STAT 4830: Numerical optimization for data science and ML

Lecture 4: Beyond Least Squares

From Manual to Automatic Differentiation

Professor Damek Davis

The Problem: Manual Gradient Computation

Consider computing this gradient by hand:

$$f(w) = \frac{1}{2}\| \tanh(W_2 \mathrm{ReLU}(W_1 x + b_1) + b_2) - y\|^2$$

Challenges:

- Complex chain rule applications
- Error-prone derivations
- Time-consuming process
- Limited to simple functions

The Solution: Automatic Differentiation

PyTorch provides:

```
# Define complex function
def f(x, W1, b1, W2, b2):
    h = torch.relu(W1 @ x + b1)
    return 0.5 * torch.sum(
        (torch.tanh(W2 @ h + b2) - y)**2
    )

# Get gradient automatically
f.backward()
```

Key benefits:

1. Automatic gradient computation
2. Handles any differentiable function
3. Memory efficient implementation
4. Scales to large problems

Three Key Ideas

1. **Computational Graph**
2. **Reverse-Mode Differentiation**
3. **Memory-Efficient Implementation**

Outline

1. Computing Gradients

`Function → Graph → Gradient`

2. Gradient Descent

`Gradient → Update → Repeat`

3. Neural Networks

`Features → Layers → Loss`

A Simple Example: Polynomial Function

Let's start with a one-dimensional function:

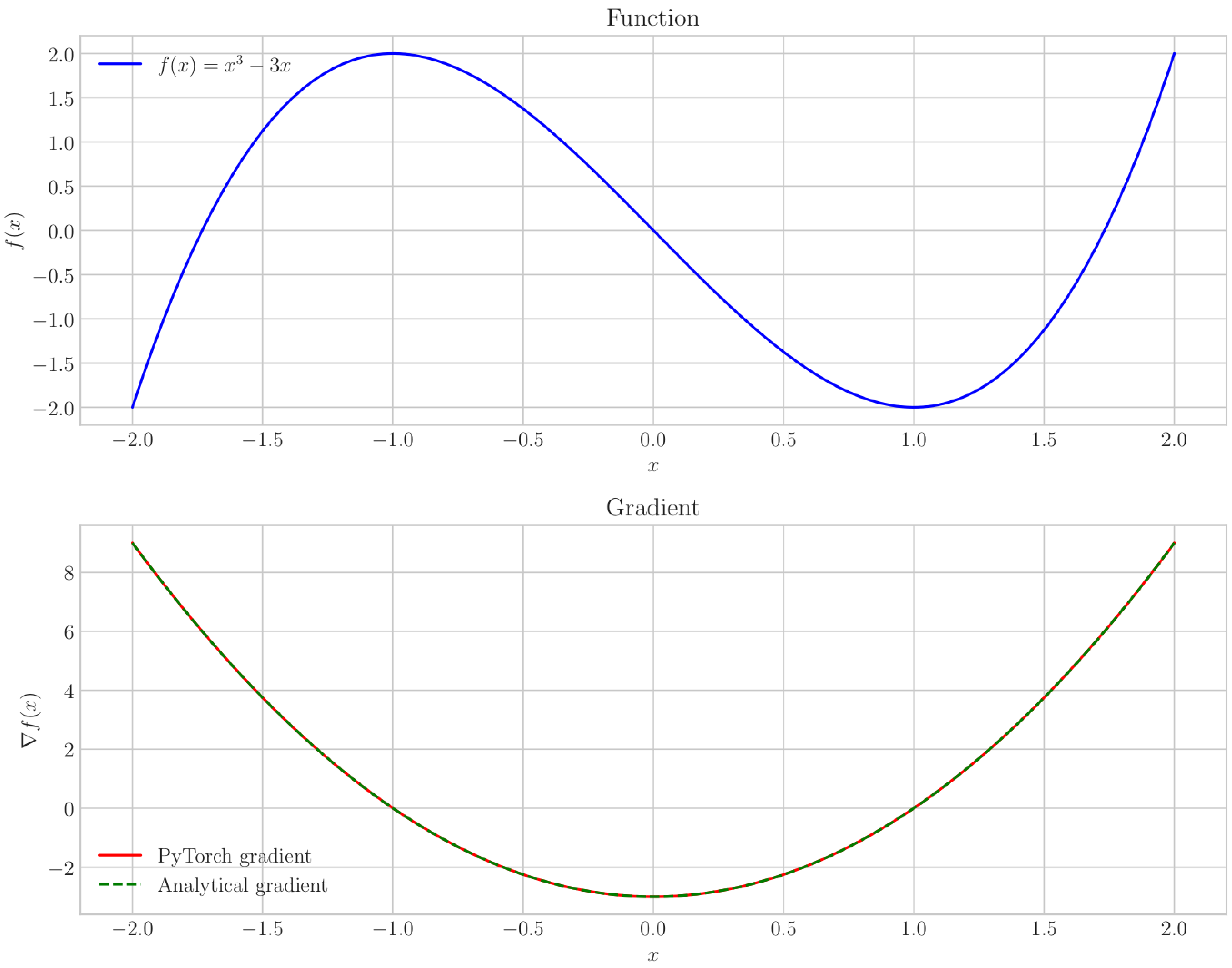
$$f(x) = x^3 - 3x$$

Manual gradient computation:

$$\frac{d}{dx} f(x) = 3x^2 - 3$$

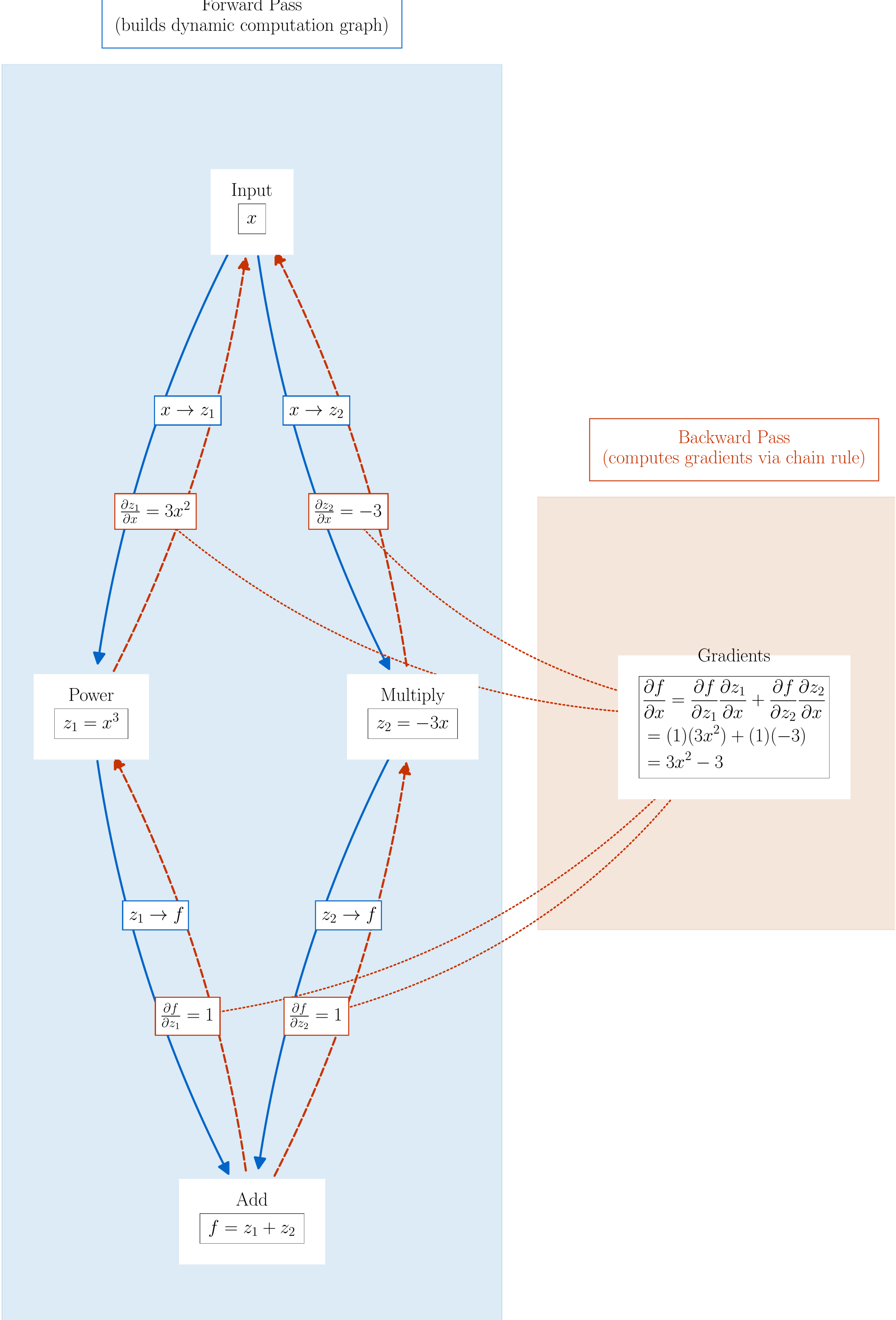
PyTorch automates this:

```
x = torch.tensor([1.0], requires_grad=True) # Gradient Tracking
y = x**3 - 3*x # Forward Pass
y.backward() # Backward Pass
print(f"f'(1) = {x.grad}") # Gradient Access
```


Function
$f(x) = x^3 - 3x$
$f(x)$
x
Gradient
PyTorch gradient
Analytical gradient
$\nabla f(x)$
x

How does PyTorch do this?

- **Forward pass:** When you evaluate a function, PyTorch computes a *computational graph* that records all operations like addition, multiplication, powers, etc.
- **Backward pass:** PyTorch traverses the graph in reverse order to compute the gradient, using what is essentially an efficient implementation of the chain rule.


Forward Pass
(builds dynamic computation graph)
Input
x
$x \to z_1$
$x \to z_2$
$\frac{\partial z_1}{\partial x} = 3x^2$
$\frac{\partial z_2}{\partial x} = -3$
Power
$z_1 = x^3$
Multiply
$z_2 = -3x$
$z_1 \to f$
$z_2 \to f$
$\frac{\partial f}{\partial z_1} = 1$
$\frac{\partial f}{\partial z_2} = 1$
Add
$f = z_1 + z_2$
Backward Pass
(computes gradients via chain rule)
Gradients
$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial z_1}\frac{\partial z_1}{\partial x} + \frac{\partial f}{\partial z_2}\frac{\partial z_2}{\partial x}$
$= (1)(3x^2) + (1)(-3)$
$= 3x^2 - 3$

Building the Computational Graph

Each node in the graph:

- Stores output value from forward pass
- Contains function for local gradients
- Maintains references to inputs

The graph structure:

- Records operations
- Stores values
- Enables gradient flow

For $f(x) = x^3 - 3x$, we build:

1. Input node storing x
2. Power node computing $z_1 = x^3$
3. Multiply node computing $z_2 = -3x$
4. Add node forming $f = z_1 + z_2$

Computing Gradients: The Process

Starting State:

- Initialize $\frac{\partial f}{\partial f} = 1$ at output
- All other gradients start at 0

Algorithm:

1. Process nodes in reverse order
2. Compute local gradients
3. Multiply by incoming gradient
4. Add to input gradients

Key Features:

- Reverse topological sort
- Chain rule at each step
- Gradient accumulation
- Memory efficient

Gradient Flow: Step by Step

Output Node ($f = z_1 + z_2$):

- $\frac{\partial f}{\partial f} = 1$
- $\frac{\partial f}{\partial z_1} = 1$, $\frac{\partial f}{\partial z_2} = 1$
- Propagate to both input nodes

Power Node ($z_1 = x^3$):

- Incoming gradient: 1
- Local gradient: $\frac{\partial z_1}{\partial x} = 3x^2$
- Contribute: $\frac{\partial f}{\partial x} += 3x^2$

Multiply Node ($z_2 = -3x$):

- Incoming gradient: 1
- Local gradient: $\frac{\partial z_2}{\partial x} = -3$
- Contribute: $\frac{\partial f}{\partial x} += -3$

Input Node (x):

Building a Computational Graph

Let's see how PyTorch builds a graph for:

$$f(x) = x^3 - 3x$$

Step 1: Create input node

```
x = torch.tensor([1.0],
                 requires_grad=True)
```

Key properties:

- Tracks gradients
- Stores value
- Records operations

Building a Computational Graph

Step 2: Power operation $z_1 = x^3$

```
z1 = x**3
```

Graph grows:

- New operation node
- Stores intermediate value
- Records connection to input

Building a Computational Graph

Step 3: Linear term $z_2 = -3x$

```
z2 = -3 * x
```

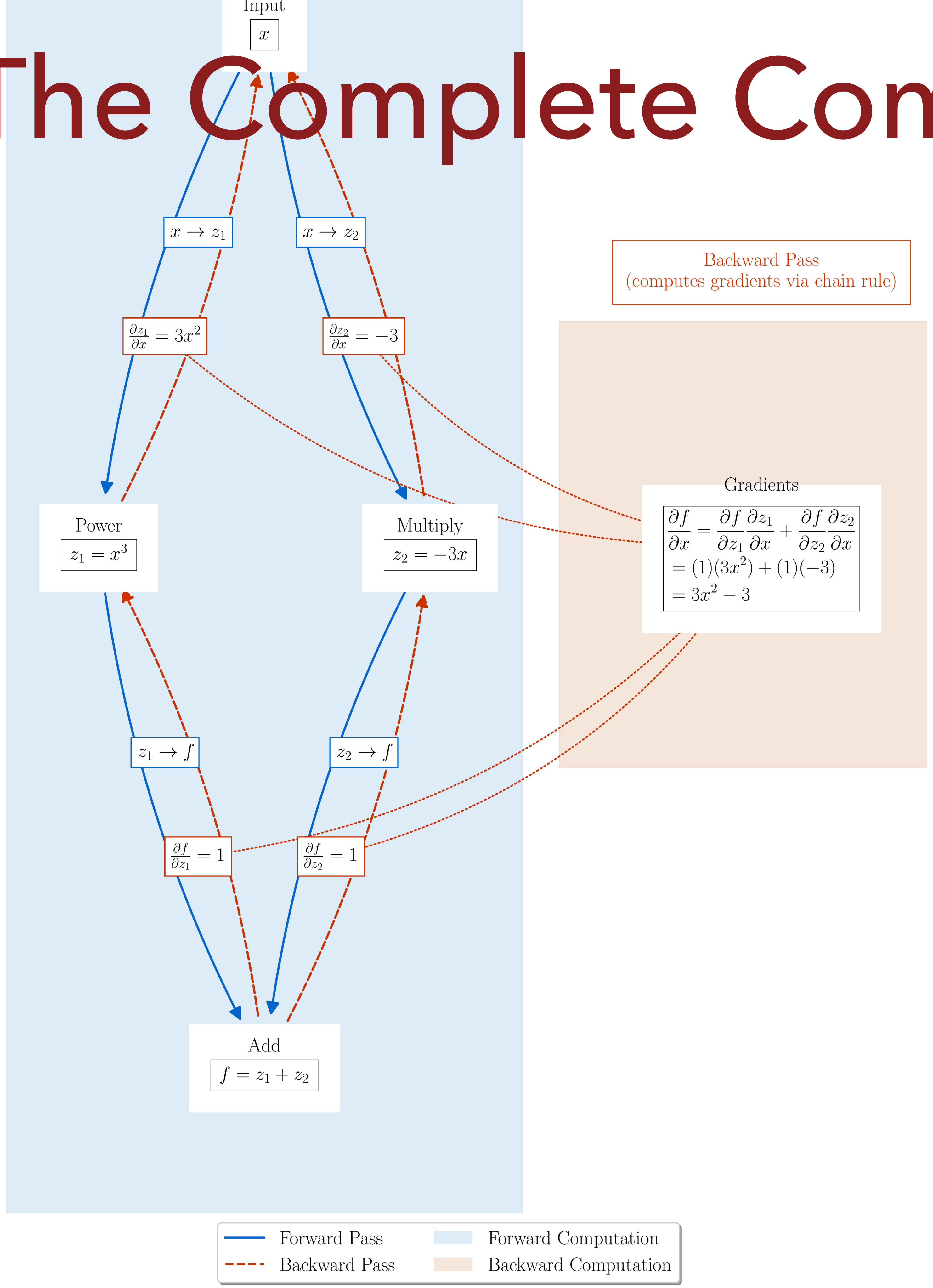
Note:

- Reuses input node
- Creates new operation
- Stores scalar multiplier

Multiple paths from x:

- Through power node
- Through multiply node
- Will need accumulation

The Complete Computational Graph



Forward Pass: Computing Values

For input $x = 1$:

1. Power node:

$$z_1 = 1^3 = 1$$

2. Multiply node:

$$z_2 = -3(1) = -3$$

3. Add node:

$$f = 1 + (-3) = -2$$

```python
# Forward computation
x = torch.tensor([1.0],
                 requires_grad=True)
z1 = x**3
z2 = -3 * x
f = z1 + z2

print(f"z1: {z1.item()}")  # 1.0
print(f"z2: {z2.item()}")  # -3.0
print(f"f: {f.item()}")    # -2.0
```

Backward Pass: Computing Gradients

Starting from output:

1. Initialize $\frac{\partial f}{\partial f} = 1$
2. Flow through power path:

$$\frac{\partial f}{\partial x} += 3x^2$$

3. Flow through multiply path:

$$\frac{\partial f}{\partial x} += -3$$

Gradient accumulation:

- Sum contributions
- Multiple paths
- Chain rule at each step

The Chain Rule: A Visual Guide

For a chain of operations:

$$x \xrightarrow{g} z \xrightarrow{h} y$$

The chain rule states:

$$\frac{dy}{dx} = \frac{dy}{dz} \cdot \frac{dz}{dx}$$

Gradient flows backward:

1. Start at output
2. Multiply derivatives
3. Follow paths back

Chain Rule in PyTorch

```python
# Forward pass
z = g(x)  # First function
y = h(z)  # Second function

# Backward pass
dy_dz = h.grad_fn(z)     # Local grad
dz_dx = g.grad_fn(x)     # Local grad
dy_dx = dy_dz * dz_dx    # Chain rule
```

Each node stores:

1. Forward function
2. Gradient function
3. Input references

PyTorch handles:

- Function composition
- Gradient computation
- Memory management

Two Implementation Approaches

1. Using `backward()`

```python
# Create graph
x.requires_grad = True
z = g(x)
y = h(z)

# Compute gradients
y.backward()
grad = x.grad  # Stored in tensor
```

Best for:

- Training loops
- Multiple gradients
- Memory efficiency

2. Using `autograd.grad()`

```python
# Create graph
x.requires_grad = True
z = g(x)
y = h(z)

# Direct computation
grad = torch.autograd.grad(y, x)[0]
```

Best for:

- One-off gradients
- Direct access
- Higher derivatives

Memory Management: Theory vs Practice

Manual gradient:

```
# Forms huge matrices
XtX = X.T @ X  # O(p²) memory
grad = XtX @ w  # Matrix-vector
```

Problems:

- Excessive memory use
- Poor cache utilization
- Limited scalability

PyTorch gradient:

```
# Matrix-vector only
Xw = X @ w      # O(p) memory
grad = X.T @ Xw  # Matrix-vector
```

Benefits:

- Minimal memory use
- Cache-friendly
- Scales to large problems

Best Practices for Memory

During Training

```
optimizer.zero_grad()  # Clear gradients
loss.backward()        # Compute gradients
optimizer.step()       # Update weights
```

2.

During Evaluation

```
with torch.no_grad():  # No gradients needed
    model.eval()       # Evaluation mode
    predictions = model(data)
```

3.

Memory Management

From Simple to Complex: Least Squares

Manual gradient:

$$\nabla f = X^\top (Xw - y)$$

Requires:

- Matrix formation
- Careful derivation
- Memory allocation

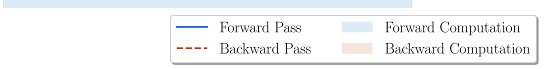
PyTorch gradient:

```
pred = X @ w
loss = 0.5*((pred - y)**2).sum()
loss.backward()
grad = w.grad
```

Benefits:

- Automatic computation
- Memory efficient
- Scales naturally

- Output: $z_1 \in \mathbb{R}^n$



The Least Squares Graph: Step by Step

2. Subtract:

$$\mathbf{z}_2 = \mathbf{z}_1 - \mathbf{y}$$

   - Input: $z_1, y \in \mathbb{R}^n$
   - Output: $z_2 \in \mathbb{R}^n$

3.

Gradient Flow in Least Squares

Total derivatives:

1. $\frac{\partial f}{\partial z_3} = \frac{1}{2}$
2. $\frac{\partial z_3}{\partial \mathbf{z}_2} = 2\mathbf{z}_2^\top$
3. $\frac{\partial \mathbf{z}_2}{\partial \mathbf{w}} = \mathbf{X}$

Chain rule:

$$\frac{\partial f}{\partial \mathbf{w}} = \frac{1}{2} \cdot 2\mathbf{z}_2^\top \cdot \mathbf{X}$$

Key insights:

1. Row vector gradients
2. Matrix-vector products
3. No matrix formation
4. Memory efficient

Final gradient:

$$\nabla f = \mathbf{X}^\top (\mathbf{X}\mathbf{w} - \mathbf{y})$$

Building Neural Networks: The Architecture

Layer composition:

```
Input → Linear₁ → Tanh → Linear₂ → Output
ℝᵈ      ℝʰˣᵈ      ℝʰ     ℝ¹ˣʰ      [0,1]
```

Each layer:

1. Linear transform
2. Nonlinear activation
3. Gradient tracking

PyTorch handles:

- Parameter management
- Forward computation
- Backward gradients

Neural Network Implementation

```python
    def forward(self, x):
        # Hidden features
        h = torch.tanh(self.linear1(x))
        # Output probability
        return torch.sigmoid(
            self.linear2(h)
        )
```

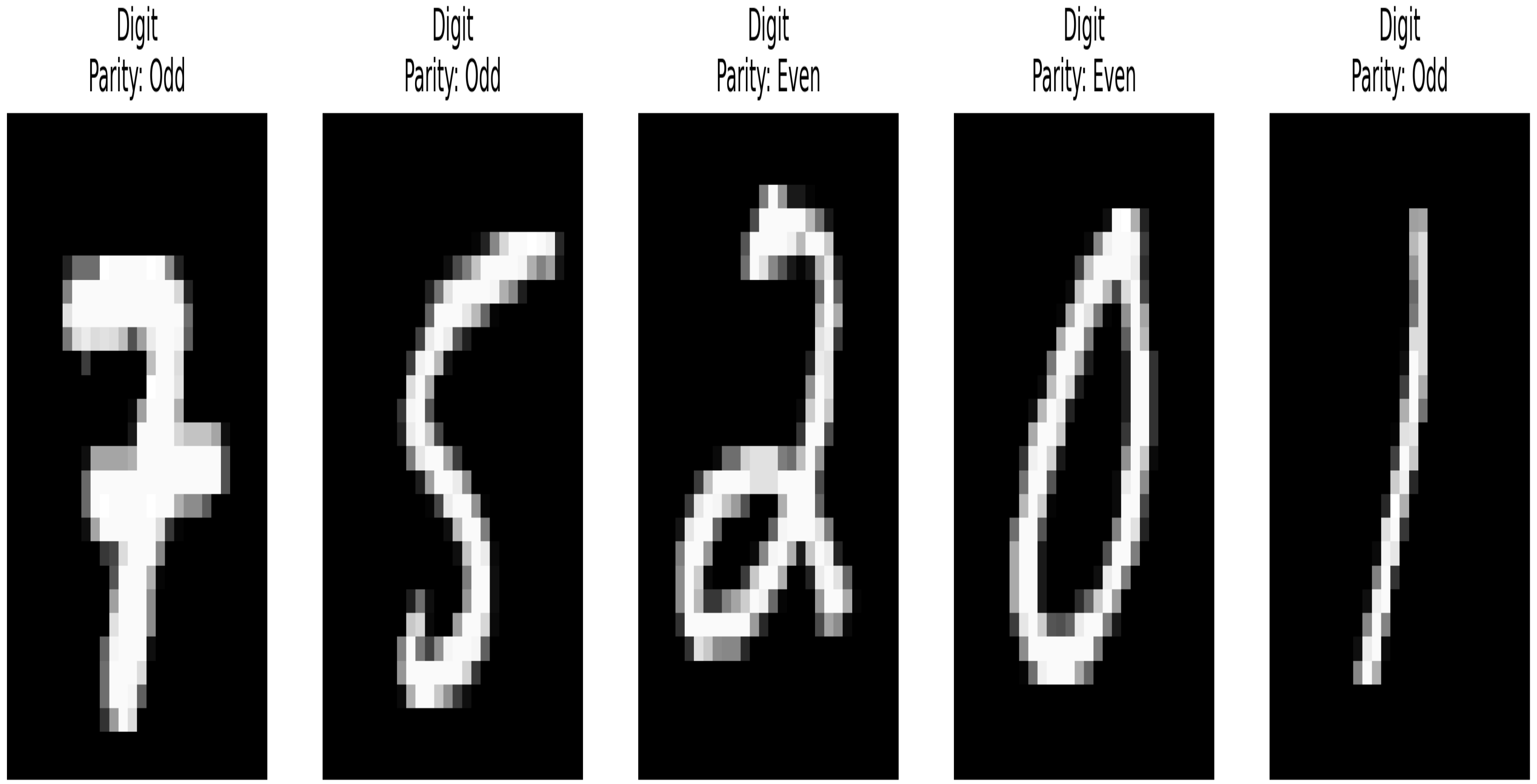
- Memory optimization

2. **Layer Organization**

   - Modular design
   - Easy composition
   - Clear data flow

3.

MNIST Classification: The Task



Dataset:

- 60,000 training images
- 10,000 test images
- 28×28 pixels each
- Binary labels (odd/even)

Preprocessing:

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(
        (0.1307,), (0.3081,)
    )
])

# Load data
train_dataset = datasets.MNIST(
    './data',
    train=True,
    transform=transform
)
```

Model Comparison: Architecture

Logistic Regression:

```
class Logistic(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(784, 1)

    def forward(self, x):
        # Single linear layer
        return torch.sigmoid(
            self.linear(x.view(-1, 784))
        )
```

Neural Network:

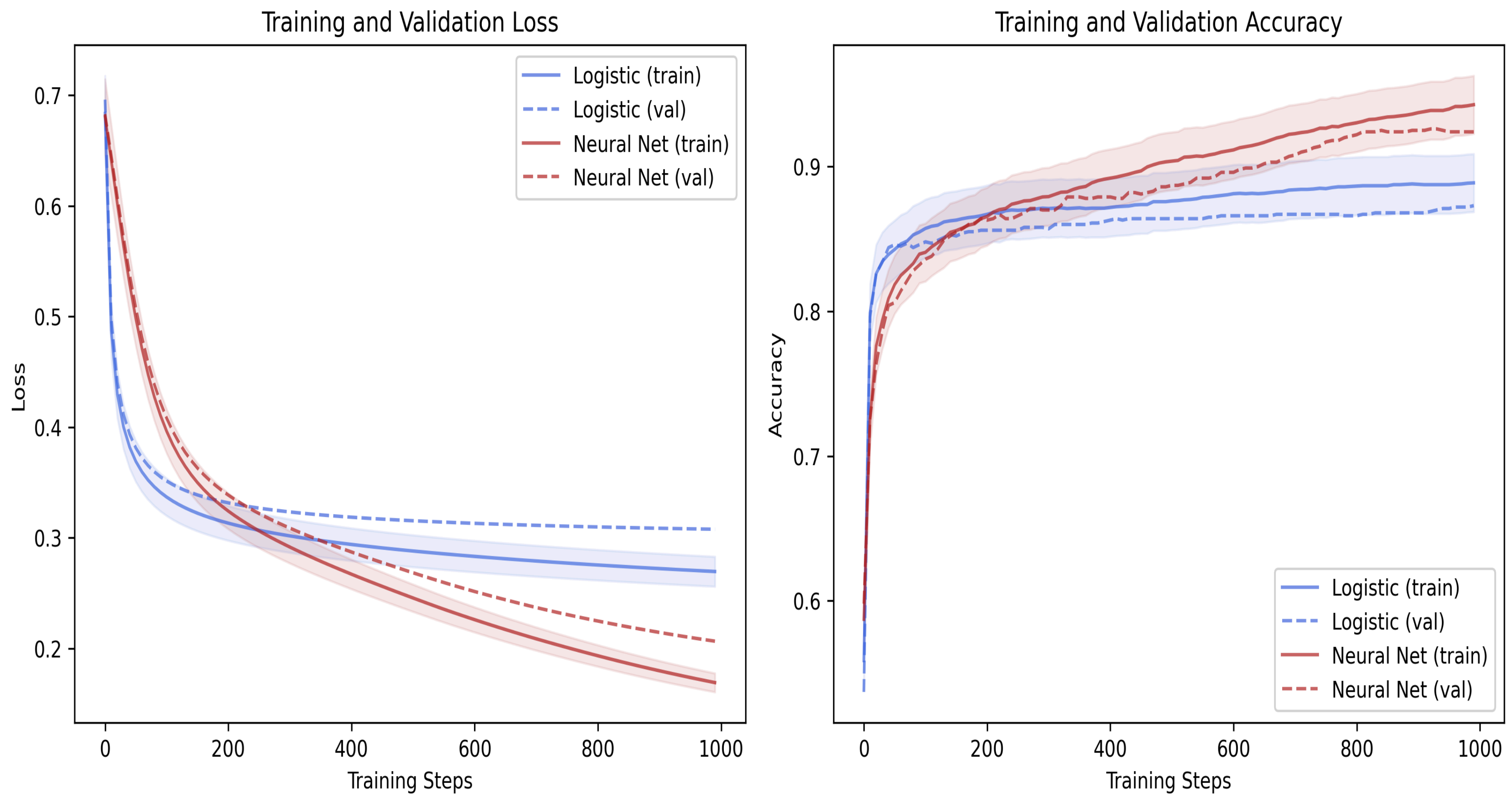
```
class SimpleNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 32)
        self.fc2 = nn.Linear(32, 1)

    def forward(self, x):
        # Hidden layer with ReLU
        h = torch.relu(
            self.fc1(x.view(-1, 784))
        )
        # Output layer
        return torch.sigmoid(self.fc2(h))
```

Training Process: Step by Step

```
def train_model(model, train_loader, optimizer, epochs=5):
    criterion = nn.BCELoss()
    for epoch in range(epochs):
        for batch_idx, (data, target) in enumerate(train_loader):
            # 1. Zero gradients
            optimizer.zero_grad()

            # 2. Forward pass
            output = model(data)
            loss = criterion(output, target.float())

            # 3. Backward pass
            loss.backward()

            # 4. Update weights
            optimizer.step()

            # 5. Log progress
```

PyTorch handles all gradient computation automatically.

Results Analysis



Training progress:

- Faster neural net learning
- Higher final accuracy
- Better generalization

Final Results:

- Logistic: 87.30% accuracy
- Neural Net: 92.40% accuracy

Key differences:

1. Feature learning
2. Nonlinear boundary
3. Better capacity

◦ Scales to complex networks

Key Takeaways

2.

Memory Efficiency

- Never forms large matrices
- Uses matrix-vector products
- Enables large-scale optimization
- Scales to deep networks

2.

Next Steps

- Stochastic gradients
- Adaptive methods
- Second-order techniques

2. **Deep Learning**
   - Complex architectures
   - Custom loss functions
   - Training strategies

Questions?

- Course website: https://damek.github.io/STAT-4830/
- Office hours: Listed on course website
- Email: damek@wharton.upenn.edu
- Discord: Check email for invite