

# 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:

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

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:

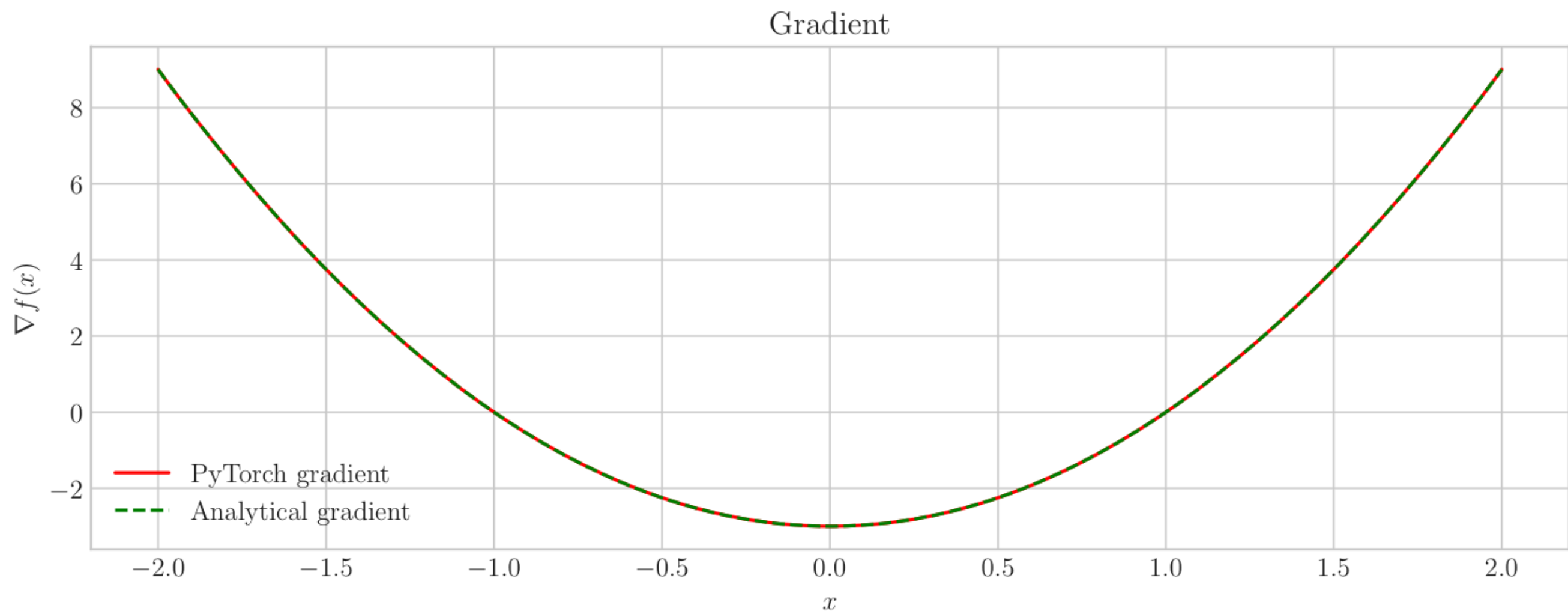
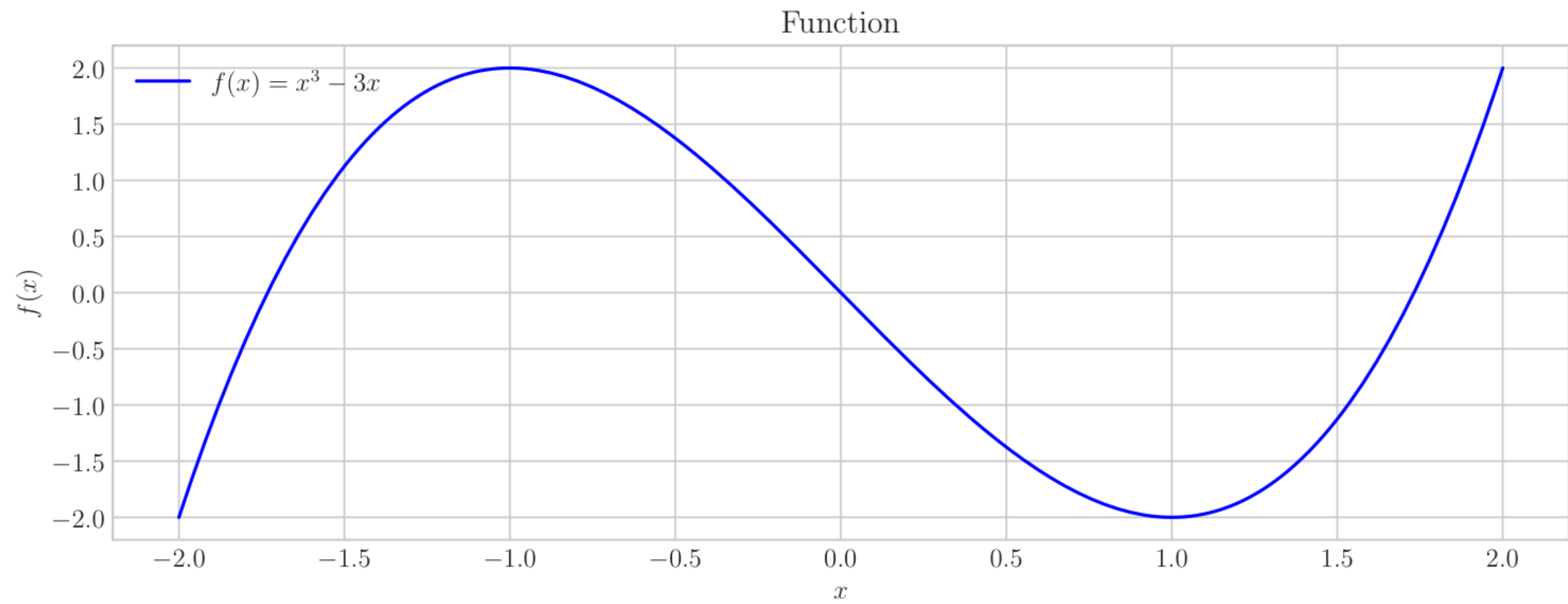
$$f(x) = x^3 - 3x$$

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
```

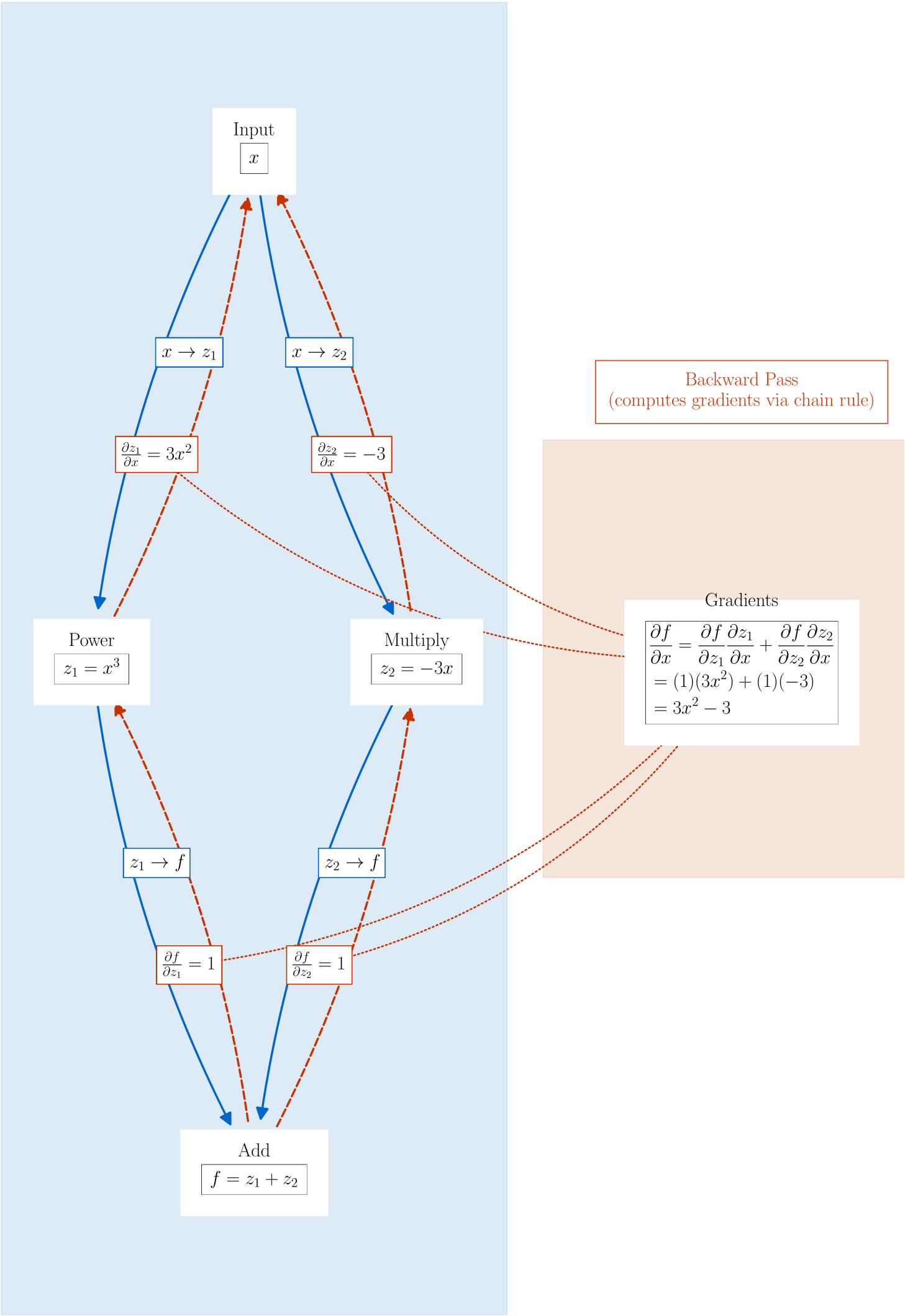


# How does PyTorch do this?

- **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.



# The graph



# Building the Computational Graph

Each node in the graph:

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

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

# backward(): Step by Step

## 1. 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

## 2. Power Node ( $z_1 = x^3$ ):

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

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

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

## 4. Input Node ( $x$ ):

- Accumulates from both paths
- $(-3)$  from multiply node
- $(3x^2)$  from power node
- Final gradient:  $\frac{\partial f}{\partial x} = 3x^2 - 3$

# Two Implementation Approaches

## 1. Using `backward()`

```
# 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()`

```
# 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

# Beyond 1d: Least Squares

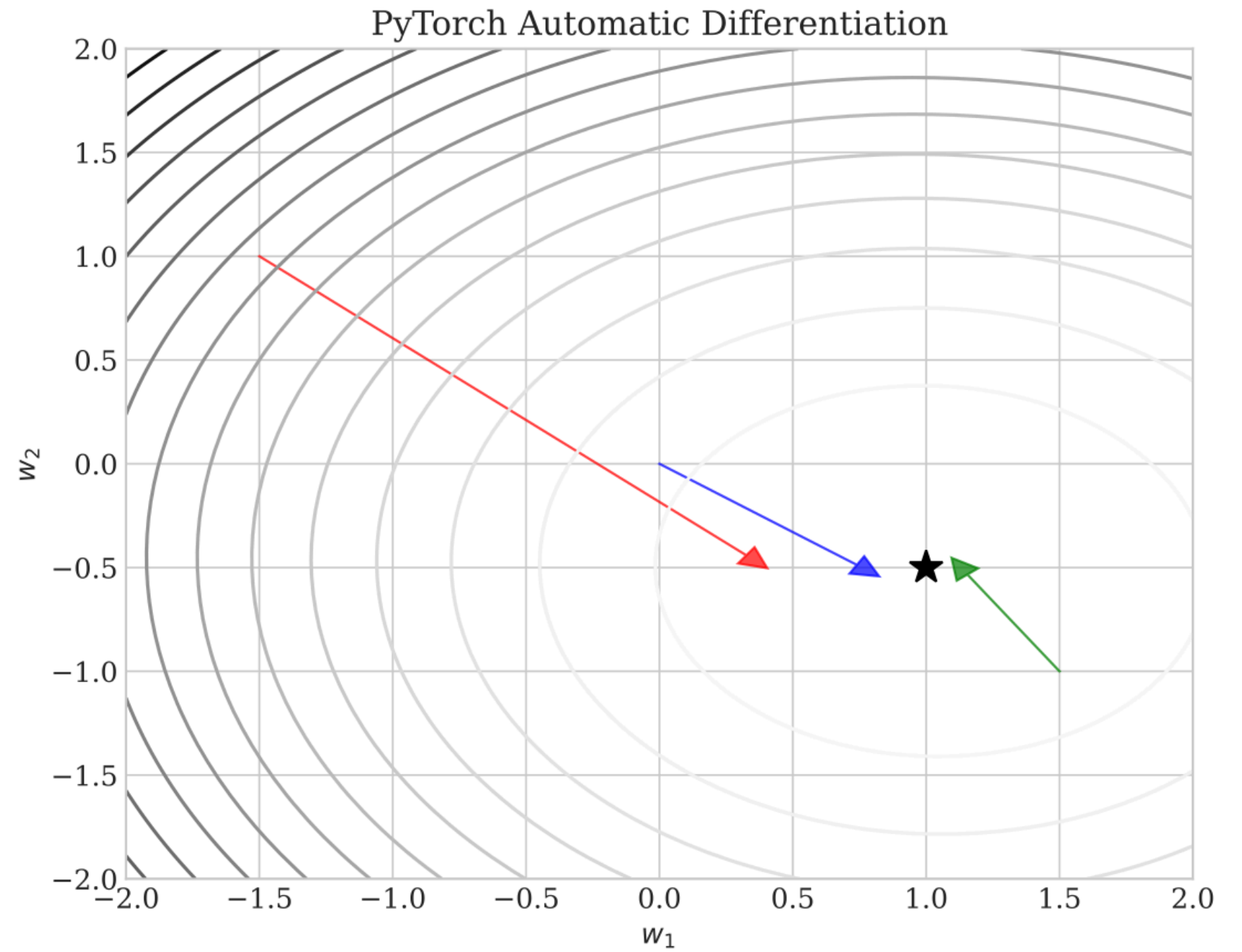
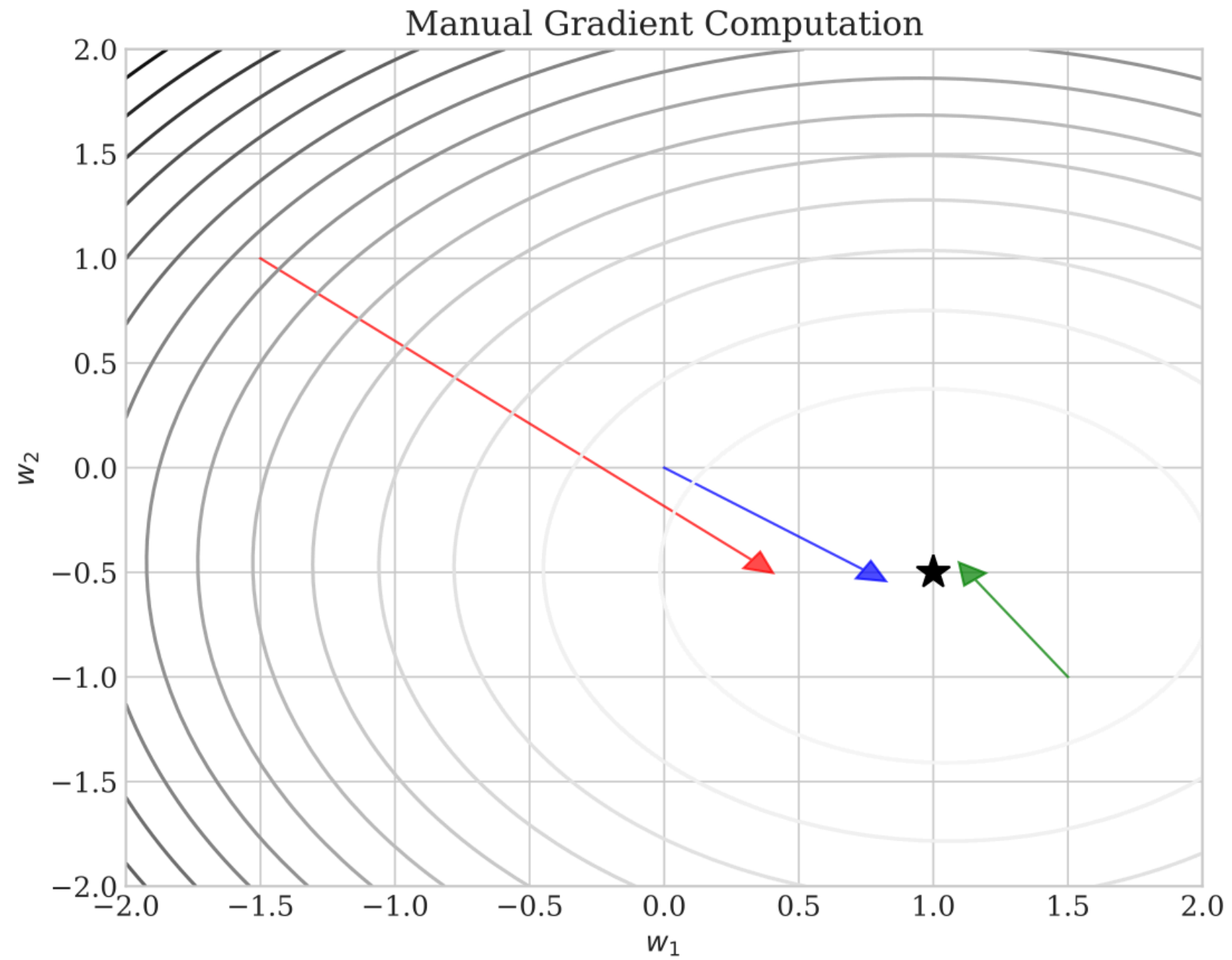
Manual gradient:

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

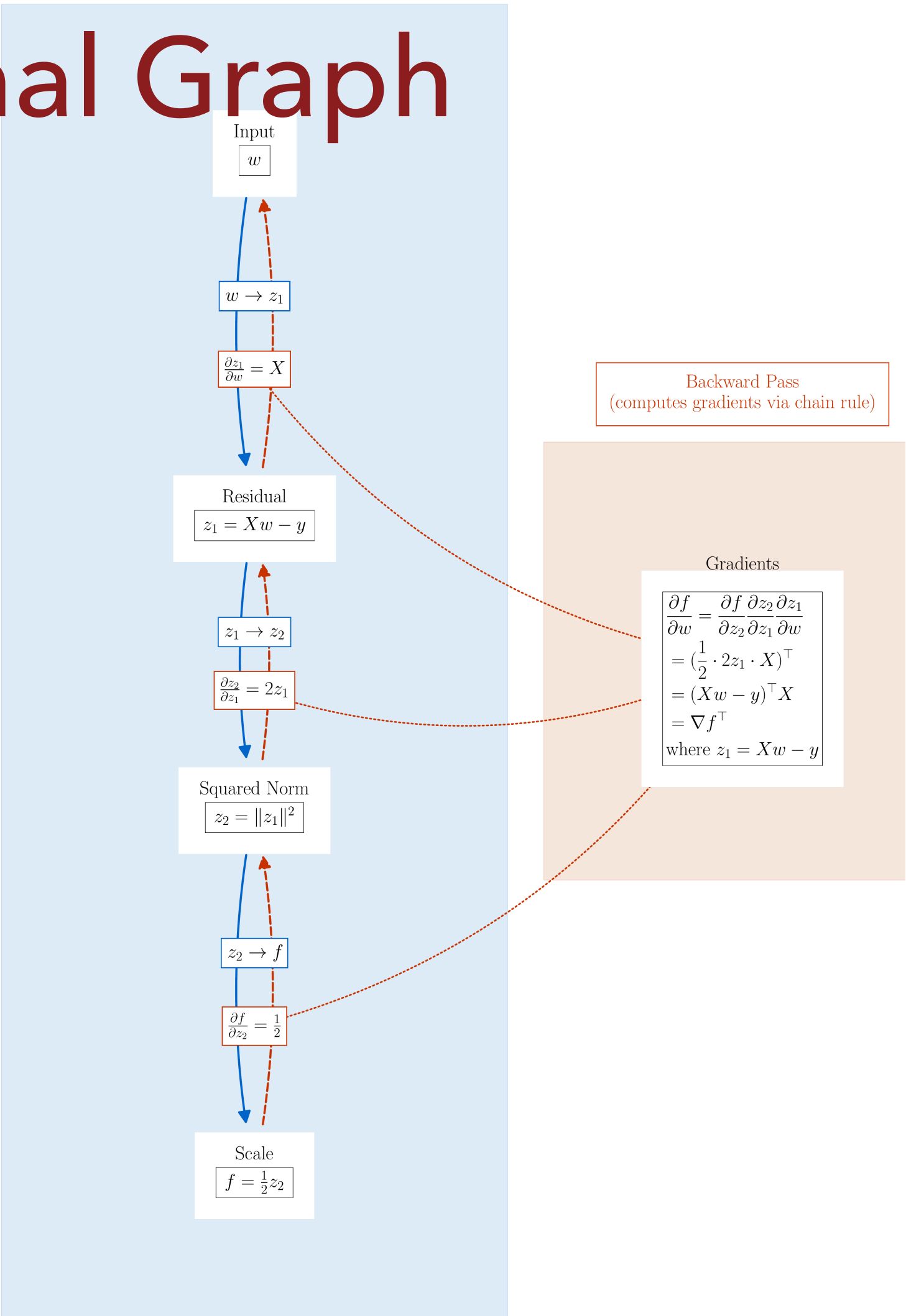
PyTorch gradient:

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

# Agreement between manual and PyTorch



# Computational Graph





# Building the Least Squares Graph

For  $f(w) = \frac{1}{2} \|Xw - y\|^2$ , we build:

1. Input nodes storing  $\mathbf{w}$
2. Residual node computing  $\mathbf{z}_1 = \mathbf{X}\mathbf{w} - \mathbf{y}$
3. Square norm node computing  $z_2 = \|\mathbf{z}_1\|^2$
4. Scale node forming  $f = \frac{1}{2} z_2$

# Computing Gradients: The Process

Subtlety: Total derivative vs :

## 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 total derivative
4. Add to input total derivative

# Least Squares: backward() Step 1

**Output Node** ( $f = \frac{1}{2} z_2$ ):

- Incoming gradient:  $\frac{\partial f}{\partial f} = 1$  (scalar)
- Total derivative:  $\frac{\partial f}{\partial z_2} = \frac{1}{2}$  (scalar)
- Propagate to  $z_2$  node:  $\frac{\partial f}{\partial z_2} = \frac{1}{2}$  ( $1 \times 1$  matrix)

# Least Squares: backward() Step 2

**Square Norm Node** ( $z_2 = \|\mathbf{z}_1\|^2$ ):

- Incoming total derivative:  $\frac{\partial f}{\partial z_2} = \frac{1}{2}$  ( $1 \times 1$  matrix)
- Local total derivative:  $\frac{\partial z_2}{\partial \mathbf{z}_1} = 2\mathbf{z}_1^\top$  ( $1 \times n$  matrix)
- Propagate to  $\mathbf{z}_1$  node:  $\frac{\partial f}{\partial \mathbf{z}_1} = \frac{\partial f}{\partial z_2} \frac{\partial z_2}{\partial \mathbf{z}_1} = \mathbf{z}_1^\top$  ( $1 \times n$  matrix)

# Least Squares: backward() Step 3

**Residual Node ( $\mathbf{z}_1 = \mathbf{X}\mathbf{w} - \mathbf{y}$ ):**

- Incoming total derivative:  $\frac{\partial f}{\partial \mathbf{z}_1} = \mathbf{z}_1^\top$  ( $1 \times n$  matrix)
- Local total derivative:  $\frac{\partial \mathbf{z}_1}{\partial \mathbf{w}} = \mathbf{X}$  ( $n \times p$  matrix)
- Total derivative to  $\mathbf{w}$  node:  $\frac{\partial f}{\partial \mathbf{w}} = \mathbf{z}_1^\top \mathbf{X}$  ( $1 \times p$  matrix)

# Least Squares: Final Step

Input Node ( $\mathbf{w}$ ):

- Total derivative:  $\frac{\partial f}{\partial \mathbf{w}} = \mathbf{z}_1^\top \mathbf{X}$  ( $1 \times p$  matrix)
- Convert to gradient:  $\nabla f = \left(\frac{\partial f}{\partial \mathbf{w}}\right)^\top = \mathbf{X}^\top \mathbf{z}_1$  ( $p \times 1$  matrix)

Final computation:

$$\nabla f = \left( \frac{\partial z_1}{\partial w} \frac{\partial z_2}{\partial z_1} \frac{\partial f}{\partial z_2} \right)^\top = \mathbf{X}^\top (\mathbf{X}\mathbf{w} - \mathbf{y})$$

# Applying Gradient Descent

Minimize least squares loss:

$$f(w) = \frac{1}{2} \|Xw - y\|_2^2$$

Manual implementation:

```
def manual_gradient(X, y, w):  
    return X.T @ (X @ w - y)  
  
w = torch.zeros(p) # Initialize  
for step in range(max_iters):  
    grad = manual_gradient(X, y, w)  
    w = w - alpha * grad
```

# PyTorch Implementation

Same algorithm, automatic gradients:

```
w = torch.zeros(p, requires_grad=True) # Initialize weights and require gradients

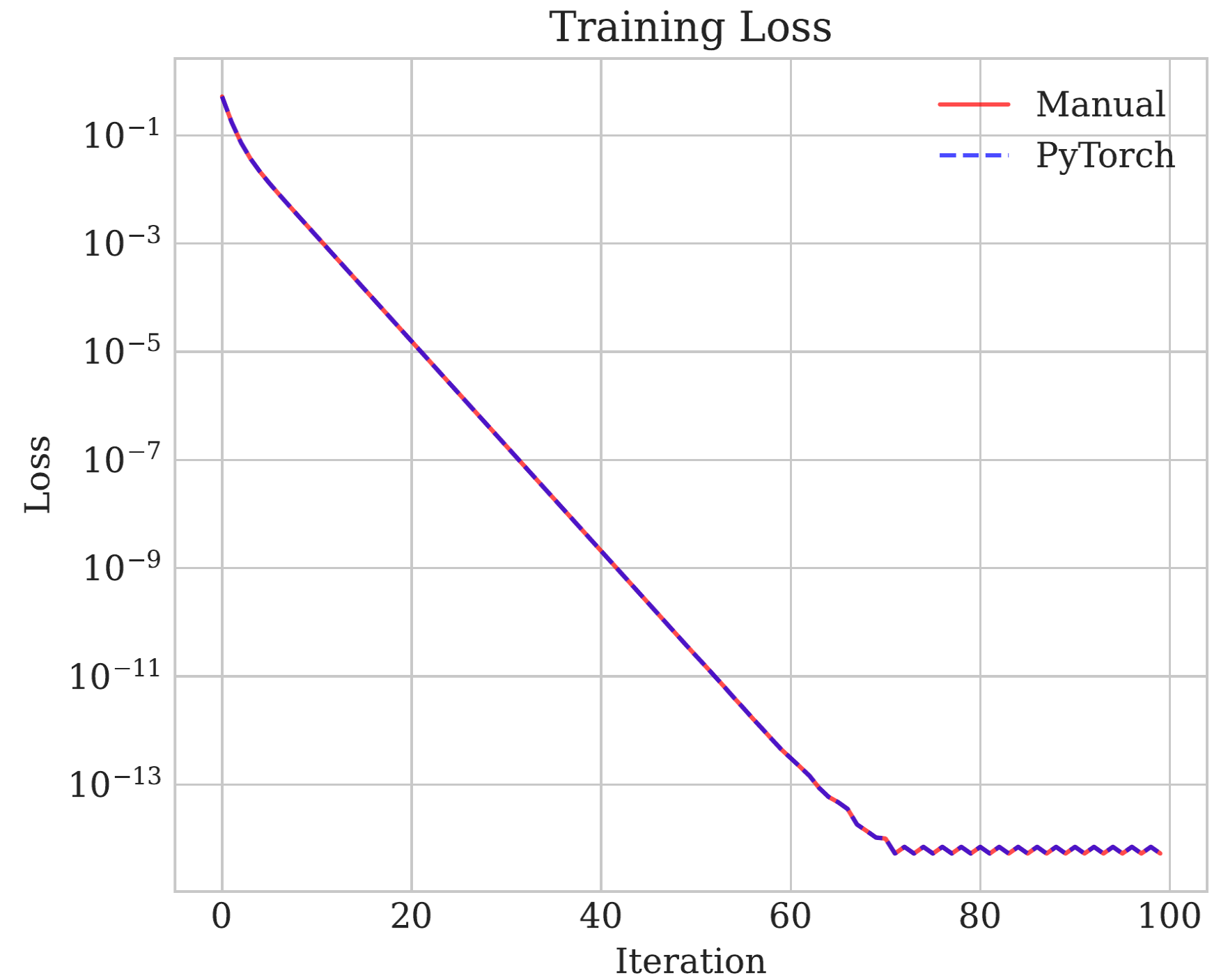
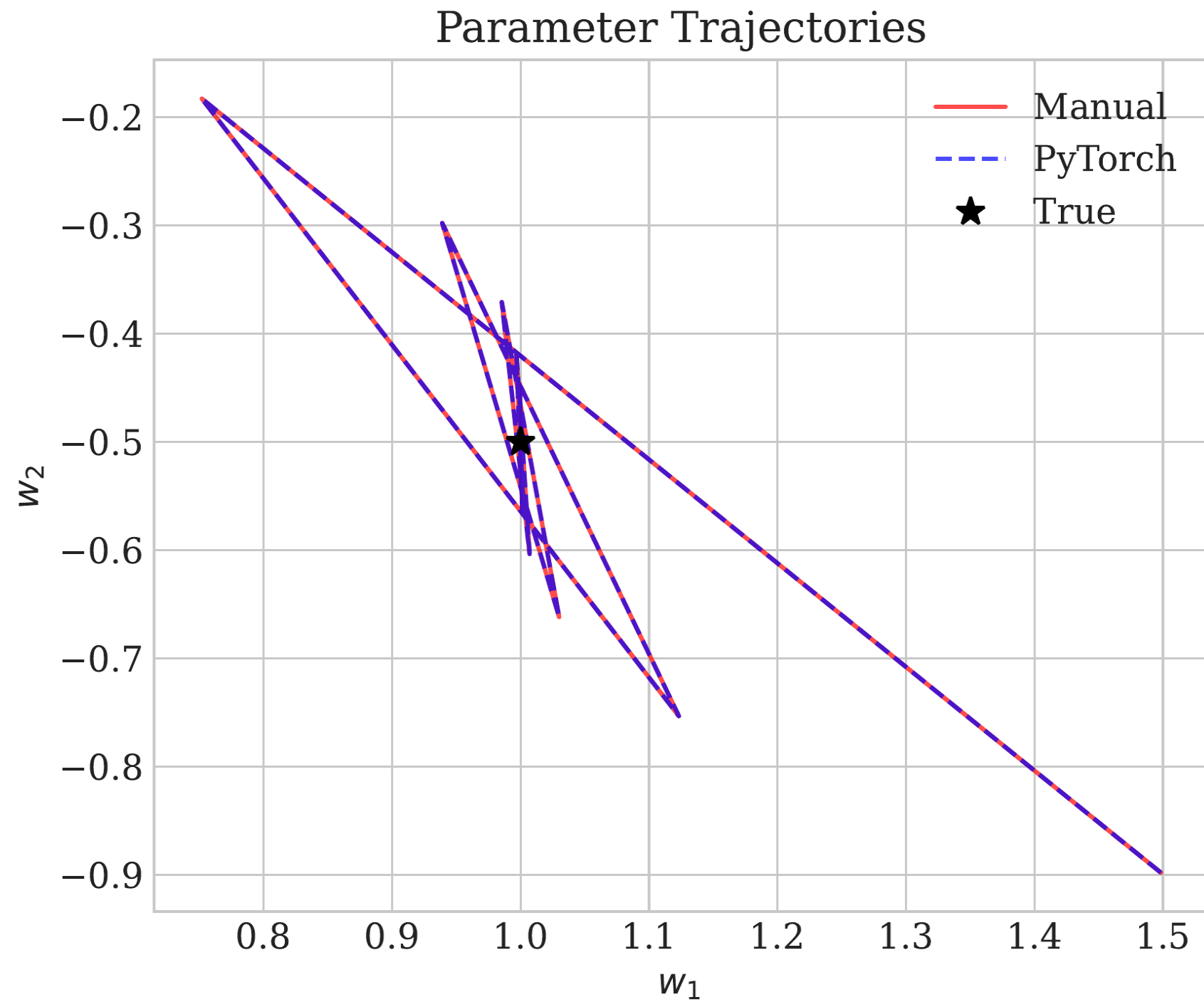
for step in range(max_iters):
    # Forward pass
    pred = X @ w
    loss = 0.5 * ((pred - y)**2).sum()

    # Backward pass
    loss.backward()

    # Update
    with torch.no_grad(): # Do not modify the computational graph
        w -= alpha * w.grad # update the weights
        w.grad.zero_() # reset the gradient to zero to avoid accumulation
```



# Comparison of Approaches



# From Linear to Neural Networks

Linear Model:

$$\mathbf{y} = \mathbf{X}\mathbf{w}$$

Neural Network:

$$\mathbf{h} = \tanh(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$$

$$p = \sigma(\mathbf{w}_2^\top \mathbf{h} + b_2)$$

Key differences:

- Multiple transformations
- Nonlinear activations
- Learnable features

# Neural Network Architecture

Layer composition:

Input  $\rightarrow$  Linear<sub>1</sub>  $\rightarrow$  Tanh  $\rightarrow$  Linear<sub>2</sub>  $\rightarrow$  Sigmoid  
 $\mathbb{R}^d$   $\mathbb{R}^{h \times d}$   $\mathbb{R}^h$   $\mathbb{R}^{1 \times h}$   $[0, 1]$

Dimensions:

- Input:  $\mathbf{x} \in \mathbb{R}^d$
- Hidden:  $\mathbf{h} \in \mathbb{R}^h$
- Output:  $p \in [0, 1]$

Each layer adds:

- Linear transform
- Nonlinearity
- Learnable parameters

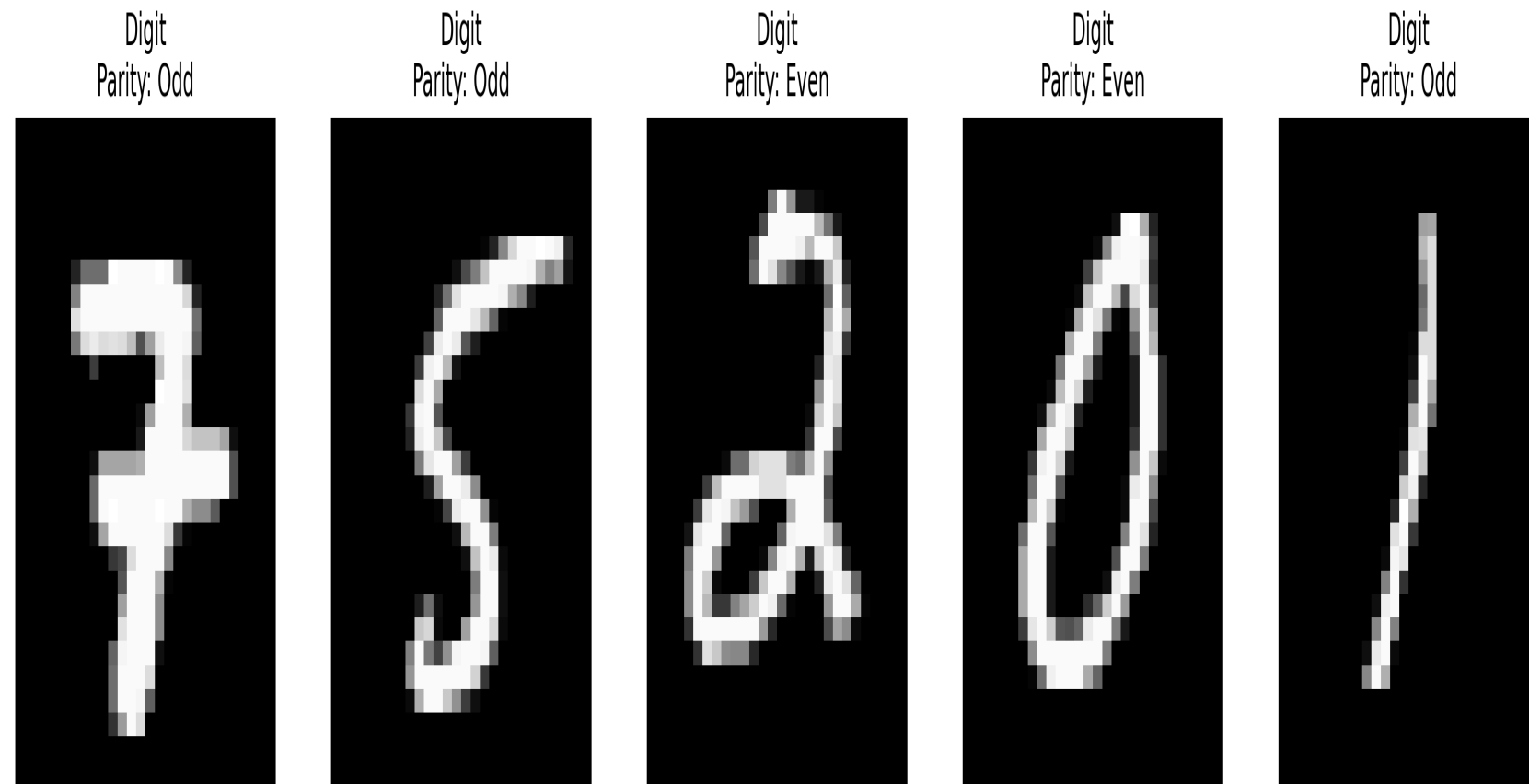
# PyTorch Implementation

```
class BinaryClassifier(nn.Module):  
    def __init__(self, d=784, h=32):  
        super().__init__()  
        #  $\mathbb{R}^d \rightarrow \mathbb{R}^h$   
        self.linear1 = nn.Linear(d, h)  
        #  $\mathbb{R}^h \rightarrow \mathbb{R}$   
        self.linear2 = nn.Linear(h, 1)  
  
    def forward(self, x):  
        # Hidden features  
        h = torch.tanh(self.linear1(x))  
        # Probability output  
        return torch.sigmoid(  
            self.linear2(h)  
        )
```

# Training Loop

```
def train_step(model, x, y, optimizer):  
    # 1. Forward: compute prediction and loss  
    pred = model(x)  
    loss = criterion(y_pred.squeeze(), y_train)  
  
    # 2. Backward: compute gradients  
    optimizer.zero_grad()  
    loss.backward()  
  
    # 3. Update: apply gradients  
    optimizer.step()  
  
    return loss.item()
```

# 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:

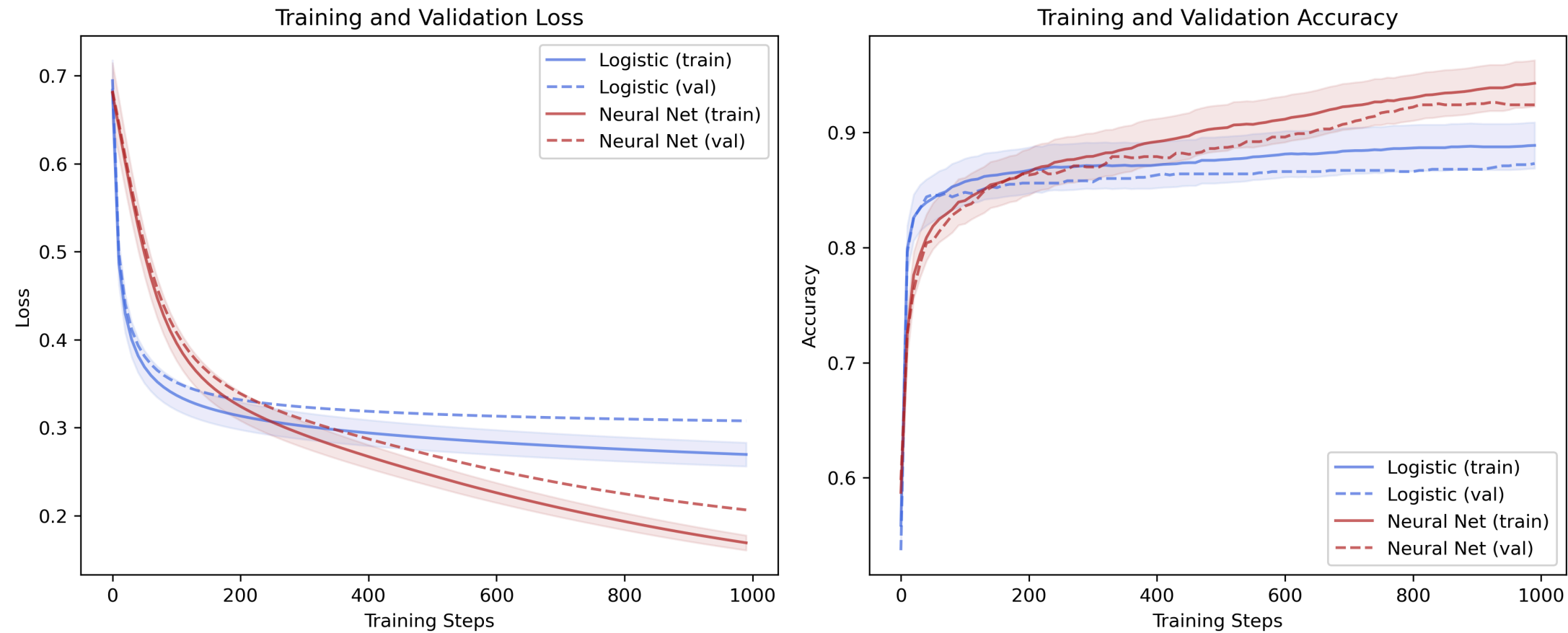
```
class SimpleNN(nn.Module):  
    def __init__(self):  
        super().__init__()  
        self.fc1 = nn.Linear(784, 32)  
        self.fc2 = nn.Linear(32, 1)  
  
    def forward(self, x):  
        # Hidden layer with ReLU  
        h = torch.relu(  
            self.fc1(x.view(-1, 784))  
        )  
        # Output layer  
        return torch.sigmoid(self.fc2(h))
```

# Training Process: Step by Step

```
def train_model(model, X_train, y_train, X_val, y_val, alpha=0.01, n_steps=1000):  
    for step in range(n_steps):  
        # Forward pass  
        y_pred = model(X_train)  
        loss = criterion(y_pred.squeeze(), y_train)  
  
        # Backward pass  
        loss.backward()  
  
        # Update parameters  
        with torch.no_grad(): # Do not modify the computational graph  
            for param in model.parameters():  
                param -= alpha * param.grad # update the parameters  
                param.grad.zero_() # reset the gradient to zero to avoid accumulation
```



# Results Analysis



## Final Results:

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

# Questions?

- Course website: <https://damek.github.io/STAT-4830/>
- Office hours: Listed on course website
- Email: [damek@wharton.upenn.edu](mailto:damek@wharton.upenn.edu)