STAT424: Optimization with Statistical Applications

Donlapark Ponnoprat

Contents

1 Calculus Review for Optimization

1.1 Introduction: Why Calculus?

Welcome to the course! At its core, optimization is the process of finding the *best* possible solution from a set of available alternatives. In the context of statistics and machine learning, this often means finding the parameters of a model that best fit a given dataset. For example, in linear regression, we want to find the slope and intercept that produce the line with the smallest overall error.

Calculus, the mathematics of change, provides the fundamental toolkit for optimization. It gives us a precise way to answer questions like:

- If we change a model parameter slightly, does our model get better or worse?
- Which direction of change will improve our model the most?
- How do we know when we have found the absolute best set of parameters?

In this lecture, we will review the key concepts from single and multi-variable calculus that form the bedrock of almost every optimization algorithm we will encounter.

1.2 Single-Variable Calculus

Let's start with a function of a single variable, $f : \mathbb{R} \to \mathbb{R}$.

First Order Derivative and Monotonicity

The first derivative of a function, denoted $f'(x)$ or $\frac{df}{dx}$, measures the instantaneous rate of change of the function at a point x. Geometrically, it represents the **slope of the tangent line** to the function's graph at that point.

There is a fundamental relationship between the sign of the derivative and the behavior of the function:

Definition 1.1 (Increasing and Decreasing Functions)**.** *A differentiable function f is said to be **increasing** at a point x_0 if $f'(x_0) > 0$. Conversely, it is **decreasing** at x_0 if $f'(x_0) < 0$.*

Intuitively, if $f'(x_0) > 0$, a small step to the right increases the function value, while a small step to the left decreases it. This logic is crucial for optimization algorithms: if we want to minimize f and we are at a point where $f'(x) > 0$, we know we should move to the left.

Defining Optimality

Before we calculate solutions, we must strictly define what "best" means.

Definition 1.2 (Global and Local Minima)**.** *Let f be a function defined on a domain D.*

- *A point x^* is a **global minimum** if $f(x^*) \leq f(x)$ for all $x \in D$.*
- *A point x^* is a **local minimum** if there exists a neighborhood around x^* (e.g., an open interval $(x^* - \delta, x^* + \delta)$) such that $f(x^*) \leq f(x)$ for all x in that neighborhood.*

Maximums are defined similarly by reversing the inequalities.

Critical Points

At a local minimum or maximum, the function cannot be increasing ($f' > 0$) nor decreasing ($f' < 0$). It must be momentarily "flat". This gives us a necessary condition for optimality.

Definition 1.3 (Critical Point)**.** *A point x_0 is a **critical point** of a differentiable function f if $f'(x_0) = 0$.*

Example 1.1 (Finding a critical point)**.** *Consider the simple quadratic function $f(x) = (x-3)^2$. This function measures the squared distance from a point x to the number 3. We know intuitively that the minimum should occur at $x = 3$, where the function value is 0. Let's verify this using calculus.*

Proof: *First, we compute the derivative of $f(x)$:*

$$f'(x) = \frac{d}{dx}(x-3)^2 = 2(x-3) \cdot 1 = 2x - 6.$$

To find the critical points, we set the derivative to zero and solve for x:

$$f'(x) = 0 \implies 2x - 6 = 0 \implies 2x = 6 \implies x = 3.$$

Thus, $x = 3$ is the only critical point of the function.

```python
import jax
import jax.numpy as jnp

# Define the function f(x) = (x-3)^2
def f(x):
  return (x - 3.0)**2

# Use jax.grad to get a new function that computes the derivative of f
```

```
grad_f = jax.grad(f)

# Evaluate the derivative at the critical point x=3
critical_point = 3.0
derivative_at_3 = grad_f(critical_point)

print(f"The function f(x) = (x-3)^2")
print(f"The derivative at x={critical_point} is: {derivative_at_3}")

# We can also evaluate it at another point, e.g., x=5
derivative_at_5 = grad_f(5.0)
print(f"The derivative at x=5.0 is: {derivative_at_5}")
```
Listing 1: Computing the first derivative in JAX.

Second Order Derivative

Knowing that $f'(x) = 0$ isn't enough to determine if a point is a minimum, a maximum, or neither (like an inflection point in $f(x) = x^3$). We need to understand the function's **curvature** at that point, which is given by the second derivative, $f''(x)$.

- If $f''(x) > 0$, the function is **concave up** (shaped like a valley $\cup$). A critical point here is a **local minimum**.
- If $f''(x) < 0$, the function is **concave down** (shaped like a hill $\cap$). A critical point here is a **local maximum**.
- If $f''(x) = 0$, the test is inconclusive.

Example 1.2 (Classifying the critical point)**.** *Let's continue with our function $f(x) = (x-3)^2$. We found the critical point at $x = 3$. Now let's classify it.*

Proof: *We compute the second derivative of $f(x)$:*

$$f''(x) = \frac{d}{dx}(f'(x)) = \frac{d}{dx}(2x-6) = 2.$$

Since $f''(x) = 2$ for all x, it is also true for our critical point: $f''(3) = 2$. Because $f''(3) > 0$, the function is concave up at this point, which confirms that $x = 3$ is a local minimum. (In this case, it is also the global minimum).

```
import jax
import jax.numpy as jnp

# Define the function f(x) = (x-3)^2
def f(x):
  return (x - 3.0)**2

# To get the second derivative, we compose jax.grad with itself
# grad_f computes the first derivative
grad_f = jax.grad(f)
# grad_grad_f computes the derivative of the derivative
grad_grad_f = jax.grad(grad_f)

# Evaluate the second derivative at the critical point x=3
critical_point = 3.0
second_derivative_at_3 = grad_grad_f(critical_point)

print(f"The function is f(x) = (x-3)^2")
print(f"The second derivative at x={critical_point} is: {second_derivative_at_3
   }")
print("Since f''(3) > 0, the critical point is a local minimum.")
```
Listing 2: Computing the second derivative by applying 'grad' twice.

1.3 Multivariate Calculus

In statistics and machine learning, our objective functions almost always depend on many parameters (e.g., the weights of a neural network). So, we need to extend our calculus tools to functions of multiple variables, $f : \mathbb{R}^n \to \mathbb{R}$.

Partial Derivatives and Gradient Vector

For a function of multiple variables, like $f(x, y)$, we can no longer talk about the slope. Instead, we have a slope in every possible direction. The simplest directions are along the coordinate axes.

Definition 1.4 (Partial Derivative)**.** *The* ***partial derivative*** *of f with respect to a variable (e.g., x) is the derivative of the function with respect to that variable, treating all other variables as constants. It is denoted $\frac{\partial f}{\partial x}$.*

We can collect all the partial derivatives into a single vector. This vector is called the gradient.

Definition 1.5 (The Gradient)**.** *The* ***gradient*** *of a function $f(x_1, \ldots, x_n)$, denoted ∇f, is the vector of its partial derivatives:*

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}.$$

The gradient is the single most important concept in multi-variable optimization.

- **Geometric Interpretation:** The gradient vector $\nabla f(x)$ at a point x points in the direction of the **steepest ascent** of the function.
- **Optimization Insight:** Consequently, the negative gradient, $-\nabla f(x)$, points in the direction of **steepest descent**. This is the core idea behind the *gradient descent* algorithm.

A critical point in the multi-variable case is where the gradient is the zero vector, $\nabla f(x) = \mathbf{0}$. This means the function is "flat" in all axial directions at that point.

Example 1.3 (The Gradient of a Paraboloid)**.** *Consider the function $f(x, y) = x^2 + y^2$. This function describes a paraboloid centered at the origin. We can intuitively see that its minimum is at $(0, 0)$. Let's find its gradient.*

Proof: *We first compute the partial derivatives:*

$$\frac{\partial f}{\partial x} = 2x \quad \textit{and} \quad \frac{\partial f}{\partial y} = 2y.$$

The gradient is therefore the vector:

$$\nabla f(x, y) = \begin{bmatrix} 2x \\ 2y \end{bmatrix}.$$

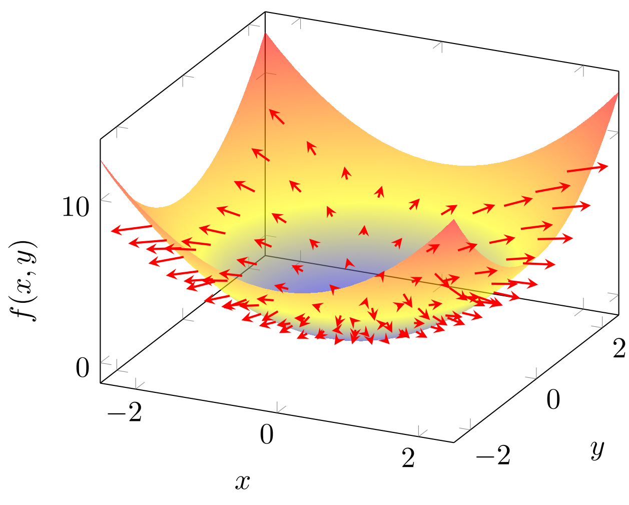
To find the critical point, we set the gradient to the zero vector:

$$\begin{bmatrix} 2x \\ 2y \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

which has the unique solution $(x, y) = (0, 0)$, as expected.

Visualization: *The figure below shows the surface and gradient field of $f(x, y)$. Notice how each gradient vector originates from a point on the surface and points horizontally in the direction of steepest ascent.*

Surface Plot of $f(x, y) = x^2 + y^2$ with Gradient Field



```
import jax
import jax.numpy as jnp

# Define the function f(x) where x is a vector [x1, x2]
def paraboloid(x):
  return x[0]**2 + x[1]**2

# jax.grad computes the gradient with respect to the first argument
grad_paraboloid = jax.grad(paraboloid)

# Find the gradient at the critical point (0, 0)
critical_point = jnp.array([0.0, 0.0])
gradient_at_0 = grad_paraboloid(critical_point)
print(f"The gradient at {critical_point} is {gradient_at_0}")

# Find the gradient at another point (3, 4)
point = jnp.array([3.0, 4.0])
gradient_at_point = grad_paraboloid(point)
print(f"The gradient at {point} is {gradient_at_point}")
print("This vector [6., 8.] points in the direction of steepest ascent.")
```

Listing 3: Computing the gradient of a multi-variable function in JAX.

The Hessian Matrix

Just as with single-variable functions, we need a way to test for curvature. This is the role of the **Hessian matrix**, which is the multi-variable equivalent of the second derivative.

Definition 1.6 (The Hessian)**.** *The **Hessian matrix** of f, denoted $\nabla^2 f$, is the matrix of all second-order partial derivatives:*

$$\nabla^2 f(x) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$

For most well-behaved functions in statistics, this matrix is symmetric.

The classification of critical points depends on the definiteness of the Hessian, which we will explore in detail in the next chapter using matrix algebra.

1.4 Taylor Approximations

Most objective functions in the real world are complex. We can't find their minimum directly. The central idea of many optimization algorithms (like gradient descent and Newton's method) is to approximate the complex function with a simpler one (a line or a parabola) in the neighborhood of our current guess, and then find the minimum of that simpler approximation. These simple approximations come from Taylor's theorem.

- **First-Order Taylor Approximation (Linear):** This approximates a function near a point a with its tangent plane. It's the basis for gradient descent.

$$f(x) \approx f(a) + \nabla f(a)^T (x - a)$$

- **Second-Order Taylor Approximation (Quadratic):** This provides a better approximation by also matching the function's curvature. It's the basis for Newton's method.

$$f(x) \approx f(a) + \nabla f(a)^T (x - a) + \frac{1}{2}(x - a)^T \nabla^2 f(a)(x - a)$$

2 Matrix Calculus and Linear Algebra for Optimization

2.1 Introduction: From Single Variables to Datasets

In Lecture 1, we reviewed the calculus of single and multi-variable functions. However, in modern statistics and machine learning, we rarely work with individual numbers. Instead, we work with entire datasets, which are naturally represented by **vectors** and **matrices**.

Linear algebra provides the language to describe operations on these objects, while matrix calculus gives us the tools to differentiate functions involving them. This combination is incredibly powerful, as it allows us to derive optimization steps for complex models that operate on millions of data points, all in a compact and elegant notation.

In this lecture, we will review the key linear algebra concepts and matrix calculus rules needed to set up and solve real-world statistical optimization problems.

2.2 A Quick Review of Linear Algebra

Let's establish our notation. We will assume all vectors are **column vectors** unless stated otherwise.

- A vector $x \in \mathbb{R}^n$ is a column of n real numbers: $x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$.
- The transpose of x, denoted x^T, is a row vector: $x^T = \begin{bmatrix} x_1 & \cdots & x_n \end{bmatrix}$.
- The **dot product** between two vectors $x, y \in \mathbb{R}^n$ is $x^T y = \sum_{i=1}^n x_i y_i$.
- A matrix $A \in \mathbb{R}^{m \times n}$ is a grid of numbers with m rows and n columns.
- The **Euclidean norm** (or ℓ_2-norm) of a vector x is $||x||_2 = \sqrt{x^T x} = \sqrt{\sum_{i=1}^n x_i^2}$. When the context is clear, we may simply write $||x||$.

2.3 Matrix Calculus

Matrix calculus is a notation for computing derivatives of functions whose inputs or outputs are vectors and matrices. The key is to be consistent with shapes. We will adopt the **denominator layout**, where the derivative of a function with respect to a vector takes the same shape as that vector.

Definition 2.1 (Gradient Shape Convention). *If $f : \mathbb{R}^n \to \mathbb{R}$ is a function that maps a vector to a scalar, its gradient $\nabla f(x)$ is a column vector in $\mathbb{R}^n$.*

$$\nabla_x f(x) = \frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}.$$

Below are the most important gradient identities for our course.

2.4 Gradient of a Linear Form

Let $a \in \mathbb{R}^n$ be a constant vector. Consider the linear function $f(x) = a^T x = \sum_{i=1}^n a_i x_i$.

The gradient is:

$$\nabla_x (a^T x) = a$$

Proof: We compute the partial derivative with respect to an arbitrary component x_k:

$$\frac{\partial f}{\partial x_k} = \frac{\partial}{\partial x_k}\left(\sum_{i=1}^{n} a_i x_i\right) = \frac{\partial}{\partial x_k}(a_1x_1 + \cdots + a_kx_k + \cdots + a_nx_n) = a_k.$$

Since this is true for every component k, stacking them into a vector gives:

$$\nabla_x f(x) = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = a.$$

```
import jax
import jax.numpy as jnp

# Define the function f(x, a) = a^T x
# JAX differentiates with respect to the first argument by default.
def linear_form(x, a):
  return jnp.dot(a, x)

# Create a gradient function for f with respect to x (arg 0)
grad_linear_form = jax.grad(linear_form, argnums=0)

# Define some data
key = jax.random.PRNGKey(0)
a_vec = jax.random.normal(key, (5,))
x_vec = jax.random.normal(key, (5,))

# Compute the gradient
gradient = grad_linear_form(x_vec, a_vec)

print(f"The vector a is:\n{a_vec}")
print(f"The computed gradient of a^T x w.r.t x is:\n{gradient}")
print(f"Are they equal? {jnp.allclose(gradient, a_vec)}")
```

Listing 4: Verifying the gradient of a linear form in JAX.

2.5 Gradient of a Quadratic Form

Let $A \in \mathbb{R}^{n\times n}$ be a constant matrix. Consider the quadratic function $f(x) = x^T Ax$.

The gradient is:

$$\nabla_x(x^T Ax) = (A + A^T)x$$

If A is a symmetric matrix (i.e., $A = A^T$), this simplifies to:

$$\nabla_x(x^T Ax) = 2Ax \quad \text{(if } A \text{ is symmetric)}$$

Proof: First, let's write out the quadratic form using summations:

$$f(x) = x^T Ax = \sum_{i=1}^{n}\sum_{j=1}^{n} A_{ij}x_ix_j.$$

Now we take the partial derivative with respect to a component x_k. We need to be careful, as

x_k appears in the sum when $i = k$ and when $j = k$.

$$\begin{aligned}
\frac{\partial f}{\partial x_k} &= \frac{\partial}{\partial x_k}\left(\sum_{i=1}^{n}\sum_{j=1}^{n} A_{ij}x_i x_j\right) \\
&= \sum_{i=1}^{n}\sum_{j=1}^{n} A_{ij}\frac{\partial}{\partial x_k}(x_i x_j) \\
&= \sum_{i=1}^{n}\sum_{j=1}^{n} A_{ij}(\delta_{ik}x_j + x_i\delta_{jk}) \quad (\text{where } \delta_{ik} \text{ is 1 if } i = k,\ 0 \text{ otherwise}) \\
&= \sum_{j=1}^{n} A_{kj}x_j + \sum_{i=1}^{n} A_{ik}x_i.
\end{aligned}$$

The first sum, $\sum_{j=1}^{n} A_{kj}x_j$, is the k-th element of the vector Ax. The second sum, $\sum_{i=1}^{n} A_{ik}x_i$, is the k-th element of the vector $A^T x$. Therefore, the k-th element of the gradient is $(Ax)_k+(A^T x)_k$. Stacking these for all k gives the vector result:

$$\nabla_x f(x) = Ax + A^T x = (A + A^T)x.$$

```
import jax
import jax.numpy as jnp

# Define the function f(x, A) = x^T A x
def quadratic_form(x, A):
  return x.T @ A @ x

# Create a gradient function for f with respect to x
grad_quadratic_form = jax.grad(quadratic_form, argnums=0)

# Define some data
key = jax.random.PRNGKey(42)
A_mat = jax.random.normal(key, (3, 3))
x_vec = jax.random.normal(key, (3,))

# --- Case 1: General non-symmetric matrix ---
jax_gradient = grad_quadratic_form(x_vec, A_mat)
analytic_gradient = (A_mat + A_mat.T) @ x_vec

print("--- General Matrix A ---")
print(f"JAX gradient: {jax_gradient}")
print(f"Analytic gradient (A+A^T)x: {analytic_gradient}")
print(f"Are they equal? {jnp.allclose(jax_gradient, analytic_gradient)}")

# --- Case 2: Symmetric matrix ---
A_sym = (A_mat + A_mat.T) / 2.0 # Make A symmetric
jax_gradient_sym = grad_quadratic_form(x_vec, A_sym)
analytic_gradient_sym = 2 * A_sym @ x_vec

print("\n--- Symmetric Matrix A ---")
print(f"JAX gradient: {jax_gradient_sym}")
print(f"Analytic gradient 2Ax: {analytic_gradient_sym}")
print(f"Are they equal? {jnp.allclose(jax_gradient_sym, analytic_gradient_sym)}
    ")
```

Listing 5: Verifying the gradient of a quadratic form in JAX.

2.6 Application: Ordinary Least Squares (OLS)

This is a cornerstone example of applying matrix calculus. In linear regression, we have a matrix of data $X \in \mathbb{R}^{m\times n}$ (where m is the number of samples and n is the number of features) and a vector of target values $y \in \mathbb{R}^m$. We want to find the vector of parameters $\beta \in \mathbb{R}^n$ that minimizes the sum of squared errors.

Example 2.1 (Deriving the Normal Equations)**.** *The objective function for OLS is to minimize the squared Euclidean norm of the residual vector $X\beta - y$.*

$$f(\beta) = ||X\beta - y||_2^2.$$

Find the optimal β that minimizes this function.

Proof:

1. ***Rewrite the objective function.*** *The squared norm is a dot product:*

$$f(\beta) = (X\beta - y)^T(X\beta - y).$$

2. ***Expand the expression.*** *Using the rule $(A - B)^T = A^T - B^T$:*

$$\begin{aligned} f(\beta) &= (\beta^T X^T - y^T)(X\beta - y) \\ &= \beta^T X^T X\beta - \beta^T X^T y - y^T X\beta + y^T y. \end{aligned}$$

Note that $\beta^T X^T y$ and $y^T X\beta$ are both scalars. A scalar is equal to its own transpose, so $(\beta^T X^T y)^T = y^T (X^T)^T (\beta^T)^T = y^T X\beta$. Thus, the two middle terms are identical.

$$f(\beta) = \beta^T (X^T X)\beta - 2y^T X\beta + y^T y.$$

3. ***Compute the gradient.*** *We now take the gradient of $f(\beta)$ with respect to β:*

$$\nabla_\beta f(\beta) = \nabla_\beta(\beta^T (X^T X)\beta) - \nabla_\beta(2y^T X\beta) + \nabla_\beta(y^T y).$$

Let's handle each term using our rules:

- *For the quadratic term, the matrix is $A = X^T X$, which is symmetric. So, $\nabla_\beta(\beta^T(X^T X)\beta) = 2(X^T X)\beta$.*
- *For the linear term, let $a^T = 2y^T X$. Then $a = (2y^T X)^T = 2X^T y$. So, $\nabla_\beta(2y^T X\beta) = 2X^T y$.*
- *The term $y^T y$ is a constant with respect to β, so its gradient is $\mathbf{0}$.*

Combining these gives the full gradient:

$$\nabla_\beta f(\beta) = 2X^T X\beta - 2X^T y.$$

4. ***Set the gradient to zero and solve.*** *To find the minimum, we set the gradient to the zero vector:*

$$\begin{aligned} 2X^T X\beta - 2X^T y &= \mathbf{0} \\ X^T X\beta &= X^T y. \end{aligned}$$

This famous result is known as the ***Normal Equations****. Assuming the matrix $X^T X$ is invertible (which is true if the features are linearly independent), we can solve for the optimal parameters $\hat{\beta}$:*

$$\hat{\beta} = (X^T X)^{-1} X^T y.$$

This is a closed-form solution for the parameters of a linear regression model, derived entirely through optimization.

2.7 The Hessian and Positive Definiteness

As in the single-variable case, we need a way to confirm that a critical point is a minimum. This involves the Hessian matrix and its eigenvalues. Before we state the formal definition, let's look at a geometric example to motivate the concepts.

Example: Motivating Curvature through Eigenvalues

Consider the quadratic form $f(x) = x^T Ax$. The Hessian of this function is $2A$ (assuming A is symmetric). The shape of this function depends entirely on the matrix A. Let's examine two specific cases in $\mathbb{R}^2$ that correspond to the functions $x^2 + y^2$ and $x^2 - y^2$.

1. **Elliptic Paraboloid:** $f_1(x, y) = x^2 + y^2$. In matrix form, we can write this using the identity matrix I:
$$f_1(x) = \begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}.$$
The Hessian is $\nabla^2 f_1 = 2I = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$. The eigenvalues are $\lambda_1 = 2$ and $\lambda_2 = 2$. **Result:** The function curves **up** in all directions. It is "bowl-shaped", and the origin is a **global minimum**.

2. **Hyperbolic Paraboloid:** $f_2(x, y) = x^2 - y^2$. In matrix form:
$$f_2(x) = \begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}.$$
The Hessian is $\nabla^2 f_2 = \begin{bmatrix} 2 & 0 \\ 0 & -2 \end{bmatrix}$. The eigenvalues are $\lambda_1 = 2$ and $\lambda_2 = -2$. **Result:** The function curves **up** along the x-axis (positive eigenvalue) but **down** along the y-axis (negative eigenvalue). It is "saddle-shaped" (like a Pringle chip). The origin is a **saddle point**, not a minimum.

Conclusion: For a point to be a minimum, the function must curve up in *every* direction. In terms of linear algebra, this means all eigenvalues of the Hessian must be positive. This leads to the definition of a Positive Definite matrix.

Formal Definitions and OLS

Definition 2.2 (Eigenvalues and Eigenvectors). *For a square matrix A, an eigenvector v and its corresponding eigenvalue λ satisfy the equation $Av = \lambda v$. Geometrically, this means that the matrix A only stretches or shrinks the vector v, without changing its direction.*

Definition 2.3 (Positive Definite Matrix). *A symmetric matrix $A \in \mathbb{R}^{n \times n}$ is **positive definite** if any of the following equivalent conditions hold:*

1. *For any non-zero vector $x \in \mathbb{R}^n$, the quadratic form is strictly positive: $x^T Ax > 0$.*
2. *All the eigenvalues of A are strictly positive: $\lambda_i > 0$ for all i.*

Example 2.2 (Hessian of OLS). *Let's find the Hessian of the OLS objective function $f(\beta) = \beta^T (X^T X)\beta - 2y^T X\beta + y^T y$.*

Proof: *Our gradient was $\nabla_\beta f(\beta) = 2X^T X\beta - 2X^T y$. We take the derivative of the gradient with respect to β again:*
$$\nabla^2_\beta f(\beta) = \nabla_\beta (2X^T X\beta - 2X^T y) = 2X^T X.$$

The matrix $X^T X$ is positive semidefinite. If its columns are linearly independent, it is positive definite. This means all its eigenvalues are positive, confirming that the solution $\hat{\beta}$ we found is indeed a minimum.

```python
import jax
import jax.numpy as jnp

# Define the OLS loss function
def ols_loss(beta, X, y):
  residuals = X @ beta - y
  return jnp.sum(residuals**2)

# Create functions to compute the gradient and Hessian w.r.t. beta
grad_ols = jax.grad(ols_loss, argnums=0)
hessian_ols = jax.hessian(ols_loss, argnums=0)

# Generate some synthetic data
key = jax.random.PRNGKey(123)
m, n = 50, 5 # 50 samples, 5 features
X_data = jax.random.normal(key, (m, n))
true_beta = jnp.array([1.5, -2.0, 3.0, 0.5, -1.0])
y_data = X_data @ true_beta + 0.5 * jax.random.normal(key, (m,))
beta_guess = jnp.zeros(n)

# --- 1. Compute the gradient and verify the formula ---
jax_gradient = grad_ols(beta_guess, X_data, y_data)
analytic_gradient = 2 * X_data.T @ (X_data @ beta_guess - y_data)
print("--- Gradient Verification ---")
print(f"Are JAX and analytic gradients equal? {jnp.allclose(jax_gradient,
    analytic_gradient)}")

# --- 2. Compute the Hessian and verify the formula ---
jax_hessian = hessian_ols(beta_guess, X_data, y_data)
analytic_hessian = 2 * X_data.T @ X_data
print("\n--- Hessian Verification ---")
print(f"Are JAX and analytic Hessians equal? {jnp.allclose(jax_hessian,
    analytic_hessian)}")

# --- 3. Motivating Example: Hyperbolic Paraboloid Eigenvalues ---
# Let's verify the eigenvalues of the saddle point example f(x,y) = x^2 - y^2
H_saddle = jnp.array([[2.0, 0.0], [0.0, -2.0]])
eigenvals = jnp.linalg.eigvals(H_saddle)
print("\n--- Saddle Point Verification ---")
print(f"Eigenvalues of Hessian for x^2 - y^2: {eigenvals}")
print("Mixed signs imply a saddle point (not a minimum).")
```

Listing 6: Computing the gradient and Hessian of the OLS objective in JAX.

3 Introduction to Optimization in Statistics

3.1 The Anatomy of an Optimization Problem

In the previous lectures, we reviewed the calculus and linear algebra tools needed to find the minimum of a function. We saw that setting the gradient to zero is a key first step. Now, let's formalize the structure of the problems we aim to solve.

Every optimization problem has three core components:

1. **The Objective Function.** This is the function we want to minimize or maximize. It quantifies the "cost" or "reward" of a particular solution. We typically denote it by $f(x)$.
2. **The Decision Variables.** These are the variables we can control or adjust to change the value of the objective function. We denote them by a vector x.
3. **The Constraints / Feasible Set.** This is a set of conditions that the decision variables must satisfy. It defines the "space of possible solutions." We denote this set by Ω.

We can write a general minimization problem in a standard form:

$$\min_x f(x) \quad \text{subject to} \quad x \in \Omega,$$

and the **argmin** of f over the set Ω is the point $x^* \in \Omega$ at which the function attains its minimum value. We write:

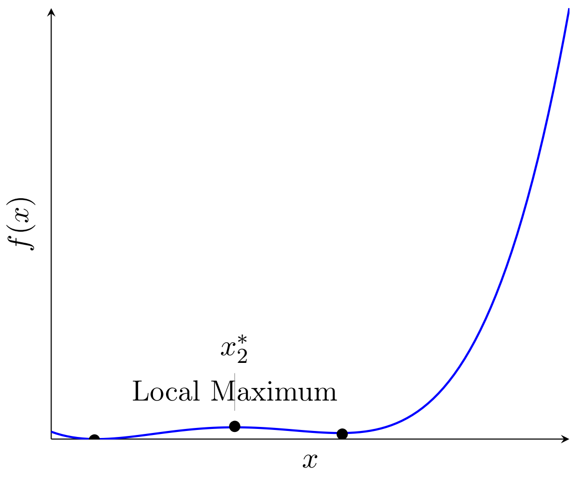
$$x^* = \operatorname*{argmin}_{x \in \Omega} f(x).$$

A maximization problem can be easily converted to a minimization problem, since maximizing a function is equivalent to minimizing its negative: $\max f(x) = -\min(-f(x))$.

Definition 3.1 (Global vs. Local Minimum).
- *A point $x^* \in \Omega$ is a **global minimum** of f over Ω if $f(x^*) \leq f(x)$ for all $x \in \Omega$.*
- *A point $x^* \in \Omega$ is a **local minimum** of f over Ω if there exists some small region around x^* such that $f(x^*) \leq f(x)$ for all $x \in \Omega$ within that region.*

In statistics, we almost always seek a global minimum (e.g., the "best" line in linear regression). Finding global minima can be hard, as optimization algorithms can get stuck in local minima.

A function with multiple optima



3.2 First-Order Optimality Condition (Unconstrained)

For now, let's consider the simplest case where there are no constraints. This is called **unconstrained optimization**, where the feasible set is the entire space, $\Omega = \mathbb{R}^n$.

As we saw in Lecture 1, for a function to have a local minimum at a point x^*, it must be "flat" at that point. This gives us the most fundamental condition in optimization.

First-Order Necessary Condition: If x^* is a local minimum of a differentiable function $f : \mathbb{R}^n \to \mathbb{R}$, then it must be a critical point, i.e.,

$$\nabla f(x^*) = \mathbf{0}.$$

This condition is **necessary** but not **sufficient**. A point where the gradient is zero could also be a maximum or a saddle point. However, it's our primary tool for identifying candidate solutions.

Example 3.1 (Finding a Minimum)**.** *Find a solution to the problem* $\min_{x\in\mathbb{R}} f(x)$ *where* $f(x) := -2x + e^x - 5$.

Proof:

1. ***Compute the derivative.***

$$f'(x) = \frac{d}{dx}(-2x + e^x - 5) = -2 + e^x.$$

2. ***Set the derivative to zero.***

$$-2 + e^x = 0 \implies e^x = 2.$$

3. ***Solve for x.***

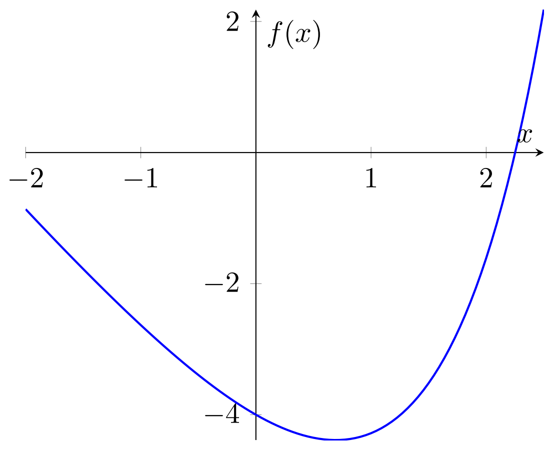
$$x^* = \ln(2) \approx 0.693.$$

4. ***Check the second derivative.*** *To confirm it's a minimum, we check the curvature:*

$$f''(x) = \frac{d}{dx}(-2 + e^x) = e^x.$$

At our critical point, $f''(\ln(2)) = e^{\ln(2)} = 2$. *Since* $f''(x^*) > 0$, *the function is concave up, and we have found a local minimum.*

$f(x) = -2x + e^x - 5$



3.3 Example: Maximum Likelihood Estimation (MLE)

Many fundamental principles in statistics are naturally framed as optimization problems. A prime example is the **Principle of Maximum Likelihood**, which states:

> Given a set of observed data and a probabilistic model parameterized by θ, the best estimate for θ is the one that **maximizes the probability** (or likelihood) of observing that data.

This is inherently a maximization problem.

Example 3.2 (MLE for a Biased Coin)**.** *Suppose we have a biased coin where the probability of getting heads is an unknown parameter θ. We flip the coin N times and observe N_H heads and N_T tails (where $N_H + N_T = N$). What is our best estimate for θ?*

Proof:

1. ***Define the Model and Likelihood.*** *A single coin flip follows a Bernoulli distribution with $P(\mathit{Heads}) = \theta$ and $P(\mathit{Tails}) = 1 - \theta$. Assuming the flips are independent, the probability of observing our specific sequence of data is:*

$$L(\theta) = \prod_{i=1}^{N} P(\mathit{flip}_i|\theta) = \theta^{N_H}(1-\theta)^{N_T}.$$

*This function, $L(\theta)$, is the **Likelihood function**. Our goal is to find the θ that maximizes it.*

2. ***Switch to Log-Likelihood.*** *Products are difficult to differentiate. Since the logarithm is a monotonically increasing function, maximizing $L(\theta)$ is equivalent to maximizing $\log(L(\theta))$. This is a standard trick in statistics.*

$$\ell(\theta) = \log(L(\theta)) = \log(\theta^{N_H}(1-\theta)^{N_T}) = N_H \log(\theta) + N_T \log(1-\theta).$$

*This is the **Log-Likelihood function**.*

3. ***Compute the Derivative.*** *We now find the derivative of $\ell(\theta)$ with respect to θ:*

$$\frac{d\ell}{d\theta} = \frac{d}{d\theta}(N_H \log(\theta) + N_T \log(1-\theta)) = \frac{N_H}{\theta} - \frac{N_T}{1-\theta}.$$

4. ***Set the Derivative to Zero and Solve.***

$$\begin{aligned} \frac{N_H}{\theta} - \frac{N_T}{1-\theta} &= 0 \\ \frac{N_H}{\theta} &= \frac{N_T}{1-\theta} \\ N_H(1-\theta) &= N_T\theta \\ N_H - N_H\theta &= N_T\theta \\ N_H &= (N_H + N_T)\theta \\ N_H &= N\theta \\ \hat{\theta} &= \frac{N_H}{N}. \end{aligned}$$

The maximum likelihood estimate for the probability of heads is simply the proportion of heads observed in the data. This is an intuitive result that we have now formally derived using the principles of optimization.

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

# Define the log-likelihood function
def log_likelihood(theta, n_heads, n_tails):
    # Add a small epsilon for numerical stability if theta is 0 or 1
    eps = 1e-7
    # Ensure theta is not exactly 0 or 1 to avoid log(0)
    theta = np.clip(theta, eps, 1 - eps)
    return n_heads * np.log(theta) + n_tails * np.log(1 - theta)

# Problem data
num_heads = 60
num_tails = 40
num_total = num_heads + num_tails

# The analytic solution (MLE)
mle_solution = num_heads / num_total

# Generate a range of theta values for plotting
theta_values = np.linspace(0, 1, 500)
log_likelihood_values = log_likelihood(theta_values, num_heads, num_tails)

# Create the plot
plt.figure(figsize=(10, 6))
plt.plot(theta_values, log_likelihood_values, label='Log-Likelihood Function')
plt.axvline(x=mle_solution, color='r', linestyle='--',
            label=f'MLE = {mle_solution:.2f}')

# Highlight the maximum point
max_log_likelihood = log_likelihood(mle_solution, num_heads, num_tails)
plt.plot(mle_solution, max_log_likelihood, 'ro') # Red dot at the max

plt.title('Log-Likelihood of a Biased Coin')
plt.xlabel(r'Parameter $\theta$ (Probability of Heads)')
plt.ylabel('Log-Likelihood')
plt.legend()
plt.grid(True)
plt.show()
```

Listing 7: Visualizing the log-likelihood function for the coin flip problem.

Example 3.3 (MLE for Poisson Distribution (Count Data)). *While the previous example dealt with binary data, often in statistics we deal with count data (e.g., number of customers arriving at a store per hour, or number of defects on a chip). This is often modeled by a Poisson distribution.*

Suppose we observe n independent data points $x_1, x_2, \ldots, x_n$, where each x_i follows a Poisson distribution with an unknown rate parameter $\lambda > 0$. The probability mass function for a single observation is:

$$P(X = k) = \frac{e^{-\lambda}\lambda^k}{k!}.$$

Proof:

1. ***Formulate the Likelihood.*** *Since the data are i.i.d., the joint likelihood is:*

$$L(\lambda) = \prod_{i=1}^{n} \frac{e^{-\lambda}\lambda^{x_i}}{x_i!}.$$

2. ***Formulate the Log-Likelihood.***

$$\begin{aligned}\ell(\lambda) &= \log\left(\prod_{i=1}^{n} \frac{e^{-\lambda}\lambda^{x_i}}{x_i!}\right)\\ &= \sum_{i=1}^{n}\Big(\log(e^{-\lambda}) + \log(\lambda^{x_i}) - \log(x_i!)\Big)\\ &= \sum_{i=1}^{n}\left(-\lambda + x_i\log(\lambda) - \log(x_i!)\right).\end{aligned}$$

We can split the sum. Note that $\sum_{i=1}^{n}(-\lambda) = -n\lambda$. The term $\sum\log(x_i!)$ does not depend on λ, so it will vanish when we differentiate.

$$\ell(\lambda) = -n\lambda + \left(\sum_{i=1}^{n} x_i\right)\log(\lambda) - constant.$$

3. ***Differentiate and Solve.*** *Take the derivative with respect to λ:*

$$\frac{d\ell}{d\lambda} = -n + \frac{1}{\lambda}\sum_{i=1}^{n} x_i.$$

Set to zero to find the critical point:

$$-n + \frac{1}{\lambda}\sum_{i=1}^{n} x_i = 0 \implies n = \frac{1}{\lambda}\sum_{i=1}^{n} x_i \implies \hat{\lambda} = \frac{1}{n}\sum_{i=1}^{n} x_i.$$

Thus, $\hat{\lambda} = \bar{x}$. The Maximum Likelihood Estimator for the Poisson rate is simply the sample mean of the observations.

```python
import jax
import jax.numpy as jnp

# 1. Generate some synthetic count data
# True rate lambda = 4.0, observing 10 samples
data = jnp.array([3, 5, 4, 2, 6, 4, 3, 5, 4, 4], dtype=jnp.float32)

# 2. Define the Negative Log-Likelihood (NLL)
# Optimization libraries usually minimize, so we minimize negative log-
    likelihood
def poisson_nll(lam, x):
    # n * lambda - sum(x) * log(lambda) + constant
    n = x.shape[0]
    # We add a small epsilon to lambda to ensure we don't take log(0)
    return n * lam - jnp.sum(x) * jnp.log(lam + 1e-7)

# 3. Compute the gradient of the NLL
grad_nll = jax.grad(poisson_nll)

# 4. Check the gradient at the sample mean (the analytical solution)
sample_mean = jnp.mean(data)
gradient_at_mean = grad_nll(sample_mean, data)

print(f"Data: {data}")
print(f"Sample Mean (Analytical MLE): {sample_mean}")
print(f"Gradient at Sample Mean: {gradient_at_mean}")
print("The gradient is effectively zero, confirming the minimum.")
```

Listing 8: Verifying the Poisson MLE using JAX automatic differentiation.

3.4 Example: Ordinary Least Squares (Revisited)

Let's formally define the OLS problem from Lecture 2 using our new framework.

- **Decision Variables:** The model parameters, $\beta \in \mathbb{R}^n$.
- **Objective Function:** The sum of squared residuals, which we want to minimize.

$$f(\beta) = ||X\beta - y||_2^2.$$

- **Feasible Set:** Since there are no restrictions on the values of β, this is an unconstrained problem.

$$\Omega = \mathbb{R}^n.$$

The full problem statement is:

$$\min_{\beta \in \mathbb{R}^n} ||X\beta - y||_2^2.$$

As we showed in Lecture 2, applying the first-order optimality condition $\nabla_\beta f(\beta) = \mathbf{0}$ leads directly to the famous Normal Equations:

$$X^T X\beta = X^T y.$$

4 Linear Programming

4.1 From Unconstrained to Constrained Optimization

In the previous lectures, we focused on finding the minimum of a function over its entire domain (e.g., $\mathbb{R}^n$). However, many real-world problems impose limitations on the solutions we can choose. For example:

- A company wants to maximize profit, but is limited by its available raw materials and labor hours.
- An investor wants to maximize returns, but is constrained by a fixed budget.
- In a statistical model, probabilities must be non-negative and sum to one.

These limitations are called **constraints**. Today, we explore the simplest, yet incredibly powerful, class of constrained optimization problems: **Linear Programming**, where both the objective function and the constraints are linear.

4.2 What is a Linear Program?

A linear program (LP) is an optimization problem where we aim to minimize or maximize a linear objective function subject to a set of linear equality and inequality constraints.

Definition 4.1 (Linear Program in Standard Form)**.** *A common standard form for a linear program is:*

$$\begin{aligned} \min_{x\in\mathbb{R}^n} \quad & c^T x \\ \textit{subject to} \quad & Ax = b \\ & x \geq \mathbf{0} \end{aligned}$$

where $c \in \mathbb{R}^n$ is the cost vector, $A \in \mathbb{R}^{m\times n}$ is the constraint matrix, $b \in \mathbb{R}^m$ is the right-hand side vector, and $x \geq \mathbf{0}$ means that each component x_i must be non-negative.

Any LP can be converted into this standard form. For example:

- A maximization problem $\max c^T x$ becomes $\min(-c)^T x$.
- An inequality constraint $a_i^T x \leq b_i$ can be converted to an equality by adding a non-negative **slack variable** s_i: $a_i^T x + s_i = b_i$, with $s_i \geq 0$.
- A variable x_j that is unrestricted in sign can be replaced by the difference of two non-negative variables: $x_j = x_j^+ - x_j^-$, with $x_j^+ \geq 0$ and $x_j^- \geq 0$.

4.3 The Geometry of Linear Programming

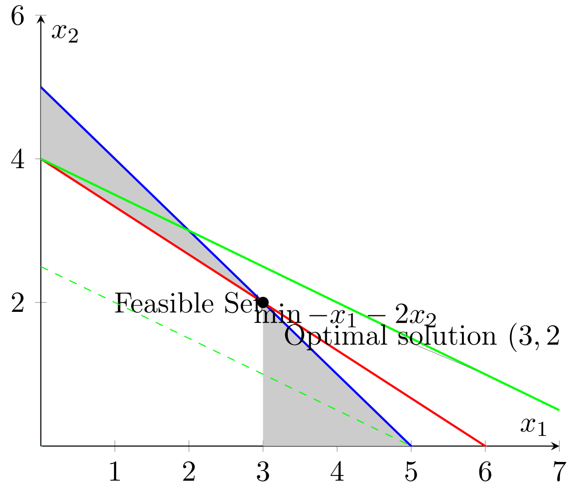
One of nice aspects of LP is its geometric interpretation.

- **The Feasible Set is a Polytope:** The set of points satisfying a system of linear inequalities forms a geometric object with flat faces, called a **polytope** (or a polyhedron if it's unbounded).
- **Optimal Solutions are at Vertices:** The fundamental theorem of linear programming states that if an LP has an optimal solution, there must be an optimal solution that is a **vertex** (a corner) of the feasible polytope.

Instead of searching an infinite number of points in the feasible region, we only need to check the finite number of vertices. Algorithms like the *Simplex method* are designed to efficiently move from vertex to vertex, always improving the objective function.

Visual Intuition: We can think of the objective function $c^T x = k$ as a family of parallel lines (or hyperplanes). To minimize $c^T x$, we are looking for the smallest value of k for which the line still touches the feasible set. This will inevitably happen at a corner.

Geometric View of a Linear Program



4.4 Application 1: Production Planning and Sensitivity Analysis

Let's explore a complete business example that demonstrates not just how to solve a linear program, but how to use the solution to make informed financial decisions. This process of evaluating how the optimal solution changes with respect to changes in the input parameters is called **sensitivity analysis**.

Example 4.1 (Production Mix at "TechPro Inc."). *TechPro Inc. manufactures two high-demand products: a "Standard Laptop" and a "Gaming Laptop." The goal is to determine the optimal number of each to produce per week to maximize profit, subject to production constraints.*

Problem Data:

- ***Profit:*** *The profit is \$300 per Standard Laptop and \$500 per Gaming Laptop.*
- ***Constraint 1: Assembly Hours.*** *The assembly department has a maximum of 1800 hours available per week. A Standard Laptop requires 3 hours of assembly, while a Gaming Laptop requires 4 hours.*
- ***Constraint 2: GPU Availability.*** *Due to supply chain limitations, TechPro can only source 350 high-end GPUs per week, which are exclusively used in the Gaming Laptops (one per unit).*
- ***Constraint 3: Testing Hours.*** *The quality assurance department has 1000 hours available for testing. Both models require 1 hour of testing.*

4.4.1 The Primal Problem (Maximizing Profit)

First, we formulate the problem as a linear program. Let:

- x_1 *= number of Standard Laptops to produce.*
- x_2 *= number of Gaming Laptops to produce.*

The primal problem is:

$$\begin{aligned} \textit{(Primal)} \quad \max_{x_1,x_2} \quad & 300x_1 + 500x_2 \quad \textit{(Profit)} \\ \textit{subject to} \quad & 3x_1 + 4x_2 \leq 1800 \quad \textit{(Assembly constraint)} \\ & x_2 \leq 350 \quad \textit{(GPU constraint)} \\ & x_1 + x_2 \leq 1000 \quad \textit{(Testing constraint)} \\ & x_1, x_2 \geq 0. \end{aligned}$$

Using a standard LP solver, we find the optimal production plan is to produce $x_1^* = 133.33$ ***Standard Laptops*** *and* $x_2^* = 350$ ***Gaming Laptops****, for a maximum profit of* ***$215,000****. (In practice, we would handle the fractional part, but we use the continuous solution for this analysis).*

4.4.2 The Dual Problem and Shadow Prices

To perform sensitivity analysis, we examine the dual problem. The dual variables will tell us the marginal value of each resource. Let y_1, y_2, y_3 *be the dual variables for the Assembly, GPU, and Testing constraints, respectively.*

The dual problem is:

$$\begin{aligned} \textit{(Dual)} \quad \min_{y_1,y_2,y_3} \quad & 1800y_1 + 350y_2 + 1000y_3 \\ \textit{subject to} \quad & 3y_1 + y_3 \geq 300 \\ & 4y_1 + y_2 + y_3 \geq 500 \\ & y_1, y_2, y_3 \geq 0. \end{aligned}$$

Solving the dual (or extracting the dual variables from the primal solution) gives the optimal dual values, which are the ***shadow prices*** *of the resources:*

- $y_1^* = \$100$
- $y_2^* = \$100$
- $y_3^* = \$0$

Interpretation of Shadow Prices:

- ***Assembly Hours (***$y_1^* = \$100$***):*** *Each additional hour of assembly time available would increase the maximum profit by $100. This is the maximum price TechPro should be willing to pay for one extra hour of assembly labor (e.g., for overtime).*
- ***GPUs (***$y_2^* = \$100$***):*** *Each additional high-end GPU sourced would increase the maximum profit by $100. This is the maximum premium TechPro should pay a supplier for an extra GPU.*
- ***Testing Hours (***$y_3^* = \$0$***):*** *Each additional hour of testing time is worthless. The shadow price is zero because this constraint is* ***not binding****. At the optimal solution, we only use* $133.33 + 350 = 483.33$ *testing hours, well below the 1000 hours available. We have a surplus, so acquiring more of this resource provides no benefit.*

4.4.3 Cost-Benefit Analysis for Decision Making

Now, TechPro's management can use these shadow prices to make data-driven decisions.

Scenario A: Overtime Opportunity

- ***Question:*** *The assembly department offers to provide 50 extra hours of overtime at a total cost of $4,000. Should management approve this?*
- ***Cost per hour:*** *$4,000 / 50 hours = $80 per hour.*
- ***Benefit per hour (Shadow Price):*** *The shadow price of an assembly hour is $100.*
- ***Decision:*** *Since the cost ($80) is less than the benefit ($100),* ***yes, they should approve the overtime****. It is projected to increase profit by* $(\$100 - \$80) \times 50 = \$1,000$.

Scenario B: Expedited GPU Shipment

- ***Question:*** *A supplier offers an expedited shipment of 20 extra GPUs, but at a premium of $120 per GPU above the normal cost. Should TechPro buy them?*
- ***Cost per GPU (Premium):*** *$120.*
- ***Benefit per GPU (Shadow Price):*** *The shadow price of a GPU is $100.*
- ***Decision:*** *Since the cost ($120) is greater than the benefit ($100),* ***no, they should reject the offer****. Paying the premium would result in a net loss of $20 for each extra GPU purchased.*

This analysis provides a clear, quantitative justification for complex resource allocation decisions, moving beyond simple intuition to optimal, profit-maximizing strategies.

```python
import numpy as np
from scipy.optimize import linprog

# --- 1. Define the Primal Linear Program ---
# Objective: max 300*x1 + 500*x2  =>  min -300*x1 - 500*x2
c = np.array([-300, -500])

# Constraints (A_ub @ x <= b_ub)
A_ub = np.array([
    [3, 4],     # Assembly constraint
    [0, 1],     # GPU constraint
    [1, 1]      # Testing constraint
])
b_ub = np.array([1800, 350, 1000])

# Bounds for x1, x2 (non-negative)
x1_bounds = (0, None)
x2_bounds = (0, None)

# --- 2. Solve the Primal LP ---
# The 'highs' solver is robust and returns dual variables (marginals)
result = linprog(c, A_ub=A_ub, b_ub=b_ub, bounds=[x1_bounds, x2_bounds], method
    ='highs')

# --- 3. Extract and Interpret the Solution ---
if result.success:
    # Primal solution
    num_standard = result.x[0]
    num_gaming = result.x[1]
    max_profit = -result.fun # Negate back to get max profit
    print("--- Optimal Production Plan ---")
    print(f"Produce {num_standard:.2f} Standard Laptops")
    print(f"Produce {num_gaming:.2f} Gaming Laptops")
    print(f"Maximum Profit: ${max_profit:,.2f}")

    # Check which constraints are binding
    resources_used = A_ub @ result.x
    print("\n--- Resource Utilization ---")
    print(f"Assembly Hours Used: {resources_used[0]:.2f} / {b_ub[0]}")
    print(f"GPUs Used: {resources_used[1]:.2f} / {b_ub[1]}")
    print(f"Testing Hours Used: {resources_used[2]:.2f} / {b_ub[2]}")

    # Dual solution (Shadow Prices)
    # The marginals are the shadow prices for the inequality constraints
    shadow_price_assembly = -result.ineqlin.marginals[0]
    shadow_price_gpu = -result.ineqlin.marginals[1]
    shadow_price_testing = -result.ineqlin.marginals[2]

    print("\n--- Sensitivity Analysis (Shadow Prices) ---")
    print(f"Marginal Profit of 1 extra Assembly Hour: ${shadow_price_assembly
    :.2f}")
    print(f"Marginal Profit of 1 extra GPU: ${shadow_price_gpu:.2f}")
    print(f"Marginal Profit of 1 extra Testing Hour: ${shadow_price_testing:.2f
    }")
    print("(Note: A zero shadow price means the resource is not a bottleneck)")

    # --- 4. Use Shadow Prices for Decision Making ---
    print("\n--- Cost-Benefit Analysis ---")

    # Scenario A: Overtime
    overtime_cost_per_hour = 80
    print(f"\nScenario A: Overtime for assembly at ${overtime_cost_per_hour}/hr
    .")
    if overtime_cost_per_hour < shadow_price_assembly:
        print(f"Decision: ACCEPT. The benefit (${shadow_price_assembly:.2f})
    exceeds the cost.")
    else:
        print(f"Decision: REJECT. The cost (${overtime_cost_per_hour:.2f})
    exceeds the benefit.")

    # Scenario B: Expedited GPUs
    gpu_premium_cost = 120
    print(f"\nScenario B: Expedited GPUs at a premium of ${gpu_premium_cost}
    each.")
    if gpu_premium_cost < shadow_price_gpu:
        print(f"Decision: ACCEPT. The benefit (${shadow_price_gpu:.2f}) exceeds
     the premium.")
    else:
        print(f"Decision: REJECT. The premium (${gpu_premium_cost:.2f}) exceeds
     the benefit.")
else:
    print("LP solver failed:", result.message)
```

Listing 9: Solving and analyzing the TechPro production problem with SciPy.

4.5 Application 2: Quantile Regression

While Ordinary Least Squares (OLS) is excellent for modeling the *mean* of a distribution, it is sensitive to outliers. **Quantile regression** is a powerful alternative that allows us to model any quantile (e.g., the median, the 25th percentile) of the response variable, making it far more robust. It turns out that this problem can be solved using linear programming.

Let's start with the simplest case: finding the median. Minimizing the sum of squared errors leads to the mean. What if we minimize the **sum of absolute errors** instead?

$$\min_{\beta} \sum_{i=1}^{m} |y_i - x_i^T \beta|$$

This is known as **Least Absolute Deviations (LAD)** or **L1 regression**. It can be shown that the solution to this problem gives the conditional **median**. However, the absolute value function is not linear.

Example 4.2 (Formulating LAD as a Linear Program)**.** *Show how to convert the LAD regression problem into a standard LP.*

Proof: *The key trick is to handle the absolute value. For any number z, we can write $|z|$ as the minimum of a variable u subject to $u \geq z$ and $u \geq -z$. We apply this to each residual term $r_i = y_i - x_i^T \beta$.*

We want to minimize $\sum_{i=1}^{m} |r_i|$. We can introduce a new variable u_i for each $|r_i|$:

$$\begin{aligned} \min_{\beta, u} \quad & \sum_{i=1}^{m} u_i \\ \text{subject to} \quad & u_i \geq |y_i - x_i^T \beta| \quad \text{for } i = 1, \dots, m. \end{aligned}$$

The constraint $u_i \geq |y_i - x_i^T \beta|$ is equivalent to the pair of linear constraints:

$$u_i \geq y_i - x_i^T \beta \quad \text{and} \quad u_i \geq -(y_i - x_i^T \beta).$$

This is almost an LP, but the variables β_j are unrestricted in sign. We can fix this by decomposing $\beta = \beta^+ - \beta^-$ where $\beta^+, \beta^- \geq \mathbf{0}$.

The final LP formulation for LAD regression is:

$$\begin{aligned} \min_{\beta^+, \beta^-, u} \quad & \mathbf{1}^T u \\ \text{subject to} \quad & u \geq y - X(\beta^+ - \beta^-) \\ & u \geq -(y - X(\beta^+ - \beta^-)) \\ & u \geq \mathbf{0}, \beta^+ \geq \mathbf{0}, \beta^- \geq \mathbf{0}. \end{aligned}$$

This is now a standard linear program that can be solved with any LP solver.

```python
import numpy as np
from scipy.optimize import linprog

# Generate some synthetic data for regression
# True relationship: y = X*beta + noise
np.random.seed(42)
m, n = 50, 5 # 50 samples, 5 features
X = np.random.randn(m, n)
true_beta = np.array([1.5, -2.0, 3.0, 0.5, -1.0])
y = X @ true_beta + 0.5 * np.random.randn(m)

# Introduce an outlier to show the robustness of L1 regression
y[10] = 50.0

# Formulate the LAD problem as a Linear Program.
# Variables are [beta+, beta-, u], where beta = beta+ - beta-
# The total number of variables is 2*n (for beta+, beta-) + m (for u)

# 1. Objective function: min sum(u_i).
# c vector should have 0s for beta+, beta- and 1s for u.
c = np.concatenate([np.zeros(2 * n), np.ones(m)])

# 2. Inequality constraints:
# u >= y - X(beta+ - beta-)  => -X@beta+ + X@beta- - u <= -y
# u >= -(y - X(beta+ - beta-)) => X@beta+ - X@beta- - u <= y
# We combine these into a single matrix A_ub.
I_m = np.eye(m)
A_ub_top = np.hstack([-X, X, -I_m])
A_ub_bottom = np.hstack([X, -X, -I_m])
A_ub = np.vstack([A_ub_top, A_ub_bottom])

b_ub = np.concatenate([-y, y])

# 3. Bounds: All variables (beta+, beta-, u) must be non-negative.
bounds = [(0, None) for _ in range(2 * n + m)]

# Solve the Linear Program
# Using 'highs' solver which is robust and available in recent SciPy
result = linprog(c, A_ub=A_ub, b_ub=b_ub, bounds=bounds, method='highs')

# Extract the solution
if result.success:
    beta_plus = result.x[:n]
    beta_minus = result.x[n:2*n]
    lad_beta = beta_plus - beta_minus

    print("LAD (L1) regression solved via LP.")
    print("True Beta:\n", true_beta)
    print("LAD Beta Estimate:\n", lad_beta.round(4))

    # For comparison, let's compute the OLS (L2) solution
    ols_beta = np.linalg.inv(X.T @ X) @ X.T @ y
    print("\nOLS (L2) Beta Estimate (sensitive to outlier):\n", ols_beta.round
    (4))
else:
    print("LP solver failed:", result.message)
```

Listing 10: Solving LAD (L1) Regression as a Linear Program using SciPy.

General Quantile Regression: We can generalize from the median (the $\tau = 0.5$ quantile) to any other quantile $\tau \in (0, 1)$ by minimizing a "tilted" absolute value function, called the check function:

$$\rho_\tau(r) = \begin{cases} \tau r & \text{if } r \geq 0 \\ (\tau - 1)r & \text{if } r < 0 \end{cases}.$$

The quantile regression problem is $\min_\beta \sum_i \rho_\tau(y_i - x_i^T \beta)$. This can also be formulated as an LP in a similar manner, which is left as an exercise for the keen student!

4.6 Application 3: Entropy-Regularized Linear Optimization

Sometimes we have a linear objective, but we also want the solution to satisfy certain properties, like being "spread out" or "smooth." This is often done by adding a **regularization term** to the objective. While this makes the problem non-linear, it is a very common extension of LP.

Consider the problem of finding a probability distribution $x = (x_1, \ldots, x_n)$ over n actions. We are given values v_i for each action and want to find a distribution that minimizes a combination of the expected value and a "regularizer".

Example 4.3 (Softmax as an Optimization Problem)**.** *The* ***entropy-regularized best re-***

sponse *is the solution to the following problem:*

$$\begin{aligned} \min_{x} \quad & \sum_{i=1}^{n} v_i x_i + \sum_{i=1}^{n} x_i \log x_i \\ s.t. \quad & x \in \Delta^n \quad (\textit{the probability simplex, i.e.,} \sum x_i = 1, x_i \geq 0). \end{aligned}$$

The term $\sum v_i x_i$ is a linear objective. The term $\sum x_i \log x_i$ is the negative ***entropy*** *of the distribution. Minimizing this term encourages the distribution to be "smoother" or more uniform.*

Solution (from lecture notes, via KKT conditions which we will see later): *The solution to this problem is the famous* ***softmax*** *distribution:*

$$x_i = \frac{e^{-v_i}}{\sum_{j=1}^{n} e^{-v_j}}.$$

This shows that actions with a lower (better) value v_i are assigned a higher probability. The softmax function is a smooth, differentiable way to approximate the 'argmin' function, which is a cornerstone of classification models in machine learning.

5 Duality and Its Interpretations

5.1 Introduction: The "Shadow" Problem

Imagine you run a furniture workshop that produces two products: tables and chairs. Your goal is to decide how many of each to produce to maximize your profit. This is a classic optimization problem.

- Let x_1 be the number of tables and x_2 be the number of chairs. - Each table yields a profit of \$15. - Each chair yields a profit of \$10.
- You have limited resources: - 20 units of wood. A table requires 4 units, a chair requires 2. - 30 hours of labor. A table requires 3 hours, a chair requires 3.

This gives us a linear program, which we call the **primal problem**:

$$\begin{aligned}
\text{(Primal)} \quad \max_{x_1,x_2} \quad & 15x_1 + 10x_2 \quad \text{(Profit)} \\
\text{subject to} \quad & 4x_1 + 2x_2 \le 20 \quad \text{(Wood constraint)} \\
& 3x_1 + 3x_2 \le 30 \quad \text{(Labor constraint)} \\
& x_1, x_2 \ge 0.
\end{aligned}$$

We can solve this to find the optimal production plan. But what if we ask a different question?

How much would I be willing to pay for one extra unit of wood? Or one extra hour of labor?

This question is not about the production plan (x_1, x_2) but about the **value of the resources**. This "shadow" problem is called the **dual problem**. Duality theory tells us that every optimization problem has a corresponding dual problem, and the solution to the dual provides insights into the primal.

5.2 Constructing the Dual Problem

For every linear program, we can mechanically construct its dual. The components of the primal map directly to the components of the dual. Let's consider a generic primal problem in maximization form:

$$\begin{aligned}
\text{(Primal)} \quad \max_{x\in\mathbb{R}^n} \quad & c^T x \\
\text{subject to} \quad & Ax \le b \\
& x \ge \mathbf{0}.
\end{aligned}$$

Its dual problem is a minimization problem:

$$\begin{aligned}
\text{(Dual)} \quad \min_{y\in\mathbb{R}^m} \quad & b^T y \\
\text{subject to} \quad & A^T y \ge c \\
& y \ge \mathbf{0}.
\end{aligned}$$

The transformation follows a simple recipe:

Primal (max)	**Dual (min)**
Objective coefficients c	Constraint RHS
Constraint RHS b	Objective coefficients
Constraint matrix A	Transposed matrix A^T
Variables x (one for each column of A)	Constraints (one for each column of A)
Constraints (one for each row of A)	Variables y (one for each row of A)
$\le$ constraints	$\ge$ variables
$\ge$ variables	$\ge$ constraints

Each variable in the dual problem corresponds to a constraint in the primal problem. This is the key to their interpretation.

Example 5.1 (Dual of the Furniture Problem)**.** *Let's construct the dual for our workshop example. The primal in matrix form is:*

$$\max \begin{bmatrix} 15 & 10 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad s.t. \quad \begin{bmatrix} 4 & 2 \\ 3 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \leq \begin{bmatrix} 20 \\ 30 \end{bmatrix}, \quad \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \geq \mathbf{0}.$$

Here, $c = \begin{bmatrix} 15 \\ 10 \end{bmatrix}$, $A = \begin{bmatrix} 4 & 2 \\ 3 & 3 \end{bmatrix}$, *and* $b = \begin{bmatrix} 20 \\ 30 \end{bmatrix}$.

The dual problem will have two variables, y_1 *(for the wood constraint) and* y_2 *(for the labor constraint).*

$$\begin{aligned} \min_{y_1, y_2} \quad & 20y_1 + 30y_2 \\ \textit{subject to} \quad & 4y_1 + 3y_2 \geq 15 \\ & 2y_1 + 3y_2 \geq 10 \\ & y_1, y_2 \geq 0. \end{aligned}$$

The dual variables y_1 *and* y_2 *can be interpreted as the "prices" or "values" we assign to our resources (wood and labor). The dual problem seeks to find the cheapest set of resource prices that still "justify" the profit from the products.*

5.3 The Duality Theorems

The relationship between the primal and dual problems is governed by two fundamental theorems.

Definition 5.1 (Weak Duality)**.** *For any feasible solution x to the primal maximization problem and any feasible solution y to the dual minimization problem, the primal objective is always less than or equal to the dual objective:*

$$c^T x \leq b^T y.$$

This means that any feasible solution to the dual problem provides an **upper bound** on the optimal value of the primal problem.

Proof: The proof is surprisingly simple and follows directly from the constraints. Since x is primal feasible, we have $Ax \leq b$ and $x \geq \mathbf{0}$. Since y is dual feasible, we have $A^T y \geq c$ and $y \geq \mathbf{0}$.

$$\begin{aligned} c^T x &\leq (A^T y)^T x \quad (\text{since } c \leq A^T y \text{ and } x \geq \mathbf{0}) \\ &= y^T A x \\ &\leq y^T b \quad (\text{since } Ax \leq b \text{ and } y \geq \mathbf{0}) \\ &= b^T y. \end{aligned}$$

This completes the proof.

While weak duality is useful, the next result is the truly powerful one.

Definition 5.2 (Strong Duality)**.** *If a linear program has an optimal solution* x^**, then its dual problem also has an optimal solution* y^**, and their optimal objective values are equal:*

$$c^T x^* = b^T y^*.$$

Strong duality tells us there is no "gap" between the primal and dual solutions. This is a result that holds for all LPs and, as we'll see later, for a broad class of optimization problems called convex problems.

5.4 Interpretation: Shadow Prices and Complementary Slackness

Strong duality is the key to interpreting the dual variables.

Definition 5.3 (Shadow Price)**.** *The optimal value y_i^* of the i-th dual variable represents the rate of improvement in the primal objective function for a marginal increase in the right-hand side of the i-th constraint. It is the* ***shadow price*** *of the i-th resource.*

In our furniture example:

- y_1^* is the additional profit we would gain from having one more unit of wood (from 20 to 21).
- y_2^* is the additional profit we would gain from having one more hour of labor (from 30 to 31).

This is invaluable information for decision-making. If the market price for an extra unit of wood is less than its shadow price y_1^*, we should buy more wood!

This leads to another crucial concept. What if we don't use up all our wood? Then getting more wood is worthless. This intuition is formalized by **complementary slackness**.

Definition 5.4 (Complementary Slackness)**.** *Let x^* be the optimal primal solution and y^* be the optimal dual solution.*

- ***Primal Slackness:*** *If a primal variable is positive ($x_j^* > 0$), then the corresponding dual constraint must be tight (met with equality): $(A^T y^*)_j = c_j$.*
- ***Dual Slackness:*** *If a dual variable is positive ($y_i^* > 0$), then the corresponding primal constraint must be tight: $(Ax^*)_i = b_i$. Conversely, if a primal constraint is not tight ($(Ax^*)_i < b_i$, i.e., we have leftover resources), then its shadow price must be zero ($y_i^* = 0$).*

This perfectly matches our intuition: if a resource is not fully used, its marginal value is zero.

Example 5.2 (Portfolio Optimization)**.** *Suppose we are building a portfolio from n assets. Let x_i be the amount of money invested in asset i, and r_i be its expected return. We have a total budget of B. The optimization problem is:*

$$\begin{aligned} \max_x \quad & r^T x \\ \textit{subject to} \quad & \mathbf{1}^T x \le B \quad \textit{(Budget constraint)} \\ & x \ge \mathbf{0}. \end{aligned}$$

The dual problem has one variable, y, for the budget constraint.

$$\begin{aligned} \min_y \quad & By \\ \textit{subject to} \quad & \mathbf{1}y \ge r \\ & y \ge 0. \end{aligned}$$

The optimal dual variable y^* represents the shadow price of our budget. It tells us the marginal increase in our portfolio's expected return for every extra dollar we are willing to invest.

6 Introduction to Convexity

6.1 Introduction: The Problem of Local Minima

In our journey so far, we have used calculus to find points where the gradient is zero. However, as we've seen, this doesn't guarantee that we've found the *best* possible solution (the global minimum). A general non-linear function can have many "valleys" (local minima), and an optimization algorithm might get stuck in a shallow one, missing the deepest valley entirely.

This is the fundamental challenge of non-linear optimization. Fortunately, a large and incredibly important class of problems does not suffer from this issue. These are **convex optimization problems**.

The Magic of Convexity: For convex optimization problems, *any local minimum is also a global minimum.* This property is so powerful that it essentially divides optimization problems into two categories: those we can solve efficiently (convex problems) and those we generally cannot (non-convex problems).

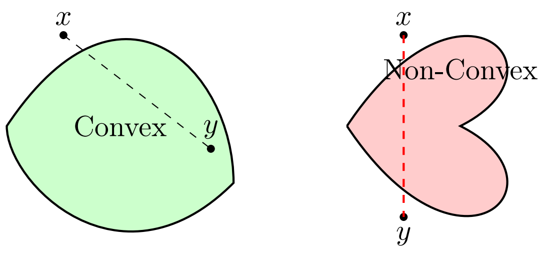
In this lecture, we will build an intuition for convexity, learning to recognize convex sets and convex functions, and understanding why they are the cornerstone of modern optimization.

6.2 Convex Sets

The first ingredient is a convex feasible set. The idea is simple: a set is convex if it has no "dents" or "holes" in it.

Definition 6.1 (Convex Set)**.** *A set Ω is* ***convex*** *if for any two points $x, y \in \Omega$, the line segment connecting them is also entirely contained in Ω. Mathematically, for any $x, y \in \Omega$ and any $\lambda \in [0, 1]$,*

$$\lambda x + (1 - \lambda)y \in \Omega.$$



Example 6.1 (Examples of Convex Sets)**.**

- *The entire space $\mathbb{R}^n$.*
- *Lines, planes, and affine subspaces (e.g., the solution set to $Ax = b$).*
- *The non-negative orthant, $\mathbb{R}^n_+ = \{x \in \mathbb{R}^n \mid x_i \geq 0$ for all $i\}$.*
- *Balls and ellipsoids (e.g., $\{x \mid ||x - x_c||_2 \leq r\}$).*
- *The probability simplex, $\Delta^n = \{x \in \mathbb{R}^n \mid \sum x_i = 1, x_i \geq 0\}$.*

A crucial property is that convexity is preserved under intersection. **Property:** If $\Omega_1, \Omega_2, \ldots, \Omega_k$ are all convex sets, then their intersection $\Omega = \bigcap_{i=1}^{k} \Omega_i$ is also a convex set.

This is why the feasible set of a linear program ($Ax \leq b, x \geq \mathbf{0}$) is convex—it's the intersection of many half-spaces, each of which is a convex set.

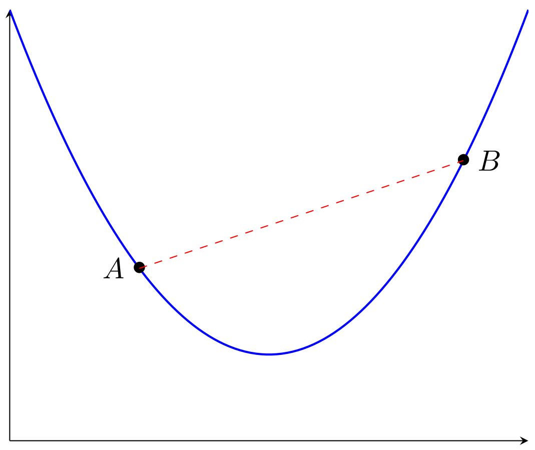
6.3 Convex Functions

The second ingredient is a convex objective function. Intuitively, a convex function is "bowl-shaped."

Definition 6.2 (Convex Function)**.** *A function f is* ***convex*** *if its domain is a convex set and for any two points x, y in its domain, the line segment connecting $(x, f(x))$ and $(y, f(y))$ lies on or above the graph of f. Mathematically, for any x, y and any $\lambda \in [0, 1]$,*

$$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda) f(y).$$

A Convex Function



For differentiable functions, there are two much easier ways to check for convexity.

- **First-Order Condition:** A differentiable function f is convex if and only if it lies above all of its tangent lines:

$$f(y) \geq f(x) + \nabla f(x)^T (y - x) \quad \text{for all } x, y.$$

- **Second-Order Condition:** A twice-differentiable function f is convex if and only if its Hessian matrix is positive semidefinite everywhere in its domain:

$$\nabla^2 f(x) \succeq 0 \quad \text{(i.e., all eigenvalues are non-negative).}$$

The second-order condition is often the easiest to use in practice.

Example 6.2 (Common Convex Functions in Statistics)**.** *1.* ***Negative Entropy:*** *$f(x) = x \log x$ for $x > 0$. This term appears in many information-theoretic models.* ***Proof of Convexity:*** *We use the second-order condition.*

$$f'(x) = \frac{d}{dx}(x \log x) = 1 \cdot \log x + x \cdot \frac{1}{x} = \log x + 1$$
$$f''(x) = \frac{d}{dx}(\log x + 1) = \frac{1}{x}$$

For $x > 0$ (the domain of the function), $f''(x) = 1/x > 0$. Since the second derivative is positive, the function is convex.

2. ***Softplus Function:*** *$f(x) = \log(1 + e^x)$. This is often used as a smooth approximation of the ReLU function in neural networks.* ***Proof of Convexity:***

$$f'(x) = \frac{1}{1+e^x} \cdot e^x = \frac{e^x}{1+e^x}$$
$$f''(x) = \frac{e^x(1+e^x) - e^x(e^x)}{(1+e^x)^2} = \frac{e^x + e^{2x} - e^{2x}}{(1+e^x)^2} = \frac{e^x}{(1+e^x)^2}$$

Since $e^x > 0$ for all x, the numerator is positive. The denominator is a square, so it is also positive. Thus, $f''(x) > 0$ for all x, and the function is convex.

3. ***L2 Regularization (Ridge):*** *$f(\beta) = ||\beta||_2^2 = \sum_{i=1}^n \beta_i^2$.* ***Proof of Convexity:*** *The gradient is $\nabla f(\beta) = 2\beta$. The Hessian is:*

$$\nabla^2 f(\beta) = 2I$$

where I is the identity matrix. The eigenvalues of $2I$ are all 2, which are positive. Thus, the Hessian is positive definite, and the L2-norm squared is convex.

4. ***L1 Regularization (Lasso):*** *$f(\beta) = ||\beta||_1 = \sum_{i=1}^n |\beta_i|$. This function is not differentiable everywhere (e.g., at $\beta_i = 0$), so we cannot use the second-order test. However, we can show it is convex by the definition. It is the sum of convex functions ($|\beta_i|$), and the sum of convex functions is convex.*

```python
import jax
import jax.numpy as jnp

# --- 1. Softplus function f(x) = log(1 + e^x) ---
def softplus(x):
    return jnp.log(1 + jnp.exp(x))

# Get the second derivative (Hessian for a scalar is just f''(x))
hessian_softplus = jax.hessian(softplus)
# Test at a point
x_test = 2.0
h_val = hessian_softplus(x_test)
print(f"For f(x) = softplus(x) at x={x_test}, f''(x) = {h_val:.4f}")
print("This is positive, so the function is convex.")

# --- 2. L2 regularization f(beta) = ||beta||^2 ---
def l2_norm_squared(beta):
    return jnp.sum(beta**2)

hessian_l2 = jax.hessian(l2_norm_squared)

# Test at a point in 3D
beta_test = jnp.array([1.0, -2.0, 3.0])
H_l2 = hessian_l2(beta_test)
eigenvalues_l2 = jnp.linalg.eigvals(H_l2)

print("\nFor f(beta) = ||beta||^2:")
print(f"Hessian is:\n{H_l2}")
print(f"Eigenvalues are: {eigenvalues_l2}")
print("All eigenvalues are non-negative, so the Hessian is positive
    semidefinite.")
print("This confirms the L2 norm squared is convex.")
```

Listing 11: Checking the convexity of functions using the Hessian in JAX.

6.4 Convex Optimization Problems

We can now put the two pieces together.

Definition 6.3 (Convex Optimization Problem). *An optimization problem of the form*

$$\min_x f(x) \quad \textit{subject to} \quad x \in \Omega$$

is a ***convex optimization problem*** *if:*

1. *The objective function $f(x)$ is a convex function.*

2. *The feasible set Ω is a convex set.*

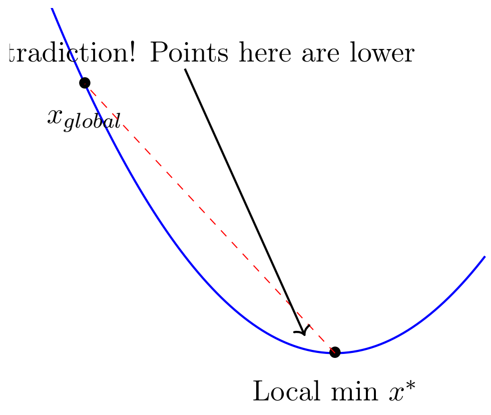
Now we arrive at the most important result of this lecture.

The Fundamental Theorem of Convex Optimization: For a convex optimization problem, any local minimum is also a global minimum.

Intuitive Proof: Suppose x^* is a local minimum, but not a global minimum. This means there exists some other point $x_{global} \in \Omega$ such that $f(x_{global}) < f(x^*)$.

- Because the set Ω is convex, the entire line segment connecting x^* and x_{global} is in Ω.
- Because the function f is convex, the graph of f along this line segment must lie below the chord connecting $(x^*, f(x^*))$ and $(x_{global}, f(x_{global}))$.
- This means that for any point z slightly along the segment from x^*, we must have $f(z) < f(x^*)$.
- But this contradicts our assumption that x^* is a local minimum! Therefore, our initial assumption must be false, and no such x_{global} can exist.

Why Local implies Global for Convex Functions



7 Convex Optimization Problems in Statistics

7.1 Recap: The Power of Convexity

In the previous lecture, we introduced the concept of convexity. We learned that a convex optimization problem consists of minimizing a convex function over a convex set. The reward for identifying such a problem is immense:

> **Any local minimum of a convex optimization problem is also a global minimum.**

This result effectively tames the complexity of optimization. It means that simple algorithms, like one that just goes "downhill" until it can't anymore, are guaranteed to find the best possible solution.

Today, we will see this principle in action. We will show that several of the most fundamental problems in modern statistics and machine learning are, in fact, convex optimization problems. This is the reason for their widespread success and the existence of efficient, reliable solvers for them.

7.2 Defining Feasible Sets with Functions

While we defined convex sets geometrically, in practice we usually define them algebraically using functions. A feasible set Ω is typically written as:

$$\Omega = \{x \in \mathbb{R}^n \mid g_i(x) \leq 0 \text{ for } i = 1, \ldots, m, \text{ and } h_j(x) = 0 \text{ for } j = 1, \ldots, p\}.$$

For Ω to be a convex set, we need:

1. Each inequality constraint function $g_i(x)$ must be **convex**.
2. Each equality constraint function $h_j(x)$ must be **affine** (i.e., linear, of the form $h_j(x) = a_j^T x - b_j$).

7.3 Application 1: Logistic Regression

Logistic regression is the workhorse model for binary classification problems, where the outcome is one of two categories (e.g., spam/not spam, disease/no disease).

Let's say our data consists of feature vectors $x_i \in \mathbb{R}^n$ and binary labels $y_i \in \{0, 1\}$. Logistic regression models the probability of the positive class ($y = 1$) using the **sigmoid function**:

$$P(y = 1 | x; \beta) = \sigma(\beta^T x) = \frac{1}{1 + e^{-\beta^T x}}.$$

The parameters of the model are the weights $\beta \in \mathbb{R}^n$. We find the best β using the principle of Maximum Likelihood Estimation (MLE).

Example 7.1 (Proving the Convexity of the Logistic Regression Objective)**.** *The likelihood of the data is the product of the probabilities of observing each label. To make the optimization problem easier, we maximize the log-likelihood. Maximizing the log-likelihood is equivalent to **minimizing the negative log-likelihood (NLL)**, which serves as our objective function.*

The NLL for logistic regression (also called the cross-entropy loss) is:

$$f(\beta) = -\sum_{i=1}^{m} \left[y_i \log(\sigma(\beta^T x_i)) + (1 - y_i) \log(1 - \sigma(\beta^T x_i)) \right].$$

Show that $f(\beta)$ is a convex function.

Proof: *A sum of convex functions is convex, so it suffices to show that the loss for a single data point is convex. Let* $z_i = \beta^T x_i$*. The single-point loss is:*

$$f_i(\beta) = -[y_i \log(\sigma(z_i)) + (1 - y_i)\log(1 - \sigma(z_i))].$$

Using $\log(\sigma(z_i)) = -\log(1 + e^{-z_i})$ *and* $\log(1 - \sigma(z_i)) = -z_i - \log(1 + e^{-z_i})$*, we get:*

$$f_i(\beta) = -[y_i(-\log(1+e^{-z_i}))+(1-y_i)(-z_i-\log(1+e^{-z_i}))] = \log(1+e^{-z_i})-y_i(-z_i) = \log(1+e^{z_i})-y_i z_i.$$

We need to compute the Hessian of $f_i(\beta)$ *with respect to* β*. We use the chain rule. First, the gradient:*

$$\begin{aligned}\nabla_\beta f_i(\beta) &= \frac{\partial f_i}{\partial z_i}\nabla_\beta z_i \\ &= \left(\frac{1}{1 + e^{z_i}} \cdot e^{z_i} - y_i\right) x_i \\ &= (\sigma(z_i) - y_i)x_i.\end{aligned}$$

Now, the Hessian. We differentiate the gradient again:

$$\begin{aligned}\nabla^2_\beta f_i(\beta) &= \nabla_\beta[(\sigma(\beta^T x_i) - y_i)x_i^T] \\ &= x_i \cdot \nabla_\beta(\sigma(\beta^T x_i) - y_i)^T \quad (\textit{using the outer product rule}) \\ &= x_i \cdot (\frac{\partial \sigma(z_i)}{\partial z_i}\nabla_\beta z_i^T) \\ &= x_i \cdot (\sigma(z_i)(1 - \sigma(z_i))x_i^T) \\ &= \sigma(\beta^T x_i)(1 - \sigma(\beta^T x_i))x_i x_i^T.\end{aligned}$$

The Hessian for the full objective $f(\beta)$ *is the sum of the Hessians for each data point:*

$$\nabla^2 f(\beta) = \sum_{i=1}^{m} \sigma(\beta^T x_i)(1 - \sigma(\beta^T x_i))x_i x_i^T.$$

To check for convexity, we need to show this matrix is positive semidefinite. Let's test it with an arbitrary non-zero vector $v \in \mathbb{R}^n$*:*

$$v^T(\nabla^2 f(\beta))v = v^T\left(\sum_{i=1}^{m} \sigma(1 - \sigma)x_i x_i^T\right)v = \sum_{i=1}^{m} \sigma(1 - \sigma)v^T x_i x_i^T v.$$

Since $v^T x_i$ *is a scalar,* $v^T x_i x_i^T v = (v^T x_i)(x_i^T v) = (x_i^T v)^2 \geq 0$*. The term* $\sigma(\beta^T x_i)$ *is a probability, so it's between 0 and 1. Therefore,* $\sigma(1 - \sigma) \geq 0$*.*

Thus, $v^T(\nabla^2 f(\beta))v$ *is a sum of non-negative terms, which means it is non-negative:*

$$v^T(\nabla^2 f(\beta))v \geq 0.$$

The Hessian is positive semidefinite. This proves that the negative log-likelihood for logistic regression is a convex function. Since this is an unconstrained problem, it is a convex optimization problem.

7.4 Application 2: Lasso Regression

In high-dimensional statistics (where the number of features n is large, possibly larger than the number of samples m), we often assume that only a small subset of features are actually relevant to the outcome. We want a model that performs both regression and **feature selection**, meaning it automatically sets the coefficients of irrelevant features to zero.

This is achieved by adding an ℓ_1-norm penalty to the OLS objective, a method called **Lasso** (Least Absolute Shrinkage and Selection Operator).

Definition 7.1 (The Lasso Problem)**.** *The Lasso objective function is:*

$$f(\beta) = \underbrace{||X\beta - y||_2^2}_{\textit{OLS Loss}} + \underbrace{\lambda||\beta||_1}_{\textit{L1 Regularization}} .$$

where $\lambda > 0$ is a tuning parameter that controls the strength of the penalty. The optimization problem is $\min_{\beta \in \mathbb{R}^n} f(\beta)$.

Example 7.2 (Proving the Convexity of the Lasso Objective)**.** *Show that the Lasso objective function is convex.*

Proof: *We will use the property that* ***the sum of convex functions is convex****. The Lasso objective is a sum of two parts:*

1. ***The OLS Loss Term:*** *Let $g(\beta) = ||X\beta - y||_2^2$. In Lecture 2, we computed its Hessian:*

   $$\nabla^2 g(\beta) = 2X^T X.$$

   For any vector v, $v^T(X^TX)v = (Xv)^T(Xv) = ||Xv||_2^2 \geq 0$. Therefore, the matrix X^TX is positive semidefinite, and the OLS loss is a convex function.

2. ***The L1 Regularization Term:*** *Let $h(\beta) = \lambda||\beta||_1 = \lambda \sum_{i=1}^n |\beta_i|$. The absolute value function $|\cdot|$ is a classic example of a convex function. A scaled sum of convex functions is also convex. Therefore, $h(\beta)$ is a convex function.*

Since the Lasso objective $f(\beta) = g(\beta) + h(\beta)$ is the sum of two convex functions, it is itself a convex function. The problem is unconstrained, so Lasso is a convex optimization problem. This guarantees that we can find the globally optimal set of sparse coefficients.

```python
import jax
import jax.numpy as jnp

# Define the two components of the Lasso objective
def ols_loss(beta, X, y):
  """The data fidelity term."""
  return jnp.sum((X @ beta - y)**2)

def l1_regularization(beta, lambda_param):
  """The regularization term."""
  return lambda_param * jnp.sum(jnp.abs(beta))

# The full Lasso objective is the sum of the two
def lasso_objective(beta, X, y, lambda_param):
  return ols_loss(beta, X, y) + l1_regularization(beta, lambda_param)

# Get the gradient of the objective. JAX handles the non-differentiable
# absolute value function by providing a subgradient.
grad_lasso = jax.grad(lasso_objective)

# Generate some synthetic data
key = jax.random.PRNGKey(123)
m, n = 50, 5
X_data = jax.random.normal(key, (m, n))
true_beta = jnp.array([1.5, -2.0, 3.0, 0.0, 0.0]) # A sparse true beta
y_data = X_data @ true_beta + 0.5 * jax.random.normal(key, (m,))
beta_guess = jnp.zeros(n)
lambda_val = 10.0

# Calculate the gradient at our initial guess
subgradient = grad_lasso(beta_guess, X_data, y_data, lambda_val)
```

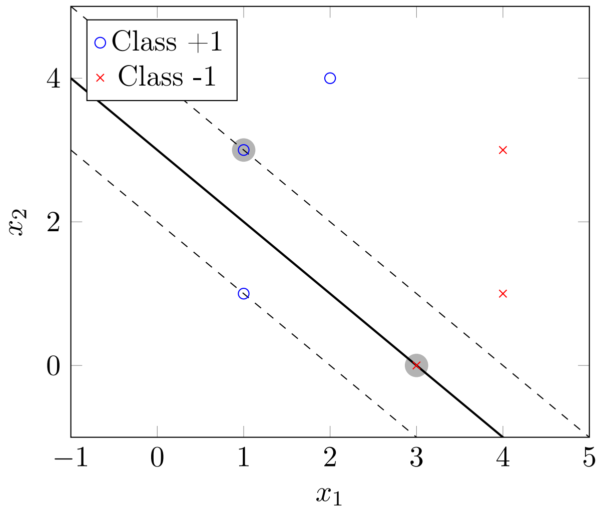
```
print("Subgradient of the Lasso objective at beta=0:\n", subgradient)
# Note: Since the problem is convex, we can use gradient descent (or more
# advanced first-order methods) to find the global minimum.
```
Listing 12: Defining the Lasso objective function in JAX.

7.5 Application 3: Support Vector Machines (SVMs)

A Support Vector Machine is another powerful model for binary classification. For linearly separable data, the goal of an SVM is to find the hyperplane that separates the two classes with the **maximum possible margin**.

The "margin" is like a "street" separating the two classes of data. The SVM finds the widest possible street. The hyperplane lies in the middle of the street, and the edges of the street are defined by the closest points from each class, which are called the **support vectors**.

SVM Maximum Margin Classifier



Example 7.3 (Formulating the SVM as a Convex Problem)**.** *Let the data be* (x_i, y_i) *where* $y_i \in \{-1, +1\}$*. A separating hyperplane is defined by* $\beta^T x + \beta_0 = 0$*. The margin is proportional to* $1/||\beta||$*. Therefore, maximizing the margin is equivalent to minimizing* $||\beta||$*, or more conveniently, minimizing* $\frac{1}{2}||\beta||^2$*.*

The constraints require that all points are correctly classified and lie outside the "gutter". This can be written as a single inequality:

$$y_i(\beta^T x_i + \beta_0) \geq 1 \quad \textit{for all } i = 1, \dots, m.$$

The full optimization problem for a hard-margin SVM is:

$$\begin{aligned} \min_{\beta, \beta_0} \quad & \frac{1}{2}||\beta||^2 \\ \textit{subject to} \quad & y_i(\beta^T x_i + \beta_0) \geq 1, \quad i = 1, \dots, m. \end{aligned}$$

Show that this is a convex optimization problem.

Proof:

1. ***The Objective Function:*** *The objective is* $f(\beta, \beta_0) = \frac{1}{2}\beta^T\beta$*. This is a quadratic function. Its Hessian is:*

$$\nabla^2 f = \begin{bmatrix} I & \mathbf{0} \\ \mathbf{0}^T & 0 \end{bmatrix}$$

where I *is the* $n \times n$ *identity matrix. This matrix is positive semidefinite (its eigenvalues are 1s and 0). Thus, the objective function is convex.*

2. ***The Feasible Set:*** *The constraints are of the form* $g_i(\beta, \beta_0) = 1 - y_i(\beta^T x_i + \beta_0) \leq 0$*. Each function* g_i *is an* ***affine*** *(linear) function of the variables* β *and* β_0*. Affine functions are both convex and concave.*

   Since the feasible set is defined by the intersection of affine inequalities, it is a convex set (a polytope).

Because we are minimizing a convex function over a convex set, the hard-margin SVM is a convex optimization problem. Specifically, it is a ***Quadratic Program (QP)****.*

8 The Lagrangian and KKT Conditions

8.1 The Challenge of Constraints

In previous lectures, we have seen that for unconstrained minimization, the condition for optimality is simple: the gradient must be zero. However, this is not sufficient for constrained problems. The optimal solution might lie on the **boundary** of the feasible set, where the gradient of the objective function is not necessarily zero.

Consider minimizing a function $f(x)$ subject to a constraint $g(x) \leq 0$. At a solution x^* on the boundary (where $g(x^*) = 0$), we can't move in certain directions. Intuitively, at this point, the "downhill" direction of our objective function, $-\nabla f(x^*)$, must be pointing "into" the infeasible region. If it were pointing into the feasible region, we could move a little further and improve our objective, meaning we weren't at an optimum.

This means that at the optimum, the gradient of the objective function must be "balanced" by the gradients of the constraints that are active. The **Lagrangian function** and the **KKT conditions** provide the mathematical framework to formalize this intuition.

8.2 The Lagrangian Function

The core idea of the Lagrangian method is to convert a constrained optimization problem into an unconstrained one by adding a "penalty" for each constraint to the objective function.

Consider a general optimization problem:

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \quad & f(x) \\ \text{subject to} \quad & g_i(x) \leq 0, \quad i = 1, \ldots, m \\ & h_j(x) = 0, \quad j = 1, \ldots, p. \end{aligned}$$

Definition 8.1 (The Lagrangian). *The **Lagrangian** function $L(x, \lambda, \mu)$ associated with this problem is:*

$$L(x, \lambda, \mu) = f(x) + \sum_{i=1}^{m} \lambda_i g_i(x) + \sum_{j=1}^{p} \mu_j h_j(x).$$

*The variables $\lambda = (\lambda_1, \ldots, \lambda_m)$ and $\mu = (\mu_1, \ldots, \mu_p)$ are called the **Lagrange multipliers** or **dual variables**. Each multiplier can be thought of as the "price" or "penalty" for violating its corresponding constraint.*

By choosing the right "prices," we can find a point where the gradient of this new, unconstrained function is zero, and that point will be the solution to our original, constrained problem.

8.3 The Karush-Kuhn-Tucker (KKT) Conditions

For problems where the objective and constraint functions are differentiable (and satisfy some regularity conditions), the **Karush-Kuhn-Tucker (KKT) conditions** provide the necessary conditions for a point x^* to be a local minimum. If the problem is convex, these conditions are also sufficient.

For a point x^* to be optimal, there must exist Lagrange multipliers λ^* and μ^* such that the following four conditions hold:

1. **Stationarity:** The gradient of the Lagrangian with respect to x must be zero.

$$\nabla_x L(x^*, \lambda^*, \mu^*) = \nabla f(x^*) + \sum_{i=1}^{m} \lambda_i^* \nabla g_i(x^*) + \sum_{j=1}^{p} \mu_j^* \nabla h_j(x^*) = \mathbf{0}.$$

Intuition: The gradient of the objective is a linear combination of the gradients of the active constraints. The forces are "balanced."

2. **Primal Feasibility:** The point must satisfy all original constraints.

$$g_i(x^*) \le 0 \text{ for all } i, \quad \text{and} \quad h_j(x^*) = 0 \text{ for all } j.$$

3. **Dual Feasibility:** The Lagrange multipliers for the inequality constraints must be non-negative.

$$\lambda_i^* \ge 0 \text{ for all } i.$$

Intuition: We only apply a penalty for $g_i(x) > 0$. A negative price would reward us for violating the constraint, which doesn't make sense.

4. **Complementary Slackness:** The product of each inequality multiplier and its constraint must be zero.

$$\lambda_i^* g_i(x^*) = 0 \text{ for all } i.$$

Intuition: This is the most crucial condition for interpretation. For each constraint i:

- If the constraint is **inactive** ($g_i(x^*) < 0$), its price must be zero ($\lambda_i^* = 0$). There is no penalty for a constraint that isn't a bottleneck.
- If the constraint's price is positive ($\lambda_i^* > 0$), the constraint must be **active** ($g_i(x^*) = 0$). The solution is pushing right up against this boundary.

8.4 Application 1: The Water-filling Algorithm

This is a classic application of KKT conditions in information theory, used for power allocation in communication systems.

Example 8.1 (Optimal Power Allocation)**.** *Imagine you have n parallel communication channels, and a total power budget of P_{total}. The capacity (data rate) of channel i is given by $C_i = \log(1 + p_i/\sigma_i^2)$, where p_i is the power allocated to channel i and σ_i^2 is the noise power in that channel. The goal is to allocate power to maximize the total capacity.*

The optimization problem is:

$$\begin{aligned} \max_{p_1,\dots,p_n} \quad & \sum_{i=1}^n \log(1 + p_i/\sigma_i^2) \\ \textit{subject to} \quad & \sum_{i=1}^n p_i \le P_{total} \\ & p_i \ge 0 \textit{ for all } i. \end{aligned}$$

This is equivalent to minimizing the negative of the objective:

$$f(p) = -\sum_{i=1}^n \log(1 + p_i/\sigma_i^2).$$

The constraints are $g_0(p) = \sum p_i - P_{total} \le 0$ and $g_i(p) = -p_i \le 0$.

Proof:

1. ***Form the Lagrangian.*** *We have one multiplier μ for the total power constraint and n multipliers λ_i for the non-negativity constraints.*

$$L(p, \lambda, \mu) = -\sum_{i=1}^n \log(1 + p_i/\sigma_i^2) + \mu \left(\sum_{i=1}^n p_i - P_{total} \right) + \sum_{i=1}^n \lambda_i(-p_i).$$

2. ***Apply Stationarity.*** *We take the partial derivative with respect to each p_k and set it to zero:*
$$\frac{\partial L}{\partial p_k} = -\frac{1}{1 + p_k/\sigma_k^2} \cdot \frac{1}{\sigma_k^2} + \mu - \lambda_k = 0 \implies \frac{1}{p_k + \sigma_k^2} = \mu - \lambda_k.$$

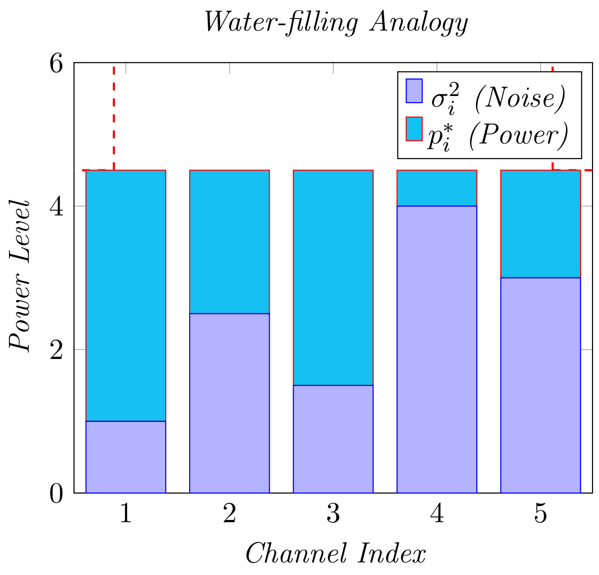
3. ***Apply Complementary Slackness.*** *We have two cases for each channel k:*

   - ***Case 1:*** $p_k^* > 0$. *This implies its non-negativity constraint is inactive, so by complementary slackness, its multiplier λ_k^* must be 0. The stationarity condition becomes:*
   $$\frac{1}{p_k^* + \sigma_k^2} = \mu^* \implies p_k^* = \frac{1}{\mu^*} - \sigma_k^2.$$
   Since $p_k^* > 0$, this can only happen if $1/\mu^* > \sigma_k^2$.
   - ***Case 2:*** $p_k^* = 0$. *In this case, λ_k^* can be non-negative. The stationarity condition gives $\frac{1}{\sigma_k^2} = \mu^* - \lambda_k^*$. Since $\lambda_k^* \geq 0$, this implies $\mu^* \geq 1/\sigma_k^2$.*

Interpretation (The Water Level): *Let's combine these insights. The dual variable μ^* acts like a constant "water level." The term $1/\mu^*$ is the height of the water. The noise powers σ_k^2 represent the "bottom floor" of each channel's container.*

The optimal power allocation strategy is:
$$p_k^* = \max\left(0, \frac{1}{\mu^*} - \sigma_k^2\right).$$

We "pour" our total power P_{total} into these containers. The water fills up the containers with the lowest floors (least noise) first. We stop pouring when the total amount of water equals P_{total}. Very noisy channels, where the floor σ_k^2 is above the final water level $1/\mu^*$, receive zero power.



8.5 Application 2: Insights into Support Vector Machines

Let's revisit the hard-margin SVM problem. The KKT conditions provide a stunning insight into the nature of its solution. The primal SVM problem is:
$$\min_{\beta,\beta_0} \frac{1}{2}||\beta||^2 \quad \text{subject to} \quad 1 - y_i(\beta^T x_i + \beta_0) \leq 0, \quad i = 1, \ldots, m.$$

The Lagrangian is (using multipliers $\alpha_i \geq 0$):

$$L(\beta, \beta_0, \alpha) = \frac{1}{2}||\beta||^2 + \sum_{i=1}^{m} \alpha_i(1 - y_i(\beta^T x_i + \beta_0)).$$

Applying the KKT conditions yields two key results:

1. **Stationarity (w.r.t. β):** Setting $\nabla_\beta L = 0$ gives:

$$\beta - \sum_{i=1}^{m} \alpha_i y_i x_i = \mathbf{0} \implies \beta^* = \sum_{i=1}^{m} \alpha_i^* y_i x_i.$$

   This is a remarkable result! It says that the optimal weight vector β^* (which defines the separating hyperplane) is simply a **linear combination of the feature vectors** x_i. The coefficients of this combination are the Lagrange multipliers.

2. **Complementary Slackness:** $\alpha_i^*(1 - y_i((\beta^*)^T x_i + \beta_0^*)) = 0$. This tells us that if a multiplier α_i^* is greater than zero, then its corresponding constraint must be active:

$$y_i((\beta^*)^T x_i + \beta_0^*) = 1.$$

   This means the data point (x_i, y_i) lies exactly on the margin. These are the crucial points that "support" the separating hyperplane. They are called the **support vectors**. For any other point not on the margin, its multiplier α_i^* must be zero.

These two conditions together imply that the solution is determined *only* by the support vectors. All the other data points could be removed without changing the solution! This is the source of the name "Support Vector Machine".

9 Mixed-Integer Programming

9.1 Introduction: Beyond Continuous Variables

All the optimization problems we have studied so far—from linear regression to support vector machines—have involved **continuous** decision variables. We have been searching for the best set of real-valued parameters $\beta \in \mathbb{R}^n$.

However, many real-world and statistical problems involve decisions that are not continuous, but discrete. For example:

- Should we include a particular feature in a model or not? (a yes/no decision)
- Which of 5 different treatments should be assigned to a patient? (a choice from a set)
- How many servers should a company purchase to handle expected web traffic? (an integer value)

These problems, which mix continuous variables with discrete or integer-constrained variables, fall into the domain of **Mixed-Integer Programming (MIP)**. MIP provides an incredibly powerful framework for modeling logical constraints and combinatorial choices, allowing us to find provably optimal solutions to problems that are otherwise computationally intractable.

9.2 What is a Mixed-Integer Program?

A Mixed-Integer Program is an optimization problem where the objective and constraints are typically linear or quadratic, but some of the decision variables are restricted to be integers.

Definition 9.1 (Mixed-Integer Linear Program (MILP))**.** *A mixed-integer linear program has the standard form:*

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \quad & c^T x \\ \textit{subject to} \quad & Ax \le b \\ & x_i \in \mathbb{Z} \quad \textit{for all } i \in \mathcal{I} \end{aligned}$$

where $\mathcal{I}$ is a subset of the variable indices $\{1, \ldots, n\}$. A particularly important special case is when the integer variables are binary, i.e., $x_i \in \{0, 1\}$.

If the objective function is quadratic, the problem is a Mixed-Integer Quadratic Program (MIQP). The key challenge of MIPs is their computational complexity. While LPs can be solved efficiently (in polynomial time), the addition of integer constraints makes the problem **NP-hard**. The feasible set is no longer convex (it's a set of disconnected points or regions), so we can't rely on simple gradient-based methods. Instead, solvers use sophisticated algorithms like *Branch and Bound* which intelligently explore a tree of possible integer solutions.

9.3 The "Big-M" Method: A Powerful Modeling Trick

A core challenge in MIP is converting logical statements into mathematical inequalities. The "Big-M" method is the standard technique for doing this. It uses a binary variable $z \in \{0, 1\}$ to control whether a constraint is "on" or "off."

Suppose we want to model the statement: "If decision A is made, then constraint B must be satisfied." We can represent decision A with a binary variable z, where $z = 1$ means "yes" and $z = 0$ means "no." Let constraint B be of the form $f(x) \le 0$.

The logical statement is: "If $z = 1$, then $f(x) \le 0$."

We can enforce this with the single linear inequality:

$$f(x) \le M(1 - z)$$

where M is a large, positive constant (the "Big M"). Let's see why this works:

- If $z = 1$ (decision is "yes"), the inequality becomes $f(x) \leq M(1-1) \implies f(x) \leq 0$. The original constraint is enforced.
- If $z = 0$ (decision is "no"), the inequality becomes $f(x) \leq M(1-0) \implies f(x) \leq M$. As long as M is chosen to be larger than any possible value of $f(x)$, this constraint becomes redundant and has no effect.

9.4 Application 1: Capital Budgeting for Project Selection

One of the most common strategic decisions a company faces is capital budgeting: deciding which potential projects or investments to fund from a limited pool of capital to achieve the best possible return. This is not just about picking the projects with the highest individual returns, as they may have different costs and be subject to various strategic, logical, or operational constraints. This makes it a perfect application for Mixed-Integer Programming.

Example 9.1 (Project Portfolio Optimization at "InnovateCorp")**.** *InnovateCorp's strategy team has identified four potential high-impact projects for the next fiscal year. The company has a total capital budget of \$10 million. The team needs to select the portfolio of projects that maximizes the total expected Net Present Value (NPV), while adhering to the budget and some key strategic dependencies.*

Problem Data: *The details for the four projects are as follows:*

Project	***Capital Cost (\$M)***	***Expected NPV (\$M)***
A: Build New Factory	*7*	*9*
B: Launch Marketing Campaign	*4*	*5*
C: Fund R&D Initiative	*3*	*4*
D: Overhaul IT Infrastructure	*3*	*4*

Strategic Constraints:

1. ***Mutual Exclusivity:*** *The New Factory and the IT Infrastructure overhaul require the same key personnel. Therefore, the company can only choose one of these two projects, not both.*
2. ***Contingency:*** *The Marketing Campaign (Project B) is only viable if the R&D Initiative (Project C) is also funded, as the campaign will be based on the new product developed.*

1. The MILP Formulation

We can model this decision problem as a Mixed-Integer Linear Program, specifically a Binary Integer Program, as each decision is a simple "yes" or "no".

Step 1: Define Decision Variables We introduce a binary variable $z_j \in \{0,1\}$ for each project $j \in \{A, B, C, D\}$.

$$z_j = \begin{cases} 1 & \text{if project } j \text{ is selected} \\ 0 & \text{if project } j \text{ is not selected} \end{cases}$$

Step 2: Define the Objective Function The goal is to maximize the total NPV of the selected projects.

$$\max_{z_A, z_B, z_C, z_D} \quad 9z_A + 5z_B + 4z_C + 4z_D$$

Step 3: Define the Constraints We translate each business rule into a mathematical inequality.

- ***Budget Constraint:*** *The total cost of selected projects cannot exceed \$10M.*

$$7z_A + 4z_B + 3z_C + 3z_D \leq 10$$

- ***Mutual Exclusivity Constraint:*** *At most one of Project A and Project D can be selected.*

$$z_A + z_D \leq 1$$

(If $z_A = 1$, then z_D must be 0, and vice-versa. If both are 0, the constraint is also satisfied).

- ***Contingency Constraint:*** *If Project B is selected ($z_B = 1$), then Project C must also be selected ($z_C = 1$).*

$$z_B \leq z_C \quad (\textit{or equivalently, } z_B - z_C \leq 0)$$

(If $z_B = 1$, this forces $z_C = 1$. If $z_B = 0$, the constraint is $0 \leq z_C$, which is always true for a binary variable).

2. Solving and Interpreting the Optimal Decision

This problem, although small, is non-trivial to solve by hand due to the interplay of the constraints. A MILP solver will explore the combinatorial space of possible project portfolios and is guaranteed to find the one that yields the highest possible NPV while respecting all rules. The solver finds the optimal solution to be:

$$z_A = 0, \quad z_B = 1, \quad z_C = 1, \quad z_D = 1$$

Business Interpretation: *The optimal decision for InnovateCorp is to fund the **Marketing Campaign**, the **R&D Initiative**, and the **IT Overhaul**, and to reject the New Factory project. Let's check the implications:*

- ***Projects Selected:*** *B, C, and D.*
- ***Total Cost:*** *\$4M (B) + \$3M (C) + \$3M (D) = \$10M. This exactly uses the entire budget.*
- ***Total NPV:*** *\$5M (B) + \$4M (C) + \$4M (D) = \$13M.*
- ***Constraints Check:*** *The mutual exclusivity ($z_A + z_D \leq 1$) is met since $0 + 1 \leq 1$. The contingency ($z_B \leq z_C$) is met since $1 \leq 1$.*

This is the provably best outcome. Notice that simply picking the project with the highest individual NPV (New Factory, \$9M) would have been a suboptimal choice. Funding it would cost \$7M, leaving only \$3M to fund Project C, for a total NPV of \$9M + \$4M = \$13M. In this case, both portfolios yield the same total NPV, but the B+C+D portfolio is more diversified. The MIP solver guarantees finding the globally best combination among all valid portfolios.

```
import numpy as np
from scipy.optimize import milp, LinearConstraint

# --- 1. Define the Problem Data ---
# Projects: A, B, C, D
project_names = np.array(['A: New Factory', 'B: Marketing Campaign', 'C: R&D
    Initiative', 'D: IT Overhaul'])
npv = np.array([9, 5, 4, 4])
costs = np.array([7, 4, 3, 3])
budget = 10.0

# SciPy's milp minimizes, so we use the negative of the NPV for the cost vector
    `c`.
c = -npv

# --- 2. Define Constraints ---
# The variables are z_A, z_B, z_C, z_D
# Constraints are formulated as A_ub @ z <= b_ub
```

```python
# Constraint 1: Budget
# 7*zA + 4*zB + 3*zC + 3*zD <= 10
budget_constraint_row = costs

# Constraint 2: Mutual Exclusivity (A and D)
# zA + zD <= 1
exclusivity_constraint_row = np.array([1, 0, 0, 1])

# Constraint 3: Contingency (B requires C)
# zB - zC <= 0
contingency_constraint_row = np.array([0, 1, -1, 0])

# Combine into a single matrix and vector for SciPy
A_ub = np.vstack([
    budget_constraint_row,
    exclusivity_constraint_row,
    contingency_constraint_row
])

b_ub = np.array([budget, 1, 0])

# Create the LinearConstraint object
constraints = LinearConstraint(A_ub, lb=-np.inf, ub=b_ub)

# --- 3. Define Integrality and Bounds ---
# All variables are binary (integer-constrained and bounded between 0 and 1)
integrality = np.ones_like(c) # 1 for integer
bounds = (0, 1) # Bounds for all variables (can be a single tuple for all)

# --- 4. Solve the MILP ---
print("Solving the Capital Budgeting MILP...")
result = milp(c=c, constraints=constraints, integrality=integrality, bounds=
    bounds)

# --- 5. Interpret the Solution ---
if result.success:
    selected_projects_mask = np.round(result.x).astype(bool)
    selected_names = project_names[selected_projects_mask]
    total_cost = np.sum(costs[selected_projects_mask])
    total_npv = np.sum(npv[selected_projects_mask])

    print("\n--- Optimal Project Portfolio ---")
    print("The following projects should be funded:")
    for name in selected_names:
        print(f"  - {name}")

    print("\n--- Financial Summary ---")
    print(f"Total Capital Cost: ${total_cost:,.2f}M / ${budget:,.2f}M")
    print(f"Total Expected NPV: ${total_npv:,.2f}M")

    # Verify constraints were met
    z = result.x
    print("\nConstraint Verification:")
    print(f"Mutual Exclusivity (A+D <= 1): {z[0] + z[3]:.0f} <= 1 (Met)")
    print(f"Contingency (B <= C): {z[1]:.0f} <= {z[2]:.0f} (Met)")
else:
    print("MILP solver failed to find a solution.")
    print("Message:", result.message)
```

Listing 13: Solving the Capital Budgeting problem with SciPy MILP.

9.5 Application 2: Sparse Linear Regression (Best Subset Selection)

In our lecture on Lasso, we sought a sparse regression vector β by adding an ℓ_1 penalty. Lasso is a *convex relaxation* that encourages, but does not guarantee, sparsity. The original, statistically ideal problem is **Best Subset Selection**: find the linear model that minimizes the squared error using *at most k non-zero coefficients.*

The objective is $\min ||X\beta - y||_2^2$, subject to the constraint $||\beta||_0 \leq k$, where the ℓ_0-"norm" simply counts the number of non-zero elements in β. This counting function is non-convex and combinatorial, making the problem NP-hard. However, we can formulate it exactly as a MIQP.

Example 9.2 (Formulating Best Subset Selection as a MIQP). *Show how to formulate the problem of finding the best regression model with at most k features as a Mixed-Integer Quadratic Program.*

Formulation:

1. ***The Objective Function.*** *The goal is to minimize the sum of squared errors, which is a convex quadratic function of β:*

$$\min_{\beta} \quad ||X\beta - y||_2^2.$$

2. ***Introducing Binary Variables.*** *To model the "on/off" nature of each feature, we introduce a binary variable $z_j \in \{0, 1\}$ for each coefficient β_j. Our intention is for $z_j = 1$ if β_j is non-zero, and $z_j = 0$ if β_j is zero.*

3. ***The Cardinality Constraint.*** *With these binary variables, the constraint of having at most k non-zero coefficients is simple to write:*

$$\sum_{j=1}^{n} z_j \leq k.$$

4. ***Linking β and z with Big-M.*** *Now we must enforce the logic: if a feature is "off" ($z_j = 0$), its coefficient must be zero ($\beta_j = 0$). This is a perfect use case for the Big-M method. We need to ensure that β_j is bounded by some constant M when $z_j = 1$, and is forced to be 0 when $z_j = 0$. This requires two constraints:*

$$\beta_j \leq M z_j$$
$$\beta_j \geq -M z_j$$

If $z_j = 1$, these become $\beta_j \leq M$ and $\beta_j \geq -M$, which simply bound the coefficient. If $z_j = 0$, they become $\beta_j \leq 0$ and $\beta_j \geq 0$, which forces $\beta_j = 0$. M must be chosen as an upper bound on the possible magnitude of any coefficient.

5. ***The Full MIQP Formulation.*** *Assembling all the pieces, we get the following MIQP:*

$$\begin{aligned} \min_{\beta \in \mathbb{R}^n, z \in \{0,1\}^n} \quad & ||X\beta - y||_2^2 \\ \textit{subject to} \quad & \sum_{j=1}^{n} z_j \leq k \\ & \beta_j \leq M z_j, \quad j = 1, \ldots, n \\ & \beta_j \geq -M z_j, \quad j = 1, \ldots, n. \end{aligned}$$

This problem is non-convex due to the integer variables, but it can be solved to proven optimality by modern MIP solvers, giving the true "best subset" solution.

9.6 Solving and Practical Considerations

While JAX is designed for gradient-based optimization on continuous variables, specialized MIP solvers are required for integer problems. Libraries like ‘CVXPY‘ provide a high-level interface to these solvers. The code below shows how to formulate and solve the MIQP for best subset selection.

The trade-off is clear:

- **Lasso (Convex Relaxation):** Very fast and scalable, provides a good approximate solution.
- **MIP (Exact Formulation):** Computationally expensive, but guarantees finding the provably optimal feature subset. It is feasible for problems where the number of features n is in the dozens or hundreds, but not for the millions of features common in deep learning.

```
import numpy as np
from scipy.optimize import milp, LinearConstraint
from sklearn.linear_model import Lasso

# Generate synthetic data with a known sparse structure
np.random.seed(1)
m, n = 50, 20  # 50 samples, 20 features
X = np.random.randn(m, n)
true_beta = np.zeros(n)
true_beta[:5] = np.array([5, -4, 3, -2, 1]) # Only 5 non-zero features
y = X @ true_beta + 0.5 * np.random.randn(m)

# --- 1. Formulate Best Subset Selection with L1 loss as a MILP ---
# The problem is min ||X*beta - y||_1 s.t. ||beta||_0 <= k.
k = 5       # We want to find the best model with at most 5 features
M = 10.0    # A reasonable upper bound for the coefficients

# The decision variables are stacked into a single vector ‘x‘:
# x = [beta+, beta-, z, u]
# beta = beta+ - beta- to handle unrestricted sign
# z are binary variables to indicate non-zero betas
# u are slack variables to linearize the L1 norm
# Sizes: beta+(n), beta-(n), z(n), u(m) -> Total vars: 3n + m
num_vars = 3 * n + m

# --- Objective Function ---
# We want to minimize sum(u_i), which is the L1 loss.
# The cost vector ‘c‘ has 1s for ‘u‘ variables and 0s elsewhere.
c = np.concatenate([np.zeros(3 * n), np.ones(m)])

# --- Integrality Constraints ---
# beta+, beta-, u are continuous (0); z are integers (1).
integrality = np.concatenate([np.zeros(2 * n), np.ones(n), np.zeros(m)])

# --- Bounds ---
# beta+, beta-, u are non-negative. z are binary (0 or 1).
bounds = [(0, None)] * (2*n) + [(0, 1)] * n + [(0, None)] * m

# --- Constraint Matrix (A_ub @ x <= b_ub) ---
A_ub_list = []
b_ub_list = []

# 1. Cardinality constraint: sum(z) <= k
A_card = np.concatenate([np.zeros(2 * n), np.ones(n), np.zeros(m)]).reshape(1,
    -1)
b_card = np.array([k])
A_ub_list.append(A_card)
```

```python
b_ub_list.append(b_card)

# 2. Big-M constraints: beta+ <= M*z and beta- <= M*z
# beta+ - M*z <= 0
A_bigM_p = np.hstack([np.eye(n), np.zeros((n, n)), -M * np.eye(n), np.zeros((n,
    m))])
# beta- - M*z <= 0
A_bigM_n = np.hstack([np.zeros((n, n)), np.eye(n), -M * np.eye(n), np.zeros((n,
    m))])
A_ub_list.extend([A_bigM_p, A_bigM_n])
b_ub_list.extend([np.zeros(n), np.zeros(n)])

# 3. L1 loss linearization constraints:
# y - X(beta+ - beta-) <= u  =>  -X@beta+ + X@beta- - u <= -y
# -(y - X(beta+ - beta-)) <= u => X@beta+ - X@beta- - u <= y
A_l1_top = np.hstack([-X, X, np.zeros((m, n)), -np.eye(m)])
A_l1_bot = np.hstack([X, -X, np.zeros((m, n)), -np.eye(m)])
A_ub_list.extend([A_l1_top, A_l1_bot])
b_ub_list.extend([-y, y])

# Assemble final constraint matrix and vector
A_ub = np.vstack(A_ub_list)
b_ub = np.concatenate(b_ub_list)
constraints = LinearConstraint(A_ub, -np.inf, b_ub)

# Solve the MILP
print("Solving Best Subset L1 Regression MILP...")
res = milp(c=c, constraints=constraints, bounds=bounds, integrality=integrality
   )

# --- 2. For comparison, solve with Lasso (convex relaxation) ---
lasso = Lasso(alpha=0.2) # alpha chosen to give a sparse-ish solution
lasso.fit(X, y)
beta_lasso = lasso.coef_

# --- 3. Print and compare the results ---
print("\n--- Results ---")
print("True non-zero indices:", np.where(true_beta != 0)[0])
if res.success:
    beta_p = res.x[0:n]
    beta_n = res.x[n:2*n]
    milp_solution = beta_p - beta_n
    milp_solution[np.abs(milp_solution) < 1e-5] = 0
    print("\nMILP Solution (Best Subset with L1 Loss):")
    print("Non-zero indices:", np.where(milp_solution != 0)[0])
    print("Beta values:\n", milp_solution.round(3))
else:
    print("\nMILP solver failed:", res.message)

print("\nLasso Solution (Convex Relaxation):")
print("Non-zero indices:", np.where(np.abs(beta_lasso) > 1e-5)[0])
print("Beta values:\n", beta_lasso.round(3))
```
Listing 14: Solving Best Subset Selection with L1 Loss as a MILP.

10 Dynamic Optimization and Optimal Control

10.1 Introduction: Optimization over Time

In all our previous discussions, the problems have been *static*. We sought a single, optimal decision vector x^* that minimized a function, assuming all information was available at once. For example, in linear regression, we find the single best set of coefficients β based on the entire dataset.

However, many of the most important problems in statistics, economics, and engineering are *dynamic*. Decisions are not made once, but sequentially over time. The outcome of today's decision affects the situation we will face tomorrow, which in turn affects our decision tomorrow, and so on.

Consider these scenarios:

- **An Inventory Problem:** How many units of a product should a store order each week? Ordering too much incurs storage costs, while ordering too little leads to lost sales. The decision this week depends on current inventory and affects next week's starting inventory.
- **A Clinical Trial:** A doctor administers one of several treatments to a patient. Each month, the patient's health state is observed, and the doctor must decide whether to continue the current treatment or switch to another. The goal is to find a sequence of treatments—a *policy*—that maximizes the patient's long-term health.
- **Training an AI to play a game:** At each turn, a program must choose a move from a set of legal moves. Each move leads to a new board state. The goal is not to get the best immediate reward, but to make a sequence of moves that leads to winning the game.

These problems require a new set of tools. **Dynamic Programming (DP)** is the mathematical framework for solving such sequential decision-making problems under uncertainty. It provides a way to break down a complex, multi-stage problem into a series of simpler, single-stage problems.

10.2 The Framework of Markov Decision Processes (MDPs)

The standard mathematical model for sequential decision-making is the **Markov Decision Process (MDP)**. An MDP formalizes the problem by defining its essential components.

Definition 10.1 (Markov Decision Process)**.** *An MDP is a 5-tuple (S, A, P, R, γ) where:*

1. *S is a finite set of **states**. A state $s \in S$ is a complete description of the system at a particular time.*
2. *A is a finite set of **actions**. In each state s, we can choose an action $a \in A$.*
3. *$P(s'|s, a)$ is the **transition probability function**. It gives the probability of moving to state s' in the next time step, given that we were in state s and took action a.*
4. *$R(s, a, s')$ is the **reward function**. It gives the immediate reward received after transitioning from state s to state s' due to action a.*
5. *$\gamma \in [0, 1)$ is the **discount factor**. It determines the present value of future rewards. A reward received k steps in the future is worth only γ^k times what it would be worth today.*

*A key assumption is the **Markov Property**: the transition probability $P(s'|s, a)$ depends only on the current state and action, not on the entire history of previous states and actions.*

Our goal in an MDP is to find a **policy**, denoted π. A policy is a rule that tells us which action to take in any given state.

Definition 10.2 (Policy)**.** *A policy $\pi(a|s)$ is a function that gives the probability of taking action a when in state s. A* ***deterministic policy*** *is a special case where for each state s, there is a single action $a = \pi(s)$ that is always chosen.*

The objective is to find an **optimal policy** π^* that maximizes the expected cumulative discounted reward, starting from any state s.

10.3 The Bellman Equation

How can we determine if a policy is optimal? The key insight, developed by Richard Bellman, is that the problem has an optimal recursive structure. The value of being in a state today is related to the values of the states we might end up in tomorrow.

Let's define the **value function** $V^\pi(s)$ as the expected cumulative discounted reward starting from state s and following policy π thereafter.

$$V^\pi(s) = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \mid s_0 = s\right].$$

The Bellman equation provides a recursive definition for this value function. It decomposes the value of a state into two parts: the immediate expected reward, and the discounted expected value of the next state.

The Bellman Equation for a Policy π:

$$V^\pi(s) = \sum_a \pi(a|s) \sum_{s'} P(s'|s,a)\left[R(s,a,s') + \gamma V^\pi(s')\right].$$

This is a system of linear equations, one for each state s, that can be solved to find the value function for a given policy.

More importantly, the *optimal* value function $V^*(s) = \max_\pi V^\pi(s)$ must satisfy a special, non-linear version of this equation.

The Bellman Optimality Equation:

$$V^*(s) = \max_{a\in A} \sum_{s'} P(s'|s,a)\left[R(s,a,s') + \gamma V^*(s')\right].$$

The 'max' operator is crucial. It says that the value of an optimal state is obtained by taking the action that leads to the best possible outcome, assuming we act optimally forever after. Once we have found the optimal value function $V^*(s)$, we can easily find the optimal policy π^* by choosing the action that achieves this maximum at each state:

$$\pi^*(s) = \arg\max_{a\in A} \sum_{s'} P(s'|s,a)\left[R(s,a,s') + \gamma V^*(s')\right].$$

10.4 Algorithms for Solving MDPs

The Bellman Optimality Equation provides a condition for optimality, but we need an algorithm to find the V^* that satisfies it. The most fundamental DP algorithm is **Value Iteration**.

Definition 10.3 (Value Iteration)**.** *Value Iteration is an iterative algorithm that finds the optimal value function V^* by repeatedly applying the Bellman Optimality Equation as an update rule.*

1. *Initialize the value function $V_0(s) = 0$ for all states s.*

2. *For $k = 0, 1, 2, \ldots$ until convergence:*

$$V_{k+1}(s) \leftarrow \max_{a\in A} \sum_{s'} P(s'|s,a)\left[R(s,a,s') + \gamma V_k(s')\right] \quad \textit{for all } s \in S.$$

3. *Stop when the maximum change in the value function is small:* $\max_s |V_{k+1}(s) - V_k(s)| < \epsilon$.

This algorithm is guaranteed to converge to the unique optimal value function V^*. This connection between an operator (the Bellman backup) and a fixed-point iteration is a powerful concept known as the Banach fixed-point theorem.

10.5 Application 1: Personalized Treatment Strategy

Let's consider a more realistic and statistically relevant problem: determining an optimal treatment policy for a chronic illness over time. This is a core problem in personalized medicine.

Imagine a patient's health can be categorized into one of a few discrete states. Each month, a doctor observes their state and must decide which treatment to administer. Treatments have different effects and risks, which are known from clinical trial data.

- **States (S):** The patient's health state. Let's define 4 states: 0='Healthy', 1='Mildly Ill', 2='Severely Ill', 3='Deceased' (a terminal state).
- **Actions (A):** The available treatments. Let's say there are two: 0='Treatment A' (e.g., an aggressive, high-risk/high-reward drug) and 1='Treatment B' (e.g., a safer, standard-of-care drug).
- **Transitions ($P(s'|s,a)$):** The probability of the patient's health transitioning to state s' next month, given their current state s and the chosen treatment a. For example, Treatment A might have a higher chance of moving a 'Severely Ill' patient to 'Healthy', but also carry a higher risk of transitioning them to 'Deceased'.
- **Rewards ($R(s)$):** We assign a numerical value to being in each state for one month. A high reward for being healthy, and negative rewards (costs) for being ill or deceased. For instance: Healthy=+10, Mildly Ill=-2, Severely Ill=-10, Deceased=-50.
- **Discount Factor (γ):** $\gamma = 0.95$. This reflects that immediate health is valued more than health in the distant future.

The goal is to find an optimal policy $\pi^*(s)$ that tells the doctor which treatment to choose for each health state (Healthy, Mildly Ill, Severely Ill) to maximize the patient's expected cumulative (discounted) quality of life.

We can solve this problem using Value Iteration. The algorithm will weigh the immediate rewards against the long-term consequences of each treatment, propagating the "value" of future health states backward in time to inform the best decision in the present. The final policy might reveal non-intuitive strategies, such as when it's optimal to choose a risky treatment versus a safer one.

```python
import numpy as np

# --- 1. Define the Personalized Treatment MDP Environment ---
# States: 0=Healthy, 1=Mildly Ill, 2=Severely Ill, 3=Deceased
NUM_STATES = 4
STATE_NAMES = ['Healthy', 'Mildly Ill', 'Severely Ill', 'Deceased']

# Actions: 0=Treatment A (Aggressive), 1=Treatment B (Standard)
NUM_ACTIONS = 2
ACTION_NAMES = ['Treatment A', 'Treatment B']

# Rewards for being in a state for one month
REWARDS = np.array([10, -2, -10, -50])

# Discount factor
```

```
DISCOUNT_FACTOR = 0.95

# Transition Probabilities: P[action, current_state, next_state]
# A 3D array representing P(s' | s, a)
transitions = np.zeros((NUM_ACTIONS, NUM_STATES, NUM_STATES))

# -- Treatment A (Aggressive) --
# From Healthy: 85% stay Healthy, 14% get Mildly Ill, 1% get Severely Ill
transitions[0, 0, :] = [0.85, 0.14, 0.01, 0.0]
# From Mildly Ill: 60% get Healthy, 30% stay Mildly Ill, 9% get Severely Ill,
    1% Deceased
transitions[0, 1, :] = [0.60, 0.30, 0.09, 0.01]
# From Severely Ill: 30% get Healthy, 30% Mildly Ill, 30% stay Severe, 10%
    Deceased
transitions[0, 2, :] = [0.30, 0.30, 0.30, 0.10]
# From Deceased (terminal state): 100% stay Deceased
transitions[0, 3, :] = [0.0, 0.0, 0.0, 1.0]

# -- Treatment B (Standard/Safer) --
# From Healthy: 98% stay Healthy, 2% get Mildly Ill
transitions[1, 0, :] = [0.98, 0.02, 0.0, 0.0]
# From Mildly Ill: 30% get Healthy, 69% stay Mildly Ill, 1% get Severely Ill
transitions[1, 1, :] = [0.30, 0.69, 0.01, 0.0]
# From Severely Ill: 5% get Healthy, 20% Mildly Ill, 74% stay Severe, 1%
    Deceased
transitions[1, 2, :] = [0.05, 0.20, 0.74, 0.01]
# From Deceased (terminal state): 100% stay Deceased
transitions[1, 3, :] = [0.0, 0.0, 0.0, 1.0]


# --- 2. Value Iteration Algorithm ---
def value_iteration(transitions, rewards, gamma, tol=1e-4):
    values = np.zeros(NUM_STATES)
    for i in range(1000): # Max iterations
        prev_values = np.copy(values)

        # Q-value matrix: Q[action, state]
        # This calculates the expected value for each action-state pair in one
    go
        q_values = np.einsum('ast,t->as', transitions, rewards + gamma *
    prev_values)

        # The optimal value for each state is the max over actions
        values = np.max(q_values, axis=0)

        # Check for convergence
        if np.max(np.abs(values - prev_values)) < tol:
            print(f"Converged at iteration {i+1}")
            break
    return values

# --- 3. Extract the Optimal Policy from the Value Function ---
def extract_policy(values, transitions, rewards, gamma):
    q_values = np.einsum('ast,t->as', transitions, rewards + gamma * values)
    # The optimal policy is to take the action with the highest Q-value in each
     state
    optimal_policy_indices = np.argmax(q_values, axis=0)
    return [ACTION_NAMES[i] for i in optimal_policy_indices]


# --- 4. Run the algorithm and print results ---
optimal_values = value_iteration(transitions, REWARDS, DISCOUNT_FACTOR)
optimal_policy = extract_policy(optimal_values, transitions, REWARDS,
    DISCOUNT_FACTOR)

print("\n--- Dynamic Optimization for Personalized Treatment ---")
print("\nOptimal Value Function (Long-term value of each health state):")
for i, name in enumerate(STATE_NAMES):
    print(f"  - {name}: {optimal_values[i]:.2f}")

print("\nOptimal Treatment Policy (The best action to take in each state):")
# We don't need a policy for the terminal 'Deceased' state
for i, name in enumerate(STATE_NAMES[:-1]):
    print(f"  - If patient is {name}, the optimal action is: '{optimal_policy[i
    ]}'")
```

Listing 15: Value Iteration for a Personalized Treatment MDP.

Interpretation: The policy for 'Severely Ill' is 'Treatment A'. This means the algorithm decided that the high potential reward of becoming healthy (30% chance) outweighs the higher risk of death (10% vs 1%) compared to the safer Treatment B, which has a very low chance (5%) of major improvement. For a 'Healthy' patient, the policy is 'Treatment B', as it's safer and has a high probability of maintaining the high-reward 'Healthy' state.

10.6 Application 2: A Personalized Digital Tutor

Dynamic programming is also a natural framework for creating adaptive learning systems. Imagine a digital tutor designed to help a student master a specific concept. The tutor can choose from several teaching methods, and its goal is to find the most efficient policy to guide the student to mastery.

- **States (S):** The student's level of understanding. We'll define 4 states: 0='Novice', 1='Competent', 2='Proficient', and 3='Mastered' (a terminal state).
- **Actions (A):** The teaching strategies the tutor can employ. We'll define 3 actions: 0='Lecture' (a passive explanation), 1='Practice Problems' (active reinforcement), and 2='Project' (application of knowledge).
- **Transitions ($P(s'|s,a)$):** The probability that a student will transition to a new level of understanding based on their current level and the teaching method used. For example, a 'Project' might be highly effective for a 'Proficient' student but confusing for a 'Novice', leading to little or no progress.
- **Rewards ($R(s)$):** We want to encourage progress and efficiency. We can assign rewards for being in a certain state for a study session. For instance: Novice=-1 (cost of time), Competent=+2, Proficient=+5, and a large reward for reaching the 'Mastered' state, +20.
- **Discount Factor (γ):** $\gamma = 0.9$. Future learning is valuable, but we prefer to achieve mastery sooner rather than later.

The objective is to find the optimal teaching policy, $\pi^*(s)$, that tells the tutor which teaching method to use for a student at any level of understanding. This policy will maximize the student's long-term, discounted knowledge gain. Value Iteration will determine, for example, whether it's better to give a 'Novice' a safe 'Lecture' or riskier but potentially faster 'Practice Problems'.

```
import numpy as np

# --- 1. Define the Personalized Tutoring MDP Environment ---
# States: 0=Novice, 1=Competent, 2=Proficient, 3=Mastered
NUM_STATES = 4
STATE_NAMES = ['Novice', 'Competent', 'Proficient', 'Mastered']
```

```
# Actions: 0=Lecture, 1=Practice Problems, 2=Project
NUM_ACTIONS = 3
ACTION_NAMES = ['Lecture', 'Practice Problems', 'Project']

# Rewards for being in a state for one session
REWARDS = np.array([-1, 2, 5, 20])

# Discount factor
DISCOUNT_FACTOR = 0.9

# Transition Probabilities: P[action, current_state, next_state]
# A 3D array representing P(s' | s, a)
transitions = np.zeros((NUM_ACTIONS, NUM_STATES, NUM_STATES))

# -- Action 0: Lecture (Safe, slow progress) --
transitions[0, 0, :] = [0.7, 0.3, 0.0, 0.0]  # Novice -> 70% stay, 30%
    Competent
transitions[0, 1, :] = [0.0, 0.6, 0.4, 0.0]  # Competent -> 60% stay, 40%
    Proficient
transitions[0, 2, :] = [0.0, 0.0, 0.8, 0.2]  # Proficient -> 80% stay, 20%
    Mastered
transitions[0, 3, :] = [0.0, 0.0, 0.0, 1.0]  # Mastered (terminal)

# -- Action 1: Practice Problems (Effective, moderate risk) --
transitions[1, 0, :] = [0.4, 0.6, 0.0, 0.0]  # Novice -> 40% stay, 60%
    Competent
transitions[1, 1, :] = [0.1, 0.2, 0.7, 0.0]  # Competent -> 10% regress, 20%
    stay, 70% Proficient
transitions[1, 2, :] = [0.0, 0.0, 0.3, 0.7]  # Proficient -> 30% stay, 70%
    Mastered
transitions[1, 3, :] = [0.0, 0.0, 0.0, 1.0]  # Mastered (terminal)

# -- Action 2: Project (High risk/reward, needs prerequisite knowledge) --
transitions[2, 0, :] = [0.9, 0.1, 0.0, 0.0]  # Novice -> 90% stay (ineffective)
    , 10% Competent
transitions[2, 1, :] = [0.0, 0.4, 0.5, 0.1]  # Competent -> 40% stay, 50%
    Proficient, 10% Mastered
transitions[2, 2, :] = [0.0, 0.0, 0.5, 0.5]  # Proficient -> 50% stay, 50%
    Mastered
transitions[2, 3, :] = [0.0, 0.0, 0.0, 1.0]  # Mastered (terminal)


# --- 2. Value Iteration Algorithm (reusable from previous example) ---
def value_iteration(transitions, rewards, gamma, tol=1e-5):
    values = np.zeros(NUM_STATES)
    for i in range(1000):
        prev_values = np.copy(values)
        q_values = np.einsum('ast,t->as', transitions, rewards + gamma *
    prev_values)
        values = np.max(q_values, axis=0)
        if np.max(np.abs(values - prev_values)) < tol:
            print(f"Converged at iteration {i+1}")
            break
    return values

# --- 3. Extract Optimal Policy (reusable from previous example) ---
def extract_policy(values, transitions, rewards, gamma):
    q_values = np.einsum('ast,t->as', transitions, rewards + gamma * values)
    optimal_policy_indices = np.argmax(q_values, axis=0)
    return [ACTION_NAMES[i] for i in optimal_policy_indices]


# --- 4. Run the algorithm and display the results ---
```

```
optimal_values = value_iteration(transitions, REWARDS, DISCOUNT_FACTOR)
optimal_policy = extract_policy(optimal_values, transitions, REWARDS,
    DISCOUNT_FACTOR)

print("\n--- Dynamic Optimization for a Personalized Tutor ---")
print("\nOptimal Value Function (Expected long-term knowledge gain):")
for i, name in enumerate(STATE_NAMES):
    print(f"  - {name}: {optimal_values[i]:.2f}")

print("\nOptimal Teaching Policy (Best action for each student state):")
# We don't need a policy for the terminal 'Mastered' state
for i, name in enumerate(STATE_NAMES[:-1]):
    print(f"  - If student is {name}, the optimal action is: '{optimal_policy[i
    ]}'")
```

Listing 16: Value Iteration for a Personalized Digital Tutor MDP.

Interpretation: The optimal policy for both 'Novice' and 'Proficient' students is 'Practice Problems'. For a Novice, this is the fastest way to become Competent. For a Proficient student, it offers the highest probability (70%) of reaching Mastery. For a 'Competent' student, the policy is 'Project'. This is interesting. While 'Practice Problems' is safer and more likely to make them Proficient (70% vs 50%), the 'Project' offers a small but valuable 10% chance of jumping directly to the high-reward 'Mastered' state, making its expected value slightly higher. This showcases how DP can find the best long-term strategy, not just the safest next step.

11 Gradient Descent

11.1 From Optimality Conditions to Algorithms

In the first part of this course, we focused on characterizing the solutions to optimization problems. We learned that for unconstrained problems, the gradient must be zero at a minimum. For constrained problems, the KKT conditions describe a delicate balance between the objective and the active constraints.

This is a powerful theoretical framework, but it doesn't always tell us *how* to find the solution. For Ordinary Least Squares, we were able to solve $\nabla f(\beta) = \mathbf{0}$ algebraically to get the Normal Equations. For most other problems in statistics and machine learning, this is impossible. The equations are too complex.

We need a different approach: an **iterative algorithm**. Instead of solving for the optimum in one shot, we start with an initial guess and take a series of steps to gradually improve it, getting closer and closer to the solution. The most fundamental of these algorithms is Gradient Descent.

11.2 The Intuition: Walking Downhill

Imagine you are standing on a hillside in a thick fog and you want to get to the bottom of the valley. You can't see the bottom, but you can feel the slope of the ground beneath your feet. What is your strategy?

The most natural strategy is to:

1. Check which direction is the steepest downhill.
2. Take a small step in that direction.
3. Repeat.

This simple, intuitive process is the core idea of gradient descent. In the language of calculus:

- The "hillside" is the graph of our objective function $f(x)$.
- The "slope of the ground" at a point x_k is given by the gradient, $\nabla f(x_k)$.
- The direction of steepest *ascent* is $\nabla f(x_k)$.
- The direction of steepest *descent* is therefore $-\nabla f(x_k)$.

11.3 The Gradient Descent Algorithm

We can translate this "walking downhill" strategy into a formal algorithm. We start with an initial guess, x_0, and iteratively update it using the following rule.

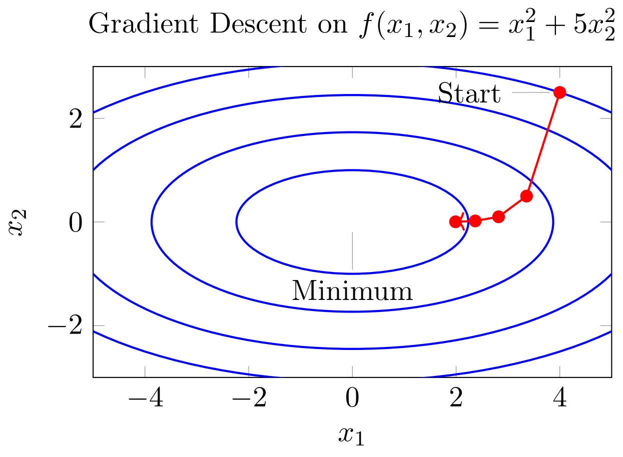
Definition 11.1 (The Gradient Descent Algorithm). *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a differentiable objective function. The gradient descent algorithm generates a sequence of points $\{x_k\}$ via the update rule:*

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

where:

- *x_k is our current guess at iteration k.*
- *$\nabla f(x_k)$ is the gradient of the function evaluated at x_k.*
- *$\eta > 0$ is a small scalar called the **step size** or **learning rate**.*

The algorithm stops when a chosen criterion is met, for example, when the gradient becomes very small ($||\nabla f(x_k)|| < \epsilon$) or when the position changes very little ($||x_{k+1} - x_k|| < \epsilon$).



11.4 Application 1: Solving Linear Regression Iteratively

In Lecture 2, we found a closed-form solution for Ordinary Least Squares (OLS) called the Normal Equations: $\hat{\beta} = (X^T X)^{-1} X^T y$.

However, there are two major practical problems with this formula:

1. **Computational Cost:** If the number of features n is very large (e.g., millions), then forming the matrix $X^T X$ (which is $n \times n$) can be too memory-intensive, and inverting it (an $O(n^3)$ operation) can be prohibitively slow.
2. **Collinearity:** If features are highly correlated, $X^T X$ can be ill-conditioned or non-invertible, making the solution numerically unstable.

Gradient descent provides an alternative way to find the optimal β that avoids matrix inversion entirely.

Example 11.1 (Gradient Descent for OLS)**.** *Derive the gradient descent update rule for the OLS objective function $f(\beta) = ||X\beta - y||_2^2$.*

Proof:

1. ***Recall the OLS Gradient.*** *From Lecture 2, we know the gradient of the OLS objective function is:*
$$\nabla f(\beta) = 2X^T(X\beta - y).$$
2. ***Plug into the Gradient Descent Update Rule.*** *We substitute this expression into the general update rule $\beta_{k+1} = \beta_k - \eta \nabla f(\beta_k)$:*
$$\beta_{k+1} = \beta_k - \eta \left(2X^T(X\beta_k - y)\right).$$
3. ***The Algorithm in Practice.*** *At each iteration k:*
   - *Start with the current parameter guess, β_k.*
   - *Calculate the current predictions: $\hat{y}_k = X\beta_k$.*
   - *Calculate the residual vector (the errors): $r_k = \hat{y}_k - y$.*
   - *Compute the gradient: $\nabla f(\beta_k) = 2X^T r_k$. This step has a nice interpretation: it correlates each feature with the current errors.*

- *Update the parameters: $\beta_{k+1} = \beta_k - (2\eta)X^T r_k$. We move our parameters in a direction that corrects for the errors.*

By repeating these simple matrix-vector multiplications, the algorithm will converge to the same $\hat{\beta}$ found by the Normal Equations, without ever forming or inverting $X^T X$. This makes it possible to solve linear regression problems with massive numbers of features.

```python
import jax
import jax.numpy as jnp

# 1. Define the OLS loss function and its gradient
def ols_loss(beta, X, y):
  return jnp.sum((X @ beta - y)**2)

grad_ols = jax.grad(ols_loss)

# 2. Generate synthetic data
key = jax.random.PRNGKey(123)
m, n = 100, 10
X_data = jax.random.normal(key, (m, n))
true_beta = jax.random.normal(key, (n,))
y_data = X_data @ true_beta + 0.5 * jax.random.normal(key, (m,))

# 3. Gradient Descent algorithm
learning_rate = 0.001
n_iterations = 1000
beta = jnp.zeros(n) # Initial guess

for i in range(n_iterations):
    # Compute the gradient using the full dataset
    gradient = grad_ols(beta, X_data, y_data)
    # Update the parameters
    beta = beta - learning_rate * gradient
    # Optional: print loss to monitor convergence
    if i % 100 == 0:
        loss = ols_loss(beta, X_data, y_data)
        print(f"Iteration {i}: Loss = {loss:.2f}")

# 4. Compare with the closed-form solution
closed_form_beta = jnp.linalg.inv(X_data.T @ X_data) @ X_data.T @ y_data

print("\n--- Results ---")
print(f"True beta: {true_beta.round(2)}")
print(f"GD beta:   {beta.round(2)}")
print(f"Closed-form beta: {closed_form_beta.round(2)}")
```

Listing 17: A JAX implementation of Gradient Descent for OLS.

11.5 Application 2: Training a Neural Network

This example shows the complete process of defining a Multi-Layer Perceptron (MLP), setting up an optimizer, and training the network on a simple regression problem using a gradient descent-based optimizer (Adam).

```python
import jax
import jax.numpy as jnp
import equinox as eqx
import optax
import matplotlib.pyplot as plt

# Use a specific key for reproducibility
key = jax.random.PRNGKey(42)
```

```python
# 1. Define the Neural Network model using Equinox
class MLP(eqx.Module):
    layers: list

    def __init__(self, in_size, out_size, width_size, depth, key):
        keys = jax.random.split(key, depth + 1)
        self.layers = []
        # Input layer
        self.layers.append(eqx.nn.Linear(in_size, width_size, key=keys[0]))
        # Hidden layers
        for i in range(depth - 1):
            self.layers.append(eqx.nn.Linear(width_size, width_size, key=keys[i
    +1]))
        # Output layer
        self.layers.append(eqx.nn.Linear(width_size, out_size, key=keys[-1]))

    def __call__(self, x):
        for layer in self.layers[:-1]:
            x = jax.nn.relu(layer(x))
        # No activation on the final layer for regression
        x = self.layers[-1](x)
        return x

# 2. Generate some synthetic data (e.g., a noisy sine wave)
key, data_key = jax.random.split(key)
x_data = jnp.linspace(-2 * jnp.pi, 2 * jnp.pi, 100)[:, None]
y_data = jnp.sin(x_data) + 0.1 * jax.random.normal(data_key, x_data.shape)

# 3. Define the loss function (Mean Squared Error)
def mse_loss(model, x, y):
    y_pred = jax.vmap(model)(x) # Use vmap to process batches
    return jnp.mean((y - y_pred)**2)

# 4. Define the training step
@eqx.filter_jit # JIT compile for speed
def make_step(model, opt_state, optimizer, x_batch, y_batch):
    # Calculate loss and gradients
    loss, grads = eqx.filter_value_and_grad(mse_loss)(model, x_batch, y_batch)

    # Update the model and optimizer state
    updates, opt_state = optimizer.update(grads, opt_state)
    model = eqx.apply_updates(model, updates)

    return model, opt_state, loss

# 5. The Training Loop
key, model_key = jax.random.split(key)
model = MLP(in_size=1, out_size=1, width_size=64, depth=3, key=model_key)
optimizer = optax.adam(learning_rate=3e-4)
opt_state = optimizer.init(model)

batch_size = 32
n_iterations = 2000

print("Training MLP for regression...")
for i in range(n_iterations):
    key, batch_key = jax.random.split(key)
    # Create a random mini-batch
    indices = jax.random.randint(batch_key, (batch_size,), 0, len(x_data))
    x_batch, y_batch = x_data[indices], y_data[indices]
    model, opt_state, loss = make_step(model, opt_state, optimizer, x_batch,
    y_batch)

    if i % 200 == 0:
        print(f"Iteration {i}: Loss = {loss.item():.4f}")

# 6. Visualize the results
plt.figure(figsize=(10, 6))
plt.scatter(x_data, y_data, label='True Data', alpha=0.5)
plt.plot(x_data, jax.vmap(model)(x_data), color='red', label='Model Prediction'
    )
plt.title('MLP Regression with Equinox')
plt.legend()
plt.show()
```

Listing 18: Training a simple MLP for regression using JAX and Equinox.

11.6 Application 3: Training a CNN

This example demonstrates how to build and train a Convolutional Neural Network (CNN) on the classic MNIST dataset for handwritten digit recognition. It showcases how the same gradient descent principles apply to more complex models and data.

```
import jax
import jax.numpy as jnp
import equinox as eqx
import optax
import tensorflow_datasets as tfds
import numpy as np

# Use a specific key for reproducibility
key = jax.random.PRNGKey(0)

# 1. Define the CNN model
class CNN(eqx.Module):
    layers: list

    def __init__(self, key):
        key1, key2, key3 = jax.random.split(key, 3)
        # Standard CNN architecture: Conv -> ReLU -> Pool -> Conv -> ReLU ->
    Pool -> Flatten -> Linear
        self.layers = [
            eqx.nn.Conv2d(1, 16, kernel_size=3, key=key1),
            eqx.nn.MaxPool2d(kernel_size=2),
            jax.nn.relu,
            eqx.nn.Conv2d(16, 32, kernel_size=3, key=key2),
            eqx.nn.MaxPool2d(kernel_size=2),
            jax.nn.relu,
            jnp.ravel, # Flatten the output for the linear layer
            eqx.nn.Linear(32 * 5 * 5, 10, key=key3) # 10 output classes for
    MNIST
        ]

    def __call__(self, x):
        # Add a channel dimension for the convolution
        x = x[None, :, :]
        for layer in self.layers:
            x = layer(x)
        return x

# 2. Load and preprocess the MNIST dataset
def get_datasets():
    ds_builder = tfds.builder('mnist')
    ds_builder.download_and_prepare()
    train_ds = tfds.as_numpy(ds_builder.as_dataset(split='train', batch_size
    =-1))
    test_ds = tfds.as_numpy(ds_builder.as_dataset(split='test', batch_size=-1))
    # Normalize images to [0, 1]
    train_ds['image'] = train_ds['image'].astype(np.float32) / 255.
    test_ds['image'] = test_ds['image'].astype(np.float32) / 255.
    return train_ds, test_ds

train_ds, test_ds = get_datasets()
X_train, y_train = train_ds['image'], train_ds['label']
X_test, y_test = test_ds['image'], test_ds['label']

# 3. Define loss and evaluation metrics
def loss_fn(model, x, y):
    # Get model predictions (logits)
    pred_y_logits = jax.vmap(model)(x)
    # Compute cross-entropy loss
    return optax.softmax_cross_entropy_with_integer_labels(pred_y_logits, y).
    mean()

def accuracy_fn(model, x, y):
    pred_y_logits = jax.vmap(model)(x)
    pred_y = jnp.argmax(pred_y_logits, axis=1)
    return jnp.mean(y == pred_y)

# 4. Define the training and evaluation steps
@eqx.filter_jit
def make_step(model, opt_state, optimizer, x_batch, y_batch):
    loss, grads = eqx.filter_value_and_grad(loss_fn)(model, x_batch, y_batch)
    updates, opt_state = optimizer.update(grads, opt_state)
    model = eqx.apply_updates(model, updates)
    return model, opt_state, loss

@eqx.filter_jit
def evaluate(model, x_test, y_test):
    return accuracy_fn(model, x_test, y_test)

# 5. The Training Loop
key, model_key = jax.random.split(key)
model = CNN(model_key)
optimizer = optax.adam(1e-3)
opt_state = optimizer.init(model)

batch_size = 128
n_epochs = 5
n_train_steps = len(X_train) // batch_size

print("Training CNN for MNIST classification...")
for epoch in range(n_epochs):
    key, epoch_key = jax.random.split(key)
    # Shuffle the training data each epoch
    perm = jax.random.permutation(epoch_key, len(X_train))
    X_train_shuffled = X_train[perm]
    y_train_shuffled = y_train[perm]

    total_loss = 0
    for i in range(n_train_steps):
        # Create mini-batch
        start = i * batch_size
        end = start + batch_size
        x_batch = X_train_shuffled[start:end]
        y_batch = y_train_shuffled[start:end]
        model, opt_state, loss = make_step(model, opt_state, optimizer, x_batch
, y_batch)
        total_loss += loss.item()

    avg_loss = total_loss / n_train_steps
    accuracy = evaluate(model, X_test, y_test)
    print(f"Epoch {epoch + 1}: Avg Loss = {avg_loss:.4f}, Test Accuracy = {
accuracy:.4f}")
```

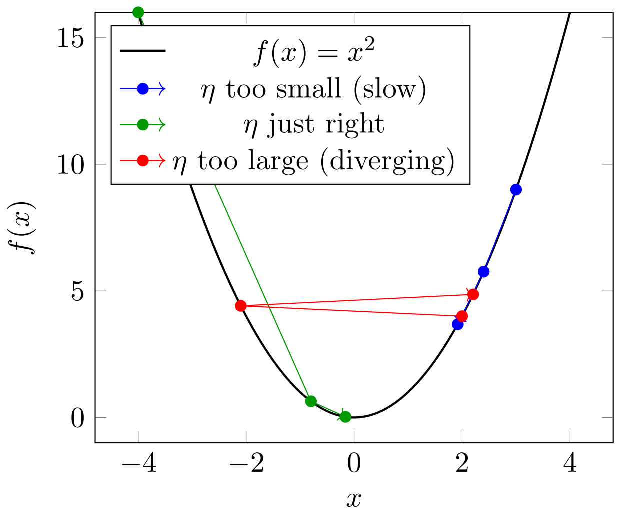
Listing 19: Training a CNN for MNIST classification using JAX and Equinox.

11.7 Practical Considerations: The Learning Rate

The learning rate, η, is the single most important parameter to tune in gradient descent. Its value has a dramatic effect on the algorithm's performance.

- **If η is too small:** The algorithm will take tiny steps and convergence will be very slow.
- **If η is too large:** The algorithm will overshoot the minimum. Instead of descending into the valley, it will bounce from one side to the other, potentially diverging to infinity.
- **If η is "just right":** The algorithm will make steady progress towards the minimum.

Effect of Learning Rate η on Convergence



A Note on Just-In-Time (JIT) Compilation with `@eqx.filter_jit`

You may have noticed the decorator `@eqx.filter_jit` applied to our `make_step` function. This is the single most important technique for achieving high performance in JAX and is worth understanding.

What is JIT Compilation?

JIT stands for **Just-In-Time**. In standard Python, the interpreter reads and executes your code line by line. This is flexible but slow, as each simple operation (like adding two numbers) has significant overhead. A JIT compiler works differently:

1. **Tracing:** The first time a JIT-compiled function is called, JAX doesn't run it in Python. Instead, it "traces" the function, recording the sequence of mathematical operations on abstract representations of the input data (based on their shape and type).

2. **Compilation with XLA:** This sequence of operations is then sent to a powerful underlying compiler, XLA (Accelerated Linear Algebra). XLA takes this graph of operations and compiles it into highly optimized machine code, specifically tailored to your hardware (CPU, GPU, or TPU). This compiled code is then cached.

3. **Execution:** On all subsequent calls to the function with data of the same shape and type, JAX skips the slow Python interpretation and directly executes the fast, pre-compiled machine code.

Why is this so much faster?

- **Reduced Python Overhead:** The expensive Python interpreter is removed from the inner loop. A `for` loop over a million elements that would be very slow in Python becomes a single, highly efficient compiled instruction.

- **Operator Fusion:** The XLA compiler is incredibly smart. It can "fuse" multiple operations together. For example, an element-wise multiplication followed by an addition (`y = a * x + b`) can be compiled into a single kernel that passes over the data only once, minimizing memory access and maximizing hardware utilization.

- **Hardware Optimization:** The compiled code is optimized for the specific parallel architecture of your GPU or TPU, taking advantage of its unique features.

What does `@eqx.filter_jit` do specifically?

A standard `jax.jit` requires that all inputs to a function be JAX-compatible types (like arrays). An Equinox model, however, is a mix of arrays (the model weights and biases, which should be compiled) and static information (like Python functions for activations or integer values for layer sizes, which should not be compiled).

The `@eqx.filter_jit` decorator is a clever wrapper that automatically handles this for us. When applied to a function that accepts an Equinox model:

- It intelligently partitions the model into its **dynamic** parts (the JAX arrays containing weights, which change during training) and its **static** parts (the model's structure and metadata).

- It then calls `jax.jit` under the hood, but only traces the function based on the dynamic parts.

This means that our entire `make_step` function—including the forward pass, loss calculation, gradient computation, and optimizer update—is compiled into a single, lightning-fast kernel. The code is only re-compiled if the *static structure* of the model changes, not when the weight values are updated, making it perfect for efficient training loops.

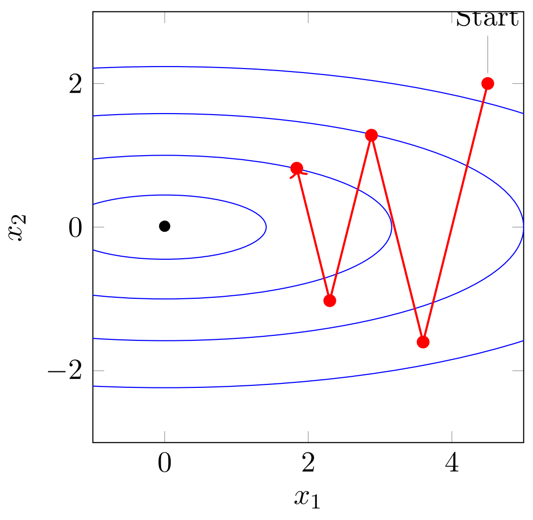
12 Momentum and Accelerated Gradient Methods

12.1 The Problem with Vanilla Gradient Descent

In the last lecture, we introduced Gradient Descent, an intuitive algorithm that iteratively steps in the direction of the negative gradient. While it is guaranteed to converge for convex problems with a suitable learning rate, its performance in practice can be quite slow.

The main issue arises when the loss surface is not perfectly spherical. If the contours of the objective function are elongated ellipses (forming a narrow valley or "ravine"), gradient descent tends to oscillate back and forth across the steep sides of the valley, while making slow progress along the shallow bottom.

Gradient Descent Oscillating in a Ravine



The gradient is always perpendicular to the contour lines. In a narrow valley, this direction points almost directly across the valley, not along it. To make faster progress, we need a way to "dampen" the oscillations and "accelerate" in the consistent downhill direction.

12.2 The Momentum Method

12.3 Intuition: The Heavy Ball

Imagine a heavy ball rolling down the loss surface.

- When it moves across the steep sides of the valley, gravity pulls it down, but its momentum carries it forward. The gravitational forces from each side tend to cancel out, dampening the oscillations.
- When it moves along the bottom of the valley, the force of gravity is consistently in the same direction. The ball picks up speed, accelerating towards the minimum.

This physical analogy is exactly what the Momentum Method implements. It adds a "velocity" term to our update rule that accumulates a running average of past gradients.

12.4 The Momentum Algorithm

The algorithm maintains a "velocity" vector v_k which is an exponentially decaying moving average of the gradients.

Definition 12.1 (The Momentum Algorithm)**.** *Start with an initial guess x_0 and initialize the velocity $v_0 = \mathbf{0}$. The update rules are:*

$$\begin{aligned} v_{k+1} &= \beta v_k + \eta \nabla f(x_k) \quad \textit{(Update velocity)} \\ x_{k+1} &= x_k - v_{k+1} \quad \textit{(Update position)} \end{aligned}$$

where:

- *η is the learning rate.*
- *$\beta \in [0, 1)$ is the **momentum coefficient**. It acts like friction. A typical value is $\beta = 0.9$.*

If $\beta = 0$, we recover vanilla Gradient Descent. As β approaches 1, we are placing more weight on the accumulated past gradients, and the "ball" becomes "heavier."

12.5 Nesterov's Accelerated Gradient (NAG)

12.6 Intuition: Look Before You Leap

In the standard momentum method, we calculate the gradient at our current position x_k and then add this to our velocity v_k to get the full step.

Nesterov's insight was that this is slightly inefficient. We already know our velocity is going to carry us forward. Why not calculate the gradient where we *expect to be* in a moment, rather than where we are now?

The NAG strategy is:

1. Make a "lookahead" step in the direction of your current velocity.
2. Calculate the gradient at this lookahead position.
3. Use this "smarter" gradient to update your velocity and make the final step.

This "look-ahead" feature allows the algorithm to correct its course faster. If the momentum is about to carry it up a hill, the gradient at the lookahead point will be pointing back towards the minimum, allowing the algorithm to slow down more quickly.

12.7 The NAG Algorithm

Definition 12.2 (Nesterov's Accelerated Gradient)**.** *Start with x_0 and $v_0 = \mathbf{0}$. The update rules are:*

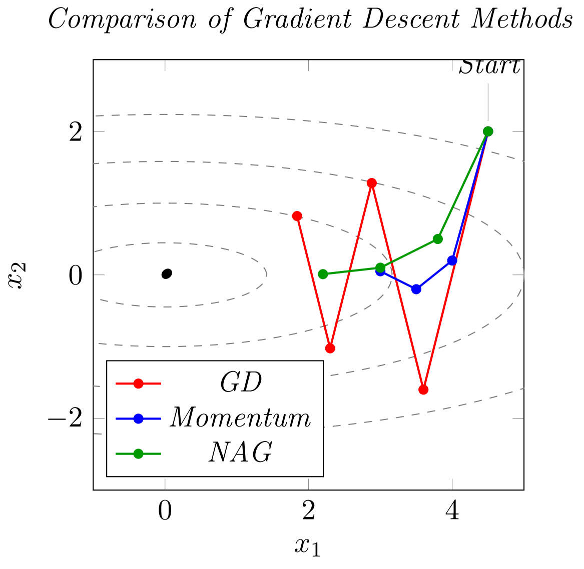
$$\begin{aligned} x_{lookahead} &= x_k - \beta v_k \quad \textit{(Lookahead step)} \\ v_{k+1} &= \beta v_k + \eta \nabla f(x_{lookahead}) \quad \textit{(Update velocity with corrected gradient)} \\ x_{k+1} &= x_k - v_{k+1} \quad \textit{(Update position)} \end{aligned}$$

The only difference is that the gradient is computed at $x_{lookahead}$ instead of x_k.

```python
import jax
import jax.numpy as jnp

# Use the same OLS setup as the previous chapter
def ols_loss(beta, X, y):
  return jnp.sum((X @ beta - y)**2)
grad_ols = jax.grad(ols_loss)
key = jax.random.PRNGKey(123)
m, n = 100, 10
X_data = jax.random.normal(key, (m, n))
true_beta = jax.random.normal(key, (n,))
```

```
y_data = X_data @ true_beta + 0.5 * jax.random.normal(key, (m,))

# --- 1. GD with Momentum ---
beta_mom = jnp.zeros(n)
velocity = jnp.zeros(n)
learning_rate = 0.001
momentum_beta = 0.9

for i in range(200): # Momentum often converges faster
    gradient = grad_ols(beta_mom, X_data, y_data)
    # Update velocity: v_new = beta*v_old + eta*grad
    velocity = momentum_beta * velocity + learning_rate * gradient
    # Update parameters: beta_new = beta_old - v_new
    beta_mom = beta_mom - velocity
    if i % 40 == 0:
        print(f"Momentum GD Iter {i}: Loss = {ols_loss(beta_mom, X_data, y_data
    ):.2f}")

# --- 2. Nesterov Accelerated Gradient (NAG) ---
beta_nag = jnp.zeros(n)
velocity_nag = jnp.zeros(n)
# Using the same hyperparameters for comparison
print("\n--- NAG ---")
for i in range(200):
    # Lookahead step
    beta_lookahead = beta_nag - momentum_beta * velocity_nag
    # Compute gradient at the lookahead position
    gradient_nag = grad_ols(beta_lookahead, X_data, y_data)
    # Update velocity
    velocity_nag = momentum_beta * velocity_nag + learning_rate * gradient_nag
    # Update parameters
    beta_nag = beta_nag - velocity_nag
    if i % 40 == 0:
        print(f"NAG Iter {i}: Loss = {ols_loss(beta_nag, X_data, y_data):.2f}")
```
Listing 20: JAX implementation of GD with Momentum and NAG.

Example 12.1 (Visual Comparison)**.** *Let's visualize the difference between the three algorithms on the narrow ravine from before.*



As the visualization shows, standard Gradient Descent oscillates wildly. Momentum dampens these oscillations and takes a more direct path. NAG, with its lookahead capability, often corrects course even more effectively, converging in fewer steps.

12.8 Application: Training Deep Neural Networks

The loss landscapes of deep neural networks are notoriously complex. They are high-dimensional and non-convex, filled with countless local minima, flat regions called plateaus, and, most frequently, **saddle points**.

A saddle point is a location where the gradient is zero, but it is not a minimum or a maximum. Think of the middle of a horse's saddle: it's a minimum along one direction (front-to-back) but a maximum along another (side-to-side).

Vanilla Gradient Descent struggles immensely with saddle points. As it approaches a saddle, the gradients become very small, and the algorithm can slow to a crawl, appearing to have converged when it has not.

This is where momentum-based methods are indispensable.

Momentum helps algorithms escape saddle points.

The "heavy ball" can use its accumulated velocity to roll right through a nearly-flat saddle point region, continuing its descent on the other side. This ability to power through areas with small or misleading gradients is a key reason why almost every state-of-the-art training algorithm for deep learning (like Adam, which we will see soon) is fundamentally based on the principle of momentum.

13 Projected, Stochastic, and Mirror Descent

13.1 The Challenge of Large-Scale Constrained Optimization

We have now entered the algorithmic part of the course. Our basic tool is Gradient Descent, which is powerful for unconstrained problems. However, real-world statistical problems often present two major challenges:

1. **Constraints:** The solution must satisfy certain conditions (e.g., probabilities must be non-negative and sum to one). A standard gradient step might violate these conditions.
2. **Scale:** The objective function is a sum over a massive dataset, making the computation of the true gradient prohibitively expensive.

Today, we will introduce three crucial modifications to Gradient Descent that address these challenges: Projected Gradient Descent for constraints, Mirror Descent for more complex geometries, and Stochastic Gradient Descent for scale.

13.2 Projected Gradient Descent (PGD)

What should we do if a gradient descent step takes us outside the feasible set Ω? The most intuitive solution is to simply put it back.

Definition 13.1 (Projected Gradient Descent (PGD))**.** *The PGD algorithm consists of two steps at each iteration:*

1. ***Standard Gradient Step:*** *Take a normal gradient descent step to a temporary point* $y_{k+1} = x_k - \eta \nabla f(x_k)$.
2. ***Projection Step:*** *Find the point in the feasible set* Ω *that is closest to* y_{k+1} *and set that as our next iterate. This is the **Euclidean projection** of* y_{k+1} *onto* Ω.

$$x_{k+1} = \Pi_\Omega(y_{k+1}) := \arg\min_{z\in\Omega} ||z - y_{k+1}||_2^2.$$

This algorithm is effective as long as the projection step itself is easy to compute.

Example 13.1 (Projection onto the Probability Simplex)**.** *This is a crucial application where the feasible set is the probability simplex,* $\Omega = \Delta^n = \{x \in \mathbb{R}^n \mid \sum x_i = 1, x_i \geq 0\}$. *Finding the projection* $\Pi_\Omega(y)$ *is a convex QP.*

Solution: *While there is no trivial closed-form solution, the projection can be computed very efficiently. The solution has the form* $x_i^* = \max(0, y_i - \alpha)$, *where a single scalar* α *is chosen such that the elements sum to 1. This can be found with a fast sorting-based algorithm. This makes PGD a viable method for optimizing over probability distributions.*

```python
import jax
import jax.numpy as jnp

# Objective: Minimize f(x) = ||Ax - b||^2
# Constraint: x must be in a box, i.e., lower_bound <= x <= upper_bound
def objective_fn(x, A, b):
    return jnp.sum((A @ x - b)**2)

grad_fn = jax.grad(objective_fn)

# Define the projection function for a box constraint
def project_to_box(x, lower, upper):
    return jnp.clip(x, lower, upper)
```

```
# Generate data
key = jax.random.PRNGKey(0)
A = jax.random.normal(key, (10, 5))
b = jax.random.normal(key, (10,))
lower_bounds = -1.0 * jnp.ones(5)
upper_bounds =  1.0 * jnp.ones(5)

# PGD algorithm
x = jnp.zeros(5) # Initial guess
learning_rate = 0.01

print("Running Projected Gradient Descent...")
for i in range(200):
    # 1. Gradient step
    gradient = grad_fn(x, A, b)
    x_temp = x - learning_rate * gradient

    # 2. Projection step
    x = project_to_box(x_temp, lower_bounds, upper_bounds)

    if i % 40 == 0:
        loss = objective_fn(x, A, b)
        print(f"Iter {i}: Loss = {loss:.2f}")

print("\nFinal solution x:\n", x.round(3))
print("Note that all values are within the [-1, 1] bounds.")
```

Listing 21: JAX implementation of Projected Gradient Descent onto a box.

13.3 Mirror Descent

Projected Gradient Descent is fundamentally tied to Euclidean distance (finding the *closest* point). But is Euclidean distance always the "right" way to measure proximity?

- For probability distributions, a measure like KL-Divergence might be more natural.
- For other problems, the geometry might be non-Euclidean.

Mirror Descent is a powerful generalization of PGD that allows us to choose a "distance" measure that is tailored to the geometry of our feasible set.

Bregman Divergence:

Mirror descent replaces the squared Euclidean distance with a **Bregman divergence**, $D_\phi(x||y)$. This "divergence" is generated by a strictly convex function ϕ, called the **distance-generating function (DGF)**.

Definition 13.2 (Bregman Divergence)**.** *Let ϕ be a strictly convex, differentiable function. The Bregman divergence between x and y is the gap between the value of ϕ at x and its first-order Taylor approximation at y:*

$$D_\phi(x||y) = \phi(x) - \phi(y) - \nabla\phi(y)^T(x - y).$$

Because ϕ is convex, this gap is always non-negative.

Example 13.2 (Key Examples of Bregman Divergences)**.** *1.* ***DGF = Squared Euclidean***

Norm: *Let $\phi(x) = \frac{1}{2}||x||_2^2$.* ***Proof:*** *The gradient is $\nabla\phi(y) = y$.*

$$\begin{aligned}
D_\phi(x||y) &= \frac{1}{2}||x||_2^2 - \frac{1}{2}||y||_2^2 - y^T(x-y) \\
&= \frac{1}{2}x^Tx - \frac{1}{2}y^Ty - y^Tx + y^Ty \\
&= \frac{1}{2}x^Tx - y^Tx + \frac{1}{2}y^Ty = \frac{1}{2}(x-y)^T(x-y) = \frac{1}{2}||x-y||_2^2.
\end{aligned}$$

The Bregman divergence is just half the squared Euclidean distance.

2. ***DGF = Negative Entropy:*** *Let $\phi(x) = \sum_i x_i \log x_i$ (for x in the probability simplex).* ***Proof:*** *The gradient is $\nabla\phi(y)_i = \log y_i + 1$.*

$$\begin{aligned}
D_\phi(x||y) &= \sum_i x_i \log x_i - \sum_i y_i \log y_i - \sum_i (\log y_i + 1)(x_i - y_i) \\
&= \sum_i x_i \log x_i - \sum_i y_i \log y_i - \sum_i x_i \log y_i - \sum_i x_i + \sum_i y_i \log y_i + \sum_i y_i \\
&= \sum_i x_i(\log x_i - \log y_i) \quad (\textit{since } \sum x_i = \sum y_i = 1) \\
&= \sum_i x_i \log\left(\frac{x_i}{y_i}\right) = D_{KL}(x||y).
\end{aligned}$$

The Bregman divergence is the ***Kullback-Leibler (KL) divergence****, a fundamental measure of distance between probability distributions.*

The Mirror Descent Algorithm

The algorithm replaces the simple projection step with a **proximal step** that uses the Bregman divergence. The update is a trade-off between moving in the gradient direction and staying close to the current point, as measured by our chosen divergence.

Definition 13.3 (Mirror Descent Algorithm)**.** *The update rule is:*

$$x_{k+1} = \arg\min_{z\in\Omega} \left(\eta\nabla f(x_k)^T z + D_\phi(z||x_k)\right).$$

Key Connections:

- If we choose $\phi(x) = \frac{1}{2}||x||_2^2$, Mirror Descent **becomes exactly Projected Gradient Descent**. (Theorem L14.2)
- If we are optimizing over the probability simplex and choose the negative entropy DGF, the Mirror Descent update takes a simple, elegant multiplicative form often called the **exponentiated gradient** update:

$$(x_{k+1})_i \propto (x_k)_i \exp(-\eta\nabla f(x_k)_i).$$

  This ensures the iterates remain positive and can be re-normalized to sum to 1.

13.4 Stochastic Gradient Descent (SGD)

The core idea of SGD is simple:

If the full gradient is too expensive, use a **cheap, noisy estimate** of it instead.

Instead of summing over all m data points, we can get a much faster, albeit noisy, estimate of the gradient by using just a small, random sample of the data.

Definition 13.4 (The SGD Algorithm). *At each iteration k, we first sample a small* ***mini-batch*** *of data, $\mathcal{B}_k$, from the full training set. We then compute the gradient using only the data in this mini-batch:*

$$g_k = \frac{1}{|\mathcal{B}_k|} \sum_{i \in \mathcal{B}_k} \nabla f_i(x_k).$$

This stochastic gradient g_k is then used in the standard gradient descent update rule:

$$x_{k+1} = x_k - \eta g_k.$$

13.5 Why Does This Work? The Unbiased Gradient

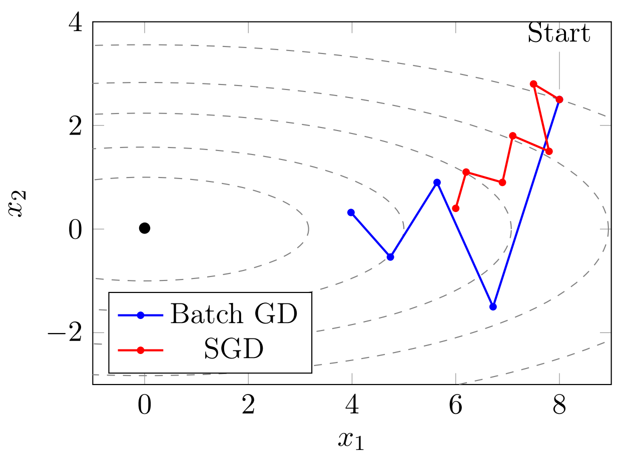
The magic of SGD lies in the fact that the stochastic gradient g_k is an **unbiased estimator** of the true gradient $\nabla f(x_k)$. This means that, on average, the stochastic gradient points in the same direction as the true gradient:

$$\mathbb{E}[g_k] = \nabla f(x_k).$$

While any single step might be slightly "wrong," over many iterations, these noisy steps average out to a path that moves, on the whole, towards the minimum.

The path taken by SGD is often described as a "drunkard's walk." It's erratic and noisy, but it has a general tendency to stumble downhill.

Batch GD vs. Stochastic GD on $f(x_1, x_2) = x_1^2 + 10x_2^2$



```
import jax
import jax.numpy as jnp

# The loss for a single data point (or a mini-batch)
def ols_loss_batch(beta, X_batch, y_batch):
  return jnp.mean((X_batch @ beta - y_batch)**2)

grad_ols_batch = jax.grad(ols_loss_batch)

# Generate a larger dataset
key = jax.random.PRNGKey(42)
m, n = 5000, 10
X_data = jax.random.normal(key, (m, n))
true_beta = jax.random.normal(key, (n,))
y_data = X_data @ true_beta + 0.5 * jax.random.normal(key, (m,))

# SGD Algorithm
learning_rate = 0.1
```

```python
n_epochs = 10
batch_size = 32
beta = jnp.zeros(n)
n_batches = m // batch_size

for epoch in range(n_epochs):
    # Shuffle the data at the start of each epoch
    key, subkey = jax.random.split(key)
    permutation = jax.random.permutation(subkey, m)
    X_shuffled, y_shuffled = X_data[permutation], y_data[permutation]

    epoch_loss = 0.0
    for i in range(n_batches):
        # Get a mini-batch
        start = i * batch_size
        end = start + batch_size
        X_batch, y_batch = X_shuffled[start:end], y_shuffled[start:end]

        # Compute gradient on the mini-batch and update
        gradient = grad_ols_batch(beta, X_batch, y_batch)
        beta = beta - learning_rate * gradient
        epoch_loss += ols_loss_batch(beta, X_batch, y_batch)

    print(f"Epoch {epoch}: Avg Loss = {epoch_loss / n_batches:.4f}")

print("\n--- Results ---")
print(f"True beta: {true_beta.round(2)}")
print(f"SGD beta:  {beta.round(2)}")
```

Listing 22: JAX implementation of Stochastic Gradient Descent (SGD) for OLS.

13.6 Practical Advantages of SGD

1. **Speed:** The computational cost of one SGD update is independent of the total dataset size m. This allows us to make thousands of updates in the time it would take for a single batch gradient descent update.

2. **Regularization Effect:** The noise in the gradient updates can help the algorithm escape from shallow local minima and saddle points, which can be a significant advantage in non-convex optimization (like training neural networks).

3. **Online Learning:** SGD is naturally suited for "online" settings where data arrives in a continuous stream. The model can be updated with each new piece of data without needing to store the entire dataset.

13.7 Practical Considerations: Learning Rate Schedule

Because the gradient in SGD is noisy, the algorithm never truly converges to the exact minimum. Instead, it tends to "bounce around" near the minimum. To mitigate this, it is common to use a **learning rate schedule**, where the learning rate η is gradually decreased over time. A common strategy is to start with a relatively large η to make rapid progress and then reduce it to allow the algorithm to settle into a good minimum.

14 Adaptive Learning Rate Methods

14.1 The Challenge of a Single Learning Rate

In our journey from Gradient Descent to Momentum and NAG, we have developed sophisticated ways to choose the *direction* of our next step. However, one major challenge remains: all these methods rely on a single, global learning rate η that is applied uniformly to all parameters.

This "one size all" approach can be inefficient. Consider two scenarios:

1. **Sparse Data:** In a natural language model, some words (like "the") appear constantly, while others (like "aardvark") are very rare. We might want to take large steps to update the parameters for rare words (since we have little information about them) but smaller, more careful steps for common words.

2. **Ill-Conditioned Problems:** In the "ravine" landscapes we've discussed, the surface is much steeper in one direction than another. A good learning rate for the steep direction might be too small for the shallow direction, and a good rate for the shallow direction might cause divergence in the steep direction.

This motivates the need for **adaptive learning rate methods**, which automatically adjust a per-parameter learning rate during training.

14.2 AdaGrad: The Adaptive Gradient Algorithm

AdaGrad was one of the first and most influential adaptive methods. Its core idea is intuitive:

> Parameters that have received large gradients in the past should have their learning rate reduced.

This is a way of saying, "If you've been moving a lot in one direction, be more cautious in that direction in the future."

14.3 The AdaGrad Algorithm

AdaGrad achieves this by accumulating the square of the gradients for each parameter over all past iterations.

Definition 14.1 (The AdaGrad Algorithm)**.** *Initialize a "cache" vector $G_0 = \mathbf{0}$. At each iteration k:*

1. *Compute the gradient (or stochastic gradient) $g_k = \nabla f(x_k)$.*

2. *Accumulate the square of each component of the gradient into the cache:*

$$G_k = G_{k-1} + g_k \odot g_k \quad (\textit{element-wise product}).$$

3. *Update the parameters using a per-parameter learning rate:*

$$x_{k+1} = x_k - \frac{\eta}{\sqrt{G_k + \epsilon}} \odot g_k.$$

Here, η is a global learning rate, and ϵ is a small constant (e.g., 10^{-8}) to prevent division by zero. The division and square root are performed element-wise.

How it works:

- If a parameter x_i has consistently large gradients, its corresponding entry $G_{k,i}$ will grow quickly, and its effective learning rate $\eta/\sqrt{G_{k,i} + \epsilon}$ will shrink.

- If a parameter x_j has small or infrequent gradients, $G_{k,j}$ will grow slowly, and its effective learning rate will remain large.

Main Drawback: The cache G_k only grows. After many iterations, the accumulated sum can become so large that the effective learning rate becomes infinitesimally small, effectively stopping the learning process.

14.4 RMSProp: A Fix for AdaGrad

To address AdaGrad's aggressively decaying learning rate, RMSProp (Root Mean Square Propagation) was proposed. It modifies AdaGrad with one simple but crucial change.

> Instead of accumulating all past squared gradients, use an **exponentially decaying moving average**.

This prevents the cache from growing indefinitely and allows the algorithm to adapt to more recent gradient information.

Definition 14.2 (The RMSProp Algorithm)**.** *Initialize cache $G_0 = \mathbf{0}$. At each iteration k:*

1. *Compute the gradient $g_k = \nabla f(x_k)$.*
2. *Update the cache using a decay rate γ (e.g., 0.9):*
$$G_k = \gamma G_{k-1} + (1-\gamma)(g_k \odot g_k).$$
3. *Update the parameters:*
$$x_{k+1} = x_k - \frac{\eta}{\sqrt{G_k + \epsilon}} \odot g_k.$$

RMSProp is not as commonly used on its own, but it forms a critical component of the most popular optimizer in deep learning: Adam.

14.5 Adam: Adaptive Moment Estimation

The Adam optimizer can be thought of as the "best of both worlds." It combines the idea of **momentum** (which stores a moving average of the gradients themselves) with the adaptive learning rate idea from **RMSProp** (which stores a moving average of the squared gradients).

- The "first moment" is the mean of the gradients (the momentum term).
- The "second moment" is the uncentered variance of the gradients (the RMSProp term).

Definition 14.3 (The Adam Algorithm)**.** *Initialize a first moment vector $m_0 = \mathbf{0}$ and a second moment vector $v_0 = \mathbf{0}$. At each iteration k:*

1. *Compute the gradient $g_k = \nabla f(x_k)$.*
2. *Update the biased first moment estimate:*
$$m_k = \beta_1 m_{k-1} + (1-\beta_1) g_k.$$
3. *Update the biased second moment estimate:*
$$v_k = \beta_2 v_{k-1} + (1-\beta_2)(g_k \odot g_k).$$

4. *(Optional but important) Compute bias-corrected estimates:*

$$\hat{m}_k = \frac{m_k}{1-\beta_1^k}, \quad \hat{v}_k = \frac{v_k}{1-\beta_2^k}.$$

   This correction step helps during the initial phases of training when the moment estimates are biased towards zero.

5. *Update the parameters:*

$$x_{k+1} = x_k - \frac{\eta}{\sqrt{\hat{v}_k + \epsilon}} \odot \hat{m}_k.$$

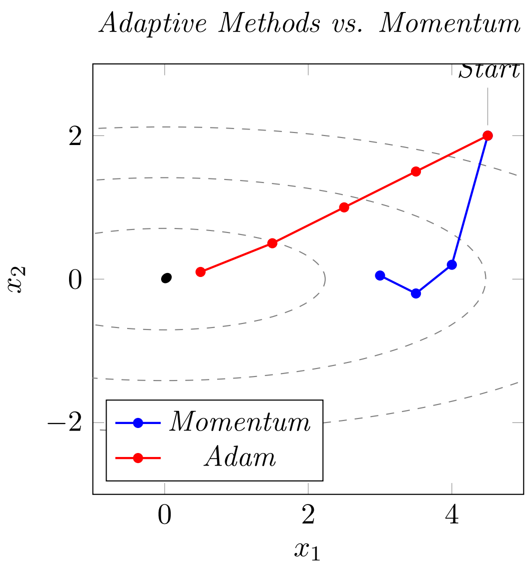
Typical hyperparameter values are $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\eta = 0.001$.

```python
import jax
import jax.numpy as jnp

# Use the same OLS setup
def ols_loss(beta, X, y):
  return jnp.sum((X @ beta - y)**2)
grad_ols = jax.grad(ols_loss)
key = jax.random.PRNGKey(123)
m, n = 100, 10
X_data = jax.random.normal(key, (m, n))
true_beta = jax.random.normal(key, (n,))
y_data = X_data @ true_beta + 0.5 * jax.random.normal(key, (m,))

# Adam Algorithm
beta = jnp.zeros(n)
m_t = jnp.zeros(n) # First moment vector
v_t = jnp.zeros(n) # Second moment vector

# Hyperparameters
learning_rate = 0.1
beta1 = 0.9
beta2 = 0.999
epsilon = 1e-8
n_iterations = 200

for t in range(1, n_iterations + 1):
    gradient = grad_ols(beta, X_data, y_data)

    # Update biased moment estimates
    m_t = beta1 * m_t + (1 - beta1) * gradient
    v_t = beta2 * v_t + (1 - beta2) * (gradient**2)

    # Compute bias-corrected estimates
    m_hat = m_t / (1 - beta1**t)
    v_hat = v_t / (1 - beta2**t)

    # Update parameters
    update_vec = learning_rate * m_hat / (jnp.sqrt(v_hat) + epsilon)
    beta = beta - update_vec

    if (t-1) % 40 == 0:
      print(f"Adam Iter {t-1}: Loss = {ols_loss(beta, X_data, y_data):.2f}")

print("\n--- Results ---")
print(f"True beta: {true_beta.round(2)}")
print(f"Adam beta: {beta.round(2)}")
```

Listing 23: A JAX implementation of the Adam optimizer.

Adam is incredibly successful and is the default choice for most deep learning applications because it is robust, computationally efficient, and performs well across a wide range of problems.

Example 14.1 (Visual Comparison in a Ravine). *Adaptive methods can navigate narrow valleys even more effectively than momentum. Because they scale down the learning rate in the steep direction (where squared gradients are large) and maintain a larger learning rate in the shallow direction (where squared gradients are small), they can take a much more direct path to the minimum.*



15 Second-Order Optimization & Newton's Method

15.1 Recap: The Limits of First-Order Methods

The family of first-order algorithms we have studied, from Gradient Descent to Adam, all share a common strategy: they approximate the objective function locally with a *linear* model (its tangent plane) and take a step in the direction of steepest descent.

While simple and scalable, this linear approximation ignores a crucial piece of information: the function's **curvature**. This is why first-order methods struggle in long, narrow valleys—they can only see the steepest local slope, which points across the valley, not along it.

To achieve faster convergence, we can use a more sophisticated local model of our function. This leads us to **second-order optimization methods**.

15.2 The Second-Order Approximation

Instead of a line, we can approximate our function $f(x)$ at a point x_k with a quadratic function that matches its value, its gradient, and its curvature (Hessian). This is the second-order Taylor expansion:

$$f(x) \approx \underbrace{f(x_k) + \nabla f(x_k)^T(x - x_k) + \frac{1}{2}(x - x_k)^T \nabla^2 f(x_k)(x - x_k)}_{m_k(x)}.$$

This quadratic model $m_k(x)$ is a much more faithful approximation of $f(x)$ near x_k than the simple linear model used by gradient descent. The logical next step is to find the minimum of this approximation and jump there directly.

15.3 Newton's Method

Definition 15.1 (Newton's Method)**.** ***Newton's method*** *is an iterative algorithm where each step jumps to the minimum of the local quadratic approximation of the objective function.*

To find the minimum of $m_k(x)$, we set its gradient with respect to x to zero:

$$\nabla_x m_k(x) = \nabla f(x_k) + \nabla^2 f(x_k)(x - x_k) = \mathbf{0}.$$

Solving for the next iterate, $x_{k+1} = x$, we get:

$$\begin{aligned} \nabla^2 f(x_k)(x_{k+1} - x_k) &= -\nabla f(x_k) \\ x_{k+1} - x_k &= -(\nabla^2 f(x_k))^{-1} \nabla f(x_k) \\ x_{k+1} &= x_k - (\nabla^2 f(x_k))^{-1} \nabla f(x_k). \end{aligned}$$

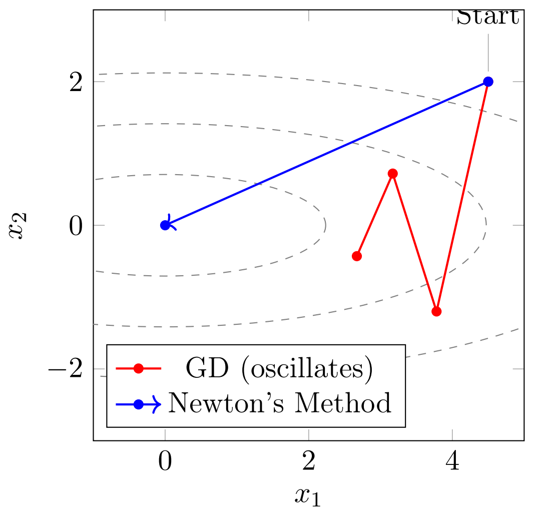
This defines the pure **Newton step**.

15.4 Affine Invariance: Preconditioning the Gradient

The Newton step can be rewritten as taking the gradient $\nabla f(x_k)$ and multiplying it by the matrix $H_k^{-1} = (\nabla^2 f(x_k))^{-1}$. This matrix is called a **preconditioner**. It "preconditions" the gradient by warping the geometry of the space.

This preconditioning makes the optimization problem immune to linear transformations of the coordinates (**affine invariant**). If we stretch a variable, gradient descent's performance changes dramatically, but Newton's method is unaffected. It automatically adapts the step for each direction based on the curvature, effectively turning a narrow ravine into a perfectly circular bowl.

Newton's Method vs. Gradient Descent



15.5 Convergence Rate

Near a minimum, Newton's method exhibits **quadratic convergence**. This means that the number of correct digits in the solution roughly doubles at each iteration. This is in stark contrast to the **linear convergence** of gradient descent, where the error is reduced by a constant factor at each step. For high-precision solutions, Newton's method is astronomically faster.

15.6 The Catch: Prohibitive Cost

With such amazing properties, why isn't Newton's method the default? The reason is its massive computational cost for high-dimensional problems (large n). At each step, we must:

1. **Compute the Hessian matrix, $\nabla^2 f(x_k)$:** This is an $n \times n$ matrix with $O(n^2)$ entries. If your model has 1 million parameters ($n = 10^6$), the Hessian has 10^{12} entries. Storing it would require thousands of terabytes of RAM.
2. **Solve the linear system $H_k d_k = -\nabla f(x_k)$:** Solving this system for the Newton step d_k is equivalent to inverting the Hessian. This is an $O(n^3)$ operation. For $n = 10^6$, this is beyond the capability of any supercomputer.

Because of this, pure Newton's method is only practical for problems with a few thousand parameters at most.

15.7 Practical Second-Order Methods

15.8 Damped Newton's Method

The pure Newton step can be too aggressive if the quadratic approximation is poor (when far from a minimum). A simple fix is to introduce a step size η, just like in gradient descent.

$$x_{k+1} = x_k - \eta(\nabla^2 f(x_k))^{-1}\nabla f(x_k).$$

This is called the **Damped Newton's Method**. The step size η is typically chosen via a line search to ensure sufficient decrease in the objective.

15.9 Quasi-Newton Methods: L-BFGS

The biggest hurdle is the Hessian. **Quasi-Newton methods** are a family of algorithms that try to get the benefits of Newton's method without paying the full computational price.

The core idea is to build up an *approximation* of the inverse Hessian, $B_k \approx (\nabla^2 f(x_k))^{-1}$, using only information from the gradients over the last few steps. The most famous of these is **L-BFGS (Limited-memory Broyden–Fletcher–Goldfarb–Shanno)**.

- It does not store the full $n \times n$ approximate inverse Hessian.
- Instead, it stores the last p update vectors (e.g., $p = 10$ or 20).
- It uses these vectors to implicitly compute the product of the approximate inverse Hessian and the gradient.

L-BFGS is a workhorse for medium-sized optimization problems in classical statistics. It converges much faster than gradient descent but does not require the $O(n^2)$ storage of Newton's method.

15.10 Application: Newton's Method for Logistic Regression

Let's see what Newton's method looks like for our logistic regression example.

Example 15.1 (Iteratively Reweighted Least Squares (IRLS))**.** *Derive the Newton's method update for the logistic regression negative log-likelihood (NLL) and show its connection to weighted least squares.*

Proof:

1. ***Recall the Gradient and Hessian.*** *From Lecture 7, the NLL is* $f(\beta) = \sum_{i=1}^{m}(\log(1 + e^{\beta^T x_i}) - y_i \beta^T x_i)$. *The gradient and Hessian are:*

$$\nabla f(\beta) = \sum_{i=1}^{m}(\sigma(\beta^T x_i) - y_i)x_i = X^T(p - y)$$
$$\nabla^2 f(\beta) = \sum_{i=1}^{m}\sigma(\beta^T x_i)(1 - \sigma(\beta^T x_i))x_i x_i^T = X^T W X$$

where p is the vector of predicted probabilities $\sigma(\beta^T x_i)$, and W is a diagonal matrix with $W_{ii} = p_i(1 - p_i)$.

2. ***Form the Newton Step.*** *The Newton update for β is:*

$$\beta_{k+1} = \beta_k - (\nabla^2 f(\beta_k))^{-1}\nabla f(\beta_k) = \beta_k - (X^T W_k X)^{-1} X^T(p_k - y).$$

3. ***The Connection to Weighted Least Squares (WLS).*** *Let's rearrange the update.*

$$\beta_{k+1} = (X^T W_k X)^{-1}(X^T W_k X)\beta_k - (X^T W_k X)^{-1} X^T(p_k - y)$$
$$\beta_{k+1} = (X^T W_k X)^{-1}\left(X^T W_k(X\beta_k) - X^T(p_k - y)\right)$$

Now let's define a "working response" vector z_k:

$$z_k = X\beta_k - W_k^{-1}(p_k - y).$$

Substituting this into the update gives:

$$\beta_{k+1} = (X^T W_k X)^{-1} X^T W_k z_k.$$

This is exactly the closed-form solution to a ***Weighted Least Squares*** *problem:*

$$\min_{\beta} ||W_k^{1/2}(z_k - X\beta)||_2^2.$$

*This is the famous **Iteratively Reweighted Least Squares (IRLS)** algorithm. Newton's method for logistic regression is equivalent to solving a sequence of weighted linear regressions, where the weights are updated at each iteration based on the current predicted probabilities. This connection is a classic result in generalized linear models.*

```python
import jax
import jax.numpy as jnp
from jax.scipy.special import expit as sigmoid # Sigmoid function

# --- 1. Define the Logistic Regression NLL and its derivatives ---
def nll_logistic(beta, X, y):
    logits = X @ beta
    # A stable implementation of log-likelihood
    return -jnp.sum(y * logits - jnp.log(1 + jnp.exp(logits)))

grad_nll = jax.grad(nll_logistic)
hessian_nll = jax.hessian(nll_logistic)

# --- 2. Generate synthetic data for a classification problem ---
key = jax.random.PRNGKey(0)
m, n = 100, 5
X_data = jax.random.normal(key, (m, n))
true_beta = jnp.array([2.0, -1.5, 1.0, 0.0, -2.5])
true_logits = X_data @ true_beta
true_probs = sigmoid(true_logits)
y_data = jax.random.bernoulli(key, true_probs) # Binary labels {0, 1}

# --- 3. Newton's Method (IRLS) Algorithm ---
beta = jnp.zeros(n)
n_iterations = 10

print("Running Newton's method (IRLS)...")
for i in range(n_iterations):
    gradient = grad_nll(beta, X_data, y_data)
    hessian = hessian_nll(beta, X_data, y_data)

    # Solve H*step = -g instead of inverting H
    # This is more numerically stable
    step = jnp.linalg.solve(hessian, -gradient)

    beta = beta + step # Damped step eta=1

    loss = nll_logistic(beta, X_data, y_data)
    print(f"Iter {i}: Loss = {loss:.2f}")

print("\n--- Results ---")
print(f"True beta: {true_beta}")
print(f"Newton beta: {beta.round(2)}")
```

Listing 24: JAX implementation of Newton's method for Logistic Regression (IRLS).

16 Variational Inference

16.1 The Bayesian Challenge: The Intractable Denominator

Throughout this course, we have primarily focused on finding a single "best" set of parameters (a point estimate) that optimizes an objective function, such as in Maximum Likelihood Estimation (MLE) or Ordinary Least Squares (OLS).

The Bayesian approach to statistics offers a different perspective. Instead of seeking a single point estimate, it aims to characterize our uncertainty by finding a full probability distribution for the model parameters, known as the **posterior distribution**.

Using Bayes' theorem, the posterior distribution for parameters θ given observed data y is:

$$p(\theta|y) = \frac{p(y|\theta)p(\theta)}{p(y)} = \frac{\text{Likelihood} \times \text{Prior}}{\text{Evidence}}.$$

The components are:

- **Likelihood** $p(y|\theta)$: The probability of the data given the parameters (same as in MLE).
- **Prior** $p(\theta)$: Our beliefs about the parameters *before* seeing the data.
- **Evidence** $p(y)$: The marginal probability of the data, averaged over all possible parameters.

The evidence, $p(y)$, is the normalization constant that ensures the posterior integrates to 1. It is defined by an integral:

$$p(y) = \int p(y|\theta)p(\theta)d\theta.$$

For almost any interesting model, this integral is high-dimensional and **intractable**—it cannot be computed analytically. This "intractable denominator" is the central computational challenge in Bayesian statistics.

Traditional methods like Markov Chain Monte Carlo (MCMC) try to get around this by drawing samples from the posterior without ever computing the denominator. While powerful, MCMC can be very slow, especially for large datasets or complex models. This motivates a different approach.

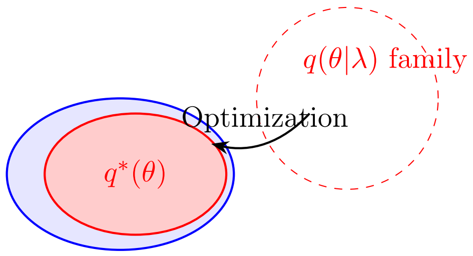
16.2 The Core Idea: Approximation via Optimization

Variational Inference (VI) turns the difficult problem of integration (computing the posterior) into a more tractable problem of **optimization**.

The central idea is:

> Instead of trying to compute the true, complex posterior $p(\theta|y)$, let's approximate it with a simpler, tractable distribution $q(\theta|\lambda)$ from a chosen family of distributions. Then, we can use optimization to find the member of that family that is "closest" to the true posterior.

The distribution $q(\theta|\lambda)$ is called the **variational distribution**, and λ are the **variational parameters** we will optimize.



16.3 Measuring Closeness: The KL-Divergence

To find the "closest" approximation, we need a way to measure the dissimilarity between two probability distributions. The standard measure used in VI is the **Kullback-Leibler (KL) divergence**.

Definition 16.1 (KL Divergence)**.** *The KL divergence from a distribution q to a distribution p is defined as:*

$$D_{KL}(q||p) = \int q(\theta) \log \left(\frac{q(\theta)}{p(\theta)} \right) d\theta = \mathbb{E}_q[\log(q(\theta)) - \log(p(\theta))].$$

The VI optimization problem is to find the variational parameters λ that minimize this divergence:

$$\lambda^* = \arg\min_{\lambda} D_{KL}(q(\theta|\lambda)||p(\theta|y)).$$

Properties of KL-Divergence:

- $D_{KL}(q||p) \geq 0$.
- $D_{KL}(q||p) = 0$ if and only if $q = p$.
- **It is not symmetric:** $D_{KL}(q||p) \neq D_{KL}(p||q)$. This asymmetry is crucial. The choice $D_{KL}(q||p)$ forces our approximation q to be zero wherever the true posterior p is zero, which often leads to underestimating the variance of the posterior.

16.4 The Evidence Lower Bound (ELBO)

The KL divergence still contains the intractable posterior $p(\theta|y)$. Here comes the central trick of variational inference. We can rearrange the KL divergence into a more useful form.

$$\begin{aligned}
D_{KL}(q(\theta|\lambda)||p(\theta|y)) &= \mathbb{E}_q[\log(q(\theta|\lambda)) - \log(p(\theta|y))] \\
&= \mathbb{E}_q[\log(q(\theta|\lambda))] - \mathbb{E}_q\left[\log\left(\frac{p(y|\theta)p(\theta)}{p(y)}\right)\right] \\
&= \mathbb{E}_q[\log q] - \mathbb{E}_q[\log(p(y|\theta)p(\theta))] + \mathbb{E}_q[\log p(y)] \\
&= \mathbb{E}_q[\log q] - \mathbb{E}_q[\log p(y,\theta)] + \log p(y) \quad \text{(since } p(y) \text{ is constant w.r.t. } \theta)
\end{aligned}$$

Rearranging this equation gives:

$$\log p(y) = D_{KL}(q||p) + \underbrace{\mathbb{E}_q[\log p(y,\theta)] - \mathbb{E}_q[\log q(\theta|\lambda)]}_{\text{Defined as the ELBO}}.$$

Definition 16.2 (The Evidence Lower Bound (ELBO))**.** *The Evidence Lower Bound is defined as:*

$$\mathcal{L}(\lambda) = \mathbb{E}_{q(\theta|\lambda)}[\log p(y,\theta)] - \mathbb{E}_{q(\theta|\lambda)}[\log q(\theta|\lambda)].$$

Since $D_{KL} \geq 0$, we have $\log p(y) \geq \mathcal{L}(\lambda)$. The ELBO is a **lower bound** on the log evidence of the data.

Now, look at the rearranged equation again. The log evidence $\log p(y)$ is a constant with respect to our choice of q. Therefore, minimizing the KL divergence is **exactly equivalent to maximizing the ELBO**.

$$\arg\min_{\lambda} D_{KL}(q||p) = \arg\max_{\lambda} \mathcal{L}(\lambda).$$

This is the key result. We have transformed the problem into maximizing an objective function, the ELBO, that we can actually compute and differentiate.

16.5 Intuition Behind the ELBO

The ELBO has an intuitive structure. Let's rewrite it:

$$\mathcal{L}(\lambda) = \underbrace{\mathbb{E}_{q(\theta|\lambda)}[\log p(y|\theta)]}_{\text{Term 1: Data Fit}} - \underbrace{D_{KL}(q(\theta|\lambda)||p(\theta))}_{\text{Term 2: Regularization}}.$$

- **Term 1: Data Fit.** This is the expected log-likelihood of the data, averaged over our approximate posterior. Maximizing this term pushes our distribution q to put its mass on parameter values θ that explain the data well.
- **Term 2: Regularization.** This is the KL divergence from our approximation q to the prior $p(\theta)$. Maximizing the ELBO means *minimizing* this KL term. This encourages our approximation to stay close to our prior beliefs, acting as a form of regularization and preventing overfitting.

VI is therefore a trade-off: find a distribution that fits the data well, but doesn't stray too far from the prior.

16.6 Variational Families: The Mean-Field Assumption

The choice of the variational family $q(\theta|\lambda)$ is crucial. We need a family that is flexible enough to be a good approximation, but simple enough to make the expectations in the ELBO tractable.

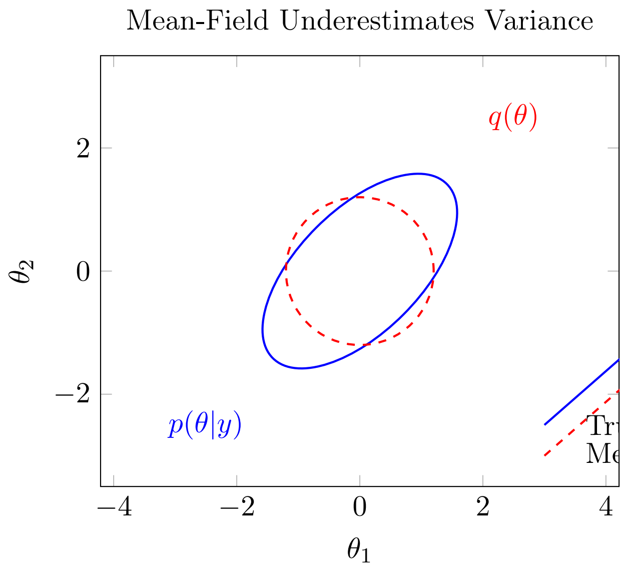
The most common choice is the **mean-field variational family**.

Definition 16.3 (Mean-Field Assumption)**.** *This assumption states that the posterior distribution over the parameters can be factorized into independent parts:*

$$q(\theta_1, \ldots, \theta_m|\lambda) = \prod_{j=1}^{m} q_j(\theta_j|\lambda_j).$$

We assume the parameters are independent in the approximate posterior, even if they are not in the true posterior.

Pros: This simplification makes the optimization much easier. We can often optimize each q_j individually. **Cons:** The mean-field assumption is a strong one. It cannot capture any correlation between the parameters in the posterior. This often leads to an approximation that is too "tight" and underestimates the true posterior variance.



16.7 Optimizing the ELBO

Once we have chosen a variational family, we are left with a standard optimization problem: $\max_\lambda \mathcal{L}(\lambda)$. We can solve this with the tools we already know. The most common method is **gradient ascent** (which is just gradient descent on the negative ELBO).

The update rule is:

$$\lambda_{k+1} = \lambda_k + \eta \nabla_\lambda \mathcal{L}(\lambda_k).$$

This is the core of modern "black-box" variational inference. Using automatic differentiation tools (like in PyTorch, TensorFlow, or Stan), we can compute the gradient of the ELBO with respect to the variational parameters and optimize it, without needing to do complex model-specific derivations by hand.

```
import jax
import jax.numpy as jnp
import jax.scipy.stats as stats

# Model: Infer the mean ‘mu‘ of a Gaussian from data ‘y‘
# Prior: mu ~ Normal(0, 1)
# Likelihood: y | mu ~ Normal(mu, 1)
# Variational Family: q(mu) = Normal(q_mu, q_sigma)

# Generate some data from a true ‘mu‘
true_mu = 5.0
key = jax.random.PRNGKey(0)
y_data = jax.random.normal(key, (100,)) + true_mu

# ELBO objective function. We want to maximize this w.r.t. ‘q_params‘.
# ‘q_params‘ is a dictionary {’mu’: ..., ’sigma’: ...}
def elbo(q_params, data):
    # We use Monte Carlo estimation for the expectations
    # 1. Sample from q(mu)
    key = jax.random.PRNGKey(1)
    samples = jax.random.normal(key, (1000,)) * q_params[’sigma’] + q_params[’
    mu’]

    # 2. Compute E_q[log p(y, mu)] = E_q[log p(y|mu) + log p(mu)]
    log_likelihood = jnp.sum(stats.norm.logpdf(data, loc=samples[:, None]),
    axis=1)
    log_prior = stats.norm.logpdf(samples, loc=0., scale=1.)

    # 3. Compute E_q[log q(mu)]
    log_q = stats.norm.logpdf(samples, loc=q_params[’mu’], scale=q_params[’
    sigma’])

    # Return the ELBO (or its estimate)
    return jnp.mean(log_likelihood + log_prior - log_q)

# We want to maximize ELBO, so we minimize its negative
def neg_elbo(q_params, data):
    return -elbo(q_params, data)

grad_neg_elbo = jax.grad(neg_elbo)

# Variational parameters to optimize
# Use softplus for sigma to ensure it’s positive
variational_params = {’mu’: 0.0, ’log_sigma’: 0.0}

def get_params(opt_params):
    return {’mu’: opt_params[’mu’], ’sigma’: jnp.exp(opt_params[’log_sigma’])}

# Simple gradient ascent (descent on negative ELBO)
learning_rate = 0.1
```

```
for i in range(1000):
    params = get_params(variational_params)
    grads = grad_neg_elbo(params, y_data)
    variational_params['mu'] -= learning_rate * grads['mu']
    variational_params['log_sigma'] -= learning_rate * grads['sigma'] # This is
     incorrect, a real implementation would use a library

    if i % 100 == 0:
        print(f"Iter {i}: ELBO = {elbo(get_params(variational_params), y_data)
    :.2f}")

final_q = get_params(variational_params)
print("\n--- Results ---")
print(f"True posterior mean is approx {jnp.mean(y_data):.2f}")
print(f"VI posterior mean: {final_q['mu']:.2f}")
print(f"VI posterior stddev: {final_q['sigma']:.2f}")
```

Listing 25: JAX code for optimizing the ELBO of a simple Bayesian model.

17 A Practical Guide & Course Review

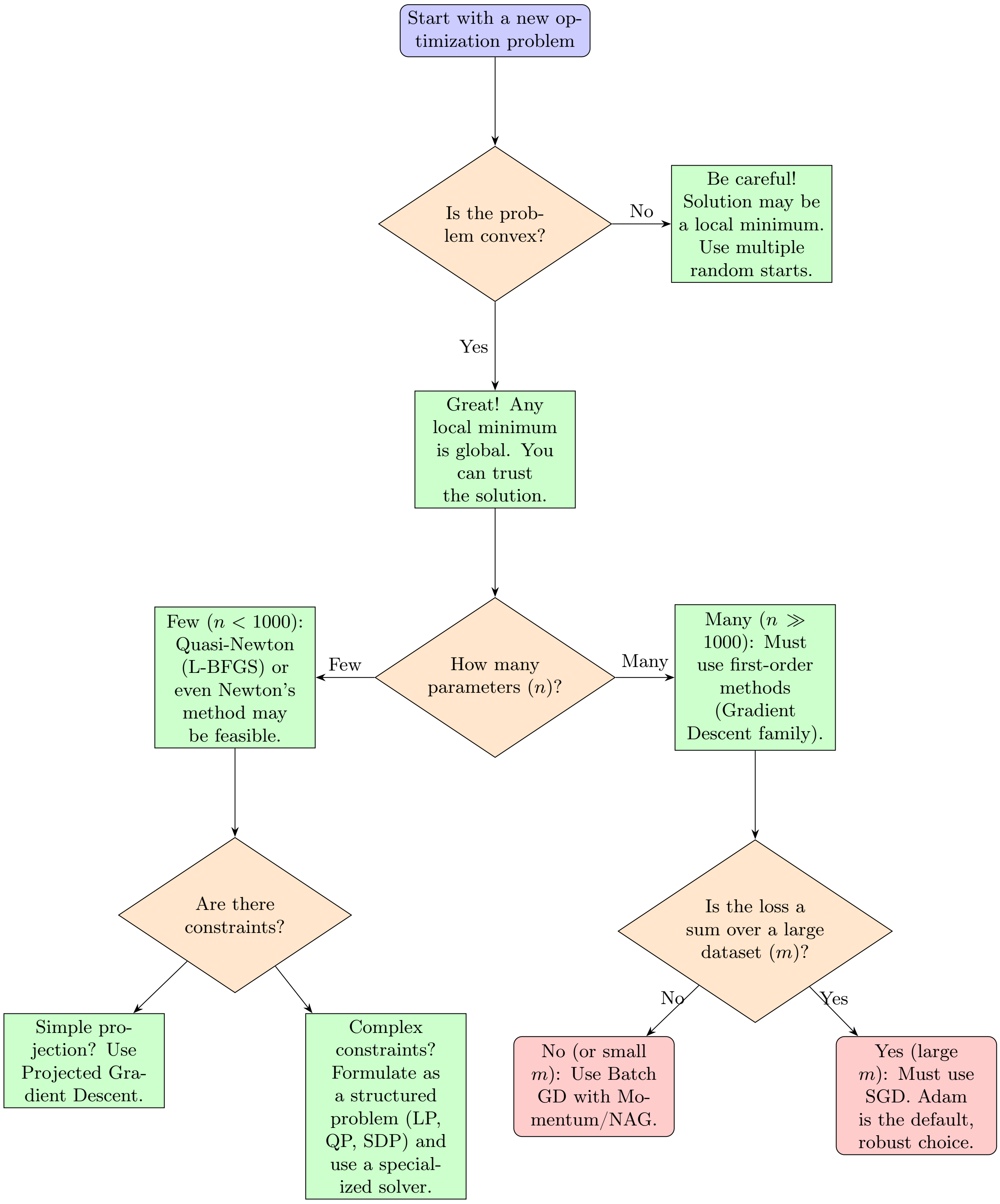
17.1 Introduction: The End of the Journey

Over the past several weeks, we have traveled from the foundational principles of calculus to the sophisticated algorithms that power modern machine learning. We have seen how abstract mathematical concepts like convexity, duality, and gradients are the essential ingredients for solving practical problems in statistics and data science.

This final lecture serves as a capstone. Our goal is to synthesize this knowledge into a practical framework. When you encounter a new problem in the wild, how do you decide which optimization tool to use? This lecture provides a flowchart for that decision-making process, reviews the family of algorithms we've studied, and points to the software that brings these ideas to life.

17.2 A Practitioner's Flowchart: Which Algorithm Do I Use?

Choosing an optimization strategy involves answering a series of questions about your problem's structure, size, and goals.



Start with a new optimization problem
Is the problem convex?
No
Be careful! Solution may be a local minimum. Use multiple random starts.
Yes
Great! Any local minimum is global. You can trust the solution.
How many parameters (n)?
Few
Few ($n < 1000$): Quasi-Newton (L-BFGS) or even Newton's method may be feasible.
Many
Many ($n \gg 1000$): Must use first-order methods (Gradient Descent family).
Are there constraints?
Simple projection? Use Projected Gradient Descent.
Complex constraints? Formulate as a structured problem (LP, QP, SDP) and use a specialized solver.
Is the loss a sum over a large dataset (m)?
No
No (or small m): Use Batch GD with Momentum/NAG.
Yes
Yes (large m): Must use SGD. Adam is the default, robust choice.

17.3 A Unified View: The Gradient Descent Family

The majority of large-scale optimization in statistics and machine learning is powered by the Gradient Descent family. This table summarizes their evolution and key ideas.

Algorithm	Update Idea	Key Contribution
GD	$x_{k+1} = x_k - \eta g_k$	The basic principle of moving against the gradient.
Momentum	$v_{k+1} = \beta v_k + \eta g_k$ $x_{k+1} = x_k - v_{k+1}$	Adds a "heavy ball" velocity to accelerate along consistent directions and dampen oscillations.
NAG	$g_k = \nabla f(x_k - \beta v_k)$	A "smarter" momentum that calculates the gradient after a lookahead step, allowing for faster course correction.
AdaGrad	$x_{k+1} = x_k - \frac{\eta}{\sqrt{G_k}} \odot g_k$	Introduces per-parameter adaptive learning rates. Slows down updates for frequently updated parameters. Good for sparse data.
RMSProp	$G_k = \gamma G_{k-1} + (1-\gamma) g_k^2$	Fixes AdaGrad's aggressive decay by using a moving average for the squared gradients instead of a sum.
Adam	Combines Momentum and RMSProp	The de facto standard. Uses moving averages for both the gradient (1st moment) and its square (2nd moment) to get the best of both worlds.

17.4 The Role of Software

While understanding these algorithms is crucial, you will rarely implement them from scratch in practice. The field relies on highly optimized and robust software packages.

17.5 Modeling Languages and Specialized Solvers

For structured convex problems (LPs, QPs, SDPs), you should use a modeling language like **CVXPY** (Python) or a specialized library like **jaxopt**. These tools separate the problem definition from the solution algorithm.

Example 17.1 (Solving Lasso with JAXOpt)**.** *Below is an implementation of Lasso using JAX-Opt.*

```python
import jax.numpy as jnp
from jaxopt import ProximalGradient
from jaxopt.loss import huber_loss # A smooth L1-like loss
import numpy as np

# Problem data.
m, n = 20, 5
X = np.random.randn(m, n)
y = np.random.randn(m)
lambda_param = 0.1

# The ‘jaxopt‘ solver needs two functions: one for the smooth part
# of the objective (OLS loss) and one for the non-smooth part (L1 norm).
def smooth_loss(beta, X, y):
    return jnp.sum((X @ beta - y)**2)

# jaxopt has built-in regularizers
# prox_l1 = ProximalGradient(fun=smooth_loss, prox=proximal_l1(lambda_param))

# Using a solver for L1-regularized least squares
pg = ProximalGradient(fun=smooth_loss, regularizer='l1', hyperparams_dtype=
    float)

# Run the solver
beta_init = jnp.zeros(n)
sol = pg.run(beta_init, hyperparams_prox=lambda_param, X=X, y=y)

print("Optimal beta:", sol.params)
```

Listing 26: A Lasso problem solved with the jaxopt library.

17.6 Libraries for Large-Scale Machine Learning

For large-scale, often non-convex problems like training neural networks, you will use libraries like **JAX** (often with higher-level libraries like **Flax**, **Haiku** or **Equinox**) or other frameworks like PyTorch and TensorFlow.

These libraries provide two critical features:

1. **Automatic Differentiation:** You define the forward pass, and the library computes the gradients.

2. **Optimizers:** Pre-built, optimized implementations of first-order methods are available. A popular choice for JAX is **Optax**.

Example 17.2. *Here's an example implementation of training a linear regression model in JAX.*

```
import jax
import jax.numpy as jnp
import optax
import equinox as eqx # Using Equinox for the model definition

# 1. Define a simple model using Equinox
class LinearModel(eqx.Module):
    linear: eqx.nn.Linear

    def __init__(self, in_features, out_features, *, key):
        self.linear = eqx.nn.Linear(in_features, out_features, key=key)

    def __call__(self, x):
        # The squeeze() is to match the output shape of the original example
        return self.linear(x).squeeze()

# 2. Define loss and the training step function
# The model itself is the first argument, containing the parameters.
def mse_loss(model, x_batch, y_batch):
    # vmap is used to apply the model to each sample in the batch
    y_pred = jax.vmap(model)(x_batch)
    return jnp.mean((y_pred - y_batch)**2)

# Use ‘eqx.filter_jit‘ to handle JIT compilation of Equinox models
@eqx.filter_jit
def train_step(model, opt_state, optimizer, x_batch, y_batch):
    # a. Compute loss and gradients w.r.t. the model's arrays
    loss, grads = eqx.filter_value_and_grad(mse_loss)(model, x_batch, y_batch)

    # b. Update the model and optimizer state
    updates, opt_state = optimizer.update(grads, opt_state)
    model = eqx.apply_updates(model, updates)
    return model, opt_state, loss

# --- Setup for a training loop ---
key = jax.random.PRNGKey(0)
in_features, out_features = 10, 1

# Model and parameters are created together
model = LinearModel(in_features, out_features, key=key)

optimizer = optax.adam(learning_rate=0.01)
# The optimizer state is initialized from the model
opt_state = optimizer.init(model)

# In a real loop, you would call train_step(..) repeatedly with batches of data
x_sample = jnp.ones((32, in_features))
y_sample = jnp.ones(32)

# The train_step now takes and returns the entire model object
model, opt_state, loss = train_step(model, opt_state, optimizer, x_sample,
    y_sample)
print(f"Loss after one step: {loss:.4f}")
```

Listing 27: A standard training step using JAX, Equinox, and Optax.

17.7 Key Takeaways from the Course

As we conclude, here are the most important high-level ideas to take away:

- **Optimization is the Engine:** Many, if not most, problems in statistics and machine learning are fundamentally optimization problems.
- **Convexity is "Easy":** Convex problems are special because local minima are global minima. Identifying convexity is a key skill.
- **The Gradient is Your Guide:** For any differentiable function, the gradient points uphill. The negative gradient is the heart of all first-order methods.
- **Duality Provides Insight:** The dual problem gives us bounds and powerful interpretations, like shadow prices and support vectors.
- **For Big Data, Go Stochastic:** When your dataset is huge, using cheap, noisy gradients (SGD) is far more efficient than using the full, expensive gradient.
- **Adam is the Robust Default:** For deep learning and other high-dimensional non-convex problems, Adam's combination of momentum and adaptive learning rates makes it a powerful and reliable first choice.
- **Use Modern Tools:** Don't reinvent the wheel. Leverage powerful modeling languages and ML libraries to solve problems efficiently and reliably.

18 Matrix Factorization and Recommender Systems

18.1 Introduction: The Recommendation Problem

In the modern digital landscape, we are constantly faced with an overwhelming amount of choice. Recommender systems are the algorithms that power personalized suggestions on platforms like Netflix, Amazon, and Spotify. They are a direct and economically vital application of statistical modeling and optimization.

The core of many recommender systems is the problem of **Collaborative Filtering**. The central idea is to leverage the preferences of a large community of users to make predictions for an individual. The data for such a system is typically represented as a large, sparse matrix called the **utility matrix**, R.

- The rows of R correspond to users.
- The columns of R correspond to items (e.g., movies, products).
- The entry r_{ui} contains the rating user u gave to item i.
- Most entries in this matrix are unknown (missing), as a typical user has only rated a tiny fraction of the available items.

The recommendation problem is thus transformed into a statistical optimization problem: **How can we best predict the missing entries of the utility matrix?**

$$R = \begin{pmatrix} & 5 & 3 & ? & 1 \\ & 4 & ? & ? & 1 \\ \text{Users} & 1 & 1 & ? & 5 \\ & ? & ? & 5 & 4 \end{pmatrix}$$
$$\text{Items}$$

18.2 Matrix Factorization: Learning Latent Factors

A naive approach, like predicting the average rating for a movie, ignores user-specific tastes. A more powerful idea is to assume that both users and items can be characterized by a small number of unobserved, or **latent**, factors.

For movies, these latent factors might represent genres like "comedy" vs. "drama", "action-packed" vs. "character-driven", or more abstract concepts that the data reveals. For users, these factors would represent their affinity for each of these concepts.

This is the core idea of **Matrix Factorization**. We hypothesize that the complete utility matrix R (of size $m \times n$, for m users and n items) can be approximated by the product of two much smaller, denser matrices:

$$R \approx P \times Q^T$$

where:

- P is a **user-factor matrix** of size $m \times k$. Each row p_u is a k-dimensional vector representing the latent preferences of user u.
- Q is an **item-factor matrix** of size $n \times k$. Each row q_i is a k-dimensional vector representing the latent characteristics of item i.

- k is the number of latent factors, a hyperparameter we choose (e.g., $k = 20$). It is typically much smaller than m or n.

The prediction for user u's rating of item i is then simply the dot product of their respective latent factor vectors:

$$\hat{r}_{ui} = p_u^T q_i = \sum_{j=1}^{k} p_{uj} q_{ij}.$$

This model assumes that a high rating occurs when a user's preferences (p_u) align well with an item's characteristics (q_i). Our goal is now to find the matrices P and Q that make these predictions as accurate as possible.

18.3 The Optimization Problem

We can find the optimal latent factor matrices P and Q by minimizing an objective function. This function has two essential components: a loss term to ensure our predictions are accurate, and a regularization term to prevent overfitting.

1. **The Loss Function:** We want our predicted ratings $\hat{r}_{ui}$ to be close to the true, known ratings r_{ui}. The most common choice is the sum of squared errors over all known ratings. Let K be the set of (u, i) pairs for which r_{ui} is known.

$$\text{Loss} = \sum_{(u,i)\in K} (r_{ui} - \hat{r}_{ui})^2 = \sum_{(u,i)\in K} (r_{ui} - p_u^T q_i)^2.$$

2. **The Regularization Term:** Minimizing only the loss can lead to severe overfitting, especially for users or items with very few ratings. To combat this, we add an L2 regularization penalty on the latent factors. This encourages the model to learn smaller, more generalizable factor vectors. The penalty is the sum of the squared Frobenius norms of the factor matrices.

$$\text{Regularizer} = \lambda \left(\sum_{u=1}^{m} ||p_u||_2^2 + \sum_{i=1}^{n} ||q_i||_2^2 \right).$$

   Here, λ is a hyperparameter that controls the strength of the regularization.

Definition 18.1 (Matrix Factorization Objective)**.** *The complete optimization problem is to find the user and item factor matrices P and Q that minimize the regularized sum of squared errors:*

$$\min_{P,Q} \sum_{(u,i)\in K} (r_{ui} - p_u^T q_i)^2 + \lambda \left(\sum_{u=1}^{m} ||p_u||_2^2 + \sum_{i=1}^{n} ||q_i||_2^2 \right).$$

This objective function is not jointly convex in P and Q, but it is biconvex (convex in P if Q is fixed, and vice-versa). This structure allows us to find good local minima, which are often sufficient for excellent recommendation quality.

18.4 Solving with Stochastic Gradient Descent (SGD)

One of the most common and scalable methods for this problem is **Stochastic Gradient Descent (SGD)**. Instead of computing the full gradient over all known ratings (which could be billions), we process one rating at a time.

For a single known rating r_{ui}, the error is $e_{ui} = r_{ui} - p_u^T q_i$. The objective for this single point is $L_{ui} = e_{ui}^2 + \lambda(||p_u||^2 + ||q_i||^2)$. The gradients with respect to the corresponding latent vectors are:

$$\nabla_{p_u} L_{ui} = \frac{\partial L_{ui}}{\partial p_u} = -2e_{ui} \cdot q_i + 2\lambda p_u$$
$$\nabla_{q_i} L_{ui} = \frac{\partial L_{ui}}{\partial q_i} = -2e_{ui} \cdot p_u + 2\lambda q_i$$

The SGD algorithm iterates through the known ratings and updates the corresponding latent vectors using these gradients.

The SGD Algorithm for Matrix Factorization:

1. Initialize all user-factor vectors p_u and item-factor vectors q_i with small random values.
2. Loop for a fixed number of epochs:
   (a) Shuffle the set of all known ratings K.
   (b) For each known rating (u, i, r_{ui}) in K:
      i. Calculate the prediction error: $e_{ui} = r_{ui} - p_u^T q_i$.
      ii. Update the user vector: $p_u \leftarrow p_u - \eta \cdot (-2e_{ui} \cdot q_i + 2\lambda p_u)$.
      iii. Update the item vector: $q_i \leftarrow q_i - \eta \cdot (-2e_{ui} \cdot p_u + 2\lambda q_i)$.

Here, η is the learning rate. By repeatedly making small updates for each rating, the algorithm gradually learns factor matrices P and Q that minimize the global objective.

18.5 Code Example: MovieLens Recommender with JAX and Equinox

Let's implement this model and train it on the classic MovieLens 100k dataset. This dataset contains 100,000 ratings from 940 users on 1680 movies.

```python
import jax
import jax.numpy as jnp
import equinox as eqx
import optax
import tensorflow_datasets as tfds
import pandas as pd
from sklearn.model_selection import train_test_split

# Use a specific key for reproducibility
key = jax.random.PRNGKey(0)

# --- 1. Load and Preprocess the MovieLens Dataset ---
def load_and_prepare_data():
    ds = tfds.load("movielens/100k-ratings", split="train")
    df = tfds.as_dataframe(ds)

    # Map raw string IDs to continuous integer indices
    user_ids = df["user_id"].astype("category").cat.codes.values
    movie_ids = df["movie_id"].astype("category").cat.codes.values
    ratings = df["user_rating"].values

    num_users = user_ids.max() + 1
    num_items = movie_ids.max() + 1

    # Create mappings to get movie titles back
    movie_id_map = df["movie_id"].astype("category").cat.categories
    movie_title_map = df["movie_title"].astype("category").cat.categories
    id_to_title = {mid: title.decode('utf-8') for mid, title in zip(
   movie_id_map, movie_title_map)}
```

```
    # Split into training and testing sets
    train_indices, test_indices = train_test_split(np.arange(len(ratings)),
    test_size=0.2, random_state=42)

    train_data = {
        "users": user_ids[train_indices],
        "items": movie_ids[train_indices],
        "ratings": ratings[train_indices]
    }
    test_data = {
        "users": user_ids[test_indices],
        "items": movie_ids[test_indices],
        "ratings": ratings[test_indices]
    }

    return train_data, test_data, num_users, num_items, id_to_title

train_data, test_data, num_users, num_items, id_to_title =
    load_and_prepare_data()

# --- 2. Define the Model using Equinox ---
class MatrixFactorization(eqx.Module):
    user_embeddings: eqx.nn.Embedding
    item_embeddings: eqx.nn.Embedding
    user_bias: eqx.nn.Embedding
    item_bias: eqx.nn.Embedding
    global_bias: jnp.ndarray

    def __init__(self, num_users, num_items, latent_dim, key):
        ukey, ikey, ubkey, ibkey = jax.random.split(key, 4)
        self.user_embeddings = eqx.nn.Embedding(num_users, latent_dim, key=ukey
    )
        self.item_embeddings = eqx.nn.Embedding(num_items, latent_dim, key=ikey
    )
        self.user_bias = eqx.nn.Embedding(num_users, 1, key=ubkey)
        self.item_bias = eqx.nn.Embedding(num_items, 1, key=ibkey)
        self.global_bias = jnp.array([jnp.mean(train_data['ratings'])])

    def __call__(self, user_idx, item_idx):
        user_vec = self.user_embeddings(user_idx)
        item_vec = self.item_embeddings(item_idx)
        u_bias = self.user_bias(user_idx).squeeze()
        i_bias = self.item_bias(item_idx).squeeze()
        # Prediction = p_u^T*q_i + b_u + b_i + global_mean
        return jnp.sum(user_vec * item_vec, axis=-1) + u_bias + i_bias + self.
    global_bias

# --- 3. Define Loss and Training Step ---
def loss_fn(model, users, items, ratings, l2_reg):
    preds = jax.vmap(model)(users, items)
    mse = jnp.mean((preds - ratings)**2)

    # Add L2 regularization
    reg_term = l2_reg * (jnp.mean(model.user_embeddings.weight**2) +
                         jnp.mean(model.item_embeddings.weight**2))

    return mse + reg_term

@eqx.filter_jit
def make_step(model, opt_state, optimizer, users, items, ratings, l2_reg):
    loss, grads = eqx.filter_value_and_grad(loss_fn)(model, users, items,
    ratings, l2_reg)
    updates, opt_state = optimizer.update(grads, opt_state)
    model = eqx.apply_updates(model, updates)
    return model, opt_state, loss

def evaluate(model, test_data):
    preds = jax.vmap(model)(test_data["users"], test_data["items"])
    # Return Root Mean Squared Error (RMSE)
    return jnp.sqrt(jnp.mean((preds - test_data["ratings"])**2))

# --- 4. The Training Loop ---
key, model_key = jax.random.split(key)
latent_dim = 50
model = MatrixFactorization(num_users, num_items, latent_dim, model_key)
optimizer = optax.adam(learning_rate=1e-3)
opt_state = optimizer.init(model)

l2_regularization = 0.01
batch_size = 128
n_epochs = 10
n_train_steps = len(train_data["ratings"]) // batch_size

print(f"Training on {num_users} users and {num_items} items...")
for epoch in range(n_epochs):
    key, epoch_key = jax.random.split(key)
    perm = jax.random.permutation(epoch_key, len(train_data["ratings"]))

    total_loss = 0
    for i in range(n_train_steps):
        indices = perm[i*batch_size : (i+1)*batch_size]
        users_batch = train_data["users"][indices]
        items_batch = train_data["items"][indices]
        ratings_batch = train_data["ratings"][indices]

        model, opt_state, loss = make_step(model, opt_state, optimizer,
                                           users_batch, items_batch,
    ratings_batch,
                                           l2_regularization)
        total_loss += loss.item()

    avg_loss = total_loss / n_train_steps
    test_rmse = evaluate(model, test_data)
    print(f"Epoch {epoch+1}: Avg Train Loss = {avg_loss:.4f}, Test RMSE = {
    test_rmse:.4f}")

# --- 5. Making Recommendations for a User ---
# Find the integer index for a user_id (e.g., user '13')
user_id_to_idx = {uid: i for i, uid in enumerate(df["user_id"].astype("category
    ").cat.categories)}
target_user_idx = user_id_to_idx[b'13']

# Predict scores for all items for this user
all_item_indices = jnp.arange(num_items)
user_indices_tiled = jnp.full_like(all_item_indices, target_user_idx)
predicted_scores = jax.vmap(model)(user_indices_tiled, all_item_indices)

# Get top 10 recommendations
top_k_indices = jnp.argsort(predicted_scores)[::-1][:10]
print(f"\nTop 10 movie recommendations for user '13':")
for item_idx in top_k_indices:
    # Map item index back to title
    movie_id = df["movie_id"].astype("category").cat.categories[item_idx]
    title = id_to_title[movie_id]
    print(f" - {title} (Predicted score: {predicted_scores[item_idx]:.2f})")
```

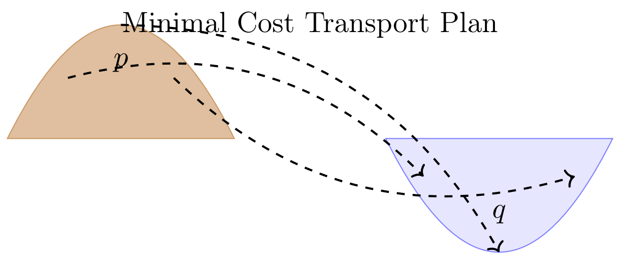
Listing 28: Matrix Factorization for MovieLens using JAX, Equinox, and Optax.

19 Optimal Transport and Entropic Regularization

19.1 Introduction: The Geometry of Comparing Distributions

So far, our optimization problems have focused on finding optimal parameters. In this lecture, we explore a different type of problem: finding the optimal way to transform one probability distribution into another. This is the core idea behind **Optimal Transport (OT)**.

The classic motivation, first posed by Gaspard Monge in the 18th century, is the "earth mover's" problem. Imagine you have a pile of earth (represented by a probability distribution p) and you want to move it to fill a hole of the same volume (represented by a distribution q). There is a cost associated with moving a single shovelful of earth from one location to another. The goal is to find a transportation plan that moves all the earth from the pile to the hole with the minimum possible total cost.



This concept provides a powerful and geometrically meaningful way to define a distance between probability distributions. Unlike measures like KL-Divergence, OT takes the underlying geometry of the sample space into account. The "distance" between two distributions depends not just on their values, but on the *cost* of moving mass between their support points.

19.2 The Classic Optimal Transport Problem

Let's formalize the discrete version of this problem. Suppose we have two discrete probability distributions (or histograms) over n bins.

- A source distribution $p \in \mathbb{R}^n_+$, where $\sum_{i=1}^n p_i = 1$.
- A target distribution $q \in \mathbb{R}^n_+$, where $\sum_{j=1}^n q_j = 1$.

We are also given a **cost matrix** $C \in \mathbb{R}^{n \times n}$, where C_{ij} is the cost of moving one unit of mass from bin i to bin j.

Our goal is to find a **transportation plan**, which is a matrix $T \in \mathbb{R}^{n \times n}$. The entry T_{ij} represents the amount of mass moved from source bin i to target bin j.

For T to be a valid plan, it must satisfy two key constraints:

1. The total mass moved *out* of source bin i must equal the mass originally in that bin: $\sum_{j=1}^n T_{ij} = p_i$ for all i.
2. The total mass moved *into* target bin j must equal the mass required by that bin: $\sum_{i=1}^n T_{ij} = q_j$ for all j.

Definition 19.1 (The Kantorovich Formulation of Optimal Transport)**.** *The discrete OT problem is to find the transport plan T that minimizes the total transportation cost. This can be* *formulated as a Linear Program (LP):*

$$\begin{aligned}
\min_{T\in\mathbb{R}^{n\times n}} \quad & \sum_{i=1}^{n}\sum_{j=1}^{n} C_{ij}T_{ij} \quad (\textit{often written as } \langle C, T\rangle) \\
\textit{subject to} \quad & \sum_{j=1}^{n} T_{ij} = p_i, \quad i = 1, \ldots, n \\
& \sum_{i=1}^{n} T_{ij} = q_j, \quad j = 1, \ldots, n \\
& T_{ij} \geq 0.
\end{aligned}$$

The optimal value of this objective is the Optimal Transport distance, also known as the Earth Mover's Distance.

While this formulation is elegant, it has significant practical drawbacks:

- **High Computational Cost:** Solving this LP is computationally intensive, typically scaling as $O(n^3 \log n)$, which is prohibitive for large n.
- **Sparsity and Non-differentiability:** The optimal solution T^* lies at a vertex of the feasible polytope and is therefore very sparse. The OT distance, as a function of p and q, is not differentiable, making it difficult to use as a loss function in gradient-based machine learning models.

19.3 Entropic Regularization

To overcome these limitations, a modern approach introduces a regularization term to the OT objective. The idea is to add a penalty that encourages "smoother" or more "spread-out" transportation plans. The perfect tool for this is **entropy**.

The entropy of a transport plan T (viewed as a joint probability distribution) is:

$$H(T) = -\sum_{i=1}^{n}\sum_{j=1}^{n} T_{ij}\log(T_{ij}).$$

Maximizing entropy favors distributions that are as uniform as possible. By adding this term to our objective, we can control the "blurriness" of the transport plan.

Definition 19.2 (Entropically Regularized Optimal Transport)**.** *The entropically regularized OT problem is:*

$$\min_{T} \quad \langle C, T\rangle - \epsilon H(T) = \sum_{i=1}^{n}\sum_{j=1}^{n} C_{ij}T_{ij} + \epsilon \sum_{i=1}^{n}\sum_{j=1}^{n} T_{ij}\log(T_{ij}).$$

The problem is still subject to the same constraints on the rows and columns of T. The parameter $\epsilon > 0$ controls the strength of the regularization.

- *As $\epsilon \to 0$, the solution converges to the classic, sparse OT solution.*
- *As $\epsilon \to \infty$, the solution ignores the cost matrix and becomes the fully independent plan $T = pq^T$.*

The new objective is strictly convex, meaning it has a unique, dense solution. Most importantly, it can be solved with a remarkably simple and fast algorithm.

19.4 Sinkhorn's Algorithm

The breakthrough for regularized OT is the realization that the optimal transport plan T^* has a specific mathematical form:

$$T^*_{ij} = u_i K_{ij} v_j, \quad \text{where } K_{ij} = e^{-C_{ij}/\epsilon}.$$

Here, $u \in \mathbb{R}^n$ and $v \in \mathbb{R}^n$ are scaling vectors (dual variables) that we need to find. The matrix K is known as the Gibbs kernel. In matrix form, $T^* = \text{diag}(u) K \text{diag}(v)$.

We don't know u and v, but we know that T^* must satisfy the row and column sum constraints. **Sinkhorn's algorithm** is a simple iterative procedure that finds u and v by alternately enforcing these constraints.

Definition 19.3 (Sinkhorn's Algorithm)**.** *1. Compute the Gibbs kernel:* $K = e^{-C/\epsilon}$.

2. Initialize $v = \mathbf{1}$ *(a vector of ones).*

3. Iterate until convergence:

   (a) Update u *to enforce the row sum constraint:* $u \leftarrow p \,/\, (Kv)$. *(Division is element-wise).*

   (b) Update v *to enforce the column sum constraint:* $v \leftarrow q \,/\, (K^T u)$.

4. Once converged, the optimal transport plan is $T = \text{diag}(u) K \text{diag}(v)$.

Each step of this algorithm is a matrix-vector multiplication, which is incredibly fast and perfectly suited for parallel hardware like GPUs. This has made OT a practical tool for modern machine learning, with a computational complexity closer to $O(n^2)$.

19.5 Application: The Word Mover's Distance

A powerful application of OT is in measuring the similarity between text documents. A simple "bag-of-words" model, which represents a document as a histogram of its word counts, fails to capture semantic meaning. For example, the documents "The king rules the nation" and "The queen governs the country" have no words in common (except "the"), but are semantically very similar.

The **Word Mover's Distance (WMD)** uses OT to solve this problem.

1. **Represent Words as Vectors:** First, we use a pre-trained model like Word2Vec or GloVe to map every word in our vocabulary into a high-dimensional vector space (e.g., 300 dimensions). In this space, the Euclidean distance between word vectors captures semantic similarity. For instance, the distance between the vectors for "king" and "queen" will be small, while the distance between "king" and "apple" will be large.

2. **Represent Documents as Distributions:** We represent each document as a normalized histogram (a probability distribution) over its words. This is our p or q.

3. **Define the Cost Matrix:** The cost C_{ij} of "transporting" word i to word j is defined as the Euclidean distance between their word vectors: $C_{ij} = ||\text{vec}(w_i) - \text{vec}(w_j)||_2$.

4. **Compute the OT Distance:** The WMD is the optimal transport cost between the two document distributions, using this semantically-aware cost matrix. It measures the minimum "work" required to transform one document into the other, where the work is cheap for semantically similar words and expensive for dissimilar words.

This allows us to find a meaningful distance between documents even if they don't share any vocabulary.

```python
import jax
import jax.numpy as jnp
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Use a specific key for reproducibility
key = jax.random.PRNGKey(0)

# --- 1. Setup: Documents, Vocabulary, and Word Embeddings ---
# Define a small vocabulary and some documents
vocab = ['king', 'queen', 'prince', 'royalty', 'apple', 'fruit', 'banana', '
    food']
doc1 = "The king and queen are royalty."
doc2 = "The prince is royalty."
doc3 = "Apple is a fruit, a type of food."

# Simulate pre-trained word embeddings. We manually make them semantically
    clustered.
# In a real application, these would come from a model like Word2Vec.
embedding_dim = 4
key, subkey = jax.random.split(key)
embeddings = jax.random.normal(subkey, (len(vocab), embedding_dim))
# Cluster "royal" words
embeddings = embeddings.at[0:4, 0:2].set(embeddings[0:4, 0:2] + 2.0)
# Cluster "fruit" words
embeddings = embeddings.at[4:8, 0:2].set(embeddings[4:8, 0:2] - 2.0)

# --- 2. Preprocessing: Convert docs to probability distributions ---
def doc_to_dist(doc_text, vocabulary):
    counts = np.array([doc_text.lower().count(word) for word in vocabulary])
    # Normalize to a probability distribution (add epsilon for stability)
    return (counts + 1e-9) / (counts.sum() + 1e-9 * len(vocabulary))

p = doc_to_dist(doc1, vocab)
q_similar = doc_to_dist(doc2, vocab)
q_dissimilar = doc_to_dist(doc3, vocab)

# --- 3. Compute the semantic cost matrix ---
# C_ij = Euclidean distance between embedding for word i and word j
def compute_cost_matrix(embeddings):
    # Expand dims to enable broadcasting for pairwise distance calculation
    diff = embeddings[:, None, :] - embeddings[None, :, :]
    return jnp.sqrt(jnp.sum(diff**2, axis=-1))

C = compute_cost_matrix(embeddings)

# --- 4. Implement Sinkhorn's Algorithm ---
@jax.jit
def sinkhorn(cost_matrix, p, q, epsilon, num_iters=100):
    K = jnp.exp(-cost_matrix / epsilon)
    v = jnp.ones_like(q)
    for _ in range(num_iters):
        u = p / (K @ v)
        v = q / (K.T @ u)

    # Compute the final transport plan and OT cost
    transport_plan = u[:, None] * K * v[None, :]
    ot_cost = jnp.sum(transport_plan * cost_matrix)
    return ot_cost, transport_plan

# --- 5. Calculate and Compare Word Mover's Distances ---
epsilon = 0.1 # Regularization strength
```

```
# Case 1: Compare two semantically similar documents
wmd_similar, T_similar = sinkhorn(C, p, q_similar, epsilon)
print(f"WMD between '{doc1}' and '{doc2}': {wmd_similar:.4f}")

# Case 2: Compare two semantically dissimilar documents
wmd_dissimilar, T_dissimilar = sinkhorn(C, p, q_dissimilar, epsilon)
print(f"WMD between '{doc1}' and '{doc3}': {wmd_dissimilar:.4f}")

# --- 6. Visualize the Transport Plan ---
fig, axes = plt.subplots(1, 2, figsize=(16, 6))

# Plot for similar documents
sns.heatmap(T_similar, xticklabels=vocab, yticklabels=vocab, annot=False, cmap=
    "Blues", ax=axes[0])
axes[0].set_title("Transport Plan (Similar Docs)")
axes[0].set_xlabel("Words in Doc 2")
axes[0].set_ylabel("Words in Doc 1")

# Plot for dissimilar documents
sns.heatmap(T_dissimilar, xticklabels=vocab, yticklabels=vocab, annot=False,
    cmap="Reds", ax=axes[1])
axes[1].set_title("Transport Plan (Dissimilar Docs)")
axes[1].set_xlabel("Words in Doc 3")
axes[1].set_ylabel("Words in Doc 1")

plt.tight_layout()
plt.show()

print("\nNotice how the transport plan for similar docs keeps mass within the '
    royal' cluster,")
print("while the plan for dissimilar docs is forced to move mass between the '
    royal' and 'fruit' clusters.")
```

Listing 29: Word Mover's Distance with Entropic Regularization using JAX.

20 Sampling as Optimization over Probability Distributions

In our last lecture, we explored Optimal Transport (OT), a powerful framework for measuring the "distance" between two probability distributions. We saw that the Wasserstein distance gives us a geometrically meaningful way to quantify how much "work" it takes to transform one distribution into another. This naturally leads to a question: if we can define a distance between distributions, can we think of the collection of all possible distributions as a single, vast geometric space? And if so, can we perform calculus and optimization on this space?

The answer is a resounding yes, and it opens up an entirely new perspective on a fundamental problem in statistics: **sampling**. To build intuition, let's consider a practical example where this viewpoint becomes essential.

Example 20.1 (Finding the Best Approximation to a Posterior)**.** *Let's revisit the biased coin problem from an earlier lecture, but from a fully Bayesian perspective. We observe $N_H = 60$ heads and $N_T = 40$ tails.*

- ***The MLE Approach (Point Estimate):*** *Our previous optimization found a single, optimal point estimate for the coin's bias: $\hat{\theta}_{MLE} = \frac{60}{100} = 0.6$. This is our "best guess," but it tells us nothing about our uncertainty.*
- ***The Bayesian Approach (Full Distribution):*** *A Bayesian statistician would seek the entire* ***posterior distribution****, $\pi(\theta|data)$. With a uniform prior, this posterior is a Beta distribution: $\pi(\theta) = Beta(\theta; 61, 41)$. This distribution is our target. It not only tells us the most likely value (its peak is at 0.6) but also quantifies our uncertainty through its width.*

Now, imagine we are working with a system that can only represent distributions using a specific, simpler family—for example, it can only store Gaussian distributions. Our true posterior π is a Beta distribution, but we need to approximate it with a Gaussian, $q(\theta; \mu, \sigma^2)$. How do we find the best Gaussian approximation?

This is no longer a search for a single point, but a search for an optimal distribution within a constrained family. We need an objective function to measure the "dissimilarity" between our approximation q and our target π. This is precisely the role of the ***Kullback-Leibler (KL) divergence****. Our problem becomes:*

$$\min_{\mu,\sigma^2} \quad KL(q(\theta; \mu, \sigma^2)||\pi(\theta)).$$

By finding the mean μ^* and variance $(\sigma^2)^*$ that minimize this KL divergence, we find the Gaussian distribution that is "closest" to the true Beta posterior. This is the core idea of Variational Inference and motivates our study of optimization over distributions.

The example above highlights the goal: we want to treat sampling and distributional approximation as optimization problems. This chapter will introduce the modern viewpoint that sampling from a complex distribution can be elegantly reframed as minimizing an objective function over the space of all probability distributions. We will see that this space, endowed with the Wasserstein distance, has a rich geometry that allows us to define concepts like "gradient flow." Most remarkably, we will discover that classic sampling algorithms like Langevin Monte Carlo are, in fact, forms of gradient descent on this incredible landscape.

20.1 The Goal: Framing Sampling as a Minimization Problem

Let's return to the core problem of Bayesian inference. We want to find the posterior distribution of some parameters θ given our data D:

$$\pi(\theta) := p(\theta|D) = \frac{p(D|\theta)p(\theta)}{p(D)} = \frac{p(D|\theta)p(\theta)}{Z}.$$

As we've seen, the denominator $Z = \int p(D|\theta)p(\theta)d\theta$, known as the evidence or normalization constant, is usually an intractable high-dimensional integral. This prevents us from evaluating the posterior density directly.

The goal of sampling is to generate a set of representative points (particles) $\{\theta_1, \ldots, \theta_M\}$ that are distributed according to π. If we can do this, we can approximate any expectation we need, for example, for Bayesian model averaging.

So, how can we turn this into an optimization problem?

1. **The Objective Function:** We need a way to measure how "close" an arbitrary distribution μ is to our target posterior π. The Kullback-Leibler (KL) divergence is the perfect tool for this. We define our objective functional $\mathcal{F}$ as:

$$\mathcal{F}(\mu) = \mathrm{KL}(\mu||\pi).$$

   We know that $\mathrm{KL}(\mu||\pi) \geq 0$ and is only zero when $\mu = \pi$. Therefore, the distribution that minimizes this objective is precisely the posterior distribution we are looking for.

2. **The Optimization Problem:** Our sampling problem is now equivalent to:

$$\min_{\mu \in \mathcal{P}(\mathbb{R}^d)} \mathcal{F}(\mu) = \min_{\mu \in \mathcal{P}(\mathbb{R}^d)} \mathrm{KL}(\mu||\pi).$$

   Here, $\mathcal{P}(\mathbb{R}^d)$ represents the space of all probability distributions over the parameter space.

3. **Handling the Intractable Constant:** A crucial advantage of using the KL divergence is that it doesn't require us to know the normalization constant Z. Recall that we can write our target as $\pi(\theta) = \frac{1}{Z}e^{-V(\theta)}$, where $V(\theta)$ is a "potential energy" function that we can compute (e.g., in Bayesian regression, $V(\theta)$ includes the sum of squared errors and the prior). The KL divergence expands to:

$$\begin{aligned}
\mathrm{KL}(\mu||\pi) &= \int \log\left(\frac{\mu(\theta)}{\pi(\theta)}\right) d\mu(\theta) \\
&= \int \log\left(\frac{\mu(\theta)}{e^{-V(\theta)}/Z}\right) d\mu(\theta) \\
&= \underbrace{\int V(\theta)d\mu(\theta)}_{\text{Potential Energy}} + \underbrace{\int \log(\mu(\theta))d\mu(\theta)}_{\text{Negative Entropy}} + \log(Z).
\end{aligned}$$

   Since we are minimizing with respect to μ, the constant term $\log(Z)$ has no effect on the solution. We are left with an objective containing only computable terms!

We now have an optimization problem, but it's unlike any we've seen before. The variable is not a vector, but an entire function—a probability distribution. To solve this, we need a notion of "gradient descent" for distributions.

20.2 The Analogy: From Euclidean to Wasserstein Gradient Flow

To understand optimization on the space of distributions, we can build a powerful analogy with the optimization we already know.

20.2.1 Optimization in Euclidean Space ($\mathbb{R}^d$)

- **The Problem:** Minimize a function $V(x)$ for $x \in \mathbb{R}^d$.
- **The Geometry:** The space is standard Euclidean space. The distance between points is $||x - y||_2$.

- **The Path of Steepest Descent:** Imagine a ball placed on the surface defined by $V(x)$. If it rolls perfectly downhill, its path $x(t)$ over time is described by an ordinary differential equation (ODE) called the **Euclidean Gradient Flow**:
$$\frac{dx(t)}{dt} = -\nabla V(x(t)).$$
This is the continuous-time limit of gradient descent.

- **The Algorithm:** We can't follow the continuous path exactly, so we approximate it with discrete steps. The simplest approximation is the **Forward Euler** method, which gives us our familiar Gradient Descent algorithm:
$$\frac{x_{k+1} - x_k}{\gamma} \approx -\nabla V(x_k) \implies x_{k+1} = x_k - \gamma \nabla V(x_k).$$

20.2.2 Optimization in Probability Space ($\mathcal{P}_2(\mathbb{R}^d)$)

- **The Problem:** Minimize a functional $\mathcal{F}(\mu) = \mathrm{KL}(\mu||\pi)$ for $\mu \in \mathcal{P}_2(\mathbb{R}^d)$, the space of distributions with finite second moments.

- **The Geometry:** The space is the space of probability distributions. The distance is the **Wasserstein-2 distance**, $W_2(\mu, \nu)$, from Optimal Transport. This endows the space with a rich "Riemannian-like" structure.

- **The Path of Steepest Descent:** Now, imagine not a ball, but a pile of sand representing our initial distribution μ_0. If this sand flows "downhill" on the landscape defined by $\mathcal{F}$ to fill the hole at the minimum, π, this continuous evolution of the distribution, $\mu(t)$, is called the **Wasserstein Gradient Flow (WGF)**. It is described by a partial differential equation (PDE):
$$\frac{\partial \mu(t)}{\partial t} = \nabla \cdot (\mu(t) \nabla_{W_2} \mathcal{F}(\mu(t))).$$
This equation looks complex, but its meaning is analogous to the Euclidean case: the distribution $\mu(t)$ evolves in the direction of steepest descent in the Wasserstein geometry.

- **The Algorithm:** Just as we discretized the Euclidean gradient flow to get gradient descent, we can discretize the Wasserstein gradient flow to get powerful sampling algorithms. The rest of this chapter explores this idea.

20.3 Langevin Monte Carlo as a Wasserstein Gradient Flow

The first and most fundamental connection comes from a process called Langevin diffusion.

Definition 20.1 (Langevin Dynamics)**.** ***Langevin dynamics*** *describes the motion of a particle in a potential field $V(x)$ subject to random collisions. It is described by the following Stochastic Differential Equation (SDE):*
$$dx_t = -\nabla V(x_t)dt + \sqrt{2}db_t.$$
This equation has two parts:

- *A* ***drift term*** *$-\nabla V(x_t)dt$: This pushes the particle "downhill" along the gradient of the potential, just like in gradient flow.*

- *A* ***diffusion term*** *$\sqrt{2}db_t$: This adds random noise, modeled by Brownian motion, that kicks the particle around.*

The remarkable property of this process is that its stationary distribution is exactly $\pi(x) \propto e^{-V(x)}$. This means that if we simulate this SDE for a long time, the particle x_t will be a sample from our target posterior!

We have the following connection between Langevin dynamics and Wasserstein Gradient Flow:

> The distribution of particles following Langevin dynamics, $\mu_t = \mathrm{Law}(x_t)$, is precisely the Wasserstein Gradient Flow of the KL divergence functional, $\mathcal{F}(\mu) = \mathrm{KL}(\mu||\pi)$.

This means that the random noise of the SDE provides exactly the right "push" to make a population of particles flow along the path of steepest descent in the Wasserstein space.

To turn this into a practical algorithm, we simply discretize the Langevin SDE using the Euler-Maruyama method. This gives us the **Langevin Monte Carlo (LMC)** algorithm.

Definition 20.2 (Langevin Monte Carlo (LMC))**.** *Given a starting point x_0, the LMC algorithm generates a sequence of samples via the update rule:*

$$x_{m+1} = x_m - \gamma \nabla V(x_m) + \sqrt{2\gamma}\eta_m,$$

where γ is the step size and $\eta_m \sim \mathcal{N}(0, I)$ is a standard Gaussian random vector.

We now have a reinterpretation of LMC as a form of stochastic gradient descent on the space of probability distributions, aiming to minimize the KL divergence to the target posterior.

20.4 Stein Variational Gradient Descent (SVGD)

LMC evolves a single particle (or many independent particles). A more recent idea, **Stein Variational Gradient Descent (SVGD)**, asks: can we evolve a set of interacting particles $\{x_i\}_{i=1}^N$ that "cooperate" to match the target distribution?

SVGD proposes a deterministic update that moves each particle x_i based on information from all other particles in the ensemble.

Definition 20.3 (Stein Variational Gradient Descent (SVGD))**.** *Given a set of initial particles $\{x_m^i\}_{i=1}^N$, the SVGD update for each particle is:*

$$x_{m+1}^i = x_m^i - \frac{\gamma}{N}\sum_{j=1}^N \left(k(x_m^i, x_m^j)\nabla V(x_m^j) + \nabla_{x_m^j} k(x_m^i, x_m^j) \right).$$

Here, $k(x, y)$ is a positive definite kernel, such as the RBF kernel $k(x, y) = \exp(-||x-y||^2/(2h^2))$.

This update has the following intuition:

- **Gradient Term ($k(\cdot)\nabla V(\cdot)$):** Each particle is pushed downhill according to a weighted average of the gradients at all other particles. The kernel k ensures that nearby particles have a stronger influence. This is the "descent" part.
- **Repulsive Term ($\nabla k(\cdot)$):** The second term, the gradient of the kernel itself, acts as a repulsive force. It pushes particles away from each other, preventing them from all collapsing into a single point at the mode of π. This forces the ensemble of particles to spread out and cover the entire distribution.

Like LMC, SVGD can also be interpreted as a gradient flow of the KL divergence, but in a different geometry induced by the kernel. It provides a powerful, deterministic alternative to MCMC methods for generating samples from a target distribution.

20.5 Example: Sampling from a Ring Distribution

We use LMC and SVGD to obtain a sample from a ring distribution:

- **The Target Distribution:** We define the distribution via its potential energy function, $V(x)$. We want the potential to be low when a point's distance from the origin is close to a specific radius, R. A simple choice is:

$$V(x) = \frac{(\|x\|_2 - R)^2}{2\sigma^2}$$

  The target density is $\pi(x) \propto e^{-V(x)}$. This creates a high-probability "ridge" in the shape of a ring with radius R and thickness controlled by σ.

Below is an implementation of LMC and SVGD in JAX.

```python
import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt

# Use a specific key for reproducibility
key = jax.random.PRNGKey(42)

# --- 1. Define the Target Distribution (a Ring) ---
# The potential V(x) is low when ||x|| is close to a target radius.
RING_RADIUS = 6.0
RING_STD = 0.3  # Controls the thickness of the ring

def potential_V(x):
    # Add a small epsilon for numerical stability of the gradient at the origin
    norm_x = jnp.sqrt(jnp.sum(x**2) + 1e-6)
    return (norm_x - RING_RADIUS)**2 / (2 * RING_STD**2)

def log_pi(x):
    return -potential_V(x)

# The gradient will be computed automatically by JAX
grad_V = jax.grad(potential_V)

# --- 2. Langevin Monte Carlo (LMC) Implementation ---
@jax.jit
def lmc_step(x, key, step_size):
    noise = jax.random.normal(key, shape=x.shape)
    grad = grad_V(x)
    x_new = x - step_size * grad + jnp.sqrt(2 * step_size) * noise
    return x_new

# --- 3. Stein Variational Gradient Descent (SVGD) Implementation ---
# RBF Kernel: k(x, y) = exp(-||x-y||^2 / (2h^2))
def rbf_kernel(x, y, h=1.0):
    return jnp.exp(-jnp.sum((x - y)**2) / (2 * h**2))

# Gradient of the kernel with respect to its first argument
grad_k_x = jax.grad(rbf_kernel, argnums=1)

@jax.jit
def svgd_step(particles, step_size):
    # Use vmap for efficient computation over all pairs of particles
    K_matrix = jax.vmap(lambda x: jax.vmap(lambda y: rbf_kernel(x, y))(
    particles))(particles)
    grad_K_matrix = jax.vmap(lambda x: jax.vmap(lambda y: grad_k_x(x, y))(
    particles))(particles)
    grads_V = jax.vmap(grad_V)(particles)

    # The two components of the SVGD update
    gradient_term = K_matrix @ grads_V
    repulsive_term = jnp.sum(grad_K_matrix, axis=0)

    phi = (gradient_term + repulsive_term) / len(particles)
    particles_new = particles - step_size * phi
    return particles_new

# --- 4. Run the simulations ---
num_particles = 250
num_iterations = 1000
key, pkey = jax.random.split(key)
# Start particles from a Gaussian blob in the center
initial_particles = jax.random.normal(pkey, (num_particles, 2)) * 2.0

# Run LMC simulation
particles_lmc = initial_particles
lmc_keys = jax.random.split(key, num_iterations)
for i in range(num_iterations):
    particles_lmc = lmc_step(particles_lmc, lmc_keys[i], step_size=0.05)

# Run SVGD simulation
particles_svgd = initial_particles
for i in range(num_iterations):
    particles_svgd = svgd_step(particles_svgd, step_size=0.3)

# --- 5. Visualize the final particle distributions ---
# Create a grid to visualize the target density
grid_range = RING_RADIUS + 3
x_grid = jnp.linspace(-grid_range, grid_range, 200)
y_grid = jnp.linspace(-grid_range, grid_range, 200)
grid_x, grid_y = jnp.meshgrid(x_grid, y_grid)
pos = jnp.dstack((grid_x, grid_y))
density = jnp.exp(jax.vmap(jax.vmap(log_pi))(pos))

fig, axes = plt.subplots(1, 2, figsize=(16, 8))

axes.contourf(grid_x, grid_y, density, levels=15, cmap='viridis')
axes.scatter(particles_lmc[:, 0], particles_lmc[:, 1], color='white', s=15,
    alpha=0.6, edgecolors='black')
axes[0].set_title(f'Langevin Monte Carlo (LMC) after {num_iterations} steps')
axes[0].set_aspect('equal', adjustable='box')
axes[0].set_xlim(-grid_range, grid_range)
axes[0].set_ylim(-grid_range, grid_range)

axes[1].contourf(grid_x, grid_y, density, levels=15, cmap='viridis')
axes[1].scatter(particles_svgd[:, 0], particles_svgd[:, 1], color='white', s
    =15, alpha=0.6, edgecolors='black')
axes[1].set_title(f'Stein Variational Gradient Descent (SVGD) after {
    num_iterations} steps')
axes[1].set_aspect('equal', adjustable='box')
axes[1].set_xlim(-grid_range, grid_range)
axes[1].set_ylim(-grid_range, grid_range)

plt.show()
```

Listing 30: Comparing LMC and SVGD for sampling from a 2D ring distribution.

20.6 Application: Bayesian Logistic Regression

Let's apply these advanced sampling methods to a classic business problem: predicting customer churn. A company has data on its customers, including their monthly spending, tenure, and number of support calls. The goal is to build a model that predicts the probability of a customer churning (canceling their service).

While a standard logistic regression would give us a single point estimate for the coefficients, a Bayesian approach will give us a full posterior distribution. This allows us to answer much richer questions, such as "How certain are we that an increase in support calls leads to higher churn?" The posterior distribution for the regression coefficients, β, provides the answer.

Our model is:

- **Likelihood:** $P(\text{churn}_i = 1|x_i, \beta) = \sigma(x_i^T \beta)$, where σ is the sigmoid function. This is a standard logistic regression model.
- **Prior:** We place a simple Gaussian prior on our coefficients, $\beta \sim \mathcal{N}(0, 10I)$, expressing a belief that the coefficients are likely centered around zero.

Our target posterior is $\pi(\beta|X, y) \propto P(y|X, \beta)P(\beta)$. The potential function we need to minimize is the negative log-posterior, $V(\beta) = -\log P(y|X, \beta) - \log P(\beta)$. We will use LMC and SVGD to draw samples from this posterior.

```python
import jax
import jax.numpy as jnp
from jax.scipy.stats import norm
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

# Use a specific key for reproducibility
key = jax.random.PRNGKey(42)

# --- 1. Generate Synthetic Customer Churn Data ---
num_samples = 1000
num_features = 3  # Features: Intercept, tenure, monthly_spend
key, subkey = jax.random.split(key)

# Define the "true" underlying relationship
true_betas = jnp.array([-2.0, 0.5, -1.5])  # Intercept, tenure_effect,
    spend_effect

X_data = jax.random.normal(subkey, (num_samples, num_features - 1))
# Add a column of ones for the intercept term
X_data_intercept = jnp.hstack([jnp.ones((num_samples, 1)), X_data])

logits = X_data_intercept @ true_betas
probabilities = 1 / (1 + jnp.exp(-logits))
y_data = jax.random.bernoulli(key, probabilities)  # True churn labels (0 or 1)

# --- 2. Define the Target Posterior Distribution ---
# V(beta) = -log(likelihood) - log(prior)
def log_likelihood_fn(beta, X, y):
    logits = X @ beta
    return jnp.sum(y * logits - jnp.log(1 + jnp.exp(logits)))

def log_prior_fn(beta):
    return jnp.sum(norm.logpdf(beta, loc=0., scale=10.))

def potential_V(beta):
    return -log_likelihood_fn(beta, X_data_intercept, y_data) - log_prior_fn(
    beta)
```

```
grad_V = jax.grad(potential_V)

# --- 3. LMC and SVGD Implementations (reusable logic) ---
@jax.jit
def lmc_step(beta, key, step_size):
    noise = jax.random.normal(key, shape=beta.shape)
    grad = grad_V(beta)
    return beta - step_size * grad + jnp.sqrt(2 * step_size) * noise

def rbf_kernel(x, y, h=0.2):
    return jnp.exp(-jnp.sum((x - y)**2) / (2 * h**2))

grad_k_x = jax.grad(rbf_kernel, argnums=1)

@jax.jit
def svgd_step(particles, step_size):
    K_matrix = jax.vmap(lambda x: jax.vmap(lambda y: rbf_kernel(x, y))(
   particles))(particles)
    grad_K_matrix = jax.vmap(lambda x: jax.vmap(lambda y: grad_k_x(x, y))(
   particles))(particles)
    grads_V = jax.vmap(grad_V)(particles)
    phi = (K_matrix @ grads_V + jnp.sum(grad_K_matrix, axis=0)) / len(particles
   )
    return particles - step_size * phi

# --- 4. Run the Simulations ---
num_particles = 300
num_iterations = 1000
key, pkey = jax.random.split(key)
initial_particles = jax.random.normal(pkey, (num_particles, num_features))

# Run LMC (vmapped for all particles)
particles_lmc = initial_particles
lmc_keys = jax.random.split(key, num_iterations * num_particles)
for i in range(num_iterations):
    step_keys = lmc_keys[i*num_particles:(i+1)*num_particles]
    particles_lmc = jax.vmap(lmc_step, in_axes=(0, 0, None))(particles_lmc,
   step_keys, 5e-5)

# Run SVGD
particles_svgd = initial_particles
for i in range(num_iterations):
    particles_svgd = svgd_step(particles_svgd, step_size=0.1)

# --- 5. Visualize the Posterior Distributions ---
def plot_posteriors(particles, title, true_params):
    df = pd.DataFrame(particles, columns=['Intercept', 'Tenure', 'Spend'])
    g = sns.pairplot(df, kind='kde', diag_kind='hist')
    g.fig.suptitle(title, y=1.02)
    # Overlay true parameters
    for i, param in enumerate(true_params):
        g.axes[i, i].axvline(param, color='red', linestyle='--')
        for j in range(len(true_params)):
            if i != j:
                g.axes[j, i].axvline(param, color='red', linestyle='--')
                g.axes[i, j].axhline(param, color='red', linestyle='--')
    plt.show()

print("Visualizing posterior samples from LMC...")
plot_posteriors(particles_lmc, 'Posterior Samples from LMC', true_betas)

print("Visualizing posterior samples from SVGD...")
```

```
plot_posteriors(particles_svgd, 'Posterior Samples from SVGD', true_betas)
```
Listing 31: Comparing LMC and SVGD for Bayesian Logistic Regression.

21 Generative Modeling with Score Matching

In our last chapter, we reframed the problem of sampling from a known target distribution $\pi(x) \propto e^{-V(x)}$ as an optimization problem on the space of probability distributions. We discovered that Langevin Monte Carlo (LMC), an algorithm that evolves particles according to a stochastic differential equation, can be seen as a form of gradient descent aiming to minimize the KL divergence to π.

The LMC update rule is:

$$x_{m+1} = x_m - \gamma \nabla V(x_m) + \sqrt{2\gamma}\eta_m.$$

The crucial term here is the drift term, $-\nabla V(x_m)$. Let's look at this more closely. Since $\pi(x) \propto e^{-V(x)}$, we have $\log \pi(x) = -V(x) - \log Z$. Taking the gradient gives:

$$\nabla_x \log \pi(x) = -\nabla_x V(x).$$

This vector field, $\nabla_x \log \pi(x)$, is known as the **score function** of the distribution π. It has a nice geometric interpretation: at any point x, the score vector points in the direction in which the log-probability density is increasing most steeply.

So, the LMC update rule is actually using the score function:

$$x_{m+1} = x_m + \gamma \nabla_x \log \pi(x_m) + \sqrt{2\gamma}\eta_m.$$

LMC works by iteratively nudging particles in the direction of the score (towards higher probability regions), while the noise term provides exploration. This reveals a deep truth: **if you know the score function of a distribution, you can generate samples from it.**

This begs a powerful question: What if we don't have an analytical form for π or $V(x)$, but we are given a dataset of samples $\{\mathbf{x}_n\}_{n=1}^N$? Can we *learn* the score function from this data and then use it to generate new, unseen samples? This is the core idea behind **score-based generative modeling**.

21.1 Denoising Score Matching: A Practical Objective

Our goal is to train a parameterized model, a neural network $s_\theta(\mathbf{x})$, to approximate the true score of the data distribution, $\nabla_{\mathbf{x}} \log p_{\text{data}}(\mathbf{x})$. A direct minimization of the squared error (the Fisher Divergence) is possible but leads to a loss function involving the trace of the model's Jacobian. A more practical and robust approach is **Denoising Score Matching (DSM)**.

The core idea is to change the problem. We first define a process for corrupting our clean data samples $\mathbf{x} \sim p_d(\mathbf{x})$ with Gaussian noise of a fixed variance σ^2:

$$q_\sigma(\tilde{\mathbf{x}}|\mathbf{x}) = \mathcal{N}(\tilde{\mathbf{x}}|\mathbf{x}, \sigma^2 I).$$

This creates a new, noisy data distribution $q_\sigma(\tilde{\mathbf{x}}) = \int q_\sigma(\tilde{\mathbf{x}}|\mathbf{x}) p_d(\mathbf{x}) d\mathbf{x}$. Instead of learning the score of the original data, our new goal is to learn the score of this perturbed distribution. The objective we wish to minimize is the Fisher divergence between our score model $s_\theta(\tilde{\mathbf{x}})$ and the true score of the noisy distribution (Equation 1):

$$\mathcal{J}(\theta) = \frac{1}{2} \int \|s_\theta(\tilde{\mathbf{x}}) - \nabla_{\tilde{\mathbf{x}}} \log q_\sigma(\tilde{\mathbf{x}})\|^2 \; q_\sigma(\tilde{\mathbf{x}}) \; \mathrm{d}\tilde{\mathbf{x}}. \tag{1}$$

This objective is intractable because we don't know the marginal distribution $q_\sigma(\tilde{\mathbf{x}})$. However, we can approximate it with an empirical distribution formed from our N data samples:

$$q_{data}(\tilde{\mathbf{x}}) = \frac{1}{N} \sum_{n=1}^{N} \mathcal{N}(\tilde{\mathbf{x}}|\mathbf{x}_n, \sigma^2 I).$$

Vincent (2011) proved a remarkable result: minimizing the Fisher divergence with respect to $q_\sigma(\tilde{\mathbf{x}})$ is equivalent to minimizing a much simpler objective. This leads to the following loss function, which replaces the intractable expectation over q_σ with an average over the dataset, where each term is an expectation over the noise added to a single data point (Equation 2):

$$\mathcal{L}(\theta) = \frac{1}{2N}\sum_{n=1}^{N}\int \|s_\theta(\tilde{\mathbf{x}}) - \nabla_{\tilde{\mathbf{x}}}\ln\mathcal{N}(\tilde{\mathbf{x}}|\mathbf{x}_n,\sigma^2 I)\|^2\ \mathcal{N}(\tilde{\mathbf{x}}|\mathbf{x}_n,\sigma^2 I)\ \mathrm{d}\tilde{\mathbf{x}}. \tag{2}$$

This is a monumental simplification. The integral is now an expectation where the distribution is a simple Gaussian, and we can analytically compute its score. As derived in the reference:

$$\begin{aligned}\nabla_{\tilde{\mathbf{x}}}\ln\mathcal{N}(\tilde{\mathbf{x}}|\mathbf{x}_n,\sigma^2 I) &= \nabla_{\tilde{\mathbf{x}}}\left(-\frac{D}{2}\ln(2\pi\sigma^2) - \frac{1}{2\sigma^2}\|\tilde{\mathbf{x}}-\mathbf{x}_n\|^2\right)\\ &= -\frac{1}{\sigma^2}(\tilde{\mathbf{x}}-\mathbf{x}_n).\end{aligned}$$

Using the reparameterization trick, we can write the noisy sample as $\tilde{\mathbf{x}}_n = \mathbf{x}_n + \sigma\epsilon$ where $\epsilon \sim \mathcal{N}(0, I)$. Substituting this into the score gives:

$$\nabla_{\tilde{\mathbf{x}}}\ln\mathcal{N}(\tilde{\mathbf{x}}_n|\mathbf{x}_n,\sigma^2 I) = -\frac{1}{\sigma^2}(\mathbf{x}_n + \sigma\epsilon - \mathbf{x}_n) = -\frac{1}{\sigma}\epsilon.$$

Plugging this result back into our tractable objective (Eq. 2) gives the final loss function.

$$\begin{aligned}\mathcal{L}(\theta) &= \frac{1}{2N}\sum_{n=1}^{N}\int \|s_\theta(\tilde{\mathbf{x}}) + \frac{1}{\sigma}\epsilon\|^2\ \mathcal{N}(\epsilon|0,\sigma^2 I)\ \mathrm{d}\epsilon\\ &= \frac{1}{2N}\sum_{n=1}^{N}\mathbb{E}_{\mathcal{N}(\epsilon|0,I)}\left[\|s_\theta(\tilde{\mathbf{x}}) + \frac{1}{\sigma}\epsilon\|^2\right].\end{aligned}$$

In summary, for each data point $\mathbf{x}_n$, we add noise to get $\tilde{\mathbf{x}}_n$, and then train our network s_θ that predicts a scaled noise $-\frac{1}{\sigma}\epsilon$.

21.2 From Score to Samples: Langevin Dynamics Revisited

Once we have successfully trained our model $s_\theta(x) \approx \nabla \log q_\sigma(x)$, we can use it as a drop-in replacement for the true score in the Langevin Monte Carlo algorithm. This allows us to generate new samples from the distribution our model has learned.

Score-Based Generation via Langevin Dynamics:

1. Start with a set of particles $\{x_0^i\}$ drawn from a simple prior distribution (e.g., random noise).

2. For $m = 0, 1, 2, \dots, M-1$:

$$x_{m+1}^i = x_m^i + \gamma s_\theta(x_m^i) + \sqrt{2\gamma}\eta_m,$$

   where $\eta_m \sim \mathcal{N}(0, I)$ is a random Gaussian vector.

The particles start as random noise and are iteratively refined by following the learned score field (the drift term) while being perturbed by injected noise (the diffusion term). After many steps, the particles are transformed from noise into realistic samples that resemble the training data.

21.3 Example: Generating Handwritten Digits

Let's implement Denoising Score Matching on a real dataset: the `Digits` dataset, which contains low-resolution (8x8) images of handwritten digits. We will train a score network to learn the structure of this data and then use Langevin dynamics to generate new, original handwritten digits.

```python
import jax
import jax.numpy as jnp
import equinox as eqx
import optax
import matplotlib.pyplot as plt
from sklearn.datasets import load_digits

# Use a specific key for reproducibility
key = jax.random.PRNGKey(42)

# --- 1. Load and Prepare the Digits Dataset ---
def load_and_prepare_data():
    digits = load_digits()
    # The data is a set of 8x8 images, flattened to 64-dim vectors
    X = digits.data
    # Normalize pixel values from to [-1, 1]
    X_normalized = (X / 8.0) - 1.0
    return jnp.array(X_normalized)

X_data = load_and_prepare_data()
DATA_DIM = X_data.shape[1]

# --- 2. Define the Score Model (an MLP) ---
class ScoreNet(eqx.Module):
    layers: list

    def __init__(self, in_dim, key):
        keys = jax.random.split(key, 4)
        self.layers = [
            eqx.nn.Linear(in_dim, 512, key=keys[0]),
            jax.nn.silu,
            eqx.nn.Linear(512, 512, key=keys[1]),
            jax.nn.silu,
            eqx.nn.Linear(512, 512, key=keys[2]),
            jax.nn.silu,
            eqx.nn.Linear(512, in_dim, key=keys[3])
        ]

    def __call__(self, x):
        for layer in self.layers:
            x = layer(x)
        return x

# --- 3. Define the Denoising Score Matching Loss ---
NOISE_STD = 0.15
def loss_fn(model, key, x_batch):
    noise = jax.random.normal(key, shape=x_batch.shape)
    x_perturbed = x_batch + NOISE_STD * noise
    target = -noise / NOISE_STD
    predicted = jax.vmap(model)(x_perturbed)
    return jnp.mean(jnp.sum((predicted - target)**2, axis=-1))

# --- 4. The Training Step ---
@eqx.filter_jit
def make_step(model, opt_state, optimizer, key, x_batch):
    loss, grads = eqx.filter_value_and_grad(loss_fn)(model, key, x_batch)
    updates, opt_state = optimizer.update(grads, opt_state)
    model = eqx.apply_updates(model, updates)
    return model, opt_state, loss

# --- 5. The Training Loop ---
key, model_key = jax.random.split(key)
score_model = ScoreNet(in_dim=DATA_DIM, key=model_key)
optimizer = optax.adam(learning_rate=1e-3)
opt_state = optimizer.init(eqx.filter(score_model, eqx.is_array))

batch_size = 128
n_iterations = 10000
print("Training score model on Digits dataset...")
for i in range(n_iterations):
    key, step_key, batch_key = jax.random.split(key, 3)
    indices = jax.random.randint(batch_key, (batch_size,), 0, len(X_data))
    x_batch = X_data[indices]

    score_model, opt_state, loss = make_step(score_model, opt_state, optimizer,
     step_key, x_batch)

    if i % 1000 == 0:
        print(f"Iteration {i}: Loss = {loss.item():.4f}")

# --- 6. Sampling using Langevin Dynamics with the Learned Score ---
@eqx.filter_jit
def langevin_sampler_step(x, key, model, step_size):
    score = jax.vmap(model)(x)
    noise = jax.random.normal(key, shape=x.shape)
    x_new = x + step_size * score + jnp.sqrt(2 * step_size) * noise
    return x_new

# Run the sampler
key, sample_key = jax.random.split(key)
num_samples = 100 # Generate 100 new digit images
# Start from pure noise, centered around 0 with some variance
initial_samples = jax.random.normal(sample_key, (num_samples, DATA_DIM))

samples = initial_samples
sampling_steps = 5000
sampling_step_size = 1e-5
sampler_keys = jax.random.split(key, sampling_steps)

for i in range(sampling_steps):
    samples = langevin_sampler_step(samples, sampler_keys[i], score_model,
    sampling_step_size)

# --- 7. Visualize the Generated Digits ---
# Un-normalize the data from [-1, 1] back to for plotting
generated_images = (jnp.clip(samples, -1.0, 1.0) + 1.0) / 2.0

fig, axes = plt.subplots(10, 10, figsize=(10, 10),
                         subplot_kw={'xticks':[], 'yticks':[]},
                         gridspec_kw=dict(hspace=0.1, wspace=0.1))

print("\nGenerating new handwritten digits from noise...")
for i, ax in enumerate(axes.flat):
    ax.imshow(generated_images[i].reshape(8, 8), cmap='gray')

plt.suptitle("Digits Generated via Learned Score and Langevin Dynamics",
    fontsize=16)
```

```
plt.show()
```

Listing 32: Score-based generative modeling for the Digits dataset in JAX/Equinox.