CSC 4315: INTRODUCTION TO ARTIFICIAL INTELLIGENCE

Week 4, Lecture 8
Gradient Descent and Cost Functions

Lecture Overview

Gradient descent is the fundamental optimization algorithm used to train most machine learning models. This lecture explores how gradient descent works, its variants, and how to implement it from scratch for linear regression.

Topics Covered:

- Optimization in machine learning
- The gradient descent algorithm
- Learning rate and convergence
- Batch, stochastic, and mini-batch gradient descent
- Implementing gradient descent for linear regression
- Visualizing the optimization process

1. Optimization in Machine Learning

1.1 The Optimization Problem

Training a machine learning model is an optimization problem: we want to find the parameters that minimize the cost function. For linear regression, we seek w and b that minimize MSE.

```
import numpy as np

print("Optimization in Machine Learning")
print("="*70)

print("The Goal:")
print("  Find parameters (w, b) that minimize the cost function J(w, b)")
print("\nFor Linear Regression:")
print("  J(w, b) = (1/2m) * sum((h(x_i) - y_i)^2)")
print("  Where h(x) = w*x + b")

# Sample data
np.random.seed(42)
X = np.array([1, 2, 3, 4, 5], dtype=float)
y = np.array([2.1, 4.0, 5.8, 8.1, 9.9], dtype=float)

def compute_cost(X, y, w, b):
    """Compute MSE cost function"""
    m = len(y)
    predictions = w * X + b
    cost = (1 / (2 * m)) * np.sum((predictions - y) ** 2)
    return cost

# Explore the cost surface
print("\nExploring Cost Function:")
print(f"Data: X = {X}, y = {y}")
print("\n{:^10} {:^10} {:^12}".format("w", "b", "Cost"))
print("-" * 35)

for w in [0.5, 1.0, 1.5, 2.0, 2.5]:
    for b in [-0.5, 0.0, 0.5]:
        cost = compute_cost(X, y, w, b)
        print(f"{w:^10.1f} {b:^10.1f} {cost:^12.4f}")
```

```
Optimization in Machine Learning
======================================================================
The Goal:
  Find parameters (w, b) that minimize the cost function J(w, b)

For Linear Regression:
  J(w, b) = (1/2m) * sum((h(x_i) - y_i)^2)
  Where h(x) = w*x + b

Exploring Cost Function:
Data: X = [1. 2. 3. 4. 5.], y = [2.1 4.  5.8 8.1 9.9]

    w          b          Cost
-----------------------------------
   0.5       -0.5        7.7720
   0.5        0.0        5.6620
   0.5        0.5        3.9520
   1.0       -0.5        2.7720
   1.0        0.0        1.4620
   1.0        0.5        0.5520
   1.5       -0.5        0.7720
   1.5        0.0        0.2620
```

1.5	0.5	0.1520
2.0	-0.5	1.7720
2.0	0.0	2.0620
2.0	0.5	2.7520
2.5	-0.5	5.7720
2.5	0.0	6.8620
2.5	0.5	8.3520

2. The Gradient Descent Algorithm

2.1 Intuition: Finding the Minimum

Imagine standing on a hilly landscape in fog, trying to reach the lowest point. Gradient descent works by taking steps in the direction of steepest descent (negative gradient).

```
# Gradient Descent Intuition
print("Gradient Descent Intuition")
print("="*70)

print("Algorithm Overview:")
print("  1. Start with initial parameter values (random or zeros)")
print("  2. Compute the gradient (direction of steepest ascent)")
print("  3. Update parameters in the opposite direction (descent)")
print("  4. Repeat until convergence")

print("\nUpdate Rule:")
print("  w := w - alpha * (dJ/dw)")
print("  b := b - alpha * (dJ/db)")
print("\nWhere:")
print("  - alpha: learning rate (step size)")
print("  - dJ/dw: partial derivative of cost with respect to w")
print("  - dJ/db: partial derivative of cost with respect to b")

print("\nKey Insight:")
print("  - If gradient is positive, decrease parameter")
print("  - If gradient is negative, increase parameter")
print("  - This moves us toward the minimum!")
```

```
Gradient Descent Intuition
======================================================================
Algorithm Overview:
  1. Start with initial parameter values (random or zeros)
  2. Compute the gradient (direction of steepest ascent)
  3. Update parameters in the opposite direction (descent)
  4. Repeat until convergence

Update Rule:
  w := w - alpha * (dJ/dw)
  b := b - alpha * (dJ/db)

Where:
  - alpha: learning rate (step size)
  - dJ/dw: partial derivative of cost with respect to w
  - dJ/db: partial derivative of cost with respect to b

Key Insight:
  - If gradient is positive, decrease parameter
  - If gradient is negative, increase parameter
  - This moves us toward the minimum!
```

2.2 Computing Gradients

```
# Computing Gradients for Linear Regression
print("Computing Gradients for Linear Regression")
print("="*70)

print("Mathematical Derivation:")
print("  Cost: J(w,b) = (1/2m) * sum((wx + b - y)^2)")
print("\nPartial Derivatives:")
print("  dJ/dw = (1/m) * sum((wx + b - y) * x)")
print("  dJ/db = (1/m) * sum((wx + b - y))")

def compute_gradients(X, y, w, b):
    """
    Compute gradients for linear regression

    Args:
        X: Input features
        y: Target values
        w: Current weight
        b: Current bias

    Returns:
        dw: Gradient with respect to w
        db: Gradient with respect to b
    """
    m = len(y)

    # Predictions
    predictions = w * X + b

    # Errors
    errors = predictions - y

    # Gradients
    dw = (1 / m) * np.sum(errors * X)
    db = (1 / m) * np.sum(errors)

    return dw, db

# Example: compute gradients at w=1.0, b=0.0
w, b = 1.0, 0.0
dw, db = compute_gradients(X, y, w, b)

print(f"\nAt w={w}, b={b}:")
print(f"  Cost: {compute_cost(X, y, w, b):.4f}")
print(f"  dJ/dw = {dw:.4f}")
print(f"  dJ/db = {db:.4f}")

print("\nInterpretation:")
print(f"  - dw < 0 means: increasing w will decrease cost")
print(f"  - db < 0 means: increasing b will decrease cost")
```

```
Computing Gradients for Linear Regression
======================================================================
Mathematical Derivation:
  Cost: J(w,b) = (1/2m) * sum((wx + b - y)^2)

Partial Derivatives:
  dJ/dw = (1/m) * sum((wx + b - y) * x)
  dJ/db = (1/m) * sum((wx + b - y))

At w=1.0, b=0.0:
  Cost: 1.4620
```

```
  dJ/dw = -1.5400
  dJ/db = -0.1400

Interpretation:
  - dw < 0 means: increasing w will decrease cost
  - db < 0 means: increasing b will decrease cost
```

3. Learning Rate

3.1 Effect of Learning Rate

```
# Effect of Learning Rate
print("Effect of Learning Rate")
print("="*70)

def gradient_descent_demo(X, y, w_init, b_init, alpha, n_iterations):
    """Run gradient descent and return history"""
    w, b = w_init, b_init
    history = {'w': [w], 'b': [b], 'cost': [compute_cost(X, y, w, b)]}

    for i in range(n_iterations):
        dw, db = compute_gradients(X, y, w, b)
        w = w - alpha * dw
        b = b - alpha * db
        cost = compute_cost(X, y, w, b)
        history['w'].append(w)
        history['b'].append(b)
        history['cost'].append(cost)

    return w, b, history

print("Starting from w=0, b=0, running 10 iterations:")
print("-" * 60)

# Test different learning rates
learning_rates = [0.01, 0.1, 0.5]

for alpha in learning_rates:
    w, b, hist = gradient_descent_demo(X, y, 0.0, 0.0, alpha, 10)
    print(f"\nLearning rate = {alpha}:")
    print(f"  Final w: {w:.4f}, b: {b:.4f}")
    print(f"  Final cost: {hist['cost'][-1]:.6f}")
    print(f"  Cost history: {[f'{c:.4f}' for c in hist['cost'][:5]]}...")

print("\nKey Observations:")
print("  - Too small (0.01): Slow convergence")
print("  - Good (0.1): Steady progress toward minimum")
print("  - Too large: May overshoot or diverge")
```

```
Effect of Learning Rate
======================================================================
Starting from w=0, b=0, running 10 iterations:
------------------------------------------------------------

Learning rate = 0.01:
  Final w: 0.2810, b: 0.0117
  Final cost: 3.998216
  Cost history: ['5.9360', '5.4776', '5.0551', '4.6659', '4.3075']...

Learning rate = 0.1:
  Final w: 1.8191, b: 0.1033
  Final cost: 0.085647
  Cost history: ['5.9360', '1.6696', '0.5109', '0.1849', '0.0983']...

Learning rate = 0.5:
  Final w: 1.9593, b: 0.0968
  Final cost: 0.076851
  Cost history: ['5.9360', '0.6700', '0.5508', '0.2058', '0.0996']...

Key Observations:
```

- Too small (0.01): Slow convergence
- Good (0.1): Steady progress toward minimum
- Too large: May overshoot or diverge

4. Complete Gradient Descent Implementation

4.1 Full Implementation

```
# Complete Gradient Descent for Linear Regression
print("Complete Gradient Descent Implementation")
print("="*70)

class LinearRegressionGD:
    """Linear Regression using Gradient Descent"""

    def __init__(self, learning_rate=0.01, n_iterations=1000, verbose=False):
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.verbose = verbose
        self.w = None
        self.b = None
        self.cost_history = []

    def fit(self, X, y):
        """Train using gradient descent"""
        # Initialize parameters
        self.w = 0.0
        self.b = 0.0
        m = len(y)

        for i in range(self.n_iterations):
            # Predictions
            y_pred = self.w * X + self.b

            # Compute gradients
            dw = (1/m) * np.sum((y_pred - y) * X)
            db = (1/m) * np.sum(y_pred - y)

            # Update parameters
            self.w = self.w - self.learning_rate * dw
            self.b = self.b - self.learning_rate * db

            # Track cost
            cost = (1/(2*m)) * np.sum((y_pred - y) ** 2)
            self.cost_history.append(cost)

            # Print progress
            if self.verbose and (i % 100 == 0 or i == self.n_iterations - 1):
                print(f"  Iteration {i:4d}: cost = {cost:.6f}")

        return self

    def predict(self, X):
        return self.w * X + self.b

# Train model
model = LinearRegressionGD(learning_rate=0.1, n_iterations=500, verbose=True)
print("Training model...")
model.fit(X, y)

print(f"\nFinal Parameters:")
print(f"  w = {model.w:.6f}")
print(f"  b = {model.b:.6f}")
print(f"\nPredictions vs Actual:")
predictions = model.predict(X)
for xi, yi, pred in zip(X, y, predictions):
    print(f"  x={xi:.1f}: actual={yi:.1f}, predicted={pred:.2f}")
```

```
Complete Gradient Descent Implementation
======================================================================
Training model...
  Iteration    0: cost = 5.936000
  Iteration  100: cost = 0.076832
  Iteration  200: cost = 0.076832
  Iteration  300: cost = 0.076832
  Iteration  400: cost = 0.076832
  Iteration  499: cost = 0.076832

Final Parameters:
  w = 1.960000
  b = 0.100000

Predictions vs Actual:
  x=1.0: actual=2.1, predicted=2.06
  x=2.0: actual=4.0, predicted=4.02
  x=3.0: actual=5.8, predicted=5.98
  x=4.0: actual=8.1, predicted=7.94
  x=5.0: actual=9.9, predicted=9.90
```

5. Variants of Gradient Descent

5.1 Batch, Stochastic, and Mini-Batch

```
# Gradient Descent Variants
print("Gradient Descent Variants")
print("="*70)

print("1. Batch Gradient Descent:")
print("   - Uses ALL training examples per update")
print("   - Most stable, but slow for large datasets")
print("   - w = w - alpha * (1/m) * sum(gradient)")

print("\n2. Stochastic Gradient Descent (SGD):")
print("   - Uses ONE random example per update")
print("   - Fast but noisy, may not converge smoothly")
print("   - w = w - alpha * gradient_i")

print("\n3. Mini-Batch Gradient Descent:")
print("   - Uses a BATCH of examples (e.g., 32, 64)")
print("   - Balances speed and stability")
print("   - Most commonly used in practice")

# Demonstrate SGD
def sgd_step(X, y, w, b, learning_rate):
    """Single SGD update using random sample"""
    i = np.random.randint(len(y))
    xi, yi = X[i], y[i]

    # Prediction and error for single point
    pred = w * xi + b
    error = pred - yi

    # Update
    w = w - learning_rate * error * xi
    b = b - learning_rate * error

    return w, b

# Run SGD
np.random.seed(42)
w, b = 0.0, 0.0
print("\nSGD Demo (100 updates):")
for i in range(100):
    w, b = sgd_step(X, y, w, b, 0.05)
    if i % 20 == 0:
        cost = compute_cost(X, y, w, b)
        print(f"  Update {i:3d}: w={w:.4f}, b={b:.4f}, cost={cost:.4f}")
```

```
Gradient Descent Variants
======================================================================
1. Batch Gradient Descent:
   - Uses ALL training examples per update
   - Most stable, but slow for large datasets
   - w = w - alpha * (1/m) * sum(gradient)

2. Stochastic Gradient Descent (SGD):
   - Uses ONE random example per update
   - Fast but noisy, may not converge smoothly
   - w = w - alpha * gradient_i

3. Mini-Batch Gradient Descent:
   - Uses a BATCH of examples (e.g., 32, 64)
```

```
   - Balances speed and stability
   - Most commonly used in practice

SGD Demo (100 updates):
  Update   0: w=0.4830, b=0.2300, cost=4.5080
  Update  20: w=1.5954, b=0.2512, cost=0.2149
  Update  40: w=1.8508, b=0.1748, cost=0.0883
  Update  60: w=1.9163, b=0.1345, cost=0.0795
  Update  80: w=1.9440, b=0.1134, cost=0.0774
```

6. Practical Considerations

6.1 Feature Scaling

```
# Feature Scaling for Gradient Descent
print("Feature Scaling")
print("="*70)

# Problem: Features on different scales
print("Problem: Features on different scales")
print("  e.g., size (1000-3000) vs bedrooms (1-5)")
print("  Causes elongated cost surface, slow convergence")

# Create data with different scales
X_multi = np.array([
    [1000, 2],  # size, bedrooms
    [1500, 3],
    [2000, 3],
    [2500, 4],
    [3000, 5]
], dtype=float)

print(f"\nOriginal features:")
print(f"  Feature 1 (size): min={X_multi[:,0].min()}, max={X_multi[:,0].max()}")
print(f"  Feature 2 (beds): min={X_multi[:,1].min()}, max={X_multi[:,1].max()}")

# Standardization: (x - mean) / std
def standardize(X):
    mean = np.mean(X, axis=0)
    std = np.std(X, axis=0)
    return (X - mean) / std, mean, std

X_scaled, mean, std = standardize(X_multi)

print(f"\nAfter standardization:")
print(f"  Feature 1: min={X_scaled[:,0].min():.2f}, max={X_scaled[:,0].max():.2f}")
print(f"  Feature 2: min={X_scaled[:,1].min():.2f}, max={X_scaled[:,1].max():.2f}")

# Min-max scaling: (x - min) / (max - min)
def min_max_scale(X):
    min_val = np.min(X, axis=0)
    max_val = np.max(X, axis=0)
    return (X - min_val) / (max_val - min_val)

X_minmax = min_max_scale(X_multi)
print(f"\nAfter min-max scaling (0-1 range):")
print(f"  Feature 1: min={X_minmax[:,0].min():.2f}, max={X_minmax[:,0].max():.2f}")
print(f"  Feature 2: min={X_minmax[:,1].min():.2f}, max={X_minmax[:,1].max():.2f}")
```

```
Feature Scaling
======================================================================
Problem: Features on different scales
  e.g., size (1000-3000) vs bedrooms (1-5)
  Causes elongated cost surface, slow convergence

Original features:
  Feature 1 (size): min=1000.0, max=3000.0
  Feature 2 (beds): min=2.0, max=5.0

After standardization:
  Feature 1: min=-1.41, max=1.41
  Feature 2: min=-1.34, max=1.34

After min-max scaling (0-1 range):
```

```
Feature 1: min=0.00, max=1.00
Feature 2: min=0.00, max=1.00
```

6.2 Convergence Checking

```
# Checking Convergence
print("Checking Convergence")
print("="*70)

def gradient_descent_with_convergence(X, y, alpha, max_iter=10000, tol=1e-6):
    """Gradient descent with convergence checking"""
    w, b = 0.0, 0.0
    m = len(y)
    prev_cost = float('inf')

    for i in range(max_iter):
        # Predictions and gradients
        y_pred = w * X + b
        dw = (1/m) * np.sum((y_pred - y) * X)
        db = (1/m) * np.sum(y_pred - y)

        # Update
        w = w - alpha * dw
        b = b - alpha * db

        # Check convergence
        cost = (1/(2*m)) * np.sum((y_pred - y) ** 2)
        if abs(prev_cost - cost) < tol:
            print(f"  Converged at iteration {i}")
            break
        prev_cost = cost

    return w, b, i+1

print("Running until convergence (tolerance = 1e-6):")
w, b, iterations = gradient_descent_with_convergence(X, y, 0.1)
print(f"  Final: w={w:.6f}, b={b:.6f}")
print(f"  Iterations: {iterations}")

print("\nConvergence Criteria:")
print("  1. Cost decrease below threshold")
print("  2. Gradient magnitude below threshold")
print("  3. Maximum iterations reached")
print("  4. Cost starts increasing (may need smaller lr)")
```

```
Checking Convergence
======================================================================
Running until convergence (tolerance = 1e-6):
  Converged at iteration 87
  Final: w=1.960000, b=0.100000
  Iterations: 88

Convergence Criteria:
  1. Cost decrease below threshold
  2. Gradient magnitude below threshold
  3. Maximum iterations reached
  4. Cost starts increasing (may need smaller lr)
```

Summary

Key Takeaways:

- Gradient descent iteratively minimizes the cost function
- The learning rate controls step size (too small: slow, too large: diverge)
- Gradient = direction of steepest ascent; we go opposite direction
- Batch GD uses all data, SGD uses one sample, mini-batch is in between
- Feature scaling helps gradient descent converge faster
- Monitor convergence using cost history

Practice Exercises:

1. Implement mini-batch gradient descent
2. Plot the cost function surface and gradient descent path
3. Compare convergence with different learning rates
4. Implement gradient descent with momentum
5. Add early stopping based on validation loss