# Module 2: Linear Regression and Gradient Descent

## Overview

This module introduces the fundamental algorithm for learning model parameters: gradient descent. Using linear regression as our example, we explore how to define a loss function, compute gradients, and iteratively update parameters to minimize error. These concepts form the basis for training nearly all modern machine learning models.

---

## 1. Linear Regression

### The Model

Linear regression predicts a continuous output as a weighted sum of inputs:

$y = w x + w x + \ldots + w x + b$

Or in vector notation:

$y = w \cdot x + b$

Where: - **x**: Input features (what we observe) - **w**: Weights (learned parameters) - **b**: Bias/intercept (learned parameter) - **y**: Predicted output

### Why Linear?

Linear models are: - Simple to understand and interpret - Fast to train - Often surprisingly effective - Foundation for more complex models

### Limitations

Linear regression assumes a linear relationship between inputs and outputs. If the true relationship is non-linear (curved), linear regression will underfit.

---

## 2. Loss Functions

### Purpose of Loss

A loss function measures how wrong our predictions are. We need this to know: - How good/bad the current model is - Whether we're improving during training - Which direction to adjust parameters

### Mean Squared Error (MSE)

The most common loss for regression:

$MSE = (1/n) \times \Sigma(y\_pred - y\_true)^2$

**Properties:** - Always non-negative (zero is perfect) - Squaring makes all errors positive - Large errors are penalized more than small ones - Differentiable (important for gradient descent)

### The Loss Landscape

Imagine the loss as a surface over the space of all possible parameter values. Training is about finding the lowest point on this surface. For linear regression with MSE, this surface is a convex bowl—there's exactly one minimum.

---

## 3. Gradient Descent

### The Core Idea

Gradient descent is an optimization algorithm that finds parameters minimizing the loss by: 1. Start at a random point in parameter space 2. Compute the gradient (slope) of the loss 3. Take a step in the direction that decreases loss 4. Repeat until convergence

### The Gradient

The gradient is a vector of partial derivatives—it points in the direction of steepest increase. To minimize loss, we move in the **opposite** direction.

For MSE loss:

```
 L/ w = (2/n) × Σ(y_pred - y_true) × x
 L/ b = (2/n) × Σ(y_pred - y_true)
```

### The Update Rule

```
w_new = w_old -   × ( L/ w)
b_new = b_old -   × ( L/ b)
```

Where η (eta) is the learning rate.

---

## 4. Learning Rate

### What It Controls

The learning rate determines the step size in each update: - **Too high**: Steps are too big, may overshoot the minimum and diverge - **Too low**: Steps are too small, convergence is very slow - **Just right**: Smooth, efficient convergence

### Symptoms of Bad Learning Rates

**Too High:** - Loss increases instead of decreases - Loss oscillates wildly - Loss becomes NaN or infinity

**Too Low:** - Loss decreases very slowly - Training takes forever - May get stuck before reaching minimum

### Typical Values

Common starting points: 0.1, 0.01, 0.001. Often requires experimentation.

---

## 5. Gradient Descent Variants

### Batch Gradient Descent

Uses **all** training examples to compute each gradient update.

**Pros:** - Accurate gradient estimate - Smooth, stable convergence - Guaranteed to converge (for convex problems)

**Cons:** - Very slow for large datasets - Must fit all data in memory - One update per pass through data

### Stochastic Gradient Descent (SGD)

Uses **one** randomly selected example per update.

**Pros:** - Very fast updates - Can escape local minima (due to noise) - Works with infinite/streaming data

**Cons:** - Noisy updates, may not converge exactly - High variance in gradient estimates - May oscillate around minimum

### Mini-Batch Gradient Descent

Uses a **small batch** of examples (e.g., 32) per update.

**Pros:** - Balance of speed and stability - Efficient use of GPU parallelism - Most common in practice

**Cons:** - Introduces batch size as hyperparameter - Still some noise in gradients

### Comparison

| Method | Samples per Update | Speed | Stability |
|---|---|---|---|
| Batch | All | Slow | High |
| SGD | 1 | Fast | Low |
| Mini-Batch | 32-256 | Medium | Medium |

---

## 6. Convergence

### When to Stop

Training typically stops when: - Loss stops decreasing (plateaus) - Maximum iterations reached - Validation performance stops improving - Loss reaches acceptable level

### Convergence Criteria

Common approaches: - Loss change below threshold - Gradient magnitude below threshold - Fixed number of epochs

### Local vs Global Minima

For convex problems (like linear regression with MSE), there's only one minimum—global. For non-convex problems (neural networks), there may be many local minima. Modern deep learning research suggests local minima are usually fine.

---

## 7. Parameters vs Hyperparameters

### Parameters

Values **learned from data** during training: - Weights (w) - Biases (b)

These define the model's predictions.

### Hyperparameters

Values **set before training** by the practitioner: - Learning rate - Number of iterations - Batch size - Model architecture

These control how training proceeds.

---

## 8. The Training Loop

### Pseudocode

```
# Initialize
weights = random values
bias = 0

for iteration in range(max_iterations):
    # Forward pass
    predictions = X @ weights + bias

    # Compute loss
    loss = mean((predictions - y)^2)
    # Compute gradients
    grad_w = (2/n) * X.T @ (predictions - y)
    grad_b = (2/n) * sum(predictions - y)

    # Update parameters
    weights = weights - learning_rate * grad_w
    bias = bias - learning_rate * grad_b
```

### Key Components

1. **Forward pass**: Compute predictions with current parameters
2. **Loss computation**: Measure how wrong we are
3. **Backward pass**: Compute gradients (how to change parameters)
4. **Update**: Adjust parameters to reduce loss

---

## Key Takeaways

1. **Linear regression**: y = w·x + b, simple but powerful
2. **Loss function**: Measures prediction error (use MSE for regression)
3. **Gradient**: Vector pointing toward steepest increase
4. **Gradient descent**: Move opposite to gradient to minimize loss
5. **Learning rate**: Controls step size (critical hyperparameter)
6. **Batch/SGD/Mini-batch**: Tradeoffs between speed and stability
7. **Convergence**: When loss stops decreasing meaningfully

---

## Connections to Future Modules

- **Module 3**: What if we have many features? □ Multivariate regression
- **Module 4**: What about classification? □ Logistic regression (same gradient descent)
- **Module 6**: What about non-linear functions? □ Neural networks
- **Module 7**: What about non-gradient methods? □ Decision trees

---

## Further Reading

- [Links to foundational papers will be added]
- Sebastian Ruder: “An Overview of Gradient Descent Optimization Algorithms”
- Deep Learning Book, Chapter 4: Numerical Computation