Machine Learning Lab

Session 0-1: Function Minimization & Linear Regression

BFA3 – Université Paris-Dauphine

M2 Level

Duration: 2h30 (30 min lecture + 2h lab)

Date: _____

Learning Objectives

By the end of this lab, you will be able to:

1. Understand function minimization from both analytical and numerical perspectives
2. Implement the gradient descent algorithm from scratch
3. Visualize how gradient descent converges (or fails to converge) under different conditions
4. Apply gradient descent to solve the linear regression problem
5. Compare closed-form solutions with iterative numerical methods
6. Understand the impact of the step size α on convergence behavior

Prerequisites

Before starting this lab, you should be comfortable with:

- **Calculus:** Derivatives, partial derivatives, gradients, chain rule
- **Linear Algebra:** Matrix multiplication, transpose, inverse, solving linear systems
- **Python:** Basic syntax, functions, loops, conditionals
- **NumPy:** Array operations, basic linear algebra functions

Files Provided

- `ml_lab_gui.py` – The main GUI application (do not modify)
- `my_lib.py` – The library file where you will implement `gradient_descent`

To run the application:

```
python ml_lab_gui.py
```

Part I: Theoretical Background

1 Function Minimization: The Core Problem

Many problems in machine learning can be formulated as **optimization problems**: we want to find the parameters that minimize (or maximize) some objective function. Understanding how to solve these problems is fundamental to understanding how learning algorithms work.

Definition: Optimization Problem

Given a function $f : \mathbb{R}^n \to \mathbb{R}$, we want to find:

$$x^* = \arg\min_{x \in \mathcal{D}} f(x)$$

where $\mathcal{D} \subseteq \mathbb{R}^n$ is the domain of optimization.
The point x^* is called a **minimizer** and $f(x^*)$ is the **minimum value**.

1.1 Analytical Approach: Finding Critical Points

For a sufficiently smooth function, we can find minima by analyzing where the derivative equals zero.

Theorem: First-Order Necessary Condition

Let $f : \mathbb{R}^n \to \mathbb{R}$ be a differentiable function (C^1). If x^* is a local minimum of f, then:

$$\nabla f(x^*) = 0$$

where ∇f denotes the gradient of f:

$$\nabla f(x) = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}$$

Theorem: Second-Order Sufficient Condition

Let $f : \mathbb{R}^n \to \mathbb{R}$ be twice differentiable (C^2). If $\nabla f(x^*) = 0$ and the Hessian matrix $H_f(x^*)$ is **positive definite**, then x^* is a **strict local minimum**.
The Hessian matrix is:

$$H_f(x) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix}$$

Example (1D case): Consider $f(x) = (x-2)^2 + 1$.

- First derivative: $f'(x) = 2(x-2)$
- Setting $f'(x) = 0$: $2(x-2) = 0 \Rightarrow x^* = 2$
- Second derivative: $f''(x) = 2 > 0$ (positive, so it's a minimum)
- Minimum value: $f(2) = 1$

Warning

The analytical approach requires solving $\nabla f(x) = 0$, which may be:

- **Impossible:** No closed-form solution exists
- **Computationally expensive:** For high-dimensional problems
- **Ambiguous:** Multiple critical points (minima, maxima, saddle points)

This motivates **numerical methods** like gradient descent.

2 Numerical Approach: Gradient Descent

When we cannot solve $\nabla f(x) = 0$ analytically, we use **iterative numerical methods**. The most fundamental of these is **gradient descent**.

2.1 The Key Insight: Gradient Points to Steepest Ascent

Theorem: Gradient Direction

For a differentiable function f at point x, the gradient $\nabla f(x)$ points in the direction of **steepest ascent** of f.
Conversely, $-\nabla f(x)$ points in the direction of **steepest descent**.

Proof sketch: Consider moving from x in direction v (unit vector). The rate of change is:

$$\frac{d}{dt} f(x + tv)\Big|_{t=0} = \nabla f(x) \cdot v = \|\nabla f(x)\|\|v\| \cos\theta = \|\nabla f(x)\| \cos\theta$$

This is maximized when $\theta = 0$ (i.e., v aligned with $\nabla f(x)$) and minimized when $\theta = \pi$ (i.e., v opposite to $\nabla f(x)$).

2.2 The Gradient Descent Algorithm

The idea is simple: *repeatedly take small steps in the direction of steepest descent.*

Algorithm 1 Gradient Descent

Require: Function f, derivative f', step size α, initial point x_0, tolerance ϵ, max iterations N
Ensure: Approximate minimizer x^*

```
x ← x_0
for i = 1 to N do
    g ← f'(x)                          ▷ Compute gradient
    if |g| < ε then                    ▷ Check convergence
        break
    end if
    x ← x − α · g                      ▷ Update step
end for
return x
```

Definition: Gradient Descent Update Rule

At each iteration k, update the current position:

$$\boxed{x_{k+1} = x_k - \alpha \nabla f(x_k)}$$

where:

- x_k is the current position
- $\alpha > 0$ is the **step size** (or **learning rate**)
- $\nabla f(x_k)$ is the gradient at the current position
- x_{k+1} is the new position

2.3 The Role of Step Size α

The step size α is **crucial** for the behavior of gradient descent:

α **value**	**Behavior**	**Consequence**
Too small	Very slow progress	Many iterations needed, may timeout
Just right	Smooth convergence	Efficient minimization
Too large	Overshooting	Oscillation or divergence

2.4 Convex vs. Non-Convex Functions

Definition: Convex Function

A function $f : \mathbb{R}^n \to \mathbb{R}$ is **convex** if for all x, y in its domain and $t \in [0, 1]$:

$$f(tx + (1 - t)y) \leq tf(x) + (1 - t)f(y)$$

Geometrically: the line segment between any two points on the graph lies above the graph.

Key property: For convex functions, **every local minimum is a global minimum**. This means gradient descent (with appropriate α) is guaranteed to find the global minimum.

Non-convex functions (like our "egg-box" functions) have multiple local minima. Gradient descent may get stuck in a local minimum that is not the global minimum. The starting point and step size significantly affect which minimum is found.

3 Application: Linear Regression

Linear regression is one of the most fundamental machine learning algorithms. It provides an excellent case study for comparing analytical and numerical optimization approaches.

3.1 Problem Definition

Definition: Linear Regression Problem

Given a dataset of n observations $\{(x_i, y_i)\}_{i=1}^n$ where $x_i \in \mathbb{R}$ and $y_i \in \mathbb{R}$, find the line $\hat{y} = ax + b$ that best fits the data.
The parameters to find are:

- a – the **slope**
- b – the **intercept**

3.2 The Objective Function: Ordinary Least Squares (OLS)

We define "best fit" as minimizing the sum of squared errors:

Definition: OLS Objective Function

$$\mathcal{L}(a, b) = \frac{1}{n}\sum_{i=1}^{n}(y_i - (ax_i + b))^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

This is the **Mean Squared Error (MSE)** between predictions and actual values.

3.3 Closed-Form Solution

In matrix notation, with $\mathbf{y} = (y_1, \ldots, y_n)^T$ and design matrix:

$$A = \begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{pmatrix}$$

we seek $\boldsymbol{\theta} = (a, b)^T$ such that $A\boldsymbol{\theta} \approx \mathbf{y}$.

Theorem: Normal Equations

The OLS solution is:

$$\boxed{\boldsymbol{\theta}^* = (A^T A)^{-1} A^T \mathbf{y}}$$

This is obtained by setting $\nabla_{\boldsymbol{\theta}} \mathcal{L} = 0$ and solving.

3.4 Gradient Descent Solution

Alternatively, we can use gradient descent. The gradient of $\mathcal{L}$ with respect to the slope a is:

$$\frac{\partial \mathcal{L}}{\partial a} = \frac{2}{n}\sum_{i=1}^{n}(y_i - ax_i - b)(-x_i) = -\frac{2}{n}\sum_{i=1}^{n} x_i(y_i - ax_i - b)$$

Similarly for b:

$$\frac{\partial \mathcal{L}}{\partial b} = -\frac{2}{n}\sum_{i=1}^{n}(y_i - ax_i - b)$$

The gradient descent update rules become:

$$a_{k+1} = a_k - \alpha\frac{\partial \mathcal{L}}{\partial a}$$
$$b_{k+1} = b_k - \alpha\frac{\partial \mathcal{L}}{\partial b}$$

3.5 Trade-offs: Closed-Form vs. Gradient Descent

Aspect	**Closed-Form**	**Gradient Descent**
Computation	Matrix inversion $O(n^3)$	Iterations $O(kn)$
Memory	Store A^TA	Store gradient only
Exactness	Exact solution	Approximate solution
Applicability	Only when $(A^TA)^{-1}$ exists	Always applicable
Scalability	Poor for large n	Good for large n

Part II: Implementation

4 Your Task: Implement Gradient Descent

You will implement the `gradient_descent` function in `my_lib.py`. This function will be used by the GUI application to visualize the optimization process.

4.1 Function Specification

```
def gradient_descent(func, domain, derivative, alpha,
                     max_iterations=1000, epsilon=1e-6):
    """
    Find the minimum of a function using gradient descent.

    Args:
        func: The function to minimize (callable)
        domain: Tuple (a, b) representing the compact domain [a, b]
        derivative: The derivative function of func (callable)
        alpha: Step size (learning rate)
        max_iterations: Maximum number of iterations (default: 1000)
        epsilon: Convergence threshold (default: 1e-6)

    Returns:
        List of x values [x0, x1, ..., x_final] showing the path
        to minimum. The last element is the found minimum.
    """
```

Listing 1: Function signature and documentation

4.2 Student Template

Open `my_lib.py` and complete the following implementation:

```
import random

def gradient_descent(func, domain, derivative, alpha,
                     max_iterations=1000, epsilon=1e-6):
    """
    Find the minimum of a function using gradient descent.

    Args:
        func: The function to minimize (callable)
        domain: Tuple (a, b) representing the compact domain [a, b]
        derivative: The derivative function of func (callable)
        alpha: Step size (learning rate)
        max_iterations: Maximum number of iterations (default: 1000)
        epsilon: Convergence threshold (default: 1e-6)

    Returns:
        List of x values [x0, x1, ..., x_final] showing the path
        to minimum. The last element is the found minimum.
    """
    a, b = domain

    # Initialize at a random point in the domain
    x = random.uniform(a + 0.1 * (b - a), b - 0.1 * (b - a))

```

```python
    # Store the path of all visited points
    path = [x]

    for i in range(max_iterations):
        # ============================================================
        # TODO 1: Compute the gradient at current position
        # ============================================================
        # Hint: Use the 'derivative' function passed as argument
        grad = _____________  # YOUR CODE HERE

        # ============================================================
        # TODO 2: Check for convergence
        # ============================================================
        # Hint: If the absolute value of the gradient is smaller
        #       than epsilon, we have converged. Break the loop.
        if _____________:  # YOUR CODE HERE
            break

        # ============================================================
        # TODO 3: Apply the gradient descent update rule
        # ============================================================
        # Hint: x_new = x - alpha * gradient
        x_new = _____________  # YOUR CODE HERE

        # Project back onto domain if we stepped outside
        x_new = max(a, min(b, x_new))

        # Add new point to path
        path.append(x_new)

        # Check if we're stuck (not moving anymore)
        if abs(x_new - x) < 1e-10:
            break

        # Update current position for next iteration
        x = x_new

    return path
```

Listing 2: Student template with gaps to fill

4.3 Step-by-Step Implementation Guide

Hint

TODO 1 – Compute the gradient:
The gradient tells us the slope of the function at the current point. Since `derivative` is a function passed as an argument, you simply need to call it with the current position `x`.
Mathematical notation: $g = f'(x_k)$ or $g = \nabla f(x_k)$
Python: `grad = derivative(x)`

Hint

TODO 2 – Check for convergence:
We consider the algorithm converged when the gradient is very close to zero (meaning we're at or near a critical point).
Mathematical condition: $|g| < \epsilon$
Python: `if abs(grad) < epsilon:`

Hint

TODO 3 – Apply the update rule:
This is the core of gradient descent. We move in the opposite direction of the gradient, scaled by the step size α.
Mathematical formula: $x_{k+1} = x_k - \alpha \cdot g$
Python: `x_new = x - alpha * grad`

4.4 Testing Your Implementation

Before using the GUI, test your implementation with this simple example:

```python
# Test with a simple quadratic function
def f(x):
    return (x - 2)**2  # Minimum at x = 2

def df(x):
    return 2 * (x - 2)  # Derivative

# Run gradient descent
path = gradient_descent(f, domain=(-5, 5), derivative=df, alpha=0.1)

# Check result
print(f"Number of iterations: {len(path)}")
print(f"Found minimum at x = {path[-1]:.6f}")
print(f"Expected minimum at x = 2.000000")
print(f"f(x*) = {f(path[-1]):.6f}")
```

Listing 3: Test your implementation

Expected output:

```
Number of iterations: ~50-100
Found minimum at x = 2.000000 (approximately)
f(x*) = 0.000000 (approximately)
```

Part III: Experimentation & Discussion

Now that your implementation is working, use the GUI application to explore gradient descent behavior. Run the application with:

```
python ml_lab_gui.py
```

5 Panel 1: Function Optimization

5.1 Experiment 1: Impact of Step Size on Convex Functions

1. Select **"Convex"** function type
2. Try the following values of α and observe the behavior:
   - $\alpha = 0.01$ (very small)
   - $\alpha = 0.1$ (moderate)
   - $\alpha = 0.5$ (large)
   - $\alpha = 1.0$ (very large)
3. For each value, click **"Run All"** and note:
   - Number of iterations
   - Path taken to the minimum
   - Final value of $f(x)$

Discussion Questions

Q1. How does the number of iterations change as α increases? Why?

Q2. What happens when α is too large? Can you explain this behavior mathematically?

Q3. For convex functions, does gradient descent always find the global minimum? Why or why not?

5.2 Experiment 2: Impact of Step Size on Non-Convex Functions

1. Select **"Egg-box"** function type
2. Try the following values of α:
   - $\alpha = 0.01$
   - $\alpha = 0.05$
   - $\alpha = 0.1$
3. Click **"Run Step"** repeatedly to observe the algorithm step-by-step
4. Run multiple times with the same α (click "Reset" between runs)

Discussion Questions

Q4. Does gradient descent always find the global minimum for egg-box functions? What does it find instead?

Q5. Run the algorithm 5 times with the same α on an egg-box function. Do you always get the same result? Why or why not?

Q6. How does changing α affect which local minimum the algorithm finds?

6 Panel 2: Linear Regression

6.1 Experiment 3: Gradient Descent for Linear Regression

1. Switch to **Panel 2: Linear Regression**
2. Click **"Generate Dataset"** to create a new dataset
3. Observe the two plots:
   - Top plot: OLS objective function $\mathcal{L}(a)$ as a function of slope
   - Bottom plot: Dataset with regression lines
4. Try different values of α and click "Run All"
5. Use "Run Step" to watch the regression line evolve

Discussion Questions

Q7. Is the OLS objective function convex? How can you tell from the plot?

Q8. Watch the regression line evolve as gradient descent runs. Describe how the line changes from iteration to iteration.

Q9. What value of α gives good convergence for the OLS problem? Is it the same as for the simple convex functions in Panel 1?

7 Synthesis Questions

Discussion Questions

Q10. Compare the number of iterations needed for gradient descent on:

- A simple convex function (Panel 1)
- The OLS objective function (Panel 2)

What factors might explain the difference?

Q11. In practice, how would you choose a good value for α? Propose a strategy.

Q12. What are the advantages of using gradient descent over the closed-form solution for linear regression? When would you prefer one over the other?

A Complete Solution (For Instructors Only)

```python
import random

def gradient_descent(func, domain, derivative, alpha,
                     max_iterations=1000, epsilon=1e-6):
    """
    Find the minimum of a function using gradient descent.
    """
    a, b = domain

    # Initialize at a random point in the domain
    x = random.uniform(a + 0.1 * (b - a), b - 0.1 * (b - a))

    # Store the path of all visited points
    path = [x]

    for i in range(max_iterations):
        # Compute the gradient at current position
        grad = derivative(x)

        # Check convergence
        if abs(grad) < epsilon:
            break

        # Gradient descent update
        x_new = x - alpha * grad

        # Project back onto domain
        x_new = max(a, min(b, x_new))

        # Add new point to path
        path.append(x_new)

        # Check if stuck
        if abs(x_new - x) < 1e-10:
            break

        x = x_new

    return path
```

Listing 4: Complete implementation of gradient_descent