Week 2: Part2 Derivatives

Chain Rule

The chain rule is a formula for computing the derivative of the composition of two or more functions.

Basic Formula

If $y = f(g(x))$, then:

$$dy/dx = (df/dg) \times (dg/dx)$$

Or in Leibniz notation: If $y = f(u)$ and $u = g(x)$, then:

$$dy/dx = (dy/du) \times (du/dx)$$

Examples

Example 1: Find the derivative of $y = sin(x^2)$

Solution: Let $u = x^2$, so $y = sin(u)$

$dy/du = cos(u)$ and $du/dx = 2x$

Therefore, $dy/dx = cos(u) \times 2x = 2xcos(x^2)$

Example 2: Find the derivative of $y = (3x + 2)^5$

Solution: Let $u = 3x + 2$, so $y = u^5$

$dy/du = 5u^4$ and $du/dx = 3$

Therefore, $dy/dx = 5u^4 \times 3 = 15(3x + 2)^4$

Extended Chain Rule

For nested functions like $y = f(g(h(x)))$, the chain rule extends to:

$dy/dx = (df/dg) \times (dg/dh) \times (dh/dx)$

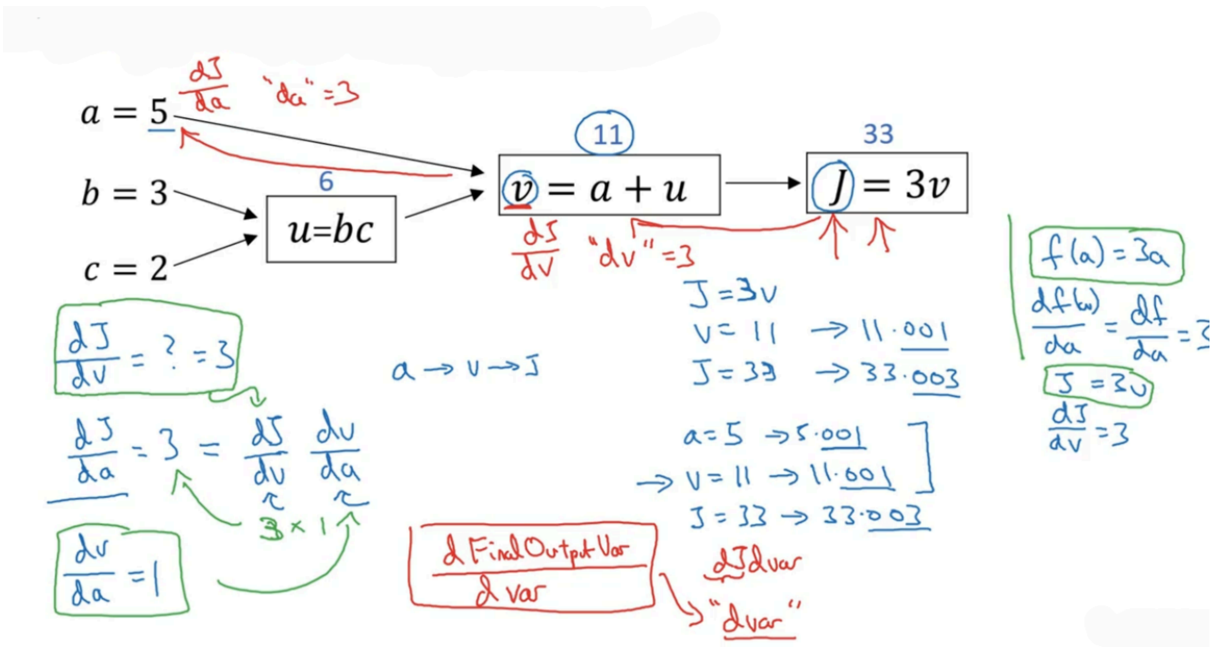
Applications

The chain rule is essential for finding derivatives of:

- Composite functions
- Functions raised to variable powers
- Nested trigonometric, logarithmic, and exponential functions

Remember that proper application of the chain rule requires identifying the composition structure of the function correctly.

Computing derivatives



Forward Pass

- Given:
    - $a = 5,\ b = 3,\ c = 2$
    - $u = bc = 3 \times 2 = 6$
- Next:
    - $v = a + u = 5 + 6 = 11$
- Finally:
    - $J = 3v = 3 \times 11 = 33$
    - $u = 6, v = 11, J = 33$

Backward Pass (Gradients)

We want to compute $\frac{\partial J}{\partial a}$.

1. $J = 3v$

$$\frac{\partial J}{\partial v} = 3$$

2. $v = a + uv$

$$\frac{\partial v}{\partial a} = 1$$

3. Chain rule:

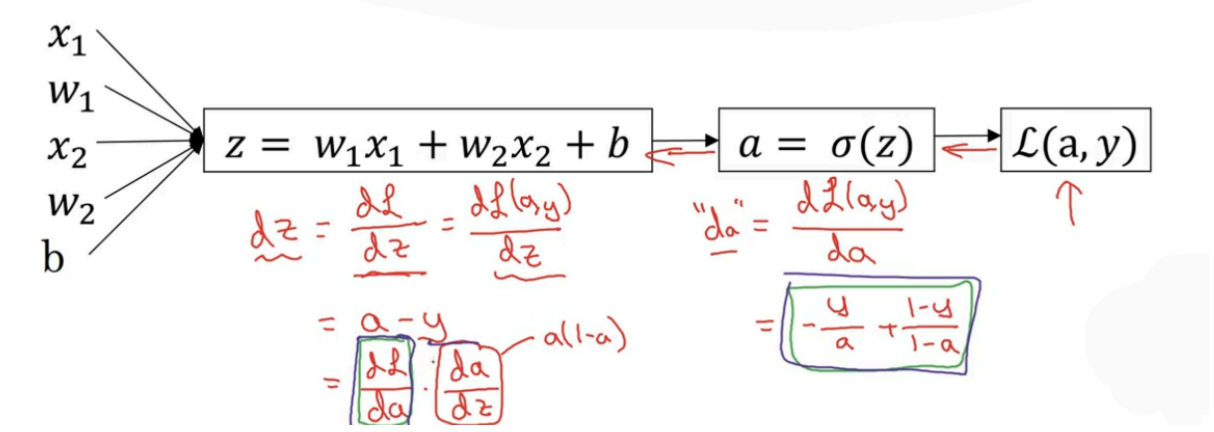
$$\frac{\partial J}{\partial a} = \frac{\partial J}{\partial v} \cdot \frac{\partial v}{\partial a}$$

$$\frac{\partial J}{\partial a} = 3 \cdot 1 = 3$$

Gradient:

$$\frac{\partial J}{\partial a} = 3$$

Logistic Regression Gradient Descent:



x_1
w_1
x_2
w_2
b
$z = w_1x_1 + w_2x_2 + b$
$a = \sigma(z)$
$\mathcal{L}(a, y)$
$dz = \frac{d\mathcal{L}}{dz} = \frac{d\mathcal{L}(a,y)}{dz}$
$= a - y$
$= \frac{d\mathcal{L}}{da} \cdot \frac{da}{dz}$
$a(1-a)$
"da" $= \frac{d\mathcal{L}(a,y)}{da}$
$= -\frac{y}{a} + \frac{1-y}{1-a}$

Forward:

$z = w_1x_1 + w_2x_2 + b, \qquad a = \sigma(z) = \frac{1}{1+e^{-z}}$

$$\mathcal{L}(a, y) = -\Big(y \log a + (1-y)\log(1-a)\Big)$$

Step 1 — derivative of loss w.r.t activation a:

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

Step 2 — derivative of activation w.r.t pre-activation z:

$$\frac{\partial a}{\partial z} = a(1-a)$$

Step 3 — chain rule to get gradient at z:

$$\frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial a} \cdot \frac{\partial a}{\partial z} = \Big(-\frac{y}{a} + \frac{1-y}{1-a}\Big)\, a(1-a) = a - y$$

Step 4 — gradients for parameters and inputs:

$$\frac{\partial \mathcal{L}}{\partial w_1} = (a-y)\,x_1, \qquad \frac{\partial \mathcal{L}}{\partial w_2} = (a-y)\,x_2, \qquad \frac{\partial \mathcal{L}}{\partial b} = a - y$$

$$\frac{\partial \mathcal{L}}{\partial x_1} = (a-y)\,w_1, \qquad \frac{\partial \mathcal{L}}{\partial x_2} = (a-y)\,w_2$$

Takeaway: with sigmoid + cross-entropy, the backprop signal into this neuron is simply $a - y$.

Gradient Descent with Vectorization

We have training data:

- Features:

$$X = \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(m)} \\ x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(m)} \end{bmatrix} \quad \in \mathbb{R}^{n_x \times m}$$

  where $n_x = 2$

- Labels:

$$Y = \big[y^{(1)}, y^{(2)}, \ldots, y^{(m)}\big] \in \mathbb{R}^{1 \times m}$$

- Parameters:

$$w = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \in \mathbb{R}^{n_x \times 1}, \quad b \in \mathbb{R}$$

1. Forward Propagation (Vectorized)

$$Z = w^T X + b \quad \in \mathbb{R}^{1 \times m}$$

$$A = \sigma(Z) = \frac{1}{1 + e^{-Z}} \quad \in \mathbb{R}^{1 \times m}$$

$$J(w, b) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)}) \right]$$

2. Backward Propagation (Vectorized)

- Error term:

$$dZ = A - Y \in R1 \times m\mathbb{R}^{1 \times m}$$

- Gradients:

$$dw = \frac{1}{m} X dZ^T \quad \in \mathbb{R}^{n_x \times 1}$$

$$db = \frac{1}{m} \sum_{i=1}^{m} (a^{(i)} - y^{(i)}) \quad \in \mathbb{R}$$

---

3. Parameter Updates

$$w := w - \alpha\, dw, \quad b := b - \alpha\, db$$

Summary of Vectorization

Instead of looping over m examples:

- Forward: `Z = np.dot(w.T, X) + b`
- Activation: `A = sigmoid(Z)`

- Backprop:
    - `dZ = A - Y`
    - `dw = (1/m) * np.dot(X, dZ.T)`
    - `db = (1/m) * np.sum(dZ)`

This way, **all examples are processed simultaneously** in matrix form.

Vector Addition

```
# Non-vectorized (slow)
def add_vectors_loop(a, b):
    c = np.zeros(len(a))
    for i in range(len(a)):
        c[i] = a[i] + b[i]
    return c

# Vectorized (fast)
def add_vectors_vectorized(a, b):
    return a + b  # NumPy handles element-wise addition
```

Computing Dot Product

```
# Non-vectorized
def dot_product_loop(a, b):
    result = 0
    for i in range(len(a)):
        result += a[i] * b[i]
    return result

# Vectorized
def dot_product_vectorized(a, b):
    return np.dot(a, b)  # Uses optimized BLAS operations
```

Logistic Regression Forward Pass

```
# Non-vectorized (for m training examples)
def forward_pass_loop(X, w, b):
    m = X.shape[0]
    A = np.zeros(m)
    for i in range(m):
        z = 0
        for j in range(len(w)):
            z += X[i,j] * w[j]
        z += b
        A[i] = 1 / (1 + np.exp(-z))
    return A

# Vectorized
def forward_pass_vectorized(X, w, b):
    Z = np.dot(X, w) + b
    A = 1 / (1 + np.exp(-Z))
    return A
```

Gradient Computation

```
# Non-vectorized gradient computation
def compute_gradients_loop(X, y, A):
    m, n = X.shape
    dw = np.zeros(n)
    db = 0

    for i in range(m):
        error = A[i] - y[i]
        db += error
        for j in range(n):
            dw[j] += error * X[i,j]

    dw /= m
    db /= m
    return dw, db

# Vectorized gradient computation
def compute_gradients_vectorized(X, y, A):
    m = X.shape[0]
    error = A - y
    dw = (1/m) * np.dot(X.T, error)
    db = (1/m) * np.sum(error)
    return dw, db
```

Key Libraries for Vectorization

- **NumPy:** Fundamental package for scientific computing with Python
- **TensorFlow/PyTorch:** Deep learning frameworks with optimized tensor operations
- **Pandas:** Data manipulation library with vectorized operations
- **JAX:** Google's library for high-performance machine learning research

Best Practices

- **Avoid explicit Python loops** when operating on arrays
- **Use broadcasting** to handle operations between arrays of different shapes
- **Leverage axis parameters** in NumPy functions for dimension-specific operations
- **Pre-allocate arrays** instead of growing them incrementally
- **Profile your code** to identify bottlenecks that could benefit from vectorization

Common Pitfalls

- **Memory issues** when working with very large matrices
- **Over-vectorization** can sometimes reduce code readability
- **Incorrect broadcasting** leading to subtle bugs
- **Overlooking specialized functions** that are already optimized

> Vectorization is essential for deep learning because it transforms computationally intensive loops into highly optimized matrix operations, enabling efficient training of complex neural networks on large datasets.

Vectorized vs non Vectorized

```
import numpy as np
import time
a = np.random.rand(1000000)
b = np.random.rand(1000000)

tic = time.time()
c = np.dot(a,b)
toc = time.time()

print(c)
print("Vectorized version:" + str(1000*(toc-tic)) +"ms")

c = 0
tic = time.time()
for i in range(1000000):
    c += a[i]*b[i]
toc = time.time()

print(c)
print("For loop:" + str(1000*(toc-tic)) + "ms")

# Output:
# 250286.989866
# Vectorized version:1.5027523040771484ms
# 250286.989866
# For loop:474.29513931274414ms
```

This code demonstrates the performance difference between vectorized and non-vectorized approaches for computing a dot product, showing that the vectorized version using np.dot() is significantly faster (about 300x) than the loop-based implementation.

Essential Vectorization Techniques for Deep Learning

1. Basic NumPy Operations

```
# Array creation
x = np.array([1, 2, 3, 4])
y = np.zeros((3, 4))
z = np.ones((2, 5))
random_array = np.random.rand(5, 5)

# Element-wise operations
a + b       # Addition
a - b       # Subtraction
a * b       # Element-wise multiplication
a / b       # Division
np.exp(a)   # Exponential
np.log(a)   # Natural logarithm
np.sqrt(a)  # Square root
```

2. Matrix Operations

```
# Matrix multiplication
np.dot(A, B)     # Dot product
A @ B            # Matrix multiplication (Python 3.5+)

# Transpose
A.T              # Transpose matrix

# Special matrix operations
np.linalg.inv(A)  # Matrix inverse
np.linalg.det(A)  # Determinant
np.linalg.eig(A)  # Eigenvalues and eigenvectors
```

3. Broadcasting

```
# Adding a scalar to an array
A + 5

# Adding a vector to each row of a matrix
# A has shape (m, n), b has shape (n,)
A + b
```

```
# Adding a vector to each column of a matrix
# A has shape (m, n), c has shape (m, 1)
A + c
```

4. Axis Operations

```
# Sum across different dimensions
np.sum(A)        # Sum all elements
np.sum(A, axis=0) # Sum each column (result has shape (n,))
np.sum(A, axis=1) # Sum each row (result has shape (m,))

# Other axis operations
np.mean(A, axis=0)  # Mean of each column
np.max(A, axis=1)   # Max of each row
np.argmax(A, axis=0) # Index of max element in each column
```

5. Reshaping and Manipulating Arrays

```
# Reshaping
A.reshape(3, 4)    # Reshape to 3×4 matrix
A.flatten()        # Flatten to 1D array
A.reshape(-1, 1)   # Convert to column vector

# Concatenation
np.concatenate([A, B], axis=0)  # Vertical stacking
np.concatenate([A, B], axis=1)  # Horizontal stacking
np.vstack([A, B])               # Vertical stack (shorthand)
np.hstack([A, B])               # Horizontal stack (shorthand)
```

6. Vectorized Implementations for Neural Networks

```
# Forward propagation for a layer
def forward_layer(X, W, b):
    Z = np.dot(X, W) + b
    A = 1 / (1 + np.exp(-Z))  # Sigmoid activation
    return A
```

```
# Vectorized forward pass for entire dataset
def forward_pass(X, W1, b1, W2, b2):
    # First layer
    Z1 = np.dot(X, W1) + b1
    A1 = np.tanh(Z1)

    # Second layer
    Z2 = np.dot(A1, W2) + b2
    A2 = 1 / (1 + np.exp(-Z2))  # Sigmoid output

    return A1, A2

# Vectorized backward pass
def backprop(X, Y, A1, A2, W2):
    m = X.shape[0]

    # Output layer gradient
    dZ2 = A2 - Y
    dW2 = (1/m) * np.dot(A1.T, dZ2)
    db2 = (1/m) * np.sum(dZ2, axis=0)

    # Hidden layer gradient
    dZ1 = np.dot(dZ2, W2.T) * (1 - np.power(A1, 2))  # tanh derivative
    dW1 = (1/m) * np.dot(X.T, dZ1)
    db1 = (1/m) * np.sum(dZ1, axis=0)

    return dW1, db1, dW2, db2
```

7. Optimized Cost Function Computation

```
# Binary cross-entropy loss
def compute_cost(A2, Y):
    m = Y.shape[0]
    cost = (-1/m) * np.sum(Y * np.log(A2) + (1 - Y) * np.log(1 - A2))
    return cost

# MSE loss
```

```
def compute_mse(predictions, targets):
    return np.mean(np.square(predictions - targets))
```

8. Mini-batch Processing

```
# Create mini-batches
def create_mini_batches(X, Y, batch_size):
    m = X.shape[0]
    mini_batches = []

    # Shuffle
    permutation = np.random.permutation(m)
    X_shuffled = X[permutation]
    Y_shuffled = Y[permutation]

    # Create mini-batches
    complete_batches = m // batch_size
    for k in range(complete_batches):
        mini_batch_X = X_shuffled[k * batch_size:(k + 1) * batch_size]
        mini_batch_Y = Y_shuffled[k * batch_size:(k + 1) * batch_size]
        mini_batch = (mini_batch_X, mini_batch_Y)
        mini_batches.append(mini_batch)

    # Handle remaining examples
    if m % batch_size != 0:
        mini_batch_X = X_shuffled[complete_batches * batch_size:]
        mini_batch_Y = Y_shuffled[complete_batches * batch_size:]
        mini_batch = (mini_batch_X, mini_batch_Y)
        mini_batches.append(mini_batch)

    return mini_batches
```

9. Data Normalization

```
# Standard normalization
def normalize(X):
    mean = np.mean(X, axis=0)
    std = np.std(X, axis=0)
    return (X - mean) / std

# Min-max scaling
def min_max_scale(X):
    min_vals = np.min(X, axis=0)
    max_vals = np.max(X, axis=0)
    return (X - min_vals) / (max_vals - min_vals)
```

10. Optimization Techniques

```
# Gradient descent update
def update_params(W, b, dW, db, learning_rate):
    W = W - learning_rate * dW
    b = b - learning_rate * db
    return W, b

# Gradient descent with momentum
def update_with_momentum(W, b, dW, db, vW, vb, learning_rate, beta):
    # Update velocity
    vW = beta * vW + (1 - beta) * dW
    vb = beta * vb + (1 - beta) * db

    # Update parameters
    W = W - learning_rate * vW
    b = b - learning_rate * vb

    return W, b, vW, vb
```

Broadcasting in Python

Broadcasting is a powerful NumPy mechanism that allows arrays with different shapes to be combined in arithmetic operations. It enables you to perform operations between arrays of different dimensions without explicitly creating copies of data.

How Broadcasting Works

Broadcasting follows specific rules to determine how arrays of different shapes can interact:

1. **Rule 1:** If arrays don't have the same rank (number of dimensions), the shape of the smaller array is padded with ones on the left until both shapes have the same length.
2. **Rule 2:** If the shape of the arrays doesn't match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.
3. **Rule 3:** If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

Broadcasting Examples

Example 1: Scalar and Array

```
import numpy as np

# Adding a scalar to each element of an array
arr = np.array([1, 2, 3, 4, 5])
result = arr + 10

print(result)  # Output: [11 12 13 14 15]
```

Here, the scalar 10 is broadcast to match the shape of arr, effectively becoming [10, 10, 10, 10, 10].

Example 2: Vector and Matrix

```
# Create a 3×4 matrix
matrix = np.array([
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12]
])

# Create a 1D array (vector)
vector = np.array([10, 20, 30, 40])
```

```
# Add the vector to each row of the matrix
result = matrix + vector

print(result)

# Output:
# [[11 22 33 44]
#  [15 26 37 48]
#  [19 30 41 52]]
```

In this case, vector with shape (4,) is broadcast to shape (3, 4) to match matrix.

Example 3: Column Vector and Matrix

```
# Same 3×4 matrix
matrix = np.array([
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12]
])

# Create a column vector
column_vector = np.array([[100], [200], [300]])  # Shape (3, 1)

# Add the column vector to each column of the matrix
result = matrix + column_vector

print(result)
# Output:
# [[101 102 103 104]
#  [205 206 207 208]
#  [309 310 311 312]]
```

Here, column_vector with shape (3, 1) is broadcast to shape (3, 4) to match matrix.

Example 4: Complex Broadcasting

```
# 3D array with shape (3, 4, 5)
array_3d = np.zeros((3, 4, 5))

# 1D array with shape (5,)
array_1d = np.array([1, 2, 3, 4, 5])

# Broadcasting works across multiple dimensions
result = array_3d + array_1d  # Shape remains (3, 4, 5)

# The 1D array is broadcast to all sub-matrices
print(result[0, 0])  # Output: [1. 2. 3. 4. 5.]
print(result[1, 2])  # Output: [1. 2. 3. 4. 5.]
```

Benefits of Broadcasting

- **Memory efficiency:** No need to physically copy data to create arrays of matching shapes
- **Performance:** Operations are optimized at the C level for speed
- **Clean code:** Reduces the need for explicit loops
- **Flexibility:** Makes it easy to perform operations between arrays of different shapes

Common Broadcasting Use Cases in Deep Learning

- **Adding biases to layers:** Adding a bias vector to each example in a batch
- **Normalizing data:** Subtracting mean and dividing by standard deviation across features
- **Element-wise operations:** Applying transformations to specific dimensions of tensors
- **Attention mechanisms:** Computing attention scores between queries and keys

> Broadcasting is one of the most powerful features of NumPy that helps make vectorized code concise and efficient. Understanding broadcasting is essential for writing optimized deep learning algorithms.