Automatic Differentiation for MLPs in Binary Classification

George Michailidis

gmichail@ucla.edu

STAT 102B

What is Automatic Differentiation?

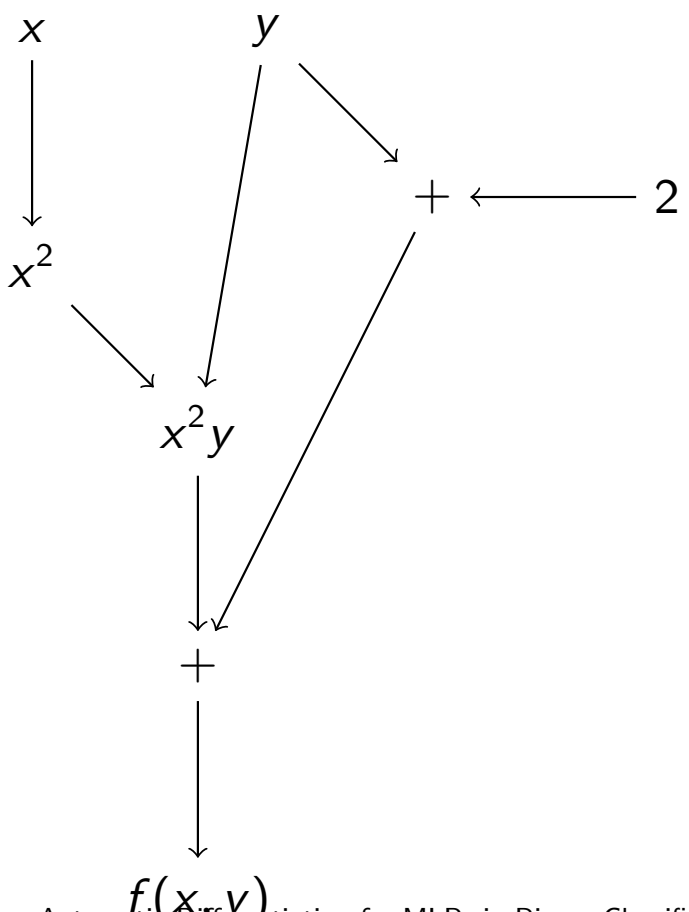
- Automatic Differentiation (AD): Technique to numerically evaluate the derivative of a function specified by a computer program
- Key idea: Apply the chain rule repeatedly to elementary operations
- Differs from:
  - Symbolic differentiation (unwieldy expressions)
  - Numerical differentiation (suffers from round-off errors)
- Two main modes:
  - Forward mode: Propagates derivatives forward through computation
  - Reverse mode: Propagates derivatives backward (backpropagation)

Why Automatic Differentiation?

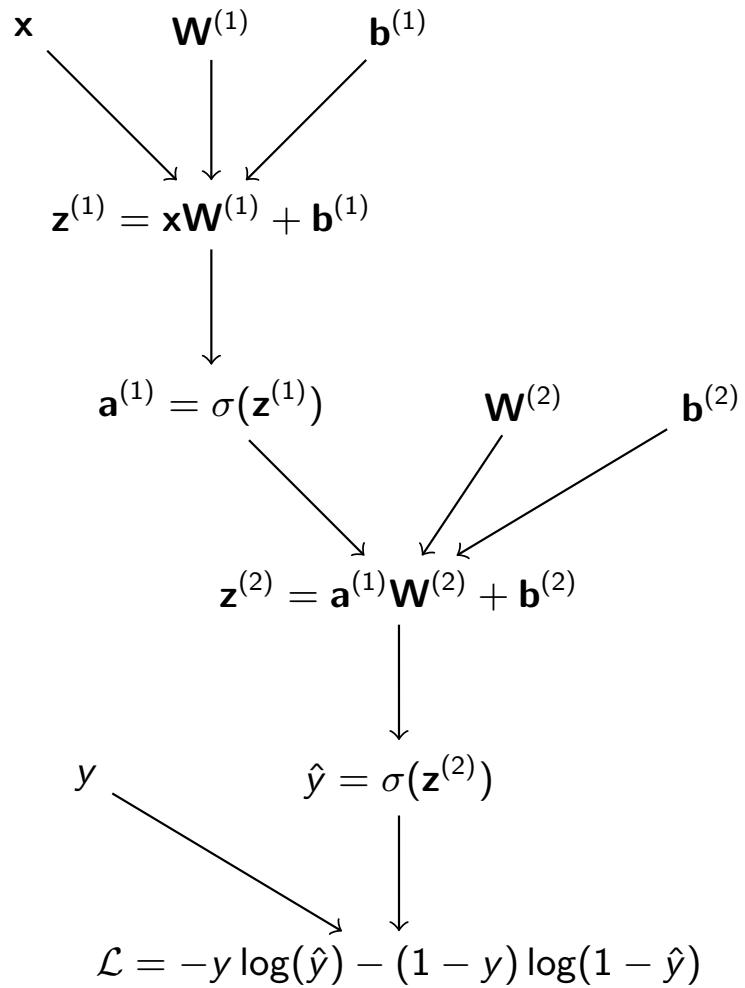
- Exact (to machine precision) derivatives
- Efficient for large models with many parameters
- Flexible - can differentiate through complex control flow
- Essential for deep learning frameworks:
  - PyTorch, TensorFlow, JAX use reverse-mode AD
  - Enables automatic optimization of neural networks
- Critical for MLPs in binary classification where we need:
  - Gradients for weight updates
  - Optimization of non-linear loss functions

Computational Graphs

- Computational Graph: Represents the computation as a directed graph
  - Nodes: Variables or operations
  - Edges: Data flow between operations
- **Example**: For $f(x, y) = x^2 y + y + 2$



Computational Graph for MLP - I



Computational Graph for MLP - II

- Each node represents a tensor operation
- σ is the activation function (typically ReLU for the hidden layer and sigmoid for the output layer)
- The computational graph enables systematic gradient computation

Tensors and Tensor Operations in Neural Networks

- Tensors: Multi-dimensional arrays that generalize scalars (0D), vectors (1D), and matrices (2D)
- Key Tensor Operations:
  - Linear transformations: $\mathbf{Wx} + \mathbf{b}$ (matrix-vector multiplication)
  - Element-wise operations: Activation functions (σ), addition, multiplication
  - Reduction operations: Loss computation (mean, sum)
  - Reshaping operations: Flattening, view, transpose
- Tensors track computation history, enabling **automatic differentiation** for backpropagation
- Modern frameworks (PyTorch, TensorFlow) optimize tensor operations for GPU acceleration

Forward Mode Autodifferentiation

- Key idea: Simultaneously compute values and derivatives
- Dual numbers: Represent $x + \dot{x}\epsilon$ where:
  - x is the primal value
  - $\dot{x}$ is the derivative value
  - ϵ is an a very small value so that $\epsilon^2 = 0$
- Process: Propagate derivatives forward through the computation

Example 1

For $f(x) = x^2 + 3x + 2$ at $x = 5$:

$$\begin{aligned} &\text{Let } x = 5 + \dot{x}\epsilon \text{ where } \dot{x} = 1 \\ &f(x) = (5+\epsilon)^2 + 3(5+\epsilon) + 2 \\ &= 25 + 2 \cdot 5 \cdot \epsilon + 15 + 3\epsilon + 2 \\ &= 42 + 13\epsilon \end{aligned}$$

Therefore, $f(5) = 42$ and $f'(5) = 13$

Reverse Mode Autodifferentiation

- Key idea: First perform forward pass to compute all intermediate values, then propagate gradients backward
- Advantages:
  - Efficient for functions with many inputs and few outputs
  - Perfect for neural networks with many parameters
- Process:
  1. Forward pass: Compute and store all intermediate values
  2. Backward pass: Compute gradients of output with respect to each intermediate value

Backpropagation is a specific implementation of reverse-mode AD for neural networks

Comparison: Forward vs. Reverse Mode

Forward Mode

- Efficient when inputs $<$ outputs
- Computes $\frac{\partial y_j}{\partial x_i}$ for all j and fixed i
- Simpler implementation
- Less memory intensive

Reverse Mode

- Efficient when inputs $>$ outputs
- Computes $\frac{\partial y_j}{\partial x_i}$ for all i and fixed j
- More complex implementation
- Higher memory requirements

For MLPs: Reverse mode is preferred since we have many parameters (inputs) but only one loss value (output)

Review: MLP Architecture for Binary Classification

Notation:

- Input: $\mathbf{x} \in \mathbb{R}^d$
- Hidden layer weights: $\mathbf{W}^{(1)} \in \mathbb{R}^{d \times h}$
- Hidden layer bias: $\mathbf{b}^{(1)} \in \mathbb{R}^h$
- Output layer weights: $\mathbf{W}^{(2)} \in \mathbb{R}^{h \times 1}$
- Output layer bias: $\mathbf{b}^{(2)} \in \mathbb{R}$
- Hidden layer activation: $\sigma_h(\cdot)$ (e.g., ReLU, tanh)
- Output layer activation: $\sigma(\cdot)$ (sigmoid for binary classification)

Forward Propagation:

$$\mathbf{z}^{(1)} = \mathbf{x}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}$$

$$\mathbf{a}^{(1)} = \sigma_h(\mathbf{z}^{(1)})$$

$$z^{(2)} = \mathbf{a}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}$$

$$\hat{y} = \sigma(z^{(2)}) = \frac{1}{1 + e^{-z^{(2)}}}$$

Review: Binary Cross-Entropy Loss

- **Binary Cross-Entropy Loss**:

$$\mathcal{L}(y, \hat{y}) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$$

- For a batch of n examples:

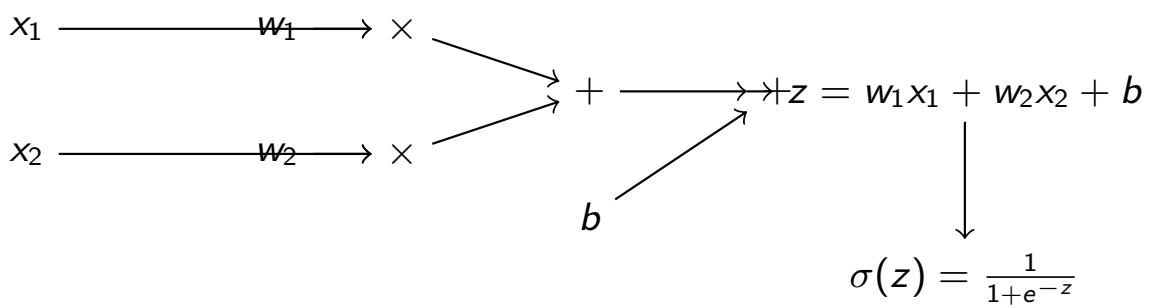
$$\mathcal{L}_{\text{batch}} = -\frac{1}{n} \sum_{i=1}^{n} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

- **Main objective**: Minimize $\mathcal{L}$ with respect to model parameters:

$$\theta = \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)}\}$$

- **Need**: $\nabla_\theta \mathcal{L}$ for gradient-based optimization

Computational Graph for Single Hidden Node



- Each operation is a node in the computational graph
- Gradients flow backward through this graph
- Each operation has a forward and backward rule

Elementary Operations and Their Derivatives

Automatic differentiation tracks derivatives through elementary operations:

Operation	**Forward**	**Backward** ($\frac{\partial L}{\partial \textbf{inputs}}$)
Addition ($z = x + y$)	$z = x + y$	$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}, \frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}$
Multiplication ($z = xy$)	$z = xy$	$\frac{\partial L}{\partial x} = y \cdot \frac{\partial L}{\partial z}, \frac{\partial L}{\partial y} = x \cdot \frac{\partial L}{\partial z}$
Sigmoid ($z = \sigma(x)$)	$z = \frac{1}{1+e^{-x}}$	$\frac{\partial L}{\partial x} = z(1-z) \cdot \frac{\partial L}{\partial z}$
ReLU ($z = \max(0, x)$)	$z = \max(0, x)$	$\frac{\partial L}{\partial x} = \mathbf{1}_{x>0} \cdot \frac{\partial L}{\partial z}$
Matrix multiply ($Z = XW$)	$Z = XW$	$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Z} W^T, \frac{\partial L}{\partial W} = X^T \frac{\partial L}{\partial Z}$

- These rules are applied automatically during backpropagation
- The chain rule connects all operations together

Detailed Backward Pass for Binary Classification MLP

Starting from the loss, working backwards:

1. **Output layer gradients**:

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}$$

$$\frac{\partial \hat{y}}{\partial z^{(2)}} = \hat{y}(1-\hat{y})$$

$$\frac{\partial \mathcal{L}}{\partial z^{(2)}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z^{(2)}} = \hat{y} - y$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(2)}} = (\mathbf{a}^{(1)})^T \cdot \frac{\partial \mathcal{L}}{\partial z^{(2)}}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(2)}} = \frac{\partial \mathcal{L}}{\partial z^{(2)}}$$

2. **Hidden layer gradients**:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(1)}} = \frac{\partial \mathcal{L}}{\partial z^{(2)}} \cdot (\mathbf{W}^{(2)})^T$$

Detailed Backward Pass (Continued)

3. **Hidden layer gradients (continued)**:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(1)}} \odot \frac{\partial \mathbf{a}^{(1)}}{\partial \mathbf{z}^{(1)}}$$

Where $\odot$ is element-wise multiplication, and $\frac{\partial \mathbf{a}^{(1)}}{\partial \mathbf{z}^{(1)}}$ depends on the activation function:

- For sigmoid: $\frac{\partial \mathbf{a}^{(1)}}{\partial \mathbf{z}^{(1)}} = \mathbf{a}^{(1)} \odot (1 - \mathbf{a}^{(1)})$
- For ReLU: $\frac{\partial \mathbf{a}^{(1)}}{\partial \mathbf{z}^{(1)}} = \mathbf{1}_{\mathbf{z}^{(1)} > 0}$
- For tanh: $\frac{\partial \mathbf{a}^{(1)}}{\partial \mathbf{z}^{(1)}} = 1 - (\mathbf{a}^{(1)})^2$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(1)}} = \mathbf{x}^T \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(1)}}$$
$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(1)}}$$

- These gradients are computed automatically in reverse order
- Intermediate values from the forward pass are reused

Autodiff Implementation in Python - Forward Pass

```
class MLP:
    def __init__(self, input_size, hidden_size):
        self.W1 = np.random.randn(input_size, hidden_size) * 0.01
        self.b1 = np.zeros((1, hidden_size))
        self.W2 = np.random.randn(hidden_size, 1) * 0.01
        self.b2 = np.zeros((1, 1))
        self.cache = {}  # Store intermediate values

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-np.clip(x, -500, 500)))

    def forward(self, X):
        # First layer
        Z1 = np.dot(X, self.W1) + self.b1
        A1 = np.maximum(0, Z1)  # ReLU activation

        # Second layer
        Z2 = np.dot(A1, self.W2) + self.b2
        A2 = self.sigmoid(Z2)
```

Autodiff Implementation - Loss and Backward Pass

```
def compute_loss(self, y_pred, y_true):
    m = y_true.shape[0]
    loss = -np.sum(
        y_true * np.log(y_pred + 1e-8) +
        (1 - y_true) * np.log(1 - y_pred + 1e-8)
    ) / m
    return loss

def backward(self, y_true):
    m = y_true.shape[0]
    X = self.cache['X']
    A1 = self.cache['A1']
    A2 = self.cache['A2']
    Z1 = self.cache['Z1']

    # Output layer
    dZ2 = A2 - y_true
    dW2 = np.dot(A1.T, dZ2) / m
    db2 = np.sum(dZ2, axis=0, keepdims=True) / m
```

Gradient Update and Training Loop

```
def update_parameters(self, grads, learning_rate):
    self.W1 -= learning_rate * grads['dW1']
    self.b1 -= learning_rate * grads['db1']
    self.W2 -= learning_rate * grads['dW2']
    self.b2 -= learning_rate * grads['db2']

def train(self, X, y, learning_rate=0.01, epochs=1000):
    losses = []

    for epoch in range(epochs):
        # Forward pass
        y_pred = self.forward(X)

        # Compute loss
        loss = self.compute_loss(y_pred, y)
        losses.append(loss)

        # Backward pass - compute gradients
        grads = self.backward(y)
```

Computational Optimizations in Autodiff

- Common subexpression elimination
  - Identify and reuse repeated computation patterns
- Checkpointing
  - Trade computation for memory by recomputing some intermediate values
  - Useful for very deep networks
- Operator fusion
  - Combine multiple operations into a single optimized kernel
  - Example: Fusing matrix multiplication and ReLU activation
- Mixed precision
  - Use lower precision (e.g., float16) for forward pass
  - Use higher precision (e.g., float32) for gradient accumulation
- Asynchronous execution
  - Overlap computation and memory transfers

Higher-Order Derivatives

- Second-order derivatives (Hessians):
  - $\mathbf{H} = \nabla^2_\theta \mathcal{L}$ contains curvature information
  - Useful for Newton's method and second-order optimizers
  - Computed by differentiating gradients
- Applications:
  - Second-order optimization methods
  - Natural gradient methods
  - Influence functions for interpretability
  - Sensitivity analysis
- Implementation:
  - Most autodiff frameworks support higher-order derivatives
  - Examples: PyTorch's `torch.autograd.grad(grad(...))`
  - Typically more memory and computation intensive

Advanced Gradient Operations

- **Gradient accumulation**
  - Process mini-batches separately but update once
  - Useful for limited memory scenarios
- **Gradient clipping**
  - Limit gradient magnitude to prevent exploding gradients
  - $\mathbf{g}_{\text{clipped}} = \text{clip}(\mathbf{g}, -c, c)$ or $\mathbf{g}_{\text{clipped}} = \mathbf{g} \cdot \min(1, \frac{c}{||\mathbf{g}||})$
- **Gradient normalization**
  - Scale gradients to have unit norm
  - $\mathbf{g}_{\text{norm}} = \frac{\mathbf{g}}{||\mathbf{g}||}$
- **Gradient noise injection**
  - Add Gaussian noise to gradients
  - Helps escape saddle points and local minima

Numerical Stability in Autodiff

- Vanishing/exploding gradients
  - Products of many derivatives can become very small or large
  - Solutions: Proper initialization, normalization layers, gradient clipping
- Stabilizing sigmoid
  - $\sigma(x) = \frac{1}{1+e^{-x}}$
  - For large negative x, use $e^{-x} \approx \infty$, so $\sigma(x) \approx 0$
  - For large positive x, use $e^{-x} \approx 0$, so $\sigma(x) \approx 1$

Summary: Chain Rule and Computation Graph

- Autodiff leverages the chain rule to compute gradients
- Constructs a graph of operations
- Backpropagates derivatives from output to inputs; e.g.,

$$\frac{\partial \mathcal{L}}{\partial W_1} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial h} \cdot \frac{\partial h}{\partial W_1}$$