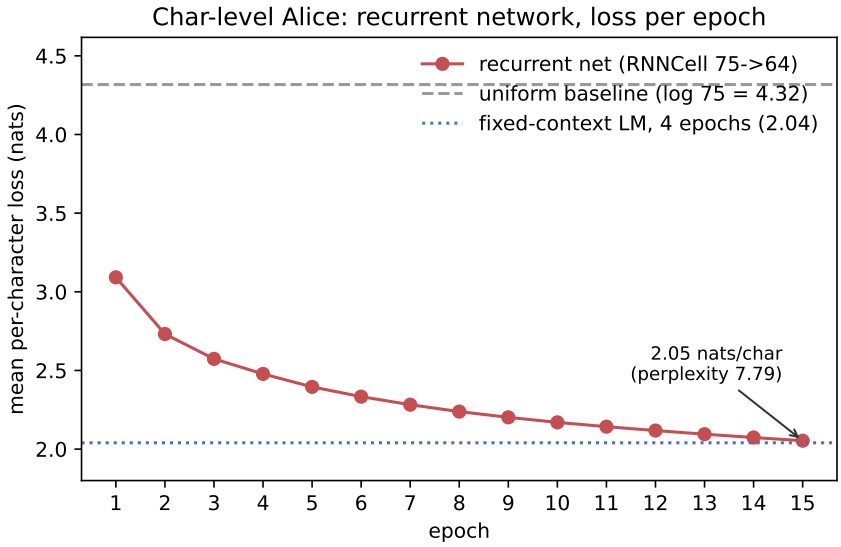

Figure 5.1: Char-level Alice: mean per-character loss per epoch for the recurrent network (`RNNCell` $75 \to 64$, then `Linear` $64 \to 75$), trained by backpropagation through time with sequence length 32 and gradient clipping. The dashed line is the uniform-random baseline, $\log 75 \approx 4.32$ nats per character. The dotted line marks the fixed-context language model of Chapter 4, which reached 2.04 nats per character in four epochs on the same corpus. Over 15 epochs the recurrent network descends to 2.05 nats per character (perplexity 7.79), a comparable endpoint reached by a different route.

temperature 0.8 (the paired notebook stores the full samples; short excerpts are quoted here). After one epoch, at loss 3.09, the output is close to random characters with spaces at roughly plausible intervals:

```
Alice wh tsbi ta teca aa mu rea meam ifs n r ne san uou d hl au t iiee
thgo to aaf moghe ior te the ...
```

By the fifth epoch, at loss 2.40, the character distribution is right and word-like fragments appear, with common short words surfacing intact:

```
Alice Aa's verind thing to Hop, ang "a thand pad wed nof the hact touro
the varnt wand wave ...
```

By the fifteenth epoch, at loss 2.05, the fragments are English-shaped, with many real words and the beginnings of local phrasing:

```
Alice out it, and the little wind she Mace sait a dit ...and hall thest
wore that hereed and ...she madeand theme was dow ...
```

The trajectory is the classic char-level recurrent arc: random-character noise, then the character distribution as vowels and consonants alternate and spacing lands, then word-like fragments and recognizable short words, then sentences that look locally English but drift in meaning across longer spans. That drift is the vanishing-gradient limit showing in the output. The state holds enough to get spelling and short words right, and not enough to keep a thought coherent across a clause. A larger hidden state, longer unrolls, and gated cells would push the coherence further; the vanilla

cell at modest scale and pure-Python compute lands here. The arc is the lesson, and the absolute quality is the next chapter's concern.

The comparison with Bengio

This is where the two architectures meet on the same data. The fixed-context model of Chapter 4 reached 2.04 nats per character in about four epochs over 29,992 independent eight-character windows. The recurrent network reaches 2.05 nats per character over 15 epochs of backpropagation through time. The endpoints are essentially the same. The routes are not, and the difference is the whole point of setting them side by side.

Start with what each one gives up. The fixed-context model gives up any reach past its window. It conditions on exactly the last N tokens and forgets everything older by construction, so a dependency that spans more than N characters is simply invisible to it. The recurrent network gives up nothing of the kind in principle: the state is a running summary of the entire past, so there is no hard wall at N. What it gives up instead is reliable access to the distant past in practice. The vanishing-gradient problem means the gradient that would teach it to use a far-back input decays before it arrives, so the effective memory is a few dozen steps, not the whole sequence. One model has a sharp, known horizon; the other has a soft, unreliable one.

Now what each one gains. The fixed-context model gains training simplicity, and it is hard to overstate how much. Every window-target pair is independent, so training is ordinary mini-batch classification with no ordering to respect, no state to thread, and no unrolling. There is no long product of Jacobians, so the gradient is well-behaved and the model converges in a handful of epochs. The recurrent network gains the unbounded-in-principle memory and the parameter economy that comes with it: one shared recurrent matrix carries the past, rather than a separate slab of head weights per window position, which is why it reaches a comparable loss with 13,835 parameters against the fixed-context model's 14,331. It pays for that reach with sequential training, a fragile gradient, and the four-times-longer schedule the trajectory shows.

For char-level Alice the contest is close to a draw, and that is itself informative. Character prediction leans heavily on local structure, morphology, word boundaries, common bigrams and trigrams, all of which live well inside an eight-character window, so the fixed-context model's hard horizon rarely bites and its training simplicity wins it a fast, comparable result. The recurrent network's extra reach buys little here because there is little long-range structure to reach for at the character level. The recurrent prior earns its premium only when the data carries dependencies longer than any fixed window one would be willing to pay for, and even then only to the extent the vanishing gradient lets the state actually carry them. That last qualification is exactly the gap the next chapter sets out to close.

5.7 The pattern repeats

Three architectures now sit on the table, and they line up along one idea.

Architecture	Weight-sharing axis	Equivariance
MLP	none	input permutation
CNN	spatial position	2-D translation in space
RNN	timestep	1-D translation in time

The recipe is the same each time. Identify a symmetry the data respects. Wire it into the architecture by reusing one set of weights along the axis the symmetry runs along. Three things

follow at once. The parameter count drops, because one shared operator replaces a separate operator per position. The data efficiency rises, because what the model learns at one position transfers to all the others for free. And the backward pass becomes a `+=` accumulation along the shared axis, the pattern the library has carried since the `Linear` layer of Section 1.3: across examples in a mini-batch, then across spatial positions in a convolution of Chapter 3, then across timesteps in a recurrent unroll. One operation, three grains.

What the recurrent network buys over a fixed-context model, for sequence data, is the state. Information can in principle flow from an arbitrarily distant past input through the hidden vector, where a fixed window simply cannot reach past its edge. What it costs is the vanishing-gradient problem. The state is a single fixed-width vector that every past input has to pass through, and the gradient that would teach the model to use a distant input has to survive a long product of Jacobians to get back there. The reach is unbounded in principle and limited in practice, and the limit is structural, not a matter of more training.

5.8 Handoff to the Transformer

The next chapter takes the other path. Rather than squeezing the past through a sequential state, the Transformer lets every position attend to every other position directly, in a single layer. That one move changes three things at once. It removes the long Jacobian product, because there is no recurrence to backpropagate through, so the vanishing-gradient problem of this chapter goes away. It restores parallelism over the sequence, because without a state to thread, the positions no longer have to be processed in order. And it trades time-translation equivariance for permutation equivariance over positions, which then has to be partly broken back with a positional encoding so the model can still tell "the cat ate the mouse" from "the mouse ate the cat."

That is a different bet about which symmetries a sequence respects. The recurrent network commits to time-translation equivariance and a running state, and pays with sequential training and a fragile gradient. The Transformer commits to permutation equivariance, breaks it deliberately with a positional encoding, and pays with a cost that grows quadratically in the sequence length. Both are entries on the same menu of architectural priors, and the right one is the one whose assumptions match the data. The next chapter builds the Transformer from the same pure-derivation, real-code footing this one used, and asks what content-addressable attention buys that a state cannot.

Bibliographic Notes

The recurrent network in the form used here, a hidden state updated by a shared computation at every step, is due to Elman (1990), *Finding Structure in Time*, which introduced the simple recurrent network and showed that the hidden state learns to encode temporal context without being told the structure in advance. Backpropagation through time, the unroll-and-chain-rule procedure of Section 5.4, is laid out by Werbos (1990), whose title, *Backpropagation Through Time: What It Does and How to Do It*, is an accurate summary of the method.

The vanishing-gradient problem of Section 5.5 was identified early. Hochreiter (1991) first analyzed it in a diploma thesis, and Bengio, Simard, and Frasconi (1994), *Learning Long-term Dependencies with Gradient Descent is Difficult*, gave the argument its standard form: the gradient through a recurrence is a long product of Jacobians, and that product almost always shrinks toward zero or grows without bound, so gradient descent cannot easily attach a prediction to an input many steps back. That diagnosis is what motivated the gated cells the chapter points at but does not build. Hochreiter and Schmidhuber (1997) introduced the long short-term memory, whose additive

cell state and learned gates create the gradient highway described in Section 5.5; the gated recurrent unit of Cho et al. (2014) is a lighter variant in the same family. Both keep the recurrent prior and the `+=` accumulation pattern, and both replace the simple tanh update with a gated one to keep gradients alive over longer spans.

For a textbook account of recurrent networks, backpropagation through time, and the gated architectures, see the sequence-modeling chapters of Goodfellow, Bengio, and Courville (2016).

Chapter 6

Attention as Content-Addressable Memory

Chapters 3 to 5 each committed to a structural prior about *which positions matter*. The convolutional network shared one kernel across space and bet on locality. The fixed-context model fixed a window and bet that the recent past was enough. The recurrent network shared one cell across time and squeezed the whole past through a single state. The transformer makes a stranger and more flexible commitment. It does not decide in advance which positions matter. It commits to a *mechanism* for deciding, from the data, position by position, on the fly.

The architectural claim of this chapter fits in one sentence. An attention head is a learned content-addressable lookup: a query matches against keys to retrieve values, exactly as a pointer dereference reads memory by address. Stacking attention layers composes lookups into multi-hop reads. That is the entire structural prior of a transformer, and the rest of the chapter is the unpacking of it.

To make the claim falsifiable we build a synthetic data-generating process whose structure *requires* content-addressable memory, train small transformers on it, and watch attention learn to do what its mechanism is built for. The task has a single-lookup variant and a multi-hop variant, so it can test both halves of the claim: that one attention layer is a lookup, and that depth is the number of composed lookups.

The chapter does two things. The first half states the task, derives why one attention layer cannot compose two lookups, and confirms the depth argument at small scale. The second half is the part the chapter did not set out to do. When we scale the task by enlarging the memory, the transformer suddenly loses to a brute-force multilayer perceptron, and we have to debug why. The debugging is where the chapter earns its real lesson, and where the book's third axis appears: the right architecture is necessary but not sufficient, and whether gradient descent realizes the bias-permitted solution depends on positional-encoding scale, initialization, head count, and depth.

This is also where the book's pure-Python through-line ends. The convolutional, fixed-context, and recurrent models were `scratchnn`, model and experiment, standard library only. The transformer here is hand-derived NumPy. Every backward pass is still written out by hand, the way Chapter 1 wrote out backpropagation for the multilayer perceptron, but the operators are vectorized in NumPy because pure-Python loops over the $T \times T$ attention matrix are not tractable. The listings in this chapter come from that NumPy implementation, `examples/transformer.py`. The chapter says plainly where the switch happens and walls off the one place a second framework appears: a scaling sweep run in PyTorch, cited as a table, with the warning that its internals differ from the NumPy model so its numbers are read only against each other.

6.1 Attention as a lookup

Scaled dot-product attention is the one operation the whole chapter turns on. For a query matrix Q, a key matrix K, and a value matrix V, it computes

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

Three steps run in sequence. Take the pairwise dot products between every query and every key. Turn each query's row of scores into a distribution over positions with a softmax. Return, for each query, the average of the values weighted by that distribution. Read operationally, that is exactly what a content-addressable memory does.

- The query at a position encodes what that position is looking for.
- The key at each past position encodes what is stored there to be matched against.
- The value at each past position encodes what would be returned if that position is the one selected.
- The softmax picks, softly, the past position whose key best matches the query, and the output is that position's value.

When the query is the encoding of an address, "return position k," and the keys encode positions, the softmax sharpens to a near one-hot spike on the addressed position, and its value is read out. That is a pointer dereference, done with differentiable parts so it can be learned by gradient descent. Note what attention does *not* have on its own: any notion of position. Permuting the inputs permutes the outputs identically, so bare attention is exactly the permutation-equivariant body the earlier chapters anticipated, and the positional encoding added back later is what breaks that symmetry to recover order.

This is where the book's pure-Python through-line ends. The operators below are hand-derived NumPy from `examples/transformer.py`, vectorized because pure-Python loops over the $T \times T$ score matrix are not tractable, but the derivation is the same one Chapter 1 carried for the multilayer perceptron. Here is the core of the forward pass, lifted from the multi-head attention class and shown for a single head: project the input into queries, keys, and values, score, mask out the future so a position can only read the past, softmax, and read out the weighted values.

```
# Q, K, V: (T, d_k) projections of the input x.
scale = 1.0 / math.sqrt(d_k)
scores = (Q @ K.T) * scale # (T, T) match scores
mask = np.tril(np.ones((T, T))) # lower triangular
scores = np.where(mask > 0, scores, -1e9) # forbid attending ahead
attn = softmax(scores, axis=-1) # (T, T) distribution
out = attn @ V # (T, d_k) read-out values
```

The masking step is what makes this a causal read: position t may match against positions 0 through t and no further, so the lookup at the final position can reach every earlier position but not the answer it is trying to predict. Whether attention *succeeds* at a given lookup depends on whether the address it needs is actually present in the query, and that is where depth enters.

6.2 A pointer-dereferencing task

To test the claim we need a task whose structure *requires* a content-addressable read, with no shortcut a fixed pattern could exploit. The task is deliberately small so the structure stays visible. Each example is a 12-bit sequence with three fields:

$$\underbrace{m_0\, m_1\, \cdots\, m_7}_{\text{8 memory bits}} \quad \underbrace{a_0\, a_1\, a_2}_{\text{3 address bits}} \quad \underbrace{y}_{\text{target}} .$$

The eight memory bits are random. The three address bits are the binary encoding of an index $a \in \{0, 1, \ldots, 7\}$, most significant bit first. The target is $y = m_a$, the memory bit at the addressed position. There are $2^{11} = 2048$ possible inputs. We draw 20,000 training examples with replacement, so duplicates abound, and 2000 held-out examples for evaluation. The model reads the first eleven bits and predicts the twelfth. The generator is `make_variant1` in `examples/simple_pointer_dgp.py`:

```
def make_variant1(num_examples, M=8, A=3, seed=0):
    """[M memory bits] [A address bits] [1 target = memory[addr]]."""
    rng = np.random.default_rng(seed)
    memory = rng.integers(0, 2, size=(num_examples, M))
    address_int = rng.integers(0, M, size=num_examples)
    # Encode the address as A binary bits, most significant first.
    address_bits = np.zeros((num_examples, A), dtype=np.int64)
    for j in range(A):
        address_bits[:, j] = (address_int >> (A - 1 - j)) & 1
    target = memory[np.arange(num_examples), address_int][:, None]
    sequences = np.concatenate([memory, address_bits, target], axis=1)
    return sequences, target.squeeze(-1)
```

This is the smallest task that requires content-addressable memory. To get the answer the model has to learn three steps: recognize that the bits at positions 8, 9, 10 are an address, decode those three bits to an integer a, and use a to read memory position m_a. The third step is the dereference, and it is dynamic: *which* memory position must be read depends on the *content* of the address bits, which are only known at inference time. No fixed wiring can stand in for it, which is exactly what makes the task a clean probe of the prior.

6.3 Why one attention layer is not enough

A subtle but load-bearing point: a one-layer, one-head transformer *cannot* solve this task, and the reason is structural rather than a matter of capacity.

The prediction is made at the last input position, position 10, using the output of attention there. The query at position 10, before attention, is computed from the input at position 10 alone: the single address bit a_2, plus a positional encoding. It does not see a_0 or a_1, which sit at positions 8 and 9. The attention layer then forms a weighted sum of past values with weights set by $\mathrm{softmax}(Q_{10} \cdot K_i)$. Since Q_{10} is fixed once the input at position 10 is fixed, those weights are not a function of the full address. The layer can learn a fixed attention pattern, always read memory position 3, say, but it cannot route to a position whose identity depends on bits it never sees.

Two layers can. The first layer lets the hidden state at position 10 pull in the address bits from positions 8 and 9 through attention. The second layer's query at position 10 is then computed from a hidden state that contains all three address bits, so it can route dynamically to the addressed memory cell. This is the precise sense in which transformer depth equals the number of composed

lookups: each additional layer adds one more stage of dynamic routing on top of the information the previous layers have already gathered. The next section puts the argument to an experiment.

6.4 Experiment 1: depth on a single lookup

We train three models on the single-lookup task, all at comparable parameter counts, all with Adam at learning rate 10^{-3} for 2000 iterations of batch size 32. The code is `examples/pointer_experiments.py`, and the paired notebook regenerates these numbers.

Model	Layers	Heads	Parameters	Test accuracy
MLP baseline	(n/a)	(n/a)	3,018	**1.000**
Transformer	1	1	8,738	0.654
Transformer	2	1	17,282	**1.000**

Three things to read off this table. The **one-layer transformer stalls** near 65 percent, modestly above the 50 percent chance line but nowhere near the ceiling. That is exactly the structural prediction of Section 6.3: the model can learn a fixed attention pattern and pick up the single bit of information the address bit at position 10 already carries, but it cannot do the dynamic lookup, and adding parameters within one layer cannot fix an architectural limit.

The **two-layer transformer reaches 100 percent**, and it gets there with a sharp phase transition rather than a smooth climb. The training loss sits at the random line, $\ln 2 \approx 0.69$, for the first several hundred iterations, then drops steeply, then settles at machine zero. The held-out accuracy stays at chance through the plateau and then jumps to one over the same short window. This is the grokking shape that recurs whenever a model finds a discrete algorithm: a long search across a flat region, then a fast collapse onto the solution once the routing falls into place. Figure 6.1 shows the trajectory.

The **MLP also reaches 100 percent**, and why is more interesting than "it memorized." The MLP has 128 hidden units, and at $M = 8$ with eight possible addresses that is about sixteen hidden units per address. With that budget it can learn a brute-force expert-per-address decomposition: a handful of units that fire when the address equals k and memory bit k is one, with the output layer summing the experts. That is an algorithm, not memorization, but it scales linearly in the number of distinct addresses, so it starts to break down once 2^A outgrows the hidden budget. The transformer is supposed to scale linearly in M regardless of A, because its attention is content-addressable rather than expert-decomposed. Section 6.7 takes the obvious next step and tests whether that scaling story holds.

6.5 Multi-head attention and parallelism

The single-head experiments already work, so what does adding heads buy? Multi-head attention runs several attention operations in parallel within one layer and combines them. Each head gets its own query, key, and value projections acting on a slice of the model dimension, so the heads can specialize: one might track a syntactic relation, another a long-range dependency, another a specific token. An output projection W_O recombines the per-head read-outs. The real NumPy forward pass, from `examples/transformer.py`, projects the input, splits it into heads, runs the masked scaled-dot-product attention of Section 6.1 per head, then merges and projects back:

```
def forward(self, x):
```

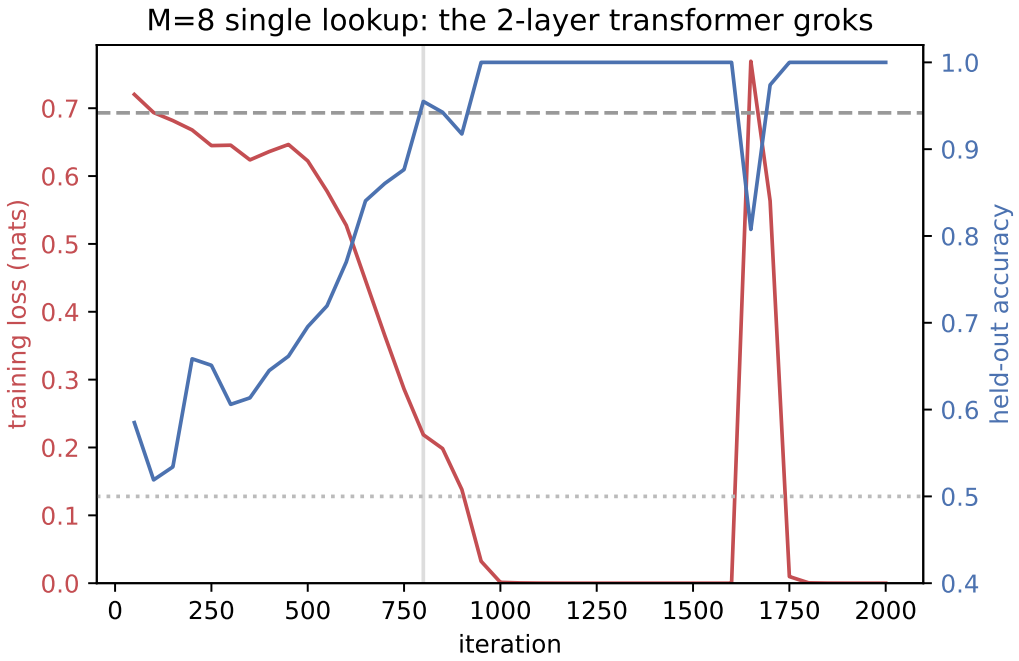

Figure 6.1: The two-layer transformer groks the $M = 8$ single-lookup task. Training loss (left axis) holds at the random line $\ln 2 \approx 0.69$ while the model searches, then collapses toward machine zero; held-out accuracy (right axis) tracks the same transition from chance to one. The brute-force MLP, by contrast, improves smoothly: fitting a lookup algorithm is a discrete jump, fitting an expert-per-address table is an interpolation.

```
T, D = x.shape
H, Dh = self.n_heads, self.head_dim
scale = 1.0 / math.sqrt(Dh)

q = self.W_q.forward(x)
k = self.W_k.forward(x)
v = self.W_v.forward(x)
# Split into heads: (T, D) -> (H, T, Dh)
Q = q.reshape(T, H, Dh).transpose(1, 0, 2)
K = k.reshape(T, H, Dh).transpose(1, 0, 2)
V = v.reshape(T, H, Dh).transpose(1, 0, 2)

scores = (Q @ K.transpose(0, 2, 1)) * scale # (H, T, T)
mask = np.tril(np.ones((T, T)))
scores = np.where(mask[None] > 0, scores, -1e9)
attn = softmax(scores, axis=-1) # (H, T, T)
heads_out = attn @ V # (H, T, Dh)
merged = heads_out.transpose(1, 0, 2).reshape(T, D) # (T, D)
return self.W_o.forward(merged)
```

The split is a reshape: the d_{model} channels are partitioned into H contiguous blocks of $d_k = d_{\text{model}}/H$, each block its own head, and the heads run independently before the merge concatenates them back. For the pointer task at $M = 8$, though, the lookup is conceptually a single operation:

one address to decode, one memory cell to read. A second head adds no qualitatively new capability at this scale, and indeed a two-layer two-head model converges to the same 100 percent accuracy as the two-layer one-head model. The heads do not specialize because there is nothing here to specialize on. That changes at larger M, where multiple heads turn out to help for a reason the investigation in Section 6.8 uncovers, and it changes again on real language, where head specialization (induction heads, copy heads, syntactic heads) is well documented. The synthetic task at $M = 8$ is simply not large enough to force it.

6.6 Experiment 2: depth on a multi-hop lookup

To probe the depth story directly, switch to a pointer-to-pointer task. The memory bit at the addressed position is combined with the address bits to form a *new* address, used for a second lookup, so the target is

$$y = m_{f(m_a, a)},$$

where f folds m_a together with the lower bits of a into the next address. The exact combination is `make_variant3` in `examples/simple_pointer_dgp.py`; the point is that the target now depends on two lookups composed in sequence. The model must decode the address, read m_a, build the new address from that bit, and read memory there. A two-layer transformer can in principle resolve the first lookup in layer one and the second in layer two; a three-layer model has a layer to spare. We train both on 20,000 examples for 2500 iterations:

Model	Layers	Parameters	Test accuracy
Transformer	2	17,282	**0.977**
Transformer	3	25,826	**0.995**

The two-layer model gets close to perfect at 97.7 percent, and the three-layer model essentially solves it at 99.5 percent; both grok within 2500 iterations. The marginal gain from the third layer is small here because the task needs only two hops, and a third layer is one more than that minimum. A three-hop task would widen the gap between two and three layers. Even on this small demonstration the trend is the one the depth-equals-composed-lookups story predicts: more layers buy more hops. The next section tries to scale the single-lookup task in the other direction, by memory size, and the clean story breaks.

6.7 Scaling beyond $M = 8$, the puzzle

So far the story holds together. One layer fails by the structural argument, two layers succeed in a clean grokking shape, and an extra hop benefits from an extra layer. The content-addressable-memory prior seems to be doing exactly what the theory predicts, so the obvious next experiment is to scale the memory size M up and watch the transformer pull ahead of the MLP cleanly. The accounting favors the transformer. Its lookup mechanism costs $O(M \cdot d^2)$ parameters, while the MLP's brute-force expert-per-address scheme costs $O(2^A \cdot \text{hidden})$. As M grows, and with it $A = \lceil \log_2 M \rceil$, the MLP's hidden-units-per-address budget shrinks while the transformer's mechanism stays the same size.

Empirically, that is not what happens.

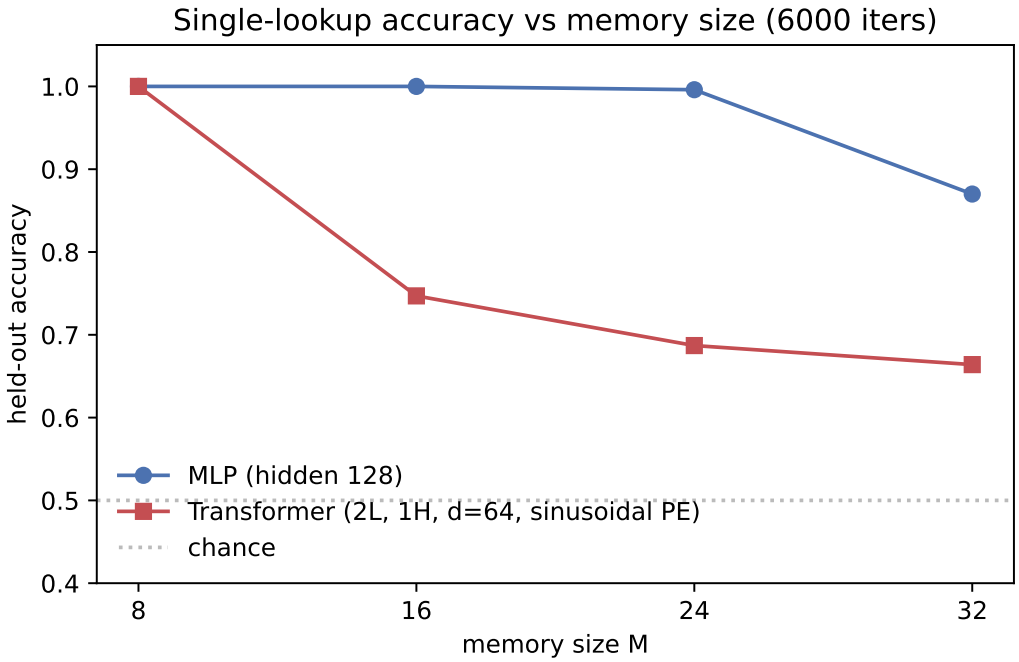

Figure 6.2: Held-out accuracy against memory size M on the single-lookup task, both models trained for 6000 iterations (the runs recorded in `examples/RESULTS.md`). The MLP tracks high until its hidden budget per address runs thin; the two-layer transformer, the model the theory expects to keep scaling, plateaus near chance from $M = 16$ on. The gap between the two is the puzzle the rest of the chapter chases down.

M	A	Input space	MLP (h=128)	Transformer (2L, 6000 it)
8	3	2^{11}	1.000	1.000
16	4	2^{20}	1.000	0.747
24	5	2^{29}	0.996	0.687
32	5	2^{37}	0.870	0.664
48	6	2^{54}	0.654	0.529
64	6	2^{70}	0.607	0.540

The MLP scales smoothly until it runs out of capacity at $M \geq 48$, where $128/2^A = 2$ hidden units per address is too few. The transformer, the model expected to keep scaling, plateaus near chance at every $M \geq 16$. The MLP *wins* at $M = 16$, 24, and 32. Figure 6.2 plots the two curves from the recorded runs in `examples/RESULTS.md`.

The same picture holds when training data is varied at fixed $M = 16$. The MLP shows the textbook sample-efficiency curve, more data buying higher accuracy, while the transformer plateaus near chance regardless of how much it is shown, reaching only 0.741 at 5000 examples and 0.738 at 20,000. The MLP's success at $M = 16$ on 5000 examples, covering half a percent of the input space, does require it to generalize rather than memorize, so this is not a case of the task being too easy. The puzzle is real, and it has two possible readings. Either the claim that attention is cheap content-addressable memory is *wrong*, which would be a genuine and interesting falsification, or something specific to this small-scale setup is keeping the prior from taking hold. Telling those two

apart is the work of the next section.

6.8 Investigation: four parallel hypotheses

To find what was actually wrong we ran four independent investigations in parallel. The results in this section are read from the recorded experiment log, `examples/RESULTS.md`, rather than re-run in the paired notebook.

1. **Audit the code** for a real bug in attention, layer norm, residuals, or the training loop.
2. **Sweep optimization recipes**: learning-rate warmup, gradient clipping, different optimizers, longer training.
3. **Try different supervision formats**, in case sparse supervision at only the final position is the bottleneck.
4. **Try architecture variants**: layer counts, head counts, model widths, learned positional encoding, no causal mask.

The audit cleared the math. A numerical gradient check on the full pipeline in float64, with $\varepsilon = 10^{-6}$, returns a worst relative error of 2.3×10^{-7} on parameters with a nonzero true gradient. Attention forward and backward, layer norm, the multi-head split and merge, the feedforward and its GELU, the residual connections, the pre-norm structure, and the sparse final-position gradient are all correct. The naive float32 check at $\varepsilon = 10^{-4}$ shows apparent errors of order 10^{-1}, which look alarming but are float32 noise from catastrophic cancellation in the forward pass, not real disagreement. The paired notebook re-runs this audit and confirms the gap: in float64 the worst relative error is machine-zero class, while the float32 check at the larger step inflates by many orders of magnitude purely from cancellation. The backward pass is not the problem.

No optimization recipe rescued the model. Warmup over 500 iterations, gradient clipping, larger batches at matched compute, longer training to 20,000 iterations, and both lower and higher learning rates all stay within a few points of the baseline at $M = 16$. The plateau is a property of the loss landscape the model is sitting in, not a tuning artifact.

Supervision shape mattered, but did not solve it. Dense per-position supervision *hurts*, dropping to 0.567: the intermediate positions carry random next-bit targets, and that noise drowns the one informative position. Reweighting the final position by ten recovers most of the baseline at 0.726. Inserting a special query token before the target, which gives the model an explicit position to predict from, lifts accuracy to 0.816, meaningfully better than the baseline but still short of the algorithm, and training longer does not grok.

Two architecture variants reached the ceiling. Holding everything else fixed at $M = 16$:

Architecture variant	$M = 16$ test accuracy
Baseline (2L, 1H, sinusoidal PE)	0.747
More layers (3L, 1H)	0.733
More layers (4L, 1H)	0.729
Wider ($d_{\text{model}} = 128$)	0.751
No causal mask	0.746
Bigger FFN ($d_{\text{ff}} = 512$)	0.726
More heads (4 heads)	**0.983**
Learned positional encoding	**1.000**

Depth, width, feedforward size, and the causal mask are all non-issues: they move the number by less than two points. Two things matter, and they matter a lot. Switching from sinusoidal to a *learned* positional encoding solves the task to 100 percent. Switching to four heads with sinusoidal encoding reaches 98.3 percent. Both findings point at the same culprit. At this scale the model is fighting its *positional* signal more than it is reading its *content* signal, and the next section pins down why.

6.9 The fix: positional-encoding scale

The audit pinned down why the choice of positional encoding mattered so much. The `Embedding` layer initializes its weight matrix at scale 0.02, the standard default tuned for large vocabularies. With a two-token bit vocabulary the input embeddings stay near that 0.02 scale, while the sinusoidal positional encoding has entries of magnitude about 1. So at the input, where embedding and position are added together, the content signal is roughly 35 times smaller than the positional signal. The model has to learn to discount the positional swamping before it can route on content. At $M = 8$ that finishes in time. At $M \geq 16$ the optimization plateaus before the embeddings have been rescaled to anything competitive with the encoding.

There are two clean fixes, and both target the same content-versus-position *scale balance* at the network's input.

1. **Rescale the embedding** so content and position start at comparable magnitudes. The original transformer paper does exactly this: it multiplies the embedding output by $\sqrt{d_{\text{model}}}$ before adding the positional encoding. A crude $25\times$ rescaling of the embedding weights alone lifts $M = 16$ from 0.67 to 0.79 with no other change, which was checked during the audit.

2. **Use a learned positional encoding**, initialized at the same small 0.02 scale as the embedding. Now content and position both start small and the model can grow them together as training proceeds. This is what takes the lookup task to 100 percent in the architecture sweep of Section 6.8.

Modern transformer libraries use either learned or rotary position representations, both of which sidestep the fixed-magnitude sinusoidal-against-tiny-embedding problem. At production scale, with large vocabularies and well-tuned initialization and warmup, the issue mostly disappears; at this small experimental scale it dominates. The multi-head result fits the same story. With several heads, at least one can find a query-key geometry that compensates for the input-scale imbalance, redundancy a single head does not have, which is why four heads recover 98.3 percent where one head reaches only 0.747.

6.10 Verification: the bias is partially confirmed

With the fix in hand we rerun the scaling experiment, changing only the positional encoding (sinusoidal versus learned, both at the same 0.02 init scale) and holding everything else at the baseline: two layers, one head, $d_{\text{model}} = 64$, 6000 iterations.

M	Sinusoidal PE	Learned PE
16	0.747	**1.000**
24	0.687	0.605
32	0.664	0.645

Layers	Parameters	Loss at 40k	Test accuracy	Loss curve
2	270k	0.6463	0.6160	flat, never transitions
3	403k	0.1154	**0.9460**	transitions at iter ∼7000

Table 6.1: The $M = 32$ depth transition, run in PyTorch with only the layer count varying (the controlled comparison from `examples/RESULTS.md`). The two-layer model is flat for the whole budget; the three-layer model phase-transitions near iteration 7000 and is still descending when stopped. The PyTorch model's initialization and internals differ from the NumPy model used elsewhere in this chapter, so these numbers are read only against each other, not against the NumPy tables.

At $M = 16$ the fix is decisive: learned positional encoding solves the lookup task to 100 percent, exactly as the content-addressable-memory theory predicts once the input-scale obstacle is removed. At $M \geq 24$, though, the same fix does not carry. Both encodings stay near chance, so the scale mismatch that dominated at $M = 16$ is one barrier among several, and another kicks in at larger M. That nuance, frustrating to hit but honest to report, sharpens this chapter's lesson rather than softening it. The architectural inductive bias is **necessary but not sufficient**. Whether a network realizes the bias depends on a co-evolved set of choices the theory glosses over: initialization, positional encoding, supervision shape, optimizer state, learning-rate schedule, width, depth, data volume, and training time.

To see how far the standard production knobs close the gap, we ran a scaled-down production recipe (the file is `examples/pointer_kitchen_sink.py`): $d_{\text{model}} = 128$ instead of 64, four heads instead of one, learned positional encoding at small init, learning-rate warmup over 500 iterations, and 8000 iterations, with the same two layers and the same 20,000 examples.

M	Sinusoidal (baseline)	Learned PE alone	Kitchen-sink recipe
16	0.747	**1.000**	**1.000**
24	0.687	0.605	**0.9975**
32	0.664	0.645	0.680

The $M = 24$ row is the headline. The combined recipe lifts accuracy from 0.605 to 0.9975, a near-complete solve at a scale that learned encoding alone could not touch. The bias was present in the architecture the whole time; the production defaults are what let it become a learned routine. At $M = 32$ the recipe barely helps, reaching only 0.680, so we chased that case down. Three knobs were left: more iterations, more width, more depth. We tried each in isolation. To run a proper sweep without waiting hours per configuration, the same architecture was reimplemented in PyTorch.

A caution that matters, and that the table below carries: the PyTorch model uses different initialization and different internals from the from-scratch NumPy code, so its absolute accuracies are *not* comparable to the NumPy numbers above. The PyTorch results are read only against each other. With that wall in place, the result is clean. More iterations alone do not solve $M = 32$: the NumPy kitchen-sink run to 30,000 iterations reached only 0.7625, and a PyTorch two-layer model run to 40,000 iterations sat at chance-plus with a loss curve flat from iteration 2000 on. More width alone does not solve it either: doubling to $d_{\text{model}} = 256$ with eight heads, four times the parameters at two layers, stayed indistinguishable from chance-plus. Depth does. Holding width, head count, data, and seed fixed and varying only the layer count over a 40,000-iteration budget:

Table 6.1 is decisive within its frame: at $M = 32$, depth is the binding knob, and neither width nor training time substitutes for it. It is tempting to package this as a tidy law, that depth equals

the number of address bits the model must compose, so larger M needs more layers. Resist it, because the data refute the clean version. $M = 24$ and $M = 32$ have the same address width, since $\lceil \log_2 24 \rceil = \lceil \log_2 32 \rceil = 5$, yet two layers solved $M = 24$ to 0.9975 and cannot solve $M = 32$ at all. The address-decode work is identical across the two. What differs is the **lookup fan-out**, the number of memory cells the second-stage query must discriminate among, 24 versus 32, together with the fact that $M = 32$ fills the entire five-bit address space while $M = 24$ uses only 24 of the 32 codes. The bottleneck at $M = 32$ is the precision of the one-of-M selection, not the depth of the address decode. The honest claim is the controlled- experiment one, that at $M = 32$ depth solves where width and time do not, and not a closed-form depth law.

This is the scaling-law picture in miniature. Capacity, depth, data, and iterations co-scale, and the binding constraint moves as the task hardens. At $M = 8$ the binding constraint was nothing; any two-layer model solves it. At $M = 16$ it was positional-encoding scale. At $M = 24$ it was the combined production recipe. At $M = 32$ it is depth. Each regime hides a different bottleneck, and finding it is the whole game.

6.11 Inductive bias, three axes

Across the book we have walked the two axes of supervised inductive bias named in Section 1.7:

- **Architecture** (this chapter and the previous two, Chapter 3 and Chapter 5): the structural prior the network commits to about *how features compose*. The CNN commits to locality and translation equivariance; the RNN to time-translation equivariance and a bottleneck state; the transformer to a content-addressable lookup, repeated and composed through depth.
- **Output head** (Section 2.9): the prior the interpretation commits to about *the distribution of the response*.

The pointer experiment forces a third dimension into the open:

- **Implementation realization**: even when the right inductive bias is structurally present in the architecture, the choice of initialization, positional encoding, supervision shape, and optimizer can keep it from materializing during training. At production scale, with the standard defaults, this dimension is mostly invisible. At small experimental scale it becomes the dominant factor, and it is where most of the debugging cycles in real transformer engineering go.

Every successful supervised network in modern practice is a triple: a body that bakes in the right architectural prior for the data, a head that bakes in the right likelihood prior for the response, and a training recipe co-evolved with the body and head so the priors can actually be discovered by gradient descent. The combinations multiply.

The transformer is the most flexible body in the book. It bakes in less structure than a CNN or an RNN: it does not commit to spatial locality, to temporal recurrence, or to a fixed context window. It commits instead to a *mechanism*, content-addressable retrieval, and lets the data dictate which positions are retrieved when. The cost of that flexibility is the $T \times T$ attention matrix, quadratic in the sequence length in both compute and memory. The benefit is that one architecture serves any task whose structure is "attend to the right earlier position." Language modeling is the canonical case (the right position depends on the syntax, the topic, the entity in play), and the same prior fits code completion, reasoning over context, and retrieval-augmented generation.

The cost is *also* the realization burden. The flexibility is genuinely accessible only with a co-evolved recipe, and much of modern transformer practice (warmup, learned or rotary positional

encodings, careful initialization, layer-norm placement, layer-wise learning-rate decay, mixed precision, gradient clipping) is the accumulated body of tricks that keeps the inductive bias usable. Section 6.9 rediscovered one slice of that recipe by debugging.

The next chapter takes the model just trained and reverse-engineers its circuit, asking whether the architectural argument of Section 6.3 is empirically vindicated and how the circuit relates to induction heads and in-context learning. The model is small enough to interpret fully, and the answer clarifies what the content-addressable-lookup primitive actually is.

The closing chapter introduces reinforcement learning, the third learning paradigm beyond supervised. The heads-as-bias frame still holds there (a policy head is a softmax over actions; a value head is identity plus mean squared error on returns), but the training signal stops being a per-example label and becomes a scalar reward over trajectories. The inductive-bias frame extends naturally: reward shaping, policy architecture, and exploration strategy are all priors. The theoretical optimum, AIXI, is Solomonoff induction (the compression-as-prediction north star of Chapter 4) joined to Bayesian decision theory and reward maximization.

Appendix: a richer DGP that did not pan out

The first design for this chapter used a recursive bit-stream generator with prefix-coded instructions (`examples/bit_dgp.py`): a continuous bit stream in which each instruction was either a literal bit (prefix code `0` for value 0, `10` for value 1) or a dereference (`11` followed by an Elias-gamma-encoded offset and the dereference's own value codeword), with the offset counted in instruction units. Multi-hop chains arose naturally whenever a dereference's target was itself a dereference.

The generator is rich and pedagogically appealing: the model must learn the prefix code, parse the stream into instructions, count instructions for the gamma-encoded address, and perform the lookup, all from raw bits. Small transformers (model width 64 to 128, one or two layers) did not learn it within our training budget; even at 5000 iterations they barely beat the marginal-prediction baseline. Parsing, counting, and addressing in one small model appears to need more capacity or far longer training than we had. The positional-encoding-scale problem of Section 6.9 likely contributed, and we did not retry the recursive generator with a learned encoding.

We kept the generator as reference because the design is interesting in its own right: it is the smallest recursive task we know that exercises stacked attention end to end with no task-specific scaffolding. Showing that a sufficiently large or sufficiently well-tuned transformer can learn it is left open.

Bibliographic Notes

The transformer and scaled dot-product attention are from Vaswani, Shazeer, Parmar, et al. (2017). That paper also multiplies the embedding by $\sqrt{d_{\text{model}}}$ before adding the positional encoding, the detail whose absence drove the scale mismatch diagnosed in Section 6.9.

The plateau-then-sudden-transition pattern of Section 6.4 is the phenomenon Power et al. (2022) named grokking: generalization arriving long after the training loss has settled, on small algorithmic tasks like the one used here.

The reverse-engineering the next chapter performs follows the transformer-circuits line of work. Elhage, Nanda, Olsson, et al. (2021) sets out the framework for reading attention-only transformers as compositions of circuits, and Olsson, Elhage, Nanda, et al. (2022) identifies induction heads and ties them to in-context learning. The pointer-dereference primitive studied in this chapter is a deliberately minimal cousin of those mechanisms.

The compression-as-prediction thread that surfaces in the closing synthesis (Section 6.11) and in Chapter 4 runs through Solomonoff induction and its decision-theoretic extension AIXI; Hutter (2005) is the book-length treatment.

Part III

Interpretation and Beyond

The first two parts ask what bias to build in. Part III asks what the network did with it, and then leaves supervised learning behind. Chapter 7 takes the trained pointer transformer apart and finds that having the right architecture is necessary but not sufficient: a third axis, implementation realization, governs whether gradient descent finds the bias-permitted solution and whether we can read it back off the weights. The answer there is a cautionary one about the limits of staring at attention maps and the power of causal probes, and it is the book's flagship honest-empiricism result. Chapter 8 then changes the learning signal entirely, trading per-example labels for a scalar reward over whole trajectories, and asks how much of the inductive-bias frame survives the move to reinforcement learning. It is the book's last chapter, and its closing section is the book's conclusion.

Chapter 7

Reverse-Engineering the Pointer Transformer

Chapter 6 built a two-layer, one-head transformer that solves the pointer-dereferencing task at $M = 8$ to 100 percent test accuracy. We argued from the architecture (Section 6.3) that the model *must* be composing two attention lookups: the first layer aggregates the address bits at the last input position, and the second layer uses the assembled address to attend to the addressed memory bit. That argument was a prediction. This chapter checks it against the trained weights.

That is the whole move of Part III. The first two parts decided what bias to build in. This chapter turns the lens around and reads the bias back out of a model that has already learned. It is the empirical payoff of the book's third axis, implementation realization: once gradient descent has run, which of the architecturally permitted solutions did it actually find, and can we recover that solution from the weights?

The $M = 8$ model is small enough to answer cleanly. It has 17,282 parameters and the task has $2^{11} = 2048$ possible inputs, so we can describe what every component does and do it from the trained weights and attention patterns directly. That is mechanistic interpretability at its cleanest: a task with known structure, a model that solves it, and a hypothesis specific enough to falsify. The first eight sections carry out that dissection and find the predicted circuit, confirmed by ablations. They also place the circuit in the literature, as a near-cousin of the induction head and a minimal instance of the mechanism behind in-context learning.

Then the chapter does the part it did not set out to do. The same task at $M = 32$ needs a third layer, and when we point the same tools at a model that solves it, the clean circuit is gone. The attention maps look illegible, and two hypotheses written before the analysis are refuted by the data. A causal probe then shows that the model is nonetheless a perfect pointer, and a causal trace recovers a mechanism as clean as the $M = 8$ one, a mechanism the attention maps simply could not surface. The lesson in one sentence: attention weight is not information flow, and the tool, not the model, decides what is legible. This honest-empiricism result is the chapter's payoff, the clearest case in the book of reading a realized solution back out of a model's weights.

A word on what is being dissected, because the chapter studies two distinct models in two frameworks and the distinction matters. Sections 1 through 8 analyze the from-scratch NumPy model of Chapter 6: two layers, one head, $d_{\text{model}} = 32$, the file `examples/pointer_interp.py`. It retrains cheaply on each run, so its exact weights move a little between runs while the circuit it learns does not. Section 9 analyzes a separately trained, deliberately introspectable PyTorch model: three layers, four heads, $d_{\text{model}} = 128$, attention written out by hand so every pattern is readable, the file `examples/pytorch/pointer_interp_deep.py`. This is *not* the Chapter 6 scaling-sweep

model, which used a library attention layer; it is a fresh model built to be read, loaded from a committed checkpoint so its numbers are fixed. The two models solve the same kind of task at different scales, but their accuracies are not comparable across the framework boundary, and the chapter never compares them as numbers. What carries across is the method and the lesson.

7.1 The setup

The model under the lens is the one Chapter 6 trained: a two-layer, one-head decoder transformer with $d_{\text{model}} = 32$ and $d_{\text{ff}} = 64$, trained for 2000 iterations with Adam at learning rate 10^{-3}, batch size 32, on 20,000 examples of the $M = 8$, $A = 3$ pointer task. On 2000 held-out examples it reaches a test accuracy of 1.000. It has 17,282 parameters, and the task has only $2^{11} = 2048$ possible inputs. That combination is what makes the dissection clean. The model solves the task perfectly, and the task is small enough that we can describe what every component does and do it from the trained weights and attention patterns directly.

Each input is an eleven-token sequence: eight memory bits, then three address bits, most significant first. The supervised target at position 10, the last input position, is the value at memory position a, where a is the integer decoded from the three address bits. The plan for the chapter is to read the architectural prediction of Chapter 6 off these trained weights and check, component by component, whether the model does what the architecture said it must.

The instrument is the attention pattern itself. Both layers cache their forward-pass attention matrix, and the model exposes a method that returns the (T, T) attention pattern for any single layer and head. We call it to extract the pattern for a given input, and we average it across the test set to ask what the model does in general rather than on one example. All of the code for this analysis is in `examples/pointer_interp.py`.

7.2 What the circuit must be

Before looking at a single weight, it is worth restating what the model *has* to be doing, because the architectural argument of Section 6.3 is specific enough to pin the circuit down to one possibility.

The model produces its prediction by passing the residual stream at position 10 through the final layer norm and an output head. Whatever the model knows about the answer has to be present in that residual stream at position 10 after the second layer. Now trace backward. Before any attention runs, position 10's residual is the embedding of token a_2, the address least significant bit, plus the positional encoding for position 10. It carries no information about a_0 or a_1, which live in positions 8 and 9. Yet the answer depends on the full address $a = 4a_0 + 2a_1 + a_2$. So for the second layer's query at position 10 to attend to memory position a, that query must already depend on all three address bits, which means the first layer must have moved a_0 and a_1 into position 10's residual. The only mechanism a self-attention layer has for that is attention from position 10 back to positions 8 and 9.

The hypothesis is therefore forced:

> **Layer 1**, at position 10, attends to positions 8, 9, and 10, the three address positions, and assembles them into position 10's residual.
>
> **Layer 2**, at position 10, uses the assembled address to attend to memory position a and copy its value into the readout.

This is the *only* way a two-layer, one-head transformer can solve the task. The claim is genuinely falsifiable. If the trained model is doing something else, the architectural argument of Chapter 6 was

wrong. If it is doing exactly this, the argument is empirically vindicated. The next three sections check each half against the weights and then confirm the whole causally.

A note on scope before we begin. The model dissected here, and through the induction-head and in-context-learning discussion that follows, is this from-scratch NumPy model at $M = 8$. The final section turns the same tools on a larger PyTorch model at $M = 32$, where the answer is more surprising.

7.3 Layer 2 attends to the addressed position

Start with the dereference, the second half of the hypothesis, because it is the easiest to see. Take four representative test examples with addresses $a \in \{0, 4, 5, 7\}$ and extract the layer-2 attention pattern. The row that matters is row 10, the query at the last input position. Its argmax, the memory position receiving the most attention, lands exactly on the address every time:

Address a	Layer-2 argmax position	Weight on that position
0	0	0.698
4	4	0.973
5	5	0.881
7	7	0.851

Every example sends most of position 10's layer-2 attention to the memory position whose index equals the address. The left panel of Figure 7.3 shows one such map: for $a = 4$ the attention mass at the last query row concentrates almost entirely on column 4, the cell m_4.

Four hand-picked examples could be a coincidence, so aggregate. (The paired notebook caches the $a = 4$ row; the other three weights above are illustrative values from the same seed-0 run. The aggregate that follows is computed and stored in the notebook.) The probe runs the layer-2 attention for many examples, takes the last row of each, and groups those rows by the decoded address:

```
def aggregate_last_row(model, X, M=8, A=3):
    """Layer-2 attention from the last position, grouped by address."""
    T = M + A
    by_addr = {a: [] for a in range(2 ** A)}
    for ids in X:
        ctx = ids[:T]
        a = address_value(ctx, M=M, A=A)
        attn_L2 = model.attention_at(ctx, layer=1, head=0)
        last_row = attn_L2[T - 1]
        by_addr[a].append(last_row)
    means = {a: np.mean(np.stack(v), axis=0) for a, v in by_addr.items() if v}
    return means
```

Averaged over 500 test examples, partitioned by encoded address, the picture is clean for every address, not just the four:

Address a	Avg weight on m_a	Avg weight on other memory
0	0.795	0.205
1	0.706	0.294
2	0.661	0.339
3	0.708	0.252
4	0.812	0.188
5	0.710	0.286
6	0.729	0.269
7	0.676	0.315

For every address from 0 to 7, the average weight on the correct memory cell m_a is between 0.66 and 0.81, and the combined weight on all other memory cells is only 0.19 to 0.34. The mapping from address to attended position is clean and consistent across the test distribution. Layer 2 is the dereference, exactly as predicted.

7.4 Layer 1 is the address aggregator

Now the other half: layer 1 should be gathering the address bits into position 10 so that layer 2's query can use them. For the same four examples, here is where layer 1 at position 10 sends its attention. The table gives the top three attended positions with weights, and the total mass on the address positions $\{8, 9, 10\}$ against the memory positions $\{0, \ldots, 7\}$:

Address a	Top-3 (position, weight)	Total on addr	Total on memory
0	(8, 0.49), (9, 0.41), (10, 0.05)	0.956	0.044
4	(8, 0.71), (9, 0.20), (7, 0.06)	0.935	0.065
5	(8, 0.73), (9, 0.19), (10, 0.06)	0.986	0.014
7	(8, 0.51), (9, 0.41), (10, 0.04)	0.965	0.035

The pattern is unambiguous. Layer 1 at position 10 puts 93 to 99 percent of its attention mass on the three address positions. The memory bits, which dominate the information content of the input by a factor of eight to three, receive almost none.

There is a subtler thing in the table worth pausing on. Within the address positions the weights are not uniform. Position 8, the most significant bit a_0, gets the most weight; position 9, a_1, less; position 10, the least significant bit a_2, the least. In the seed-0 run shown here this ordering is sensible rather than incidental, and while the exact weights move between retrains, the bias toward the more significant bits is the stable part. The most significant bit partitions memory into upper and lower halves, the next bit selects quadrants, and the least significant bit selects within pairs. A model that has to write the whole address into a single residual vector through a linear value map will weight its inputs by how much they matter for the downstream lookup, and the most significant bit matters most for telling apart memory positions that are far apart. The hypothesis is vindicated on this side too: layer 1 is gathering the address, by exactly the attention-as-content-addressable-lookup mechanism of Chapter 6, with the position-10 query learning to match the position embeddings of 8 and 9.

7.5 Causal confirmation: ablations

Everything so far is correlational. We have seen that layer 1 attends to the address positions and layer 2 attends to the correct memory cell, which is consistent with the circuit hypothesis. It does not by itself rule out a duller story: that the model leans mostly on position 10's own embedding plus the bias structure of the data, with attention going along for the ride. To separate the two we need a causal test.

The cleanest one is to replace a layer's attention pattern with the uniform-on-causal-prefix distribution, in which each query position attends with equal weight to every position at or before it, and then rerun the rest of the forward pass. This wipes out the layer's content-addressable behavior while leaving its value and feed-forward machinery untouched. If a layer is doing real work, the substitution should hurt. The same routine also supports a second intervention: shuffle the three address bits before the forward pass, which leaves the bit *set* unchanged but destroys which bit is which.

```
def ablation_shuffle_address(model, X, Y, M=8, A=3, seed=0):
    """Permute the address bits across each example before forward."""
    rng = np.random.default_rng(seed)
    correct = 0
    T = M + A
    for ids, y in zip(X, Y):
        ctx = ids[:T].copy()
        addr = ctx[M:M + A].copy()
        rng.shuffle(addr)
        ctx[M:M + A] = addr
        logits = model.forward(ctx)
        pred = int(np.argmax(logits[-1]))
        if pred == int(y):
            correct += 1
    return correct / len(X)
```

The uniform-prefix substitution is built the same way one would expect, by replacing the cached pattern with a normalized lower-triangular matrix before the value-weighted sum:

```
uniform = np.tril(np.ones((T, T), dtype=np.float32))
uniform = uniform / uniform.sum(axis=-1, keepdims=True)
```

On 500 test examples:

Intervention	Test accuracy
Baseline (no intervention)	1.000
Layer 1 attention → uniform	0.658
Layer 2 attention → uniform	0.636
Shuffle the three address bits before forward	0.776

Both layer ablations drop accuracy by a third. Either layer is necessary, neither alone is sufficient, and both have to be doing the hypothesized job. That is the causal confirmation the correlational evidence could not give on its own.

The shuffle ablation is the most instructive of the three. Permuting the address bits keeps the same multiset of bits but destroys the positional encoding of which bit is which. If the model depended only on whether each address bit was 0 or 1, a permutation-invariant function of the bits, the shuffle would not hurt. The drop from 1.000 to 0.776 says the model is using the bits with their

positional roles, exactly as a circuit computing $a = 4a_0 + 2a_1 + a_2$ must. And the fact that accuracy lands at 0.776 rather than at chance is itself informative. The memory bits are still present, so when many of them happen to agree the prediction is right regardless of which cell is addressed. The residual 0.776 is the "guess the majority bit" baseline given a scrambled address, and the 0.224 gap below the baseline is the attributable contribution of correctly routed addressing.

7.6 Is this an induction head?

The circuit we just read out is worth placing in the wider interpretability literature, because it is a near-cousin of the most studied small circuit in transformers: the induction head. The pattern our model implements is not the canonical induction-head pattern, but it is the same *primitive* deployed for a different purpose, and the distinction is exactly what makes the comparison useful.

The induction head, from Olsson and colleagues, is a two-head, two-layer circuit that implements the rule

$$\ldots\ A\ B\ \ldots\ A\ \rightarrow\ B,$$

that is, on seeing token A again, predict whatever followed the previous occurrence of A. It decomposes into two pieces. A **previous-token head** in the first layer makes every position t copy the embedding of token $t-1$ into its residual, so that position t ends up encoding "I am here, preceded by token X." An **induction head** in the second layer then queries, from position t, against earlier positions using that previous-token-shifted signature, finds the position i where the preceding token matched the current token, and copies what was at position i, which is "the thing that followed the last A."

The transformer-circuits framework of Elhage and colleagues decomposes any such head into a **QK circuit**, which determines where the attention spike lands from the query-key product, and an **OV circuit**, which determines what value gets copied once the spike is located. The induction head's QK circuit matches on previous-token-shifted features; its OV circuit copies the value at the matched position into the residual stream. Our pointer circuit fits the same frame exactly, just with different content:

	Induction head	Pointer dereference (ours)
What layer 1 writes into the residual	The previous token's identity, shifted one position	The address bits, gathered from positions 8 and 9 (and 10) into position 10
What layer 2's QK matches on	Previous-token signatures, seeking prior occurrences of the current token	The encoded address, seeking the memory position whose positional encoding matches
What layer 2's OV copies	The token value at the matched position	The memory bit at the matched position
Domain	Token identities and sequences of them	Positional encodings and an address

Both are two-layer compositions of content-addressable lookups, and the *mechanism* is identical: attention as soft retrieval, with the query learning to match the right key. Only the *content* differs, token identities for the induction head against positional addresses for ours. So the answer to "is it an induction head?" is: structurally yes, literally no. It is what one might call the address-lookup head, the same primitive in the same shape, turned on the purer content-addressable-memory task. Induction heads are the version of this circuit that the statistics of natural language ask for; ours is the version a memory-addressing task asks for.

7.7 In-context learning is content-addressable lookup, composed

The reason the induction head matters beyond its own neatness is its connection to **in-context learning**: the ability of a trained language model to use information in its prompt, with no weight update, to inform predictions on a task specified in that prompt. Show a model the prompt

```
zarp -> bumi
flom -> bumi
trel -> tev
clok -> tev
zarp -> ?
```

and it will often emit `bumi`. The model has, in context, learned a mapping from a single example with no gradient step. That capability is much of what separates large language models from the older sequence models, and it is the substance of the in-context-learning literature.

Olsson and colleagues argue that induction heads are a *mechanistic* substrate for this ability: a circuit that finds prior occurrences of the current token and copies the following value is precisely what one-shot retrieval-from-context needs. Empirically, the emergence of induction heads during pretraining coincides with the emergence of measurable in-context-learning ability.

Our pointer-lookup circuit is in the same family. Read the pointer task as a one-shot in-context-learning problem. The **context** is an eight-bit memory and a three-bit address. The **task specification** is implicit: at this position, retrieve memory bit a. The **completion** is the value at the addressed cell. The model was never trained on the specific eleven-bit sequence it sees at inference time, and most of the twenty thousand training examples never reappear; it solves each new example using the in-context memory and address, exactly as an in-context-learning system would. It just happens to do so by positional-address matching rather than by token-pattern matching, which is the same primitive applied to a slightly cleaner data-generating process.

This is the broader picture worth carrying away. In-context learning is what a transformer does when its content-addressable lookup primitive is composed over enough tokens, heads, and layers to encode arbitrary soft retrieval queries. Induction heads are one well-studied two-layer instance of that primitive, and our pointer circuit is another. A trained language model has thousands of attention heads across dozens of layers, each implementing some specialized lookup, and in-context learning is the emergent property of running them all together on a context-dependent task.

Looking back at the inductive-bias frame, this sharpens what a transformer's architectural commitment actually is. It is not just "attention." It is specifically content-addressable retrieval as a composable primitive. Depth is how many composed retrievals the model can chain. Width, the multi-head axis of Section 2.9, is how many parallel retrievals it runs per layer. In-context learning emerges where depth and width are large enough that the composed retrievals can express arbitrary soft programs over the context.

7.8 What this small model leaves out

A word of honesty about scale before we test that claim. The model interpreted here is two layers, one head, eleven tokens of context, a two-token vocabulary, and the circuit is correspondingly minimal. Three things a full interpretability analysis on a larger model would also have to cover are worth naming, both to be candid and because the last section runs into the third of them head-on.

- **Value circuits and embedding geometry.** We have not opened up the embedding vectors for tokens 0 and 1 or the positional encodings. With a two-token vocabulary the embedding

has only two rows to inspect, but how *value* information flows through the OV path of each layer is genuinely interesting at scale, and we leave it closed here.

- **Multi-head specialization.** Real transformers have many heads per layer, each potentially a different lookup. A character-level language model trained on prose, for instance, develops heads that attend to the previous space, heads that attend to the start of the current word, heads that attend to repeated characters, each a small specialized lookup that together enable rich behavior. Our one-head model has none of this structure to find. (These per-head specializations are exactly what mechanistic-interpretability studies of trained character-level language models visualize; our pointer task is deliberately too small to grow them.)

- **Activation patching.** The ablations here zero out an attention layer wholesale. A finer tool is to *patch* an activation from one input into another: copy a residual at one layer and position from one run into another and watch whether the output flips. This isolates the causal role of each component on each example, and it is the basis for most published mechanistic results on language models. The final section leans on exactly this tool, and it turns out to be the difference between seeing the mechanism and not.

The pedagogical point is that the technique applied in this chapter, state a circuit hypothesis from the architectural argument and verify it with attention extraction plus ablation, is the same technique used at scale, only with more components and sharper intervention tools. The next section puts that to the test. It takes the *same task* at $M = 32$, where Chapter 6 found a third layer is required, and asks whether the clean circuit survives the scale-up.

7.9 Scaling the lens: attention hides the circuit, causal tracing reveals it

Everything so far was the $M = 8$ model: eleven tokens, two layers, one head, a circuit clean enough to read off the attention maps. Chapter 6 found that the same task at $M = 32$ needs a *third* layer: in the controlled comparison there, depth solved the task where neither width nor a longer training budget did. We now have a full set of dissection tools. The natural question is whether they reveal the same clean circuit, just scaled up. The attention maps say no. A causal trace says yes. The gap between those two answers is the most important lesson in this chapter.

The model here is a different one, and the difference matters. It is an explicit three-layer transformer with four heads, $d_{\text{model}} = 128$, and learned positional encoding, written so that every attention matrix is directly readable, and trained to 0.9965 test accuracy on $M = 32$. It is *not* the scaling-sweep model of Chapter 6, which used a library attention layer whose weights are awkward to extract. This is a fresh, introspectable model built to be read, and its code is in `examples/pytorch/pointer_interp_deep.py`. The analysis below loads it from a committed checkpoint, so its numbers are fixed. Because it is a separate model in a separate framework, its accuracy is not comparable to the NumPy model's; we read the two as two studies of the same kind of task, not as numbers to line up.

One implementation-realization aside before the mechanism, because it is the same lesson as Section 6.9 in a new place. This model only trains if its query, key, and value projections use Xavier-uniform initialization. With PyTorch's default linear-layer initialization, Kaiming-uniform with $a = \sqrt{5}$, those projections are mis-scaled, the phase transition is pushed from around iteration 7000 out past iteration 22,000, and the model is stuck at 0.62 even at a 60,000-iteration budget. Right architecture, right depth, wrong initialization scale on one set of tensors, and the circuit

never forms. The $M = 16$ failure of Section 6.9 was a scale mismatch between content and position at the input; this is a scale mismatch inside the attention projections. Both are the third axis, implementation realization, deciding whether a bias the architecture permits is one gradient descent actually reaches. It is not a footnote.

Causally, it is a perfect pointer

Start with the question that needs no attention map at all: does the model compute exactly m_a? The test is a causal intervention. Take a correctly handled example, flip the bit at the addressed cell m_a, and see whether the prediction follows. Then flip a bit at a random *non*-addressed cell and see whether the prediction is disturbed. The probe perturbs the input and watches the output, with no assumption about what attention means:

```
@torch.no_grad()
def causal_flip_test(model, X_te, M, A, n_examples=600, seed=0):
    """Flip m_a: does the prediction follow? Flip another cell: does it stay?"""
    rng = np.random.default_rng(seed)
    T = M + A
    tracks_ma = 0
    changed_by_other = 0
    n = 0
    for ids in X_te[:n_examples]:
        ctx = ids[:T].copy()
        a = address_value(ctx, M, A)
        flipped = ctx.copy()
        flipped[a] ^= 1
        if predict_bit(model, flipped) == int(flipped[a]):
            tracks_ma += 1
        base = predict_bit(model, ctx)
        others = [j for j in range(M) if j != a]
        j = int(rng.choice(others))
        other = ctx.copy()
        other[j] ^= 1
        if predict_bit(model, other) != base:
            changed_by_other += 1
        n += 1
    return tracks_ma / n, changed_by_other / n
```

Intervention	Prediction follows?
Flip the addressed cell m_a	tracks the new value 0.995 of the time
Flip a random non-addressed cell	prediction changes only 0.005 of the time

This is airtight. The model reads the addressed cell and nothing else: flipping m_a flips the answer, flipping any other cell does nothing. Behaviorally, the three-layer model is a perfect pointer dereference, exactly like the $M = 8$ model. Whatever the attention maps say, the function being computed is the right one.

Mechanically, the clean circuit is gone

Now read the attention, hoping for the $M = 8$ story scaled up: an aggregation layer and a single lookup head spiking on m_a. Averaged over the test set, layers 1 and 2 both attend overwhelmingly

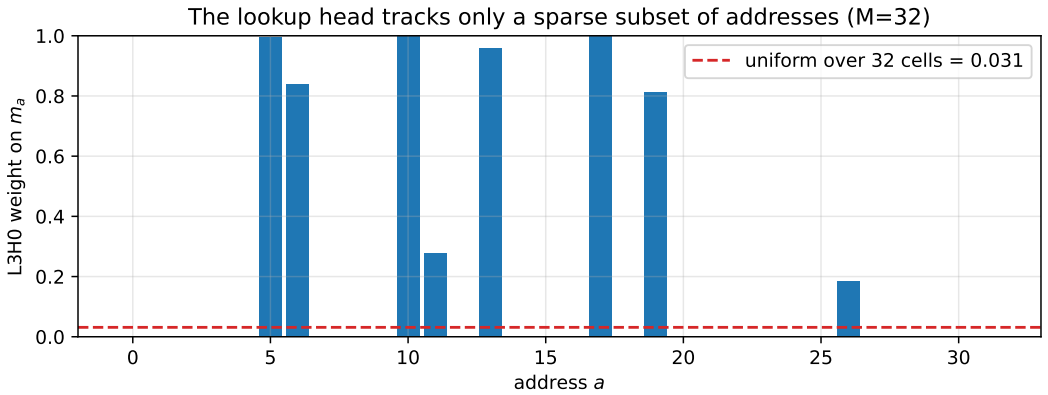

Figure 7.1: The most lookup-like head in the $M = 32$ model (layer 3, head 0) dereferences cleanly, with weight near 1, for only a sparse subset of addresses and stays near the uniform baseline (dashed) for the rest. No single head performs the general one-of-32 lookup that the $M = 8$ model performed in one head.

to the address positions, 0.51 and 0.67 of their weight, not to memory. This model distributes the address aggregation across both of its lower layers, where the $M = 8$ model used a single one. That is a fact about how this trained network used the depth it has, not a claim that the wider address *required* the extra layer: Chapter 6 was careful on exactly that point, since $M = 24$ has the same five-bit address as $M = 32$ yet two layers suffice for it, so what makes the third layer necessary is the precision of the one-of-M selection, not the width of the address decode. The dereference weight, the mass landing on the addressed cell, rises across layers, 0.016 then 0.047 then 0.080, and is largest in layer 3. So far this is consistent with "layers 1 and 2 aggregate, layer 3 dereferences."

Then look at layer 3 head by head. The weight each head puts on the addressed cell:

Layer-3 head	h0	h1	h2	h3
weight on m_a	0.198	0.089	0.031	0.002

Head 0 is the only candidate lookup head, and at 0.198 against 0.031 for uniform it is a weak one. Worse, it does not do a general one-of-32 lookup. Figure 7.1 plots its weight on m_a as a function of the address: it dereferences *cleanly*, with weight 0.8 to 1.0, for a sparse handful of addresses, $a \in \{5, 6, 10, 13, 17, 19\}$, and does essentially nothing for the other twenty-six. Counting every head in layers 2 and 3, the number of addresses that receive a clean single-head spike, weight above 0.5 on m_a, is only 10 of 32. Even summing all four layer-3 heads into an ensemble, the total weight on m_a exceeds 0.5 for just 8 of 32 addresses, mean 0.29. For most of the address space, the last-position query never visibly concentrates on the correct cell.

The per-layer ablations, replacing one layer's attention with uniform and remeasuring, confirm the load-bearing structure without identifying any clean lookup. Baseline accuracy on this harness's example subset is 0.9975, a shade above the 0.9965 full-test figure quoted earlier and measured on a different slice; sending layer 1's attention to uniform drops it to 0.540, layer 2 to 0.762, and layer 3 to 0.600. Layers 1 and 3 are essential, layer 2 is auxiliary. But "layer 3 is essential" together with "layer 3 has no clean lookup head for most addresses" is a genuine puzzle.

Reconciling the two readings

How can the model read exactly m_a while its attention barely concentrates on m_a for most addresses? Two facts dissolve the apparent contradiction.

First, **attention weight is not information flow.** A head's output is its attention-weighted sum of *value* vectors, and the value projection can map an attended-to cell to near-zero in the direction the readout cares about. High attention to a cell does not imply that cell influences the output, and diffuse attention does not imply the output is a blur. The causal flip test is the ground truth precisely because it bypasses this: it perturbs the input and watches the output, assuming nothing about what attention weight means.

Second, **multi-head and multi-layer composition spreads the read out.** Heads concatenate before the output projection, so two heads each placing 0.3 on m_a combine into a confident read that no single-head threshold catches. And the dereference can be indirect: layer 3 can attend to an intermediate position whose residual already carries m_a because an earlier layer routed it there.

It is worth being honest about the arc of this investigation, because it is how the work actually went, and the honesty is the point. The first hypothesis, written before the analysis, was that the third layer is a refinement stage that sharpens a lookup layer 2 already performs. The data refuted it: layer 2 does almost no dereferencing, 0.047 on m_a. The second hypothesis was that the lookup heads partition the address space among themselves. The data refuted that too: all heads together cleanly cover only 10 of 32 addresses. What survived is narrower and truer. The model computes m_a exactly, by the causal test; the depth buys a two-stage address aggregation across layers 1 and 2; and the dereference itself is distributed across heads and the value pathway in a way the attention maps do not show. But "attention does not show it" is not the same as "it cannot be seen." It can. We just need the right tool.

Causal tracing recovers the clean mechanism

The flip test told us m_a is read; it did not tell us *how the value gets from cell a to the readout.* Information moves between positions only through attention, so there has to be a path. To find it, trace the value causally through the residual stream.

The method is activation patching on a minimal pair. For one example, run the clean input and cache the residual stream after the embedding and after each block. Run the corrupted input, the same example with m_a flipped so the correct answer flips. Then patch the *clean* residual into the corrupted run at one layer and position at a time, and measure how much the readout recovers the clean answer:

$$\text{recovery} = \frac{\Delta_{\text{patched}} - \Delta_{\text{corrupt}}}{\Delta_{\text{clean}} - \Delta_{\text{corrupt}}}, \qquad \Delta = \text{logit[answer]} - \text{logit[other]}.$$

The margin Δ compares the output bit that is correct on the clean input, *answer*, against its complement, *other*; the three subscripted margins are that same quantity on the clean run, on the corrupted run, and on the corrupted run after a single residual is patched. Recovery near 1 marks a position that *carries* m_a, since patching it back restores the clean margin; near 0 marks one that does not. The inner loop is short:

```
for L in range(n_layers + 1):
    for p in range(T):
        vec = res_clean[L][0, p]
        patched = model.forward_patched(corrupt, L, p, vec)
        rec = (ldiff(patched) - d_corrupt) / denom
```

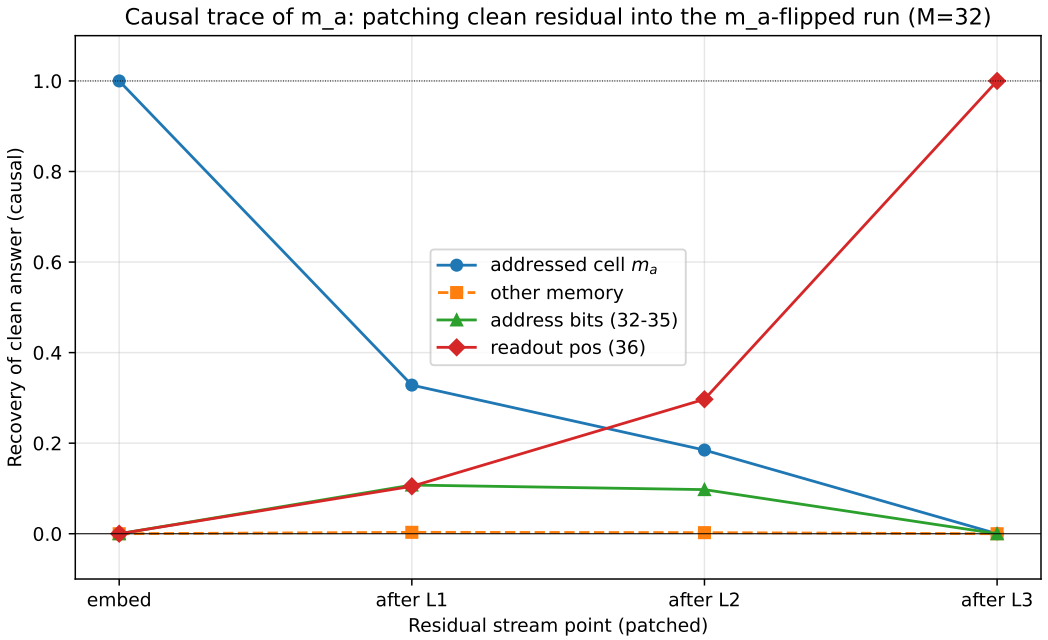

Figure 7.2: Causal trace of m_a in the $M = 32$ model. Patching the clean residual into the m_a-flipped run, the recovery at the addressed cell (top line at the embedding) decays to zero across the three layers while the recovery at the readout position rises from zero to one. The address bits are a partial intermediate conduit; every other memory cell carries nothing at any layer. The value m_a is transported from the addressed cell to the readout over the three layers, a clean mechanism the attention maps could not surface.

```
c = _position_class(p, a, M, A)
sums[(L, c)] += rec
counts[(L, c)] += 1
```

Averaging over 100 minimal pairs, grouped by position class:

Residual point	addressed m_a	other memory	address bits	readout
embedding	1.000	0.000	0.000	0.000
after layer 1	0.328	0.003	0.108	0.105
after layer 2	0.185	0.002	0.097	0.297
after layer 3	0.000	0.000	0.000	1.000

Figure 7.2 plots it. Read top to bottom. At the embedding the two inputs differ only at cell a, so all the recovery is there, 1.000, a sanity check the method passes exactly. After layer 1 the addressed cell has already shed most of its exclusivity, 0.328, and the value has begun appearing at the address bits, 0.108, and the readout position, 0.105. After layer 2 the readout position is the dominant carrier, 0.297, and the cell continues to drain, 0.185. After layer 3 the readout holds *all* of it, 1.000, and the original cell holds *none*, 0.000.

This is a clean, complete mechanism. The value m_a is transported from cell a to the readout position over the three layers, with the address bits a partial intermediate conduit, and every other memory cell carrying nothing at any layer, the flat zero line consistent with the flip test. The dereference is not a single hard attention spike at one layer. It is a gradual, distributed transport,

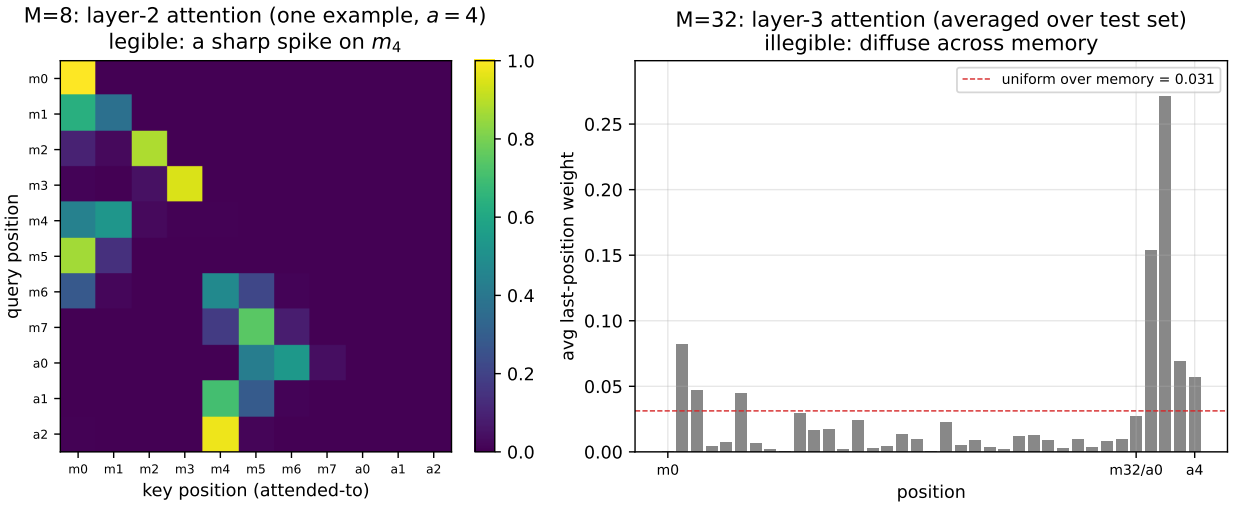

Figure 7.3: Left: in the $M = 8$ model, layer-2 attention from the last position concentrates sharply on the addressed cell m_4, a circuit readable directly from the attention map. Right: in the $M = 32$ model, the last-position attention in layer 3, averaged over the test set, is diffuse across the memory cells and barely rises above the uniform baseline (dashed). The mechanism is no messier in any deep sense, as the causal trace of Figure 7.2 shows; it is the attention instrument that has lost the ability to see it.

which is exactly why staring at any one layer's attention row showed only a fraction of it. The structure was always there; it lived in the residual-stream flow, not in the attention weights. The apparent illegibility of the previous subsection was an artifact of the tool, not a property of the model.

The tool, not the model, decides what is legible

Figure 7.3 puts the two models side by side and makes the lesson visual. On the left, the $M = 8$ layer-2 attention for one example: a sharp spike on the addressed cell, a circuit you can read straight off the map. On the right, the $M = 32$ layer-3 attention averaged over the test set: diffuse across memory, hovering near the uniform baseline, legible to no one. The same task, one more layer, and the obvious instrument goes dark.

That is the lesson in one sentence: the tool, not the model, decides what is legible. The single increment in problem size did not make the mechanism messier in any deep sense, since it is still a clean pointer that transports one bit to the readout. It made the mechanism invisible to the weak instrument while leaving it fully visible to the strong one. This is why the field leans on causal interventions, activation patching and path patching and ablation, rather than attention visualization, and why a claim like "head 7 is the lookup head" is suspect without a causal test behind it. Attention weight is suggestive, not probative. The induction-head story of Section 7.6 has the same shape at scale: in real language models induction is distributed across many heads, ablating any one barely moves the behavior because the others compensate, and the clean accounts come from causal patching, not from looking at attention. Our $M = 8$ to $M = 32$ transition is that whole arc in miniature, a legible toy circuit, an apparent loss of legibility as the task grows, and a causal tool that restores it.

7.10 Closing: the inductive-bias frame, completed

The book has organized everything around three axes (Section 6.11):

1. **Architecture**: structural priors about how features compose, such as locality, recurrence, and content-addressable lookup.
2. **Output head**: priors about the distribution of the response, such as Bernoulli, Categorical, Gaussian, and Poisson.
3. **Implementation realization**: whether the gradient procedure can actually discover the right parameters, given initialization, optimizer, data, and schedule.

This chapter has been the **interpretability lens** that sits across all three. Once a model is trained, mechanistic interpretation tells us *which* of the bias-permitted solutions the gradient procedure actually picked, and how legibly. At $M = 8$ the answer was a single, clean, fully readable circuit: layer 1 aggregates the address, layer 2 dereferences, and ablations confirm both are load-bearing. At $M = 32$, one increment of scale later, the answer was a causally perfect pointer whose mechanism the attention maps render only partially legible, and which a causal trace makes clean again.

That progression is the real shape of the field: legible circuits in the smallest models, partial legibility as scale grows, and causal probes rather than weight-reading as the reliable ground truth. The meta-lesson is the one the $M = 32$ model forced on us. Attention weight is suggestive, not probative; it is not information flow; and a computation spread across heads, layers, and the value pathway will hide from it. Reading a model is not staring at its attention maps. It is intervening on its activations and watching what changes. Interpretability is how we discover which solution gradient descent picked, and the $M = 8$ to $M = 32$ step is a warning about how quickly "which" gets hard to answer.

The closing chapter changes the setting entirely. Until now the learning signal has been a per-example label, a target the network is trained to match. The next chapter replaces it with a scalar reward over whole trajectories and turns to reinforcement learning, where the supervised toolkit assembled across the first seven chapters reappears as the inner loop of something larger. The inductive-bias frame carries over, the interpretability lens carries over, and the mechanism we just dissected becomes one tool among several for understanding what a learned agent has actually learned.

Bibliographic Notes

The circuit reading in this chapter follows the transformer-circuits line of work. Elhage, Nanda, Olsson, et al. (2021) sets out the framework for reading attention transformers as compositions of QK and OV circuits, the decomposition used throughout Section 7.6 to compare the pointer-lookup head to the induction head. Olsson, Elhage, Nanda, et al. (2022) identifies induction heads and ties their emergence during pretraining to the emergence of in-context learning, the connection Section 7.7 builds on. The pointer-dereference circuit dissected here is a deliberately minimal cousin of those mechanisms, the same content-addressable primitive turned on a purer memory-addressing task.

The causal interventions of Section 7.9 are the methodological core of modern mechanistic interpretability. Wang et al. (2022) works out a full circuit for indirect-object identification in a real language model using activation and path patching, and makes the case that attention-pattern

reading is too weak an instrument once a computation distributes across heads and layers. The activation-patching causal trace used on the $M = 32$ model, patching a clean residual into a corrupted run and measuring recovery, is the technique Meng et al. (2022) introduced for locating where a fact is stored in a transformer; the same minimal-pair idea is what Figure 7.2 applies to locate where the addressed bit is carried.

Chapter 8

Reinforcement Learning: A Scalar Reward over Trajectories

Every chapter so far has varied one of two parallel axes. The architecture moved from the multilayer perceptron of Chapter 1 through the convolutional, recurrent, fixed-context, and attention families of Part II. The output head moved through the Bernoulli, categorical, Gaussian, and Poisson choices of Chapter 2. Both axes lived entirely inside **supervised learning**. The training signal was always a label per example, and the gradient question was always the same: how do I move $f_\theta(x)$ closer to this specific y?

Reinforcement learning is the paradigm where that signal disappears at the example level and reappears as a scalar reward, summed along a whole trajectory. The distinctive hard part is not the absence of labels. It is **temporal credit assignment**: a trajectory of two hundred actions produces one number, and the learner has to work out which of those actions mattered. Every named technique in reinforcement learning, from Monte Carlo returns to temporal-difference bootstrapping to generalized advantage estimation to eligibility traces to hindsight relabeling, is a piece of credit-assignment machinery. Supervised learning never needs any of it, because it never has the problem: the label pins the blame on the input exactly, and the chain rule walks that blame backward through the network.

This chapter does the minimum to draw the line between supervised learning and reinforcement learning precisely, and then to dissolve part of it. It sets up the trichotomy and the Markov decision process, derives the REINFORCE estimator, and then reads the result back as something already familiar: softmax cross-entropy weighted by return. The per-step gradient is the very expression of Section 1.5, the predicted distribution minus the one-hot of the chosen action, now scaled by how good the trajectory turned out. The supervised toolkit assembled across Parts I and II is, quite literally, the inner loop of reinforcement learning. A real NumPy gridworld makes that concrete, AIXI names the incomputable north star (mirroring the Solomonoff ideal of Chapter 4), and the closing section ends the book.

8.1 The trichotomy

Three standard learning paradigms, with the boundary that matters drawn at the right place.

- **Supervised.** Given (x, y) pairs, learn f_θ such that $f_\theta(x) \approx y$. One gradient per example. The loss tells you the answer was wrong, *and how to be less wrong on this exact input.*

- **Unsupervised.** Given x alone, learn structure: density, clusters, manifolds, factors. There is no target.
- **Reinforcement learning.** Given an environment and a scalar reward, learn a policy $\pi_\theta(a \mid s)$ that maximizes expected return. The signal is one number per trajectory, not a label per example.

Self-supervised learning, the engine of every modern language model, is not a fourth paradigm. Next-token prediction, masked-patch reconstruction, and contrastive pretraining all manufacture their own labels from unlabeled data, and once the label exists the objective is ordinary supervised learning. It is mathematically identical and practically distinct only because the data supply is effectively unbounded, which is what changes the scaling story. A transformer trained on ten terabytes of web text is running the same loss minimization as a one-layer softmax classifier on a table of digits, with more data and a heavier model. Self-supervised learning is supervised learning that solved its label problem.

So the boundary that matters is not "do we have labels" but "do we have a *per-example* gradient signal." Supervised and self-supervised learning both do. Reinforcement learning does not, and that single absence is what the rest of the chapter is about.

8.2 The reduction loses what makes RL hard

A common move flattens the distinction: supervised learning, the argument goes, is just reinforcement learning with reward $r = -\mathrm{NLL}(y \mid x)$ at every step. This is technically true. The reduction packages each (x, y) pair as a one-step episode in which the agent takes action $\hat{y}$, the environment immediately reveals y, and the reward is the negative log-likelihood of y under $\pi_\theta(\cdot \mid x)$. Maximizing expected reward is then maximum likelihood, and the whole supervised apparatus reappears as a degenerate special case.

What the reduction quietly discards is everything that makes reinforcement learning hard:

- **Delayed reward.** Here the reward arrives at the same step as the action. In general it can arrive many steps later, or only at the end.
- **Sparse reward.** Here every action has a reward. In general most actions return nothing, and a single number arrives at the end of a long trajectory.
- **Exploration.** Here the action $\hat{y}$ and the gradient of the reward with respect to it are directly available; there is no need to try the alternatives. In general the agent only ever sees the reward of the action it actually took.
- **Off-policy correction.** Here the data and the policy are the same thing. In general there is a real distinction between the behavior that generated the data and the target being improved.

Each of those is a hard part of reinforcement learning erased. The genuine difficulty underneath all of them is **temporal credit assignment**. A trajectory $\tau = (s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_T)$ produces a single return $R(\tau) = \sum_t r_t$. The learner took, say, two hundred actions. Which of them were responsible for the return being high or low?

REINFORCE, advantage estimation, temporal-difference learning, generalized advantage estimation, eligibility traces, hindsight relabeling, off-policy corrections: every named technique in the field is a piece of that credit-assignment machinery. Supervised learning never needs any of it,

because the label tells you precisely which input produced which loss, and the chain rule walks the blame backward through the network. The reinforcement-learning loss surface, by contrast, is computed *after* the actions are committed, and the trajectory probability is a product over many policy decisions of which only some mattered. The reframing earns its keep because it lets a reader who has only done supervised work see *why* reinforcement learning is hard, not merely *that* it is.

8.3 The MDP framing

The standard formalism is the **Markov decision process**. An agent and an environment interact in a loop. At step t the agent observes a state s_t, takes an action $a_t \sim \pi_\theta(\cdot \mid s_t)$, and the environment returns a reward r_{t+1} and a next state $s_{t+1} \sim p(\cdot \mid s_t, a_t)$.

The **Markov property** is the assumption that the transition distribution $p(s', r \mid s, a)$ depends only on the current (s, a), not on the prior history. The "state" is, by definition, whatever you need to remember for that to hold.

The **discount** is a choice of $\gamma \in [0, 1)$. The discounted return from step t is

$$G_t = \sum_{k=0}^{\infty} \gamma^k \, r_{t+k+1}.$$

The discount is geometric weighting on the future. At $\gamma = 0$ the agent is myopic and only the next reward matters; as $\gamma \to 1$ a reward now and a reward in a thousand steps are valued equally. In practice $\gamma \in [0.9, 0.999]$ is typical, and the effective horizon is about $1/(1-\gamma)$ steps.

The **objective** is to maximize the expected return

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[\, R(\tau) \,],$$

where $\tau = (s_0, a_0, r_1, s_1, \ldots)$ is a trajectory sampled by following π_θ in the environment and $R(\tau) = \sum_{t \geq 0} \gamma^t r_{t+1}$. Two derived quantities organize most algorithms, the state-value and action-value functions:

$$V^\pi(s) = \mathbb{E}_\pi[\, G_t \mid s_t = s \,], \qquad Q^\pi(s, a) = \mathbb{E}_\pi[\, G_t \mid s_t = s,\, a_t = a \,].$$

A policy is a conditional distribution over actions given states; a return is the discounted sum of rewards along the trajectory the policy induces; we want the policy whose induced trajectories have the highest expected return; and V^π and Q^π are the natural quantities to estimate along the way. The next section asks how to turn samples of $R(\tau)$ into a gradient on θ at all.

8.4 Credit assignment, three families

Supervised learning gives every example its own loss, and the chain rule pins the blame precisely. Reinforcement learning gives one return per episode, or a few sparse rewards along the way, with a sequence of many actions between them, and the question of which action mattered is genuinely open. There are three families of answers, each a different bet about how to split the credit.

- **Value-based (and Monte Carlo at the extreme).** Estimate how good states or actions are. The Monte Carlo version waits until the episode ends and attributes the entire realized return G_t to every action taken, optionally discounted by recency: high variance, since one trajectory is one sample, but unbiased, since the return is what actually happened. Temporal-difference methods do not wait; they use a learned value estimate $V(s_{t+1})$ as a bootstrap target for $V(s_t)$ through the Bellman equation, which lowers variance at the cost of a bias from the accuracy of V.

- **Policy-gradient.** Skip the per-trajectory credit-assignment question entirely. Estimate the gradient of $J(\theta)$ as an expectation over trajectories, average many samples, and let the averaging do the work of variance reduction. This is the family the chapter develops, because the bridge to supervised cross-entropy is exact and the derivation is short.
- **Model-based.** Learn the environment's dynamics $p(s' \mid s, a)$ and plan against the learned model, paying the cost of fitting it in exchange for being able to imagine rollouts rather than execute them.

Actor-critic, n-step returns, generalized advantage estimation, and eligibility traces are interpolations between these: bias-variance trade-offs on the shape of the credit signal.

8.5 REINFORCE, derived

Parameterize the policy as $\pi_\theta(a \mid s)$. The mental image is a neural network outputting a softmax over discrete actions, or the parameters of a continuous distribution; the gridworld below uses the discrete-softmax form, with the same multilayer perceptron of Chapter 1 behind the head. The objective is the expected return under the policy-induced trajectory distribution,

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)] = \sum_\tau p_\theta(\tau)\, R(\tau),$$

with $p_\theta(\tau)$ the probability of sampling trajectory τ. We want $\nabla_\theta J(\theta)$.

The load-bearing identity is the **log-derivative trick**. For any parameterized distribution p_θ,

$$\nabla_\theta p_\theta(\tau) = p_\theta(\tau)\, \nabla_\theta \log p_\theta(\tau),$$

which is nothing more than $\nabla \log x = \nabla x / x$ rearranged. It turns the gradient of a probability, which is awkward because probabilities are normalized, into the gradient of a log-probability, which factors cleanly, times the probability itself, which becomes an expectation as soon as we sample. Applying it to the objective,

$$\nabla_\theta J(\theta) = \sum_\tau \nabla_\theta p_\theta(\tau)\, R(\tau) = \mathbb{E}_{\tau \sim \pi_\theta}[\nabla_\theta \log p_\theta(\tau)\, R(\tau)]\,.$$

Now factor the trajectory probability. A trajectory is sampled by alternating policy choices and environment transitions:

$$p_\theta(\tau) = p(s_0) \prod_{t \geq 0} \pi_\theta(a_t \mid s_t)\, p(s_{t+1} \mid s_t, a_t).$$

Take the log, turning the product into a sum, and differentiate with respect to θ:

$$\nabla_\theta \log p_\theta(\tau) = \sum_t \nabla_\theta \log \pi_\theta(a_t \mid s_t).$$

The initial-state distribution $p(s_0)$ and the transition kernel $p(s_{t+1} \mid s_t, a_t)$ do not depend on θ, so their gradients vanish. Only the policy factors survive. This is the structural fact that makes model-free reinforcement learning possible: you can estimate the policy gradient without ever knowing the environment's dynamics.

Substituting back, the full-return weight on each step collapses to the **reward-to-go** G_t. A reward earned *before* step t cannot have been influenced by the action a_t, so conditioning on the

history up to s_t and using the zero-mean score identity $\mathbb{E}_{a_t}[\nabla_\theta \log \pi_\theta(a_t \mid s_t)] = 0$ (the same fact the baseline below relies on), its contribution to that step's gradient is exactly zero. What survives is the reward-to-go form of REINFORCE:

$$\boxed{\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, G_t\right].}$$

In words: averaged across sampled trajectories, the gradient of the expected return is the sum over time steps of the log-likelihood gradient of the action taken, each weighted by the return earned from that step onward. Sample one trajectory, compute that sum, and you have an unbiased Monte Carlo estimate of $\nabla_\theta J$; step along it with SGD.

Variance reduction by a baseline. Subtracting any state-dependent baseline $b(s_t)$ from G_t leaves the expectation unchanged but typically reduces variance:

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\,\big(G_t - b(s_t)\big)\right].$$

The expectation is unchanged because $\mathbb{E}_a[\nabla_\theta \log \pi_\theta(a \mid s)] = 0$ for every s, so multiplying by a state-dependent constant and summing contributes nothing in expectation. How much the variance drops depends on how strongly $b(s_t)$ correlates with G_t; a learned value estimate $V_\phi(s_t)$ is the standard choice, and $G_t - V_\phi(s_t)$ is then an estimate of the advantage $A^\pi(s,a)$. That is the half-step from REINFORCE to actor-critic. The gridworld example uses a simpler scalar baseline, a moving average of recent episode returns; a global constant is the trivial state-dependent baseline, so the same unbiasedness argument applies, and it is enough to make the gradient useful at that scale.

8.6 The bridge: REINFORCE is weighted cross-entropy

This is the chapter's payoff, and it is a derivation, not an analogy. Look at the per-step term in the REINFORCE update, written as a loss to descend (the minus sign is so that descending on it ascends on expected return):

$$\ell_t(\theta) = -\log \pi_\theta(a_t \mid s_t) \cdot G_t.$$

The whole claim is that the first factor is, identically, the supervised softmax cross-entropy gradient of Section 1.5. Make the policy concrete the way the gridworld does. The network produces a vector of logits $z = g_\theta(s_t) \in \mathbb{R}^K$ over the K actions, and the policy is their softmax,

$$\pi_\theta(a \mid s_t) = \mathrm{softmax}(z)_a = \frac{e^{z_a}}{\sum_j e^{z_j}}.$$

Treat the action actually taken, a_t, as a one-hot label, and the log-policy of that action is exactly the negative cross-entropy between that one-hot and the predicted distribution:

$$-\log \pi_\theta(a_t \mid s_t) = -\log \mathrm{softmax}(z)_{a_t} = \underbrace{\log\Big(\textstyle\sum_j e^{z_j}\Big) - z_{a_t}}_{\text{softmax cross-entropy on the label } a_t}.$$

Differentiate with respect to the logits, the same one-line computation carried out in Section 1.5. The log-sum-exp term contributes $\mathrm{softmax}(z)_k = p_k$ in coordinate k, and the $-z_{a_t}$ term contributes -1 in coordinate a_t only:

$$\frac{\partial}{\partial z_k}\big(-\log \pi_\theta(a_t \mid s_t)\big) = p_k - \mathbf{1}[k = a_t], \qquad \text{that is} \qquad \left.\frac{\partial \ell_t}{\partial z}\right|_{G_t=1} = p - \mathrm{onehot}(a_t).$$

This is the predicted distribution minus the one-hot of the chosen action: the *same* expression the supervised softmax head produces, the one `SoftmaxCrossEntropy` returns from its backward pass. The full per-step REINFORCE gradient is just that vector scaled by the return:

$$\frac{\partial \ell_t}{\partial z} = (p - \text{onehot}(a_t))\, G_t.$$

Nothing else changes. The chain rule from the logits back through the head and the multilayer perceptron is identical to the supervised case, because it is the same network. The only edit to the entire backward pass is a single scalar multiply per time step.

So policy gradient is supervised softmax cross-entropy on the actions the agent took, with each per-sample loss reweighted by the return G_t (or by an advantage). Reading the same identity at three weights tells the whole family apart:

- **Behavioral cloning** is the constant-weight case. Given a dataset of expert (s, a) pairs, fit π_θ by ordinary softmax cross-entropy. It is pure supervised learning of a policy.
- **REINFORCE** is the same softmax cross-entropy on the actions the *agent* took, weighted by how good the return turned out to be.
- **Actor-critic** is the same softmax cross-entropy weighted by an advantage, with a learned baseline subtracted from G_t.

In supervised learning the dataset tells you what the right action was. In reinforcement learning the return tells you, after the fact, how good the action you took was, and you weight the gradient accordingly. The loss surgery is the same; what differs is who supplies the label, you or the environment, and how the per-sample importance is scored, uniformly or by return. This is the through-line of the whole book stated at its sharpest. The supervised toolkit assembled in Chapter 1 and Chapter 2, a `Linear` stack, a tanh nonlinearity, and a softmax cross-entropy head, is not merely *analogous* to the inner loop of reinforcement learning. It *is* the inner loop. The gridworld implements this literally: its update is a softmax over the episode's states, a subtract-the-one-hot step, and a per-step scale by the return.

```
# The REINFORCE gradient for one episode (examples/reinforce_gridworld.py).
# states: (T, 25) one-hot states; actions: (T,); weights: (T,) = G_t - baseline.
Z2 = H @ self.W2 + self.b2 # action logits, shape (T, 4)
Z2 = Z2 - Z2.max(axis=1, keepdims=True)
P = np.exp(Z2); P /= P.sum(axis=1, keepdims=True) # softmax over actions

dlogits = P.copy()
dlogits[np.arange(T), actions] -= 1.0 # p - onehot(a): the Ch.1 gradient
dlogits *= weights[:, None] # scale each step by G_t - baseline
# ... and the rest is the ordinary MLP backward pass through W2, tanh, W1.
```

The line `dlogits[np.arange(T), actions] -= 1.0` is the entire conceptual content of the chapter. It is Section 1.5's $p - \text{onehot}(a)$, evaluated at every step of the trajectory; the line below it, the scale by the return, is the only thing reinforcement learning adds to the supervised gradient.

8.7 Worked example: REINFORCE on a 5×5 gridworld

The full implementation is `examples/reinforce_gridworld.py`, around 250 lines of standalone NumPy. It is NumPy rather than the pure-Python `scratchnn` core for the same honest reason

the Transformer chapter gave: a policy-gradient run does 2000 episodes of up to 25 steps each, so roughly 5×10^4 forward passes, and the per-step Python overhead of the teaching core would dominate the actual learning. The model itself is the same multilayer perceptron of Chapter 1, just vectorized. The policy network, `PolicyMLP`, is deliberately the same `Linear(25, 64)` $\rightarrow$ `Tanh` $\rightarrow$ `Linear(64, 4)` stack with a `SoftmaxCrossEntropy` head that `scratchnn` would build, so the chapter can point at the toolkit without importing it. The interesting object is no longer the layer; it is the loss, and the loss is the one derived in Section 8.6.

Why a gridworld. A multi-armed bandit has no state and collapses credit assignment to a single step. A continuous-control task like CartPole has a dense reward, which hides the credit-assignment problem behind a forgiving signal. A 5×5 gridworld has a small discrete state space, a sparse reward (almost all of it arrives at the goal), and an obvious picture: credit assignment is on the page, and you can watch the value of progress propagate backward from the goal as training proceeds.

Setup. The state is the agent's position $(r, c) \in \{0, \ldots, 4\}^2$, one-hot-encoded as a 25-dimensional vector. The four actions are up, down, left, and right; off-grid moves are no-ops. The reward is -0.01 per step and $+1.0$ on entering the goal cell, and the episode ends at the goal or after 25 steps. The discount is $\gamma = 0.99$. The policy is the one-hidden-layer MLP above, 64 tanh units into a four-way softmax. The update is REINFORCE with a moving-average return baseline, $b_{\text{new}} = 0.9\, b_{\text{old}} + 0.1\, R_{\text{episode}}$, and the optimizer is plain SGD with learning rate 0.05.

The training loop. For each episode the loop rolls out under π_θ, computes the returns-to-go by a single backward pass over the rewards, forms the per-step weights $w_t = G_t - b$, and descends on $-\sum_t \log \pi_\theta(a_t \mid s_t)\, w_t$. That descent is the softmax cross-entropy of Section 8.6 summed over the episode and reweighted per step.

```
# The REINFORCE training loop (examples/reinforce_gridworld.py, trimmed).
for ep in range(n_episodes):
    states, actions, rewards = rollout(policy, rng) # sample a trajectory
    G = returns_to_go(rewards) # G_t, one backward pass
    episode_return = float(rewards.sum())

    # Moving-average baseline; subtracting it leaves the gradient unbiased.
    baseline = baseline_beta * baseline + (1 - baseline_beta) * episode_return
    weights = G - baseline # w_t = G_t - baseline

    grads = policy.grads_for_episode(states, actions, weights) # p - onehot, *w
    policy.step_sgd(grads, lr) # one SGD step
```

Results. The optimal trajectory is the Manhattan path of length 8, so the optimal return is $1.0 - 0.01 \times 7 = 0.93$ (seven step-penalties before the goal-reward step). The numbers below are the seed-fixed run in the paired notebook (`notebooks/ch08-rl.ipynb`; seed 0, 2000 episodes, $\gamma = 0.99$, learning rate 0.05). Before training, under the randomly initialized policy:

$$\text{mean return } -0.1258, \qquad \text{mean length } 24.20 \text{ steps}, \qquad \text{success } 11.5\%.$$

The agent wanders. About one trial in nine stumbles into the goal inside the 25-step budget, and the expected return is negative because most episodes pay the per-step cost and hit the step limit without a reward. After 2000 episodes:

$$\text{mean return } +0.9278, \qquad \text{mean length } 8.21 \text{ steps}, \qquad \text{success } 100.0\%.$$

Every sampled trajectory now reaches the goal, and the average length is barely above the optimal

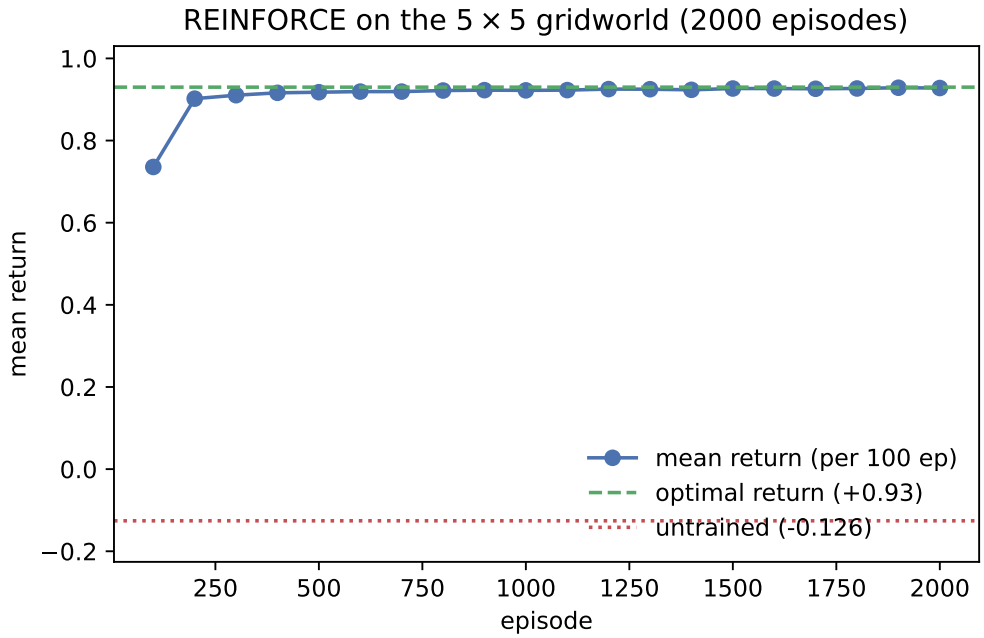

Figure 8.1: Mean return per 100 episodes against episode number for the REINFORCE agent on the 5×5 gridworld (the seed-fixed run in `notebooks/ch08-rl.ipynb`). The dashed line marks the optimal return of $+0.93$ for the eight-step Manhattan path; the dotted line marks the untrained baseline of -0.126. The curve has the shape every policy-gradient run on a sparse-reward task has: a fast climb in the first roughly 200 episodes once the first chance trajectory reaches the goal and REINFORCE reinforces every action that got it there, then a slow refinement as the policy concentrates on the shortest paths.

8. The trained greedy policy traces an exact shortest path,

$$(0,0) \to (1,0) \to (2,0) \to (2,1) \to (2,2) \to (2,3) \to (3,3) \to (3,4) \to (4,4),$$

eight steps, return $+0.93$, exactly optimal. The policy has learned to walk to the goal.

Figure 8.1 plots the smoothed learning curve, and the shape it shows is the one that matters. While the agent never reaches the goal, every episode collects only the -0.01 per-step penalty and the parameters drift on essentially uninformative gradients. The first chance trajectory that reaches the goal produces a strongly positive return, the REINFORCE update reinforces every action it took, and the policy now has a non-trivial probability of repeating that trajectory; subsequent successes refine it. The fast initial rise is the moment the algorithm finds an algorithm; the long tail is parameter tuning. This is the smallest visible version of a phenomenon that haunts larger reinforcement learning: with sparser rewards, longer horizons, or more actions, the chance trajectory that gives the first informative return can take far longer to arrive, and reward shaping, curiosity-driven exploration, and behavioral-cloning warm-starts are all interventions aimed at shortening that wait.

8.8 AIXI: the theoretical north star

Solomonoff induction, the ideal that framed Chapter 4, is the optimal incomputable *predictor*: a Bayesian mixture over all computable environments, each weighted by its program length under a universal prior. It is what an idealized agent would do if it could enumerate every Turing machine, score each by how well it explains the observations seen so far, and weight its predictions by the prior times the likelihood. Practical language models, the fixed-context network of Chapter 4 among them, are computable approximations to that ideal.

AIXI (Hutter, 2000) is its action-taking sibling. Where Solomonoff predicts, AIXI *acts*. At each step t with interaction history $h_{<t}$, AIXI selects

$$a_t^* = \arg\max_{a_t} \sum_{o_t r_t} \xi(o_t r_t \mid h_{<t}\, a_t)\, V^*(h_{<t}\, a_t\, o_t\, r_t),$$

where ξ is the Solomonoff prior over environments (a mixture over all programs that could have generated the observed history), o_t is the next observation, r_t the next reward, and V^* the optimal value function under that prior, defined by the same recursion. Decoded: AIXI maintains a posterior over every environment consistent with what it has seen, and at each step takes the action whose expected future discounted reward, averaged over that posterior, is highest. The expectation runs over the infinite-horizon trajectory under the optimal policy in each candidate environment, which is itself an AIXI-style recursion.

So AIXI is, exactly, Solomonoff induction plus Bayesian decision theory plus reward maximization, fastened together. Each piece is a component a full treatment of reinforcement learning would unpack on its own: the Solomonoff prior is a universal Bayesian prior, computable to no finite degree but the right ideal for a learner that wants robustness to model misspecification; Bayesian decision theory says the action maximizing posterior expected utility is the action a coherent agent should take; and reward maximization sets that utility to a discounted sum of rewards, a strong modeling commitment in its own right. How that reward is chosen, and how it can mislead, is the design surface the next section opens.

The parallel. The supervised arc had Solomonoff induction as the unachievable but illuminating prediction ideal, with modern language models as its practical computable approximations. Reinforcement learning has AIXI as the unachievable but illuminating *action* ideal, with modern algorithms (REINFORCE, deep Q-networks, proximal policy optimization, MuZero, decision transformers) as its practical computable approximations. In both arcs the architecture is the computable inductive bias substituted for an incomputable optimum. You cannot run Solomonoff; you can run a transformer. You cannot run AIXI; you can run proximal policy optimization. The architecture *is* where you compromise, and reading AIXI as the target sharpens what each practical method is doing. Q-learning learns a single environment model implicitly where AIXI averages over all of them; proximal policy optimization uses a parameterized policy and a clipped surrogate objective where AIXI takes an exact arg-max over an infinite expectation; decision transformers collapse policy and value into one sequence-prediction problem where AIXI keeps them separate. Each compromise can be located as a specific distance from the AIXI specification.

8.9 Inductive bias, the reinforcement-learning axis

The supervised chapters argued that architecture (Chapter 3, Chapter 5, Chapter 6) and output head (Chapter 2) are parallel axes of inductive bias. The same frame extends to reinforcement learning, with more design surface, each piece of which encodes an assumption about the task.

1. **Reward shaping.** The reward function is a prior about the value landscape. A sparse goal-reward, the gridworld setup above, is minimal-prior: the agent has to find a successful trajectory before it gets any informative gradient at all. A shaping reward, a small positive bonus for moving closer to the goal, injects information about which intermediate states count as progress. The same lever can mislead if the hint is wrong, which has its own literature under reward hacking and Goodhart's law. The reward is a teacher, and the choice of teacher is a prior.

2. **Policy architecture.** Whether π_θ is a multilayer perceptron, a convolutional network over pixels, a recurrent network over partial observations, or a transformer over state-action histories is the *same* architectural choice the supervised chapters catalogued. The convolutional locality of Chapter 3, the recurrence of Chapter 5, and the content-addressable lookup of Chapter 6 all carry over intact, now in service of action selection rather than prediction. The gridworld uses a multilayer perceptron because its state is small, discrete, and structureless; Atari from pixels uses a convolutional network; partially observed tasks with hidden state use a recurrent network or a transformer.

3. **Exploration strategy.** Epsilon-greedy, an entropy bonus, intrinsic curiosity, upper-confidence-bound action selection, Thompson sampling: each is a prior about where useful information lives. Epsilon-greedy assumes uniform-random is good enough some of the time; curiosity assumes novel states are valuable; Thompson sampling assumes a posterior over models and exploits its variance. Different priors, different exploration behavior, and a real source of difference between algorithms on sparse-reward tasks.

4. **Algorithm class.** On-policy against off-policy, model-free against model-based, value against policy against actor-critic: each commits to a structural assumption about what is cheap, what is reliable, and what generalizes. Off-policy methods assume trajectories from older policies still inform the current one; model-based methods assume a learned dynamics model will repay the cost of fitting it; value-based methods assume the Bellman recursion is more reliable than direct policy parameterization. The choice is a prior about the structure of the task and of the learning dynamics.

All of these are inductive biases in exactly the sense that translation equivariance is for a convolutional network. The heads-as-bias frame of Chapter 2 carries over cleanly too: a discrete policy is a categorical (softmax) head, and a value function is an identity-plus-squared- error (Gaussian) head, the same two output-distribution choices already in hand, now predicting actions and returns instead of labels. A real reinforcement-learning system is a stack of priors at every level, reward, policy architecture, exploration, algorithm class, and matching each to the task is a separate engineering problem. The pattern from Chapter 2 through Chapter 6 generalizes to reinforcement learning; it just has more axes.

8.10 Closing: the book, completed

This is where the book ends, so it is worth saying plainly what the eight chapters were for. The thread was a single equation with three slots: a parametric model, an output distribution, and a loss, trained by gradient descent on data. Choosing what to put in those slots is choosing an inductive bias, and the whole book was a tour of those choices along three axes (Section 6.11).

1. **Architecture.** Structural priors about how features compose: locality and weight-sharing in a convolutional network, recurrence in a recurrent one, a bounded window in the fixed-context

model, content-addressable lookup in a transformer. Each is a statement about which functions are easy to represent and which the data should not have to teach from scratch.

2. **Output head.** Priors about the distribution of the response: Bernoulli for a bit, categorical for a class, Gaussian for a real value, Poisson for a count. The head fixes what "being right" even means, and the loss follows from it.

3. **Implementation realization.** Whether gradient descent, from a given initialization and schedule, actually finds one of the bias-permitted solutions, and whether we can read that solution back off the trained weights.

The journey ran the length of all three. It opened (Chapter 1) with the smallest non-trivial case, a multilayer perceptron learning XOR, the problem a single linear boundary provably cannot solve and a hidden layer can: the first inductive bias in the book (Section 1.7), the bias toward composing simple boundaries into a curved one. It then walked the output head through its distributional choices, and the architecture through the convolutional, recurrent, fixed-context, and attention families, each matched to a structure in its data. The pointer transformer was the high-water mark of the architecture axis, a model that solves content-addressable lookup because the lookup is built into the attention operation. Then the book turned the lens around and read the bias back out of a trained model, finding that having the right architecture is necessary but not sufficient: at the smallest scale the learned circuit was perfectly legible, and one increment of scale later a causal trace, not the attention maps, was what recovered the mechanism. That was the third axis made empirical, and the book's most honest result, that what gradient descent realized is a separate question from what the architecture permits.

This final chapter changed the learning signal one last time. The per-example label disappeared and came back as a scalar reward over a whole trajectory, so the hard new problem was temporal credit assignment. And yet the inner loop turned out to be the toolkit from the very first chapter: REINFORCE is softmax cross-entropy on the actions the agent took, the predicted distribution minus the one-hot chosen action, scaled by the return. The supervised gradient of Chapter 1 is, line for line, what a policy-gradient agent runs; what reinforcement learning adds is the outer loop, the credit-assignment machinery, and the trajectory distribution over which that inner loss is weighted. The same is true above the gridworld: softmax cross-entropy is the gradient signal in behavioral cloning, in policy gradient, in the clipped surrogate of proximal policy optimization, in the policy network of AlphaZero; squared error is the gradient signal in value-function regression and in the reward-model fitting step of learning from human feedback. The supervised toolkit is the inner loop of every modern learning system; the rest is which labels you feed it and how you weight them.

Two incomputable ideals bracketed the book. Solomonoff induction was the optimal predictor that no machine can run, and the language models of Part II were its computable approximations. AIXI is its action-taking sibling, the optimal agent that no machine can run, and the REINFORCE agent of this chapter is its smallest computable shadow: a single policy network, a single return, a single gradient step. Between the incomputable optimum and the runnable approximation sits the architecture, which is exactly where the compromise lives, and which is exactly what an inductive bias is. The boundary between supervised learning and reinforcement learning, like the boundary between prediction and action, turns out to be conceptual rather than categorical, and the systems that matter most today thread back and forth across it. The math that carries across, the math worked out by hand across these eight chapters, is the part worth keeping. That is the end of the book.

Bibliographic Notes

The REINFORCE estimator derived in Section 8.5 is due to Williams (1992), who introduced the class of score-function gradient algorithms for connectionist reinforcement learning and proved that the log-likelihood-weighted update follows the gradient of expected reward in expectation. The reward-to-go form and the baseline argument used here are the standard modern presentation. Sutton and Barto (2018) is the depth reference for everything this chapter compresses: the Markov decision process formalism of Section 8.3, the value-based, policy-gradient, and model-based families of Section 8.4, the policy-gradient theorem, and the bias-variance reading of Monte Carlo against temporal-difference credit assignment.

The boundary-blurring systems named in the closing are documented in their own right. Silver et al. (2017) is AlphaZero, which trains its policy network by softmax cross-entropy and its value network by squared error against targets produced by Monte Carlo tree search, the clearest large-scale instance of the chapter's bridge: the supervised toolkit as the inner loop, with a search procedure rather than a human supplying the labels.

The theoretical north star of Section 8.8 is AIXI, set out in full by Hutter (2005). The text develops AIXI as Solomonoff induction joined to Bayesian sequential decision theory, the optimal-but-incomputable agent against which practical reinforcement-learning algorithms can be read as computable approximations, exactly as the language-modeling chapter read practical predictors against Solomonoff induction.

Bibliography

Alpaydin, E. and C. Kaynak (1998). *Optical Recognition of Handwritten Digits*. UCI Machine Learning Repository. `https://archive.ics.uci.edu/dataset/80`.

Baydin, Atilim Gunes et al. (2018). "Automatic Differentiation in Machine Learning: a Survey". In: *Journal of Machine Learning Research* 18.153, pp. 1–43.

Bengio, Yoshua, Réjean Ducharme, et al. (2003). "A Neural Probabilistic Language Model". In: *Journal of Machine Learning Research* 3, pp. 1137–1155.

Bengio, Yoshua, Patrice Simard, and Paolo Frasconi (1994). "Learning Long-term Dependencies with Gradient Descent is Difficult". In: *IEEE Transactions on Neural Networks* 5.2, pp. 157–166.

Bishop, Christopher M. (1994). "Mixture Density Networks". In: *Neural Computing Research Group Report NCRG/4288*. Aston University.

— (2006). *Pattern Recognition and Machine Learning*. Springer.

Cho, Kyunghyun et al. (2014). "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1724–1734.

Cybenko, George (1989). "Approximation by Superpositions of a Sigmoidal Function". In: *Mathematics of Control, Signals, and Systems* 2.4, pp. 303–314.

Elhage, Nelson, Neel Nanda, Catherine Olsson, et al. (2021). "A Mathematical Framework for Transformer Circuits". In: *Transformer Circuits Thread*. Anthropic.

Elman, Jeffrey L. (1990). "Finding Structure in Time". In: *Cognitive Science* 14.2, pp. 179–211.

Fukushima, Kunihiko (1980). "Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position". In: *Biological Cybernetics* 36.4, pp. 193–202.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press.

Hochreiter, Sepp (1991). "Untersuchungen zu dynamischen neuronalen Netzen". Diploma thesis; the early statement of the vanishing-gradient problem. MA thesis. Technische Universität München.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long Short-Term Memory". In: *Neural Computation* 9.8, pp. 1735–1780.

Hornik, Kurt, Maxwell Stinchcombe, and Halbert White (1989). "Multilayer Feedforward Networks are Universal Approximators". In: *Neural Networks* 2.5, pp. 359–366.

Hutter, Marcus (2005). *Universal Artificial Intelligence: Sequential Decisions Based on Algorithmic Probability*. Springer.

LeCun, Yann, Bernhard Boser, et al. (1989). "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1.4, pp. 541–551.

LeCun, Yann, Léon Bottou, et al. (1998). "Gradient-based Learning Applied to Document Recognition". In: *Proceedings of the IEEE* 86.11, pp. 2278–2324.

MacKay, David J. C. (2003). *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press.

McCullagh, Peter and John A. Nelder (1989). *Generalized Linear Models*. 2nd. Chapman and Hall.

Meng, Kevin et al. (2022). "Locating and Editing Factual Associations in GPT". In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 35.

Minsky, Marvin and Seymour Papert (1969). *Perceptrons: An Introduction to Computational Geometry*. MIT Press.

Nelder, John A. and Robert W. M. Wedderburn (1972). "Generalized Linear Models". In: *Journal of the Royal Statistical Society, Series A* 135.3, pp. 370–384.

Nix, David A. and Andreas S. Weigend (1994). "Estimating the Mean and Variance of the Target Probability Distribution". In: *Proceedings of the International Conference on Neural Networks (ICNN)*. Vol. 1, pp. 55–60.

Olsson, Catherine, Nelson Elhage, Neel Nanda, et al. (2022). "In-context Learning and Induction Heads". In: *Transformer Circuits Thread*. Anthropic.

Power, Alethea et al. (2022). "Grokking: Generalization Beyond Overfitting on Small Algorithmic Datasets". In: *arXiv:2201.02177*.

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams (1986). "Learning Representations by Back-Propagating Errors". In: *Nature* 323, pp. 533–536.

Shannon, Claude E. (1951). "Prediction and Entropy of Printed English". In: *Bell System Technical Journal* 30.1, pp. 50–64.

Silver, David et al. (2017). "Mastering the Game of Go without Human Knowledge". In: *Nature* 550.7676, pp. 354–359.

Sutton, Richard S. and Andrew G. Barto (2018). *Reinforcement Learning: An Introduction*. 2nd. MIT Press.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, et al. (2017). "Attention Is All You Need". In: *Advances in Neural Information Processing Systems*. Vol. 30.

Wang, Kevin et al. (2022). "Interpretability in the Wild: a Circuit for Indirect Object Identification in GPT-2 small". In: *arXiv:2211.00593*.

Werbos, Paul J. (1990). "Backpropagation Through Time: What It Does and How to Do It". In: *Proceedings of the IEEE* 78.10, pp. 1550–1560.

Williams, Ronald J. (1992). "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning". In: *Machine Learning* 8.3–4, pp. 229–256.