Chapter 3: Calculus and Optimization Essentials for Machine Learning

Master in Computer Science (M1) — Applications with Real Data in Python

Youssef SALMAN

Academic Year 2025–2026

Overview and Motivation

Optimization turns modeling choices into *decisions*: given a loss/objective and constraints, we search for parameters that minimize the loss while satisfying the constraints. In machine learning (ML), parameters are model weights, the loss encodes prediction error plus regularization, and constraints capture domain or resource limits. This chapter collects essential calculus and optimization ideas and connects them to practical Python implementations used in ML.

1 Introduction to Limits, Continuity, and Derivatives

Limit, Continuity, Derivative

For $f : \mathbb{R} \to \mathbb{R}$, the limit $\lim_{x\to a} f(x)$ is the value L approached by $f(x)$ as x approaches a. The function is continuous at a if $\lim_{x\to a} f(x) = f(a)$. The derivative is $f'(a) = \lim_{h\to 0} \dfrac{f(a+h) - f(a)}{h}$ when the limit exists.

Rules (single-variable). Linearity, product/quotient rules, and chain rule $(f \circ g)'(x) = f'(g(x))\, g'(x)$.

Why ML cares. Derivatives quantify *marginal change*: how much the loss changes for an infinitesimal change of a weight. They enable gradient-based training.

2 Partial Derivatives and Gradients

Partial Derivatives, Gradient, Directional Derivative

For $f : \mathbb{R}^d \to \mathbb{R}$ and $x = (x_1, \ldots, x_d)$, the partial derivative w.r.t. x_i is

$$\frac{\partial f}{\partial x_i}(x) = \lim_{h \to 0} \frac{f(x_1, \ldots, x_i + h, \ldots, x_d) - f(x)}{h}.$$

The *gradient* is

$$\nabla f(x) = \left(\frac{\partial f}{\partial x_1}(x), \ldots, \frac{\partial f}{\partial x_d}(x) \right)^\top.$$

For a unit direction $u \in \mathbb{R}^d$, the *directional derivative* is

$$D_u f(x) = \nabla f(x)^\top u.$$

These quantities are fundamental in gradient methods and first-order optimality conditions.

Chain rule (vector form). If $f = g \circ h$ with $h : \mathbb{R}^p \to \mathbb{R}^m$ and $g : \mathbb{R}^m \to \mathbb{R}$ differentiable, then $\nabla f(x) = [J_h(x)]^\top \nabla g(h(x))$.

ML example (MSE). For $J(w) = \frac{1}{2n} \|Xw - y\|_2^2$, $\nabla J(w) = \frac{1}{n} X^\top (Xw - y)$ (linear regression).

3 Multivariate Calculus Essentials

Setup and notation. Let $f : \mathbb{R}^p \to \mathbb{R}$ and $g : \mathbb{R}^p \to \mathbb{R}^m$. We use the inner product $\langle u, v \rangle = u^\top v$ and norm $\|x\| = \sqrt{x^\top x}$. The gradient is $\nabla f(x) \in \mathbb{R}^p$, the Jacobian is $J_g(x) \in \mathbb{R}^{m \times p}$, and the Hessian is $H_f(x) \in \mathbb{R}^{p \times p}$ with entries $[H_f(x)]_{ij} = \partial^2 f / \partial x_i \partial x_j$.

Directional derivative and differential

For $u \in \mathbb{R}^p$, the directional derivative of f at x in direction u is

$$D_u f(x) = \lim_{t \downarrow 0} \frac{f(x + tu) - f(x)}{t}.$$

If f is differentiable at x, then $D_u f(x) = \nabla f(x)^\top u$ for all u. Equivalently, the first differential is the linear map $Df(x)[u] = \nabla f(x)^\top u$.

Jacobian and chain rule

For $g = (g_1, \dots, g_m)$, the Jacobian $J_g(x)$ has rows $\nabla g_i(x)^\top$. If $h = g \circ F$ with $F : \mathbb{R}^p \to \mathbb{R}^m$ and $g : \mathbb{R}^m \to \mathbb{R}^r$ are differentiable, then

$$J_h(x) = J_g\Big(F(x)\Big)\, J_F(x).$$

For scalar f, this gives $\nabla(f \circ F)(x) = J_F(x)^\top \nabla f\Big(F(x)\Big)$.

Hessian and symmetry

If all second partial derivatives of f are continuous near x, then $H_f(x)$ is symmetric (Clairaut's theorem). For any u, the second directional derivative is $D_u^2 f(x) = u^\top H_f(x)\, u$.

Taylor expansions

If f is differentiable,

$$f(x+s) = f(x) + \nabla f(x)^\top s + o(\|s\|).$$

If f is twice differentiable,

$$f(x+s) = f(x) + \nabla f(x)^\top s + \tfrac{1}{2} s^\top H_f(x) s + o(\|s\|^2).$$

This quadratic model bridges to second-order tests used later.

Critical points and the Hessian test (recap).
Critical points satisfy $\nabla f(x^\star) = 0$.

- If $H_f(x^\star) \succ 0$ we have a strict local minimum;
- If $H_f(x^\star) \prec 0$ a strict local maximum;
- If $H_f(x^\star)$ is indefinite, a saddle point.

Smoothness and Lipschitz gradient

We say f has L-Lipschitz gradient if $\|\nabla f(x) - \nabla f(y)\| \le L\|x - y\|$ for all x, y. Equivalently,

$$f(y) \le f(x) + \nabla f(x)^\top (y - x) + \tfrac{L}{2}\|y - x\|^2.$$

This inequality will control step sizes in the optimization section.

Convexity and strong convexity (calculus view)

A function f is convex on a convex set $\mathcal{D}$ if $f(\theta x + (1-\theta)y) \le \theta f(x) + (1-\theta) f(y)$ for all $x, y \in \mathcal{D}$, $\theta \in [0, 1]$. If f is twice differentiable, then on $\mathcal{D}$:

$$\text{convex} \iff H_f(x) \succeq 0, \qquad \text{strongly convex with parameter } \mu > 0 \iff H_f(x) \succeq \mu I.$$

Strong convexity implies a unique minimizer.

Level sets and gradients

The level set at height c is $\mathcal{L}_c = \{x : f(x) = c\}$. At points where $\nabla f(x) \neq 0$, the gradient is orthogonal to $\mathcal{L}_{f(x)}$ and points in the direction of steepest increase; $-\nabla f(x)$ is steepest decrease.

Useful derivatives for machine learning

Let $A \in \mathbb{R}^{n\times p}$, $x, w \in \mathbb{R}^p$, $a \in \mathbb{R}^p$, $b \in \mathbb{R}^n$.

$$\nabla_x\,(a^\top x) = a, \qquad \nabla_x\,\tfrac{1}{2}\|x\|^2 = x,$$
$$\nabla_w\,\tfrac{1}{2}\|Aw-b\|^2 = A^\top(Aw-b), \qquad H_w\,\tfrac{1}{2}\|Aw-b\|^2 = A^\top A,$$
$$\text{If } f(t) = \log(1+e^{-t}),\ \sigma(t) = \tfrac{1}{1+e^{-t}},\ \text{then } \nabla_w f\Big(y\,w^\top x\Big) = -\,y\,\sigma(-y\,w^\top x)\,x.$$

These identities are reused when deriving gradients of ML losses.

Worked example (to reuse later)

For $f(x_1, x_2) = x_1^2 + 2x_2^2 + e^{-x_1-x_2}$,

$$\nabla f(x_1,x_2) = \begin{bmatrix} 2x_1 - e^{-x_1-x_2} \\ 4x_2 - e^{-x_1-x_2} \end{bmatrix}, \quad H_f(x_1,x_2) = \begin{bmatrix} 2+e^{-x_1-x_2} & e^{-x_1-x_2} \\ e^{-x_1-x_2} & 4+e^{-x_1-x_2} \end{bmatrix}.$$

The Hessian is symmetric and positive definite for all (x_1, x_2), implying a unique minimizer.

4 Optimization Algorithms

What is Optimization?

Optimization Problem

Given $f : \mathbb{R}^d \to \mathbb{R}$ and a feasible set $C \subseteq \mathbb{R}^d$, the goal is to solve

$$\min_{x\in C} f(x).$$

In ML, f is typically the empirical risk plus regularization, and x denotes the model parameters.

4.1 Gradient Descent Variants: Batch, Stochastic, Mini-batch

In supervised learning, we have:

- Training data: $(x_i, y_i)_{i=1}^n$ with inputs x_i and labels y_i.
- A model $f_w(x)$ with parameters w (vector of weights).
- A loss $\ell(f_w(x), y)$ measuring prediction error.

The **empirical risk** is $J(w) = \frac{1}{n}\sum_{i=1}^{n} \ell(f_w(x_i), y_i)$. Gradient descent updates w by

$$w_{k+1} = w_k - \eta\, g_k,$$

where g_k is an estimate of $\nabla J(w_k)$. The three variants differ only in how g_k is computed.

Batch Gradient Descent (full-batch GD)

Uses **all** n **samples** at each step:

$$g_k = \frac{1}{n}\sum_{i=1}^{n} \nabla_w \ell(f_w(x_i), y_i).$$

How it works: Each update moves parameters in the exact steepest-descent direction.
Pros. Stable and accurate steps. **Cons.** Slow when n is large.

Python (Gradient Descent on y

```
import numpy as np
import matplotlib.pyplot as plt

# Function y = x^2
def y_function(x):
    return x ** 2

# Derivative y' = 2x
def y_derivative(x):
    return 2 * x

# Values for plotting the curve
x = np.arange(-100, 100, 0.1)
y = y_function(x)

# Starting point (far from the minimum)
current_pos = (50, y_function(50))

# Learning rate
learning_rate = 0.01

# Make plot interactive
plt.ion()

# Gradient descent iterations
for _ in range(1000):
    # Gradient descent update
    new_x = current_pos[0] - learning_rate * y_derivative(current_pos[0])
    new_y = y_function(new_x)
    current_pos = (new_x, new_y)

    # Plot function and current point
    plt.plot(x, y)
```

```
    plt.scatter(current_pos[0], current_pos[1], color="red")

    plt.pause(0.001) # short delay for animation
    plt.clf() # clear figure for next step

plt.ioff()
plt.show()
```

Why this example is full-batch Gradient Descent: At each iteration, the update uses the *exact* derivative $f'(x) = 2x$ of $f(x) = x^2$, with no sampling noise. Hence it is full-batch.

Stochastic Gradient Descent (SGD)

Uses **one random sample** (x_{i_k}, y_{i_k}) per step:

$$g_k = \nabla_w \ell(f_w(x_{i_k}), y_{i_k}).$$

How it works: Updates are very fast but noisy.
Pros. Extremely cheap per step. **Cons.** Noisy path; requires many updates.

Python (SGD on y

```
import numpy as np
import matplotlib.pyplot as plt

# Target curve to visualize: f(x) = x^2 (up to an irrelevant constant)
def f(x):
    return x**2

# One-sample stochastic gradient via L_z(x) = (x - Z)^2 with Z ~ N(0,1)
rng = np.random.default_rng(0)

# Plot grid (for the parabola)
xx = np.arange(-100, 100, 0.1)
yy = f(xx)

# Initialize
x = 80.0 # starting point
eta = 0.01 # learning rate
steps = 1000 # number of SGD updates

plt.ion() # interactive mode for animation

for t in range(steps):
    z = rng.normal(0.0, 1.0) # one random sample Z ~ N(0,1)
    g = 2 * (x - z) # stochastic gradient at current x (E[g|x]=2x)
    x = x - eta * g # SGD update
```

```
    # Draw frame: curve + current point (noisy path)
    plt.plot(xx, yy)
    plt.scatter(x, f(x), color="red")
    plt.title(f"SGD on x^2 | iter={t} x={x:.2f}")
    plt.pause(0.001)
    plt.clf()

plt.ioff()
plt.show()
```

Key difference (same $f(x) = x^2$). Full-batch uses $g = 2x$; SGD uses one-sample $g = 2(x - z)$ with $z \sim \mathcal{N}(0, 1)$; mini-batch averages m samples:

$$g = \frac{2}{m} \sum_{j=1}^{m} (x - z_j), \qquad z_j \overset{\text{i.i.d.}}{\sim} \mathcal{N}(0, 1).$$

All are unbiased for $2x$, but variance decreases with $1/m$.

Mini-batch Gradient Descent

Uses a **small subset of m samples** $B_k \subset \{1, \ldots, n\}$:

$$g_k = \frac{1}{m} \sum_{i \in B_k} \nabla_w \ell(f_w(x_i), y_i).$$

How it works: Trades off speed and variance; GPU-friendly.
Pros. More stable than SGD, faster than full batch. **Cons.** Still random; depends on m.

Python (Mini-batch GD on y

```
import numpy as np
import matplotlib.pyplot as plt

# Target function (for visualization)
def f(x):
    return x**2

# --- Visualization setup ---
xx = np.arange(-100, 100, 0.1)
yy = f(xx)

# Mini-batch SGD hyperparameters
x = 80.0 # start at x0 = 80
eta = 0.01 # learning rate
steps = 600 # number of updates
m = 32 # mini-batch size
rng = np.random.default_rng(0) # reproducible randomness

plt.ion() # interactive mode for animation
```

```
for t in range(steps):
    # Draw a mini-batch Z_1,...,Z_m ~ N(0,1)
    z_batch = rng.normal(0.0, 1.0, size=m)

    # Mini-batch gradient: g = (2/m) * sum(x - z_j) = 2 * (x - mean(z))
    g = 2.0 * (x - z_batch.mean())

    # Update
    x = x - eta * g

    # Draw current frame
    plt.plot(xx, yy)
    plt.scatter(x, f(x), color="red")
    plt.xlim(-100, 100)
    plt.ylim(0, 10000)
    plt.xlabel("x"); plt.ylabel("f(x)=x^2")
    plt.title(f"Mini-batch GD on x^2 | m={m} | iter={t} | x={x:.2f} | g
        =(2/m)sum(x - z_j)")
    plt.pause(0.001)
    plt.clf()

plt.ioff()
plt.show()
```

Same objective, different gradients. All three use $x \leftarrow x - \eta\, g$; they differ only in computing g: `g = 2*x` (full-batch), `g = 2*(x - z)` (SGD), `g = 2*(x - mean(z_batch))` (mini-batch).

4.2 Newton's Method

Newton's Method (idea)

For smooth $J : \mathbb{R}^p \to \mathbb{R}$ at w_k, let $g_k = \nabla J(w_k)$ and $H_k = \nabla^2 J(w_k)$. The Newton step solves $H_k\, s_k = g_k$ and updates $w_{k+1} = w_k - s_k$.

Python (one Newton step for logistic regression)

```
import numpy as np

def sigmoid(z): return 1.0/(1.0 + np.exp(-z))

def logreg_grad(X, y, w, lam=0.0):
    p = sigmoid(X @ w)
    return (X.T @ (p - y))/len(y) + lam*w

def logreg_hess(X, y, w, lam=0.0):
    p = sigmoid(X @ w)
    W = p*(1-p) # length-n
```

```
    H = (X.T * W) @ X / len(y) + lam*np.eye(X.shape[1])
    return H

def newton_update(X, y, w, lam=0.0):
    g = logreg_grad(X, y, w, lam)
    H = logreg_hess(X, y, w, lam)
    delta = np.linalg.solve(H, g)
    return w - delta
```

5 Convex Optimization

Convex set/function

A set $C \subset \mathbb{R}^d$ is convex if $tx + (1-t)y \in C$ for all $x, y \in C$ and $t \in [0,1]$. A function $f : C \to \mathbb{R}$ is convex if $f(tx + (1-t)y) \le tf(x) + (1-t)f(y)$ for all $x, y \in C$.

Equivalent characterizations (differentiable case).

If f is differentiable on a convex domain, then f is convex iff

$$f(y) \;\ge\; f(x) \;+\; \nabla f(x)^\top (y - x).$$

If f is twice differentiable, f is convex iff $H_f(x) \succeq 0$ for all x; f is μ-strongly convex iff $H_f(x) \succeq \mu I$.

Subgradients (nonsmooth convex f).

A vector g is a *subgradient* at x if

$$f(y) \;\ge\; f(x) \;+\; g^\top (y - x) \quad \forall\, y.$$

The set $\partial f(x)$ of all such g is the *subdifferential* (e.g., $f(t) = |t|$, $\partial f(0) = [-1, 1]$).

Optimality conditions (convex problems).

Unconstrained convex f: $x^\star$ is a global minimizer iff $0 \in \partial f(x^\star)$ (differentiable: $\nabla f(x^\star) = 0$). Constrained convex $\min_{x \in C} f(x)$ with closed convex C:

$$x^\star \in C \text{ and } 0 \in \nabla f(x^\star) + N_C(x^\star),$$

equivalently $x^\star = \mathrm{proj}_C\big(x^\star - \eta\, \nabla f(x^\star)\big)$ for any $\eta > 0$.

Useful consequences. Sublevel sets $\{x : f(x) \le c\}$ of convex f are convex. Strong convexity implies uniqueness and quadratic growth: $f(x) \ge f(x^\star) + \frac{\mu}{2}\|x - x^\star\|^2$.

Example (convex quadratic)

Let $Q = Q^\top \succeq 0$, $c \in \mathbb{R}^d$. The function

$$f(x) = \tfrac{1}{2} x^\top Q x + c^\top x$$

is convex with $\nabla f(x) = Qx + c$ and $H_f(x) = Q$. If $Q \succeq \mu I$ with $\mu > 0$, then f is strongly convex.

Python (Projected Gradient on a convex quadratic with box constraints)

```
import numpy as np
import matplotlib.pyplot as plt

# Convex quadratic: f(x) = 0.5 x^T Q x + c^T x (Q PSD)
Q = np.array([[3.0, 1.0],
              [1.0, 2.0]])
c = np.array([-2.0, 1.0])

def f(x):
    return 0.5 * x.T @ Q @ x + c.T @ x

def grad(x):
    return Q @ x + c

# Projection onto a box C = [a1,b1] x [a2,b2] (component-wise clipping)
a = np.array([-2.0, -2.0])
b = np.array([ 2.0, 2.0])
def proj_box(z):
    return np.minimum(np.maximum(z, a), b)

# Step size: 1 / L with L = largest eigenvalue of Q (L-smooth gradient)
L = np.max(np.linalg.eigvals(Q).real)
eta = 1.0 / L

# Initialize inside the box
x = np.array([1.8, -1.5])
xs = [x.copy()] # store iterates for plotting
for k in range(100):
    x = proj_box(x - eta * grad(x)) # projected gradient step
    xs.append(x.copy())

# ---- Plot contours and the path ----
xs = np.array(xs)
grid_x = np.linspace(-2.2, 2.2, 200)
grid_y = np.linspace(-2.2, 2.2, 200)
X, Y = np.meshgrid(grid_x, grid_y)
Z = 0.5*(Q[0,0]*X**2 + 2*Q[0,1]*X*Y + Q[1,1]*Y**2) + c[0]*X + c[1]*Y

plt.figure()
```

```
cs = plt.contour(X, Y, Z, levels=30)
plt.clabel(cs, inline=1, fontsize=8)
plt.plot(xs[:,0], xs[:,1], marker='o', ms=3, lw=1.5, label='Projected GD
    path')
plt.axvline(a[0], color='k', ls='--', lw=0.8); plt.axvline(b[0], color='k
    ', ls='--', lw=0.8)
plt.axhline(a[1], color='k', ls='--', lw=0.8); plt.axhline(b[1], color='k
    ', ls='--', lw=0.8)
plt.xlim(-2.2, 2.2); plt.ylim(-2.2, 2.2)
plt.xlabel('x1'); plt.ylabel('x2'); plt.legend()
plt.title('Projected Gradient on a convex quadratic with box constraints'
    )
plt.gca().set_aspect('equal', 'box')
plt.show()
```

Conclusion. We focused on calculus objects (gradients, Jacobians, Hessians, smoothness, convexity) that power Python implementations of optimization. On the toy $f(x) = x^2$, we showed full-batch, stochastic, and mini-batch descent differ only in how the gradient estimate is formed (exact, one-sample, averaged). For constraints, projections give a simple fixed-point view of first-order steps. These pieces are the backbone of practical ML training.