
Learning Nonlinear Representations (continued)

ECE 57000 — October 14, 2026

David I. Inouye

| Last time: nonlinear features made **the mismatch task separable**

- Fitting the training objective and generalizing to new examples are different goals.
- One linear score cannot express XOR; two ReLU units can construct useful features.
- A hidden layer computes $\mathbf{h} = \phi(W\mathbf{x} + \mathbf{b})$.

Today: revisit that toy construction, then build a multilayer perceptron and learn how its parameters receive gradients.

| Learned features expand **what the model can represent**

Arc 5 • Section 1 of 4

1. **Representation: build nonlinear features** ← **this section**
2. Training: learn the parameters
3. Image structure: design convolutional layers
4. Evaluation: test the architectural assumptions

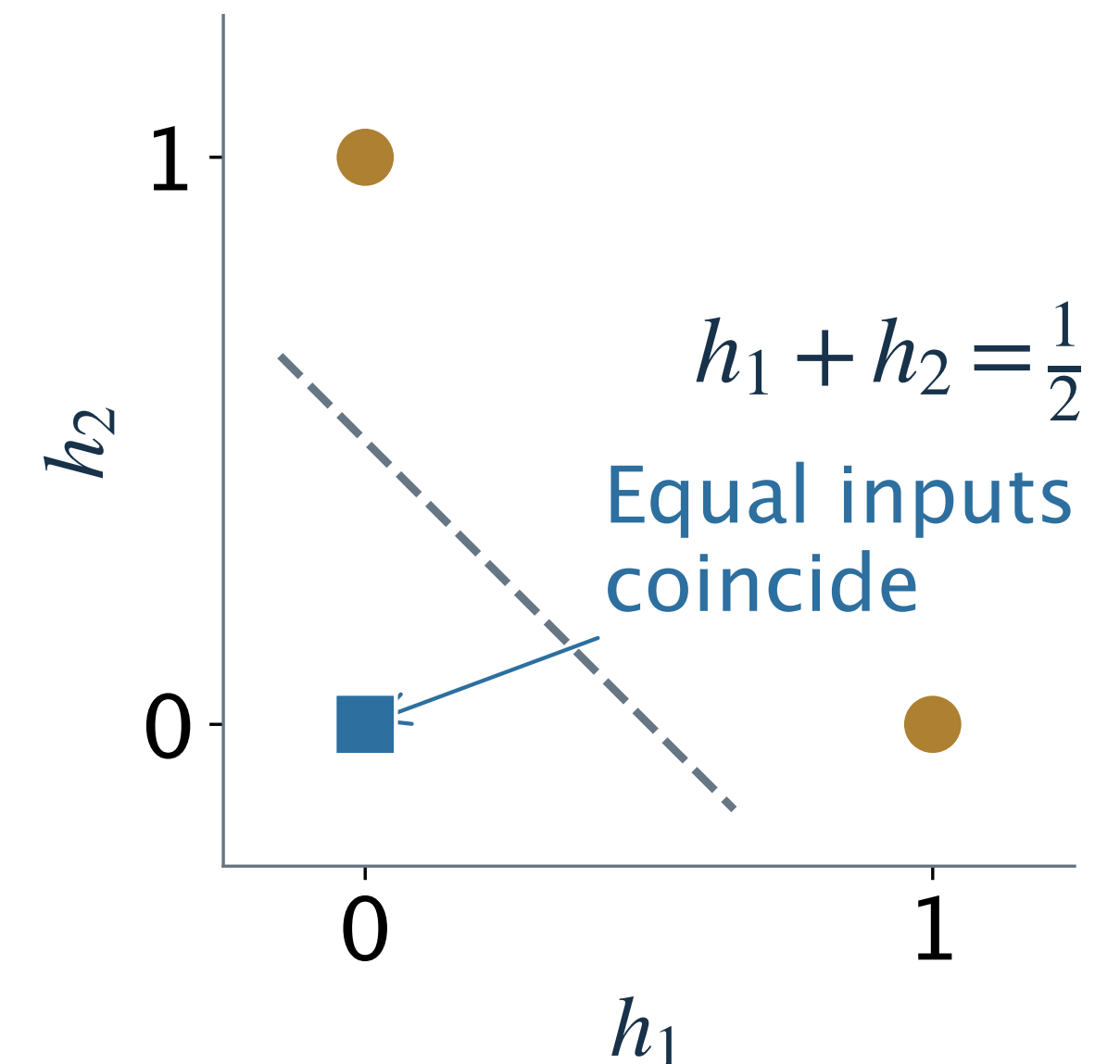
Guiding question: Can learned features improve product-image classification?

Product images → XOR and ReLU → multilayer perceptrons → held-out evidence.

| Two directional responses create a useful hidden representation

$$h_1 = \max(0, x_1 - x_2), \quad h_2 = \max(0, x_2 - x_1).$$

Input x	Hidden h	Class y
$[0, 0]^T$	$[0, 0]^T$	0
$[0, 1]^T$	$[0, 1]^T$	1
$[1, 0]^T$	$[1, 0]^T$	1
$[1, 1]^T$	$[0, 0]^T$	0



$$h_1 + h_2 = |x_1 - x_2|, \quad \hat{y} = \mathbb{1}\{h_1 + h_2 > \frac{1}{2}\}.$$

A linear decision in the hidden features solves the nonlinear input-space task.

| A hidden layer collects **several computed features into a vector**

$$h = [h_1, \dots, h_m]^T \in \mathbb{R}^m.$$

Input

x_1

x_2

Hidden representation

$$h_1 = \max(0, x_1 - x_2)$$

$$h_2 = \max(0, x_2 - x_1)$$

Decision

$$h_1 + h_2 > \frac{1}{2}$$

Each hidden unit has its own weights and bias; all see the same input.

The output layer combines the resulting features.

| Stack the neuron weights to obtain **one matrix expression**

Each row of W is one neuron's $w_j^\top$:

$$\boldsymbol{h} = \phi(W\boldsymbol{x} + \boldsymbol{b}), \quad W \in \mathbb{R}^{m \times d}, \quad \boldsymbol{b} \in \mathbb{R}^m.$$

Apply ϕ separately to each feature.

For our two-region example:

$$W = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad \boldsymbol{h} = \phi \left(\begin{bmatrix} x_1 - x_2 \\ x_2 - x_1 \end{bmatrix} \right).$$

| Dimensions account for every trainable parameter

One Fashion-MNIST image: 784 pixel features $\rightarrow$ 64 hidden units $\rightarrow$ 10 class scores.

$$h = \text{ReLU}(Wx + b), \quad s = Vh + c.$$

Layer	Weight matrix	Bias vector	Trainable parameters
Hidden layer	$W : 64 \times 784$	$b : 64 \times 1$	$64 \cdot 784 + 64 = 50,240$
Output layer	$V : 10 \times 64$	$c : 10 \times 1$	$10 \cdot 64 + 10 = 650$

Total: $50,240 + 650 = 50,890$ trainable parameters.

Activations: $x \in \mathbb{R}^{784}, h \in \mathbb{R}^{64}, s, p \in \mathbb{R}^{10}$. Softmax converts scores s to probabilities p and adds **no parameters**.

The output layer maps learned features to **multiclass probabilities**

For the product images, let $\mathbf{h} \in \mathbb{R}^{64}$ and compute

$$\mathbf{s} = V\mathbf{h} + \mathbf{c}, \quad V \in \mathbb{R}^{10 \times 64}, \quad \mathbf{c} \in \mathbb{R}^{10}.$$

Reuse softmax to normalize the ten scores into class probabilities:

$$p_c = \frac{\exp(s_c)}{\sum_{j=1}^{10} \exp(s_j)}, \quad c = 1, \dots, 10.$$

For class labels y_i , fit all parameters $\boldsymbol{\theta} = (W, \mathbf{b}, V, \mathbf{c})$ jointly using **mean cross-entropy**:

$$J(\boldsymbol{\theta}) = -\frac{1}{n} \sum_i \log p_{i,y_i}.$$

| Deeper models **compose learned representations**

$$\mathbf{h}^{(0)} = \mathbf{x},$$

$$\mathbf{h}^{(\ell)} = \phi_{\ell}(\mathbf{W}_{\ell}\mathbf{h}^{(\ell-1)} + \mathbf{b}_{\ell}), \quad \ell = 1, \dots, k.$$

$$\mathbf{x} \longrightarrow \mathbf{h}^{(1)} \longrightarrow \mathbf{h}^{(2)} \longrightarrow \text{prediction}.$$

Each layer transforms features produced by the previous layer.

An MLP—multilayer perceptron—uses dense layers with nonlinear hidden activations.

| Would extra linear layers solve **the mismatch task**?

A network has several hidden layers. Replace every hidden activation with the identity; compare the two final class scores.

Can it classify all four mismatch images correctly?

A. Yes, if the hidden layers are wide enough.

B. Yes, if gradient descent runs long enough.

C. No; the difference between its two final scores is still a linear score in the input.

C. Additional linear layers cannot supply the missing interaction.

| Composing linear layers **still gives one linear layer**

A **linear layer** computes $Wx + b$ (technically *affine* when $b \neq 0$).

$$f(x) = W_2(W_1x + b_1) + b_2$$

$$= W_2W_1x + W_2b_1 + b_2$$

$$= Ax + c$$

(two linear layers)

(distribute)

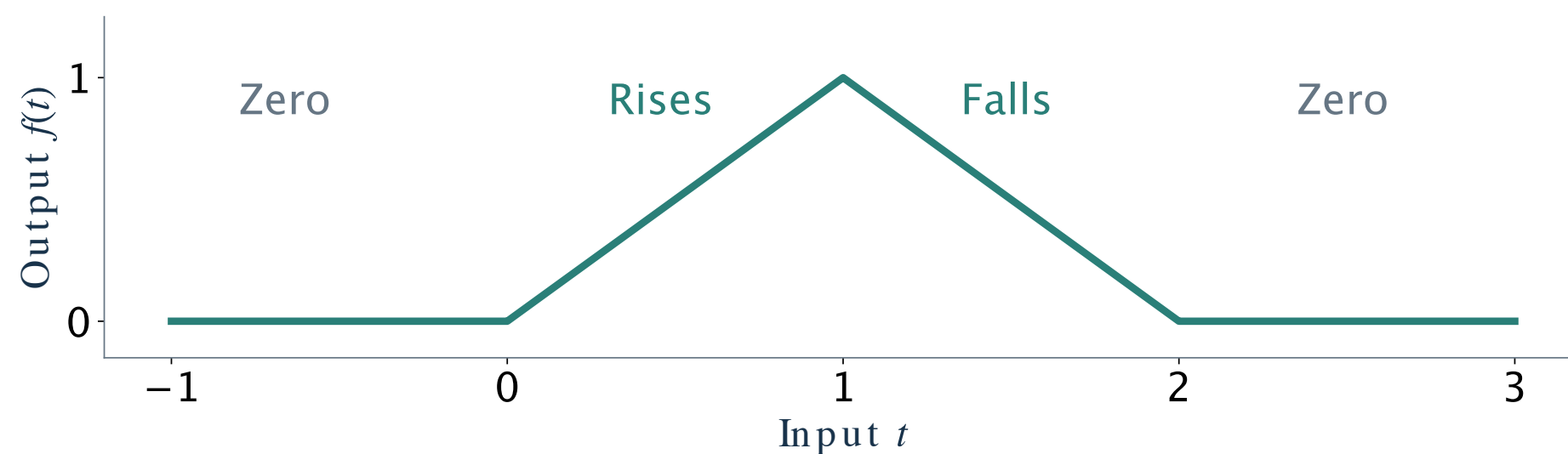
($A = W_2W_1, c = W_2b_1 + b_2$)

Repeat this argument for any number of linear layers.

Nonlinear activations let simple pieces form a new shape

$$f(t) = \phi(t) - 2\phi(t - 1) + \phi(t - 2), \quad \phi(s) = \max(0, s).$$

The three ReLU units switch on at $t = 0, 1, 2$; the output weights are $1, -2, 1$.



Input interval	Substitute the active units
$t \leq 0$	$f(t) = 0$
$0 < t \leq 1$	$f(t) = t$
$1 < t \leq 2$	$f(t) = t - 2(t - 1) = 2 - t$
$t > 2$	$f(t) = t - 2(t - 1) + (t - 2) = 0$

At $t = 1.5$: $f(t) = 1.5 - 2(0.5) + 0 = 0.5$.

Each newly active unit changes the slope: $0 \rightarrow 1 \rightarrow -1 \rightarrow 0$. Three ReLU features plus a linear output layer create a shape no single linear layer can express.

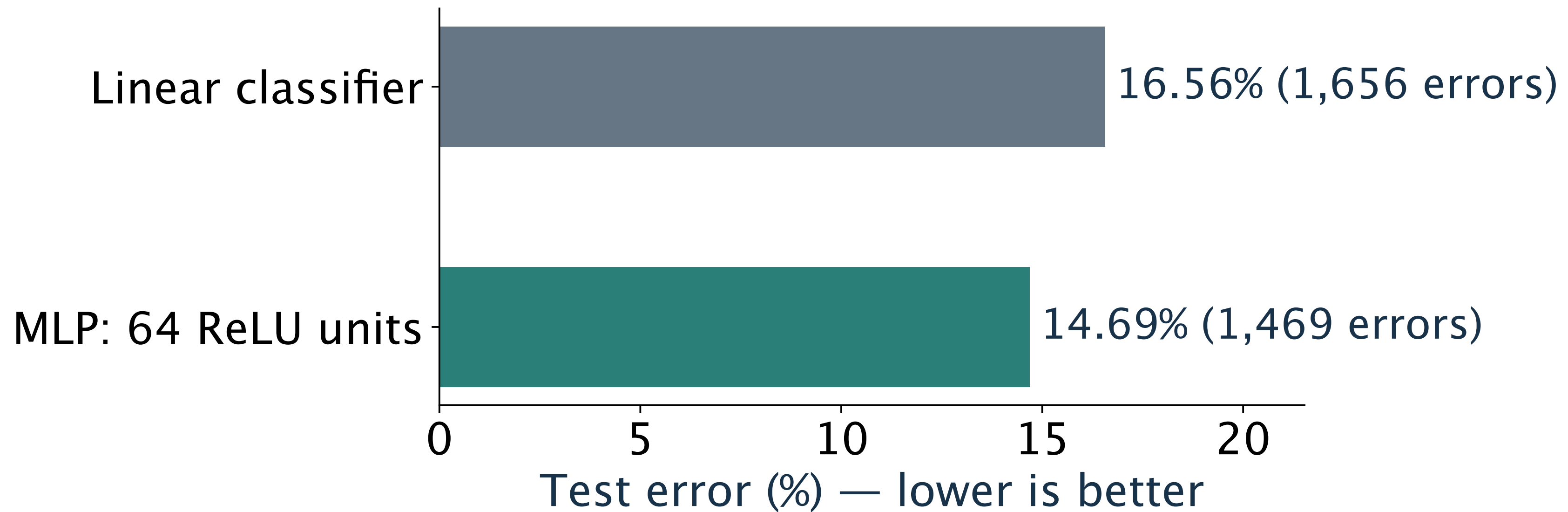
| More expressive features still need **held-out evidence**

Question	Evidence needed
Can this architecture express a useful rule?	A construction or approximation argument
Did training find useful parameters?	Training objective and optimization diagnostics
Does the fitted predictor work on new examples?	Appropriate held-out evaluation

Choose width, depth, and regularization using validation data; reserve test data for final assessment.

Our four-image construction answers only the first question.

| A small MLP makes **fewer product-image errors** in this experiment



187 fewer errors on 10,000 test images. Same inputs and split; each model selected on validation data.

| The final loss must guide every layer's parameters

Arc 5 • Section 2 of 4

1. Representation: build nonlinear features
2. **Training: learn the parameters ← this section**
3. Image structure: design convolutional layers
4. Evaluation: test the architectural assumptions

Guiding question: How can a final loss train earlier layers?

Gradient descent → atomic gradient rules → backpropagation → training diagnostics.

| We still use gradient descent—but where do the gradients come from?

$$\theta \leftarrow \theta - \eta \nabla_{\theta} J.$$

The parameters now sit inside a composition of many layers.

Backpropagation computes $\nabla_{\theta} J$ efficiently. Gradient descent (or Adam) uses those gradients to update the parameters.

Training loop: forward computation $\rightarrow$ backpropagation $\rightarrow$ parameter update.

| Backpropagation evaluates **the chain rule from the loss backward**

For $a = f(\theta)$, $b = g(a)$, and $\ell = q(b)$:

$$\frac{\partial \ell}{\partial \theta} = \frac{\partial \ell}{\partial b} \frac{\partial b}{\partial a} \frac{\partial a}{\partial \theta}.$$

Forward: $\theta \rightarrow a \rightarrow b \rightarrow \ell$; save the values needed by the derivatives.

Backward: start with $\frac{\partial \ell}{\partial \ell} = 1$, then compute

$$\frac{\partial \ell}{\partial b} = \frac{\partial q(b)}{\partial b}, \quad \frac{\partial \ell}{\partial a} = \frac{\partial \ell}{\partial b} \frac{\partial b}{\partial a}, \quad \frac{\partial \ell}{\partial \theta} = \frac{\partial \ell}{\partial a} \frac{\partial a}{\partial \theta}.$$

Same chain rule, reversed evaluation order. We need the forward output to evaluate the loss's derivative, and saved intermediate values to evaluate the local derivatives.

| A few atomic operations supply the local gradient rules

Apply the reverse chain rule to each operation: **incoming loss derivative × local derivative**.

Forward operation	Send backward to its inputs	Forward values needed
$q = a + b$	$\frac{\partial \ell}{\partial a} += \frac{\partial \ell}{\partial q}, \frac{\partial \ell}{\partial b} += \frac{\partial \ell}{\partial q}$	None
$q = ab$	$\frac{\partial \ell}{\partial a} += b \frac{\partial \ell}{\partial q}, \frac{\partial \ell}{\partial b} += a \frac{\partial \ell}{\partial q}$	a, b
$h = \text{ReLU}(z)$	$\frac{\partial \ell}{\partial z} += \mathbb{1}\{z > 0\} \frac{\partial \ell}{\partial h}$	Sign of z (or active mask)

Each rule multiplies the incoming sensitivity by a local derivative. The $+=$ means add contributions if an input is used more than once.

One forward pass supplies everything this backward pass needs

Toy scalar network: $z = wx, h = \text{ReLU}(z), s = vh, \ell = \frac{1}{2}(s - y)^2$. Set $x = 2, y = 1, w = 1, v = 1$; omit biases to keep this example small.

1. Forward: compute and save

Value	Calculation
z	$1 \cdot 2 = 2$
h	$\max(0, 2) = 2$
s	$1 \cdot 2 = 2$
ℓ	$\frac{1}{2}(2 - 1)^2 = \frac{1}{2}$

Save $h = 2$ and the **active ReLU mask**.

2. Backward: start at the loss

Sensitivity / gradient	Calculation
$\partial \ell / \partial s = s - y$	$2 - 1 = 1$
$\partial \ell / \partial v = (\partial \ell / \partial s)h$	$1 \cdot 2 = 2 \leftarrow \text{saved } h$
$\partial \ell / \partial h = (\partial \ell / \partial s)v$	$1 \cdot 1 = 1$
$\partial \ell / \partial z = (\partial \ell / \partial h)\mathbb{1}\{z > 0\}$	$1 \cdot 1 = 1 \leftarrow \text{saved mask}$

$\partial \ell / \partial w = (\partial \ell / \partial z)x$
 $1 \cdot 2 = 2$

Both weight gradients are 2. The optimizer can now update w and v .

| Libraries combine these rules **to differentiate the whole network**

1. **Forward:** record the operations and retain the values their backward rules need.
2. **Backward:** start at the scalar loss, apply local rules in reverse order, and sum contributions from shared paths.
3. **Update:** use the resulting gradient for every trainable parameter.

For our MLP, that includes W , b , V , c .

We specify the model and loss; automatic differentiation organizes the chain rule.

| Can an inactive ReLU receive a weight update from this example?

A hidden unit has $z = -1$. There is no penalty term.

What does this example contribute to the gradient of its incoming weights?

- A. Zero: the local ReLU derivative blocks this path.
- B. Nonzero: a wrong prediction always changes every weight.
- C. Undefined: ReLU has no derivative below zero.

A. Other examples may activate the unit; being inactive once does not make it permanently inactive.

| A training batch averages the example gradients

For a batch $\mathcal{B}$ of B training examples:

$$J_{\mathcal{B}}(\boldsymbol{\theta}) = \frac{1}{B} \sum_{i \in \mathcal{B}} \ell_i(\boldsymbol{\theta}), \quad \nabla J_{\mathcal{B}} = \frac{1}{B} \sum_{i \in \mathcal{B}} \nabla \ell_i.$$

One iteration: forward pass $\rightarrow$ average loss $\rightarrow$ backward pass $\rightarrow$ optimizer update.

An epoch processes the training set once. Clear old gradient accumulators before a new batch unless accumulation is intentional.

| Finite differences can **check a gradient implementation**

Hold all other parameters fixed and perturb parameter θ_j :

$$g_j^{\text{FD}} = \frac{J(\boldsymbol{\theta} + \epsilon \mathbf{e}_j) - J(\boldsymbol{\theta} - \epsilon \mathbf{e}_j)}{2\epsilon} \approx (\nabla J)_j.$$

For the scalar example, $\epsilon = 10^{-4}$ gives $g_w^{\text{FD}} \approx 2$.

Use a tiny deterministic batch, double precision, and several small ϵ values. Avoid perturbations that cross a ReLU kink.

This checks the derivative code, not the model's generalization.