Fundamental Algorithms

Chapter 3: Five Essential Supervised Learning Algorithms

Gradient Descent

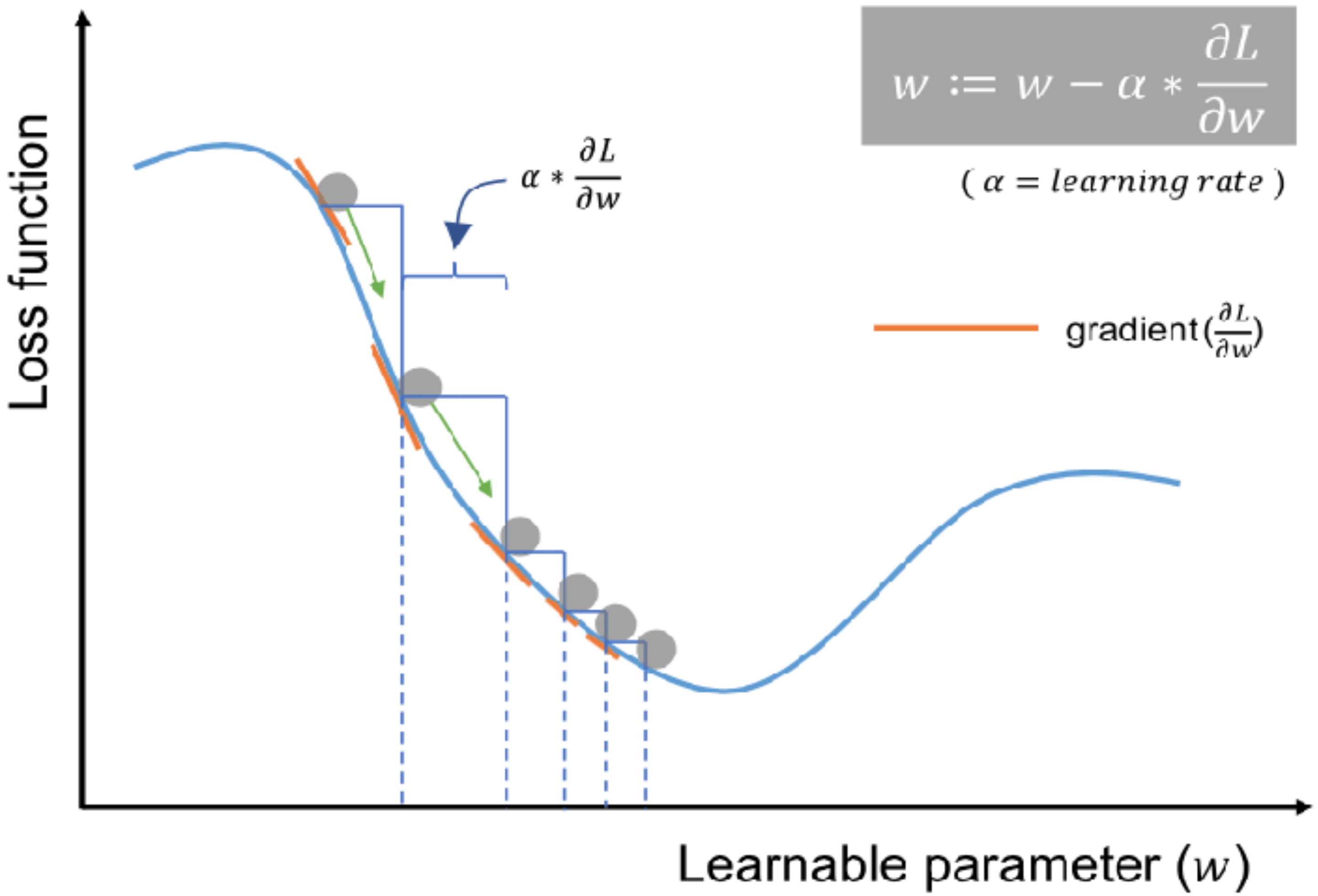
Why Start Here?

By reading modern machine learning literature, you often encounter references to **gradient descent** or **stochastic gradient descent**. These are the two most frequently used optimization algorithms when the optimization criterion is differentiable.

Universal Tool:

- Linear and Logistic Regression
- Support Vector Machines
- Neural Networks
- Most modern ML algorithms

What is Gradient Descent?



What is Gradient Descent?

Definition: An iterative optimization algorithm for finding the minimum of a function.

Core Idea:

- Start at some random point
- Take steps proportional to the **negative gradient**
- Move "downhill" toward the minimum

Mathematical Intuition:

$$\mathbf{w}_{new} = \mathbf{w}_{old} - \alpha \nabla f(\mathbf{w}_{old})$$

where α is the **learning rate** and ∇f is the gradient

Convex vs Non-Convex Functions

Convex Functions (Logistic Regression, Linear Regression, SVM):

- Have only **one minimum** (global)
- Gradient descent guaranteed to find it
- Bowl-shaped optimization landscape

Non-Convex Functions (Neural Networks):

- Multiple local minima
- Finding local minimum often sufficient in practice
- Complex optimization landscape

Key Insight: Even when global optimum isn't guaranteed, gradient descent works remarkably well!

Gradient Descent: Practical Example

Dataset: Company advertising spending vs sales

```
Company | Spendings (Crores) | Sales (Units)
   1    |     37.8           |    22.1
   2    |     39.3           |    10.4
   3    |     45.9           |     9.3
   4    |     41.3           |    18.5
  ...   |      ...           |     ...
```

Goal: Build model $f(x) = wx + b$ to predict sales from spending

Problem: What are optimal values for w and b?

The Mathematical Foundation

Objective: Minimize Mean Squared Error $\ell = \frac{1}{N}\sum_{i=1}^{N}(y_i - (wx_i + b))^2$

Step 1: Calculate partial derivatives (gradients)

$$\frac{\partial \ell}{\partial w} = \frac{1}{N}\sum_{i=1}^{N} -2x_i(y_i - (wx_i + b))$$

$$\frac{\partial \ell}{\partial b} = \frac{1}{N}\sum_{i=1}^{N} -2(y_i - (wx_i + b))$$

Step 2: Update parameters

$$w_i = w_{i-1} - \alpha\frac{\partial \ell}{\partial w}, b_i = b_{i-1} - \alpha\frac{\partial \ell}{\partial b}$$

Chain Rule in Action

Why these derivatives? For term $(y_i - (wx_i + b))^2$ with respect to w:

Chain Rule: $f = f_2(f_1)$ where:

- $f_1 = y_i - (wx_i + b)$
- $f_2 = f_1^2$

Step-by-step:

1. $\frac{\partial f_2}{\partial f_1} = 2f_1 = 2(y_i - (wx_i + b))$
2. $\frac{\partial f_1}{\partial w} = -x_i$
3. $\frac{\partial f}{\partial w} = \frac{\partial f_2}{\partial f_1} \cdot \frac{\partial f_1}{\partial w} = 2(y_i - (wx_i + b)) \cdot (-x_i)$

Result: $\frac{\partial \ell}{\partial w} = \frac{1}{N} \sum_{i=1}^{N} -2x_i(y_i - (wx_i + b))$

Gradient Descent: Python Implementation

```python
def update_w_and_b(spendings, sales, w, b, alpha):
    """
    One epoch of gradient descent
    alpha: learning rate (step size)
    """
    dl_dw = 0.0  # Gradient w.r.t. w
    dl_db = 0.0  # Gradient w.r.t. b
    N = len(spendings)

    # Calculate gradients
    for i in range(N):
        # Prediction error
        error = sales[i] - (w * spendings[i] + b)

        # Accumulate gradients
        dl_dw += -2 * spendings[i] * error
        dl_db += -2 * error

    # Update parameters
    w = w - (1/float(N)) * dl_dw * alpha
    b = b - (1/float(N)) * dl_db * alpha

    return w, b
```

Training Loop and Convergence

```
def train_linear_regression(spendings, sales, epochs=1000, alpha=0.0001):
    # Initialize parameters
    w, b = 0.0, 0.0
    losses = []

    for epoch in range(epochs):
        # Update parameters
        w, b = update_w_and_b(spendings, sales, w, b, alpha)

        # Calculate current loss
        predictions = [w * x + b for x in spendings]
        loss = sum((y - pred)**2 for y, pred in zip(sales, predictions)) / len(sales)
        losses.append(loss)

        # Print progress
        if epoch % 100 == 0:
            print(f"Epoch {epoch}: Loss = {loss:.4f}, w = {w:.4f}, b = {b:.4f}")

    return w, b, losses
```

Training Loop and Convergence

Key Concepts:

- **Iteration** Every time you look at example (01 or 01 batch)
- **Epoch:** One pass through all training examples
- **Convergence:** When w and b stop changing significantly

Learning Rate: The Critical Hyperparameter

Too Small (α too small):

- Very slow convergence, Many epochs needed
- Safe but inefficient

Too Large (α too large):

- Overshooting the minimum, Oscillations or divergence
- Fast but unstable

Just Right:

- Steady decrease in loss
- Reasonable convergence speed
- Stable parameter updates

Variants of Gradient Descent

Three Main Types Based on Training Data Usage:

1. **Batch Gradient Descent (BGD):** - Uses **entire dataset** for each parameter update
   - Most accurate gradients, guaranteed convergence, Slow for large datasets, high memory usage
2. **Stochastic Gradient Descent (SGD):** - Uses **one sample** for each parameter update
   - Fast, low memory, adds beneficial noise, Noisy gradients, may not converge exactly
3. **Mini-Batch Gradient Descent:** - Uses **small batches** (e.g., 32, 64, 128 samples)
   - **Best of both worlds -** most commonly used, Good balance of speed, accuracy, and memory

Batch vs Mini-Batch vs SGD: Comparison

Method	Batch Size	Speed	Memory	Convergence	Noise
Batch GD	Full dataset (N)	Slow	High	Smooth	None
Mini-Batch GD	32-512	Fast	Medium	Stable	Some
SGD	1	Fastest	Low	Noisy	High

Modern Practice:

- **Mini-batch** is the standard (typically 32-256)
- Enables parallelization on GPUs
- Good compromise between accuracy and efficiency

Mini-Batch Implementation

```python
def mini_batch_gradient_descent(X, y, batch_size=32, epochs=100, alpha=0.01):
    n_samples, n_features = X.shape
    w = np.random.normal(0, 0.01, n_features)
    b = 0

    for epoch in range(epochs):
        # Shuffle data each epoch
        indices = np.random.permutation(n_samples)
        X_shuffled = X[indices]
        y_shuffled = y[indices]

        # Process mini-batches
        for i in range(0, n_samples, batch_size):
            # Get current mini-batch
            batch_end = min(i + batch_size, n_samples)
            X_batch = X_shuffled[i:batch_end]
            y_batch = y_shuffled[i:batch_end]

            # Compute gradients on mini-batch
            predictions = X_batch @ w + b
            errors = predictions - y_batch

            dw = (1/len(X_batch)) * X_batch.T @ errors
            db = (1/len(X_batch)) * np.sum(errors)

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

    return w, b
```

Why Mini-Batch Works So Well

Computational Efficiency:

- **Vectorization:** Matrix operations faster than loops
- **GPU Parallelization:** Perfect for modern hardware
- **Memory Management:** Fits in GPU memory

Statistical Benefits:

- **Noise Reduction:** Averaging over batch reduces variance
- **Regularization Effect:** Some noise helps generalization
- **Escape Local Minima:** Noise helps escape shallow minima

Modern Variants of Gradient Descent

Advanced Optimizers:

- **Adagrad:** Adapts learning rate per parameter
- **Momentum:** Accelerates in relevant directions
- **RMSprop:** Addresses Adagrad's learning rate decay
- **Adam:** Combines momentum and adaptive learning rates

Key Insight: These are not ML algorithms - they are **optimization solvers**