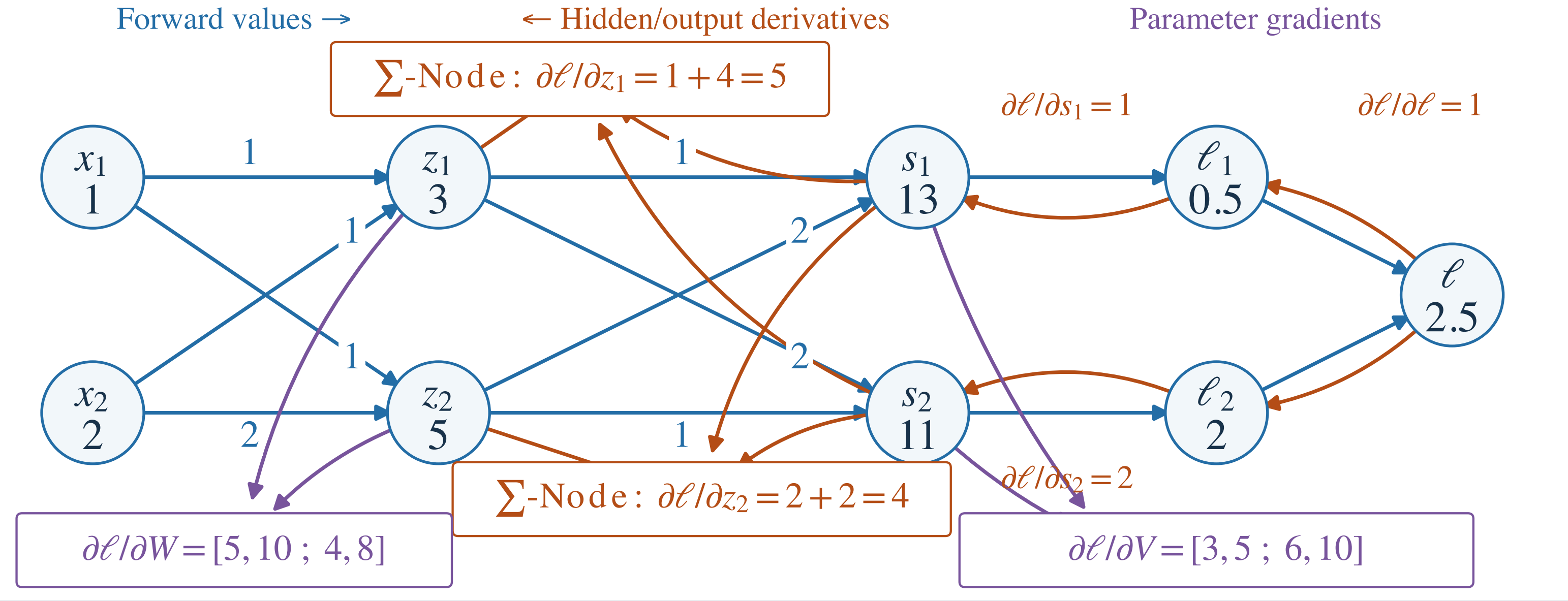

Training Neural Networks (continued)

ECE 57000 — October 16, 2026

David I. Inouye

| Backward computation sums paths at hidden features

13/13 Matrix Summary: Accumulate, Then Form Parameter Gradients



$$\frac{\partial \ell}{\partial \mathbf{z}} = \mathbf{V}^\top \frac{\partial \ell}{\partial \mathbf{s}} = [5, 4]^\top$$
$$\frac{\partial \ell}{\partial \mathbf{V}} = \begin{bmatrix} 3 & 5 \\ 6 & 10 \end{bmatrix} \qquad \frac{\partial \ell}{\partial \mathbf{W}} = \begin{bmatrix} 5 & 10 \\ 4 & 8 \end{bmatrix}$$

Matrix multiplication performs the same path sums. No gradient with respect to $\mathbf{x}$ is computed.

Backpropagation computes **all** parameter gradients efficiently

$$\text{In 1D: } \frac{\partial J}{\partial \theta} \approx \frac{J(\theta + \epsilon) - J(\theta)}{\epsilon}.$$

One objective evaluation takes **1 second**. Reuse the unperturbed value $J(\theta)$.

Finite differences: about how long for **10,000 parameters**?

- A. 2 seconds
- B. 100 seconds
- C. 10,000 seconds

C: 10,000 parameters → **about 10,000 seconds**.

One new evaluation per parameter.

Backpropagation: how long for the **same full gradient**?

Assume backward $\approx$ forward cost.

- A. 2 seconds
- B. 100 seconds
- C. 10,000 seconds

A: about 2 seconds for forward + backward.

One reverse pass reuses work for **all parameters**.

| Implementation connects **the graph to running code**

Arc 5 • Implementation in PyTorch

1. **Autograd:** inspect the scalar computation and its gradients.
2. **Training:** assemble the image model, loss, and optimizer.
3. **Diagnostics:** inspect what the training loop produces.

The preceding section explained backpropagation mathematically. Now we make each operation concrete.

| `requires_grad` identifies which inputs need derivatives

For scalar $b = f(a)$, computing $\nabla_a b$ requires marking `a` with `requires_grad=True`.

```
1 import torch
2 x = torch.tensor(2.)
3 y = torch.tensor(1.)
4 w = torch.tensor(1., requires_grad=True)
5 v = torch.tensor(1., requires_grad=True)
6 print(x.requires_grad, w.requires_grad, w.grad)
```

Output:

```
1 False True None
```

Mark the variable in the **subscript of ∇** —or the **denominator of $\partial b / \partial a$** for scalars.

Here `w` and `v` are the trainable **leaves**. Their `.grad` fields will hold the weight derivatives after backward.

| The forward pass builds values and a backward graph

```
1 print("Before computation, w.grad_fn:", w.grad_fn)
2 z = w * x
3 print("After computation, z.grad_fn:", type(z.grad_fn).__name__)
4 h = torch.relu(z)
5 s = v * h
6 r = s - y
7 loss = 0.5 * r.square()
8 print(z.item(), h.item(), s.item(), loss.item())
```

Output:

```
1 Before computation, w.grad_fn: None
2 After computation, z.grad_fn: MulBackward0
3 2.0 2.0 2.0 0.5
```

Forward values: $w, x \rightarrow z \rightarrow h$; then $v, h \rightarrow s \rightarrow r \rightarrow \ell$.

Backward graph: w is a leaf with no producing operation. Multiplying it creates z with a `.grad_fn` linking to that operation.

`requires_grad=True` enables tracking; the forward computation creates the graph.

| Backward rules need selected forward values

For our scalar example:

Operation	A value its backward rule needs	Here
$z = wx$	x to obtain $\partial \ell / \partial w$	2
$h = \text{ReLU}(z)$	Which inputs were positive	Active
$s = vh$	h for v 's derivative; v for h 's	2, 1
$\ell = \frac{1}{2}r^2$	Residual r	1

Saved forward values are not gradients. They let backward evaluate the local derivative rules.

| We can observe what autograd saves for backward

```
1 saved = []
2 def pack(t):
3     saved.append(t.detach().clone()) # snapshot for inspection
4     return t.detach()               # value kept for backward
5
6 def unpack(t):
7     return t
8
9 with torch.autograd.graph.saved_tensors_hooks(pack, unpack):
10     loss = 0.5 * (v * torch.relu(w * x) - y).square()
11 print([t.item() for t in saved])
```

Output:

```
1 [2.0, 2.0, 2.0, 1.0, 1.0]
```

These values supply the multiplication, ReLU, and squared-loss backward rules.

| One `backward()` computes both weight gradients

```
1 print("Before backward:", w.grad, v.grad)
2 loss.backward()
3 print("After backward:", w.grad.item(), v.grad.item())
4 print("Weights:", w.item(), v.item())
```

Output:

```
1 Before backward: None None
2 After backward: 2.0 2.0
3 Weights: 1.0 1.0
```

Backward starts at the scalar loss, follows the recorded graph, and adds contributions into the trainable leaves' `.grad` fields.

The weights have not changed. Ordinary backward releases saved values; recompute forward for the next batch or updated weights.

| What does `.grad` contain on the second iteration?

```
1 a = torch.tensor(1., requires_grad=True)
2 for iteration in range(2):
3     b = a.square()
4     b.backward()
5     print(iteration + 1, a.grad.item())
```

Output after iteration 1:

```
1 1 2.0
```

What value of `a.grad.item()` will iteration 2 print?

A. 2.0 B. 4.0 C. 0.0

Output after iteration 2:

```
1 2 4.0
```

Each `backward()` **adds** to the previous `.grad`: $2 + 2 = 4$. To get fresh gradients, reset them before recomputing a gradient.

| `zero_grad()` begins a new gradient accumulation

```
1 def objective():
2     return 0.5 * (v * torch.relu(w * x) - y).square()
3
4 optimizer = torch.optim.SGD([w, v], lr=0.1)
5 optimizer.zero_grad()
6 print("After reset:", w.grad)
7 loss = objective()
8 loss.backward()
9 print("After backward:", w.grad.item())
```

Output:

```
1 After reset: None
2 After backward: 2.0
```

`zero_grad()` clears the previous accumulated gradients.

It resets gradients, not parameter values. Clear before each independent batch's backward pass.

| `step()` uses the gradients to **change the weights**

```
1 print("Before: w, v =", round(w.item(), 2), round(v.item(), 2))
2 print("Before: loss =", round(loss.item(), 4))
3 optimizer.step()
4 with torch.no_grad():
5     new_loss = objective()
6 print("After:  w, v =", round(w.item(), 2), round(v.item(), 2))
7 print("After:  loss =", round(new_loss.item(), 4))
```

Output:

```
1 Before: w, v = 1.0 1.0
2 Before: loss = 0.5
3 After:  w, v = 0.8 0.8
4 After:  loss = 0.0392
```

Here $w, v : 1 \rightarrow 1 - 0.1(2) = 0.8$, and loss decreases from 0.5 to 0.0392.

`no_grad()` skips recording this evaluation; the next training forward pass records again.

| The same operations train **the product-image MLP**

Implementation • From two scalar weights to 50,890 parameters

Data batch → model → scalar loss → backward → optimizer step.

Now assemble the complete loop, including validation.

| A dataset provides examples; a loader **assembles** batches

```
1 from torch.utils.data import DataLoader, TensorDataset
2 # X_train: float32 [12000, 1, 28, 28], pixels in [0, 1]
3 # y_train: int64 [12000], class indices 0,...,9
4 train_dataset = TensorDataset(X_train, y_train)
5 train_loader = DataLoader(
6     train_dataset, batch_size=64, shuffle=True)
7 val_loader = DataLoader(
8     TensorDataset(X_val, y_val), batch_size=256)
9 xb, yb = next(iter(train_loader))
10 print(xb.shape, yb.shape)
```

Output:

```
1 torch.Size([64, 1, 28, 28]) torch.Size([64])
```

Dataset: `train_dataset[i]` returns one image–label pair; `len(train_dataset)` is the number of examples.

DataLoader: iterates over batches of those pairs, handling batch size and shuffling. Here `next(iter(train_loader))` returns 64 examples.

| `nn.Module` collects the model's trainable parameters

```
1 from torch import nn
2 torch.manual_seed(570)
3 model = nn.Sequential(
4     nn.Flatten(),
5     nn.Linear(784, 64),
6     nn.ReLU(),
7     nn.Linear(64, 10))
8 print(sum(p.numel() for p in model.parameters()))
```

Output:

```
1 50890
```

Batch shapes: $[B, 1, 28, 28] \rightarrow [B, 784] \rightarrow [B, 64] \rightarrow [B, 10]$.

`model.parameters()` collects the trainable weights and biases, already marked `requires_grad=True`.

| Cross-entropy consumes **scores** before softmax

```
1 loss_fn = nn.CrossEntropyLoss()
2 logits = model(xb)          # [B, 10] unnormalized scores
3 loss = loss_fn(logits, yb)
4 print(loss.shape)
```

Output:

```
1 torch.Size([])
```

The loss combines log-softmax with the true class's negative log probability, then averages over this batch.

Pass logits directly. Use `logits.argmax(dim=1)` to choose classes; softmax is only needed when displaying probabilities.

| One batch needs **four distinct training operations**

```
1 optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
2 model.train()                # set training mode only
3 optimizer.zero_grad(set_to_none=True) # clear
4 loss = loss_fn(model(xb), yb)      # forward
5 loss.backward()                  # differentiate
6 optimizer.step()                # update
```

`model.train()` only sets **training mode** for layers such as dropout and batch normalization. It does not compute gradients or update weights.

Operation	Changes
<code>zero_grad()</code>	Stored parameter gradients
Forward and <code>backward()</code>	Activations, graph, then gradients
<code>step()</code>	Parameter values and optimizer state

| An epoch repeats the update for every training batch

```
1 def train_one_epoch(model, loader, loss_fn, optimizer):
2     model.train() # set training mode only
3     total_loss, count = 0.0, 0
4     for xb, yb in loader:
5         optimizer.zero_grad(set_to_none=True)
6         loss = loss_fn(model(xb), yb)
7         loss.backward()
8         optimizer.step()
9         total_loss += loss.item() * len(yb)
10        count += len(yb)
11    return total_loss / count
```

Multiply each batch's mean loss by its size so a short final batch receives the correct weight.

This records the losses observed during training, as the weights change.

Validation measures predictions **without updating the model**

```
1 @torch.no_grad()
2 def evaluate(model, loader, loss_fn):
3     model.eval() # set evaluation mode only
4     total_loss, correct, count = 0.0, 0, 0
5     for xb, yb in loader:
6         logits = model(xb)
7         total_loss += loss_fn(logits, yb).item() * len(yb)
8         correct += (logits.argmax(1) == yb).sum().item()
9         count += len(yb)
10    return total_loss / count, correct / count
```

`train()` / `eval()` only select **layer behavior**—important for dropout and batch normalization. Neither runs training or evaluation.

`no_grad()` disables graph recording. **Mode and gradient tracking are separate.**

| The complete loop joins **learning** and **held-out evaluation**

```
1 optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
2 for epoch in range(5):
3     train_loss = train_one_epoch(
4         model, train_loader, loss_fn, optimizer)
5     val_loss, val_acc = evaluate(model, val_loader, loss_fn)
6     print(f"{epoch + 1}: train={train_loss:.3f}, "
7           f"val={val_loss:.3f}, accuracy={val_acc:.1%}")
```

Output:

```
1 1: train=0.843, val=0.602, accuracy=79.6%
2 2: train=0.538, val=0.525, accuracy=82.2%
3 3: train=0.489, val=0.498, accuracy=83.1%
4 4: train=0.448, val=0.482, accuracy=83.6%
5 5: train=0.420, val=0.464, accuracy=84.1%
```

Model → **loss** → **gradient** → **update** → **validation**. The functions above now form one complete training loop.

Five epochs are a demonstration budget; validation guides later choices. Keep the test set untouched.

| detach() keeps the values but cuts the gradient path

```
1 u = torch.tensor(1., requires_grad=True)
2 q = torch.tensor(1., requires_grad=True)
3 hidden = torch.relu(u * 2.).detach()
4 loss = 0.5 * (q * hidden - 1.).square()
5 loss.backward()
6 print(hidden.item(), hidden.requires_grad)
7 print(u.grad, q.grad.item())
```

Output:

```
1 2.0 False
2 None 2.0
```

The detached tensor has the **same values**, treated as constants for this backward pass. The loss trains **q**, but not **u** through this path.

Useful: train a new prediction layer on fixed features, or compute diagnostics from predictions without extending their training graph.

| Store diagnostic values without keeping their training graphs

```
1 loss_history, score_history = [], []
2 logits = model(xb)
3 loss = loss_fn(logits, yb)
4 loss_history.append(loss.item())
5 score_history.append(logits.detach().cpu().clone())
6 print(type(loss_history[-1]).__name__)
7 print(score_history[-1].requires_grad)
8 loss.backward()
```

Output:

```
1 float
2 False
```

Use `.item()` for a scalar loss; store detached scores for later plots or evaluation metrics. The original `loss` still supports backward.

Storing `loss` or `logits` directly keeps references to their graphs. **Logging should keep the numbers, not their training history.**

| Initialization is **part of the training recipe**

- **PyTorch provides useful defaults:** standard layers initialize their parameters when constructed; `nn.Linear` scales its initial values with the input width.
- **Match the method you reproduce:** follow a paper's specified initialization, including its scale, distribution, and bias settings. Defaults may differ.
- **Initialization can materially change training and the final model.** Record the strategy and random seed; use multiple seeds when comparing results.

`torch.nn.init` provides strategies such as **Xavier** and **Kaiming** initialization. Choose for the architecture and activation; do not replace a paper's recipe blindly.