Numerical Optimization

Preliminaries

Consider the following optimization problem

$$\min_{x \in \mathbb{R}^n} f(x)$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is a function. The function f is called the objective function. The set $\mathbb{R}^n$ is called the feasible set. The vector x is called the decision variable. The vector x^* is called the optimal solution. The scalar $f(x^*)$ is called the optimal value. The optimization problem is called a minimization problem. A maximization problem is defined similarly.

Convex optimization

A convex optimization problem is an optimization problem where the objective function is convex, and the feasible set is convex. A convex function is a function where the line segment between any two points on the graph of the function lies above the graph. A convex set is a set where the line segment between any two points in the set lies in the set. Convex optimization problems are easy to solve because they have a unique global minimum.

Solving convex optimization problems

Convex optimization problems can be solved using the gradient descent algorithm. The gradient descent algorithm is an iterative algorithm that starts at a random point and moves in the direction of the negative gradient until it reaches a local minimum. The gradient descent algorithm is guaranteed to converge to a local minimum if the objective function is convex and the feasible set is convex. The gradient descent algorithm is guaranteed to converge to the global minimum if the objective function is convex and the feasible set is convex and compact.

Gradient descent

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

where x_k is the current point, x_{k+1} is the next point, α_k is the step size, and $\nabla f(x_k)$ is the gradient of the objective function at the current point. The step size is a hyperparameter that controls how fast the algorithm moves towards the minimum. The step size can be constant or variable. The step size is constant if it does not change during the algorithm. The step size is variable if it changes during the algorithm. The step size can be chosen using more advanced methods such as line search or backtracking line search.

Quick gradient descent example in Python

```
import numpy as np

f = lambda x: x**2
df = lambda x: 2*x

x = 10
alpha = 0.1
tol = 1e-6
max_iter = 1000

while True:
    x_new = x - alpha*df(x)
    if np.abs(x_new - x) < tol:
        break
    x = x_new
```

Optimization in Python: SciPy

$$\min_{x\in\mathbb{R}^2} f(x) = x_1^2 + x_2^2$$

```python
import numpy as np
from scipy.optimize import minimize
f = lambda x: x[0]**2 + x[1]**2
x0 = np.array([10, 10])
res = minimize(f, x0)
print(res)
```

```
      fun: 9.714371410949269e-13
 hess_inv: array([[ 0.75000002, -0.24999998],
       [-0.24999998,  0.75000002]])
      jac: array([-1.37896909e-06, -1.37896909e-06])
  message: 'Optimization terminated successfully.'
     nfev: 12
      nit: 2
     njev: 4
   status: 0
  success: True
        x: array([-6.96935126e-07, -6.96935126e-07])
```

Performance, how fast is SciPy?

```python
import numpy as np
def my_minimize(f, x0, alpha = 0.1, tol = 1e-6, max_iter = 1000):
    df = lambda x: (f(x + tol) - f(x - tol))/(2*tol)
    x = x0
    for _ in range(max_iter):
        x_new = x - alpha*df(x)
        if np.max(np.abs(x_new - x)) < tol:
            break
        x = x_new
    return x
```

Profiling, knowing the derivative accelerates convergence

```python
import numpy as np
from scipy.optimize import minimize
f = lambda x: x**2
x0 = 10
%timeit minimize(f, x0, tol=1e-6, options={'maxiter': 1000})
%timeit my_minimize(f, x0, tol=1e-6, alpha=0.1, max_iter=1000)
```

```
588 µs ± 1.32 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
358 µs ± 1.08 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Scipy when the Jacobian is known

```python
import numpy as np
from scipy.optimize import minimize
f = lambda x: x[0]**2 + x[1]**2
df = lambda x: np.array([2*x[0], 2*x[1]])
x0 = np.array([10, 10])
res = minimize(f, x0, jac=df)
print(res)
```

```
 fun: 1.5777218104420236e-30
 hess_inv: array([[ 0.75, -0.25],
       [-0.25,  0.75]])
      jac: array([-1.77635684e-15, -1.77635684e-15])
  message: 'Optimization terminated successfully.'
     nfev: 4
      nit: 2
     njev: 4
   status: 0
  success: True
        x: array([-8.8817842e-16, -8.8817842e-16])
```

Numerical Differentiation

Computing the derivative of a function numerically is useful when the derivative is not known analytically. The derivative of a function $f : \mathbb{R}^n \to \mathbb{R}$ at a point $x \in \mathbb{R}^n$ is defined as

$$\nabla f(x) = \left[\frac{\partial f}{\partial x_1}(x), \frac{\partial f}{\partial x_2}(x), \ldots, \frac{\partial f}{\partial x_n}(x)\right]$$

where $\frac{\partial f}{\partial x_i}(x)$ is the partial derivative of f with respect to x_i at x. The partial derivative of f with respect to x_i at x is defined as

$$\frac{\partial f}{\partial x_i}(x) = \lim_{h \to 0} \frac{f(x + he_i) - f(x)}{h}$$

where e_i is the i-th standard basis vector. The partial derivative of f with respect to x_i at x is the slope of the tangent line of f at x in the direction of e_i.

Numerical Differentiation (2)

The partial derivative of f with respect to x_i at x can be approximated using the forward difference formula

$$\frac{\partial f}{\partial x_i}(x) \approx \frac{f(x + he_i) - f(x)}{h}$$

where h is a small number. It can also be approximated using the backward difference formula

$$\frac{\partial f}{\partial x_i}(x) \approx \frac{f(x) - f(x - he_i)}{h}$$

It can also be approximated using the central difference formula

$$\frac{\partial f}{\partial x_i}(x) \approx \frac{f(x + he_i) - f(x - he_i)}{2h}$$

where h is a small number. The central difference formula is more accurate than the forward difference formula and the backward difference formula.

Numerical Differentiation, higher order derivatives

The second partial derivative of f with respect to x_i and x_j at x is defined as

$$\frac{\partial^2 f}{\partial x_i \partial x_j}(x) = \lim_{h \to 0} \frac{\frac{\partial f}{\partial x_i}(x + he_j) - \frac{\partial f}{\partial x_i}(x)}{h}$$

where $\frac{\partial f}{\partial x_i}(x + he_j)$ is the partial derivative of f with respect to x_i at $x + he_j$. The second partial derivative of f with respect to x_i and x_j at x can be approximated using the central difference formula

Numerical Differentiation, higher order derivatives (2)

Replacing the partial derivative of f with respect to x_i at x with the central difference formula gives

$$\frac{\partial^2 f}{\partial x_i \partial x_j}(x) \approx \frac{\frac{f(x+he_i+he_j)-f(x+he_j)}{h} - \frac{f(x+he_i)-f(x)}{h}}{h}$$

arranging terms

$$\frac{\partial^2 f}{\partial x_i \partial x_j}(x) \approx \frac{f(x+he_i+he_j) - f(x+he_i) - f(x+he_j) + f(x)}{h^2}$$

Differentiation in Python

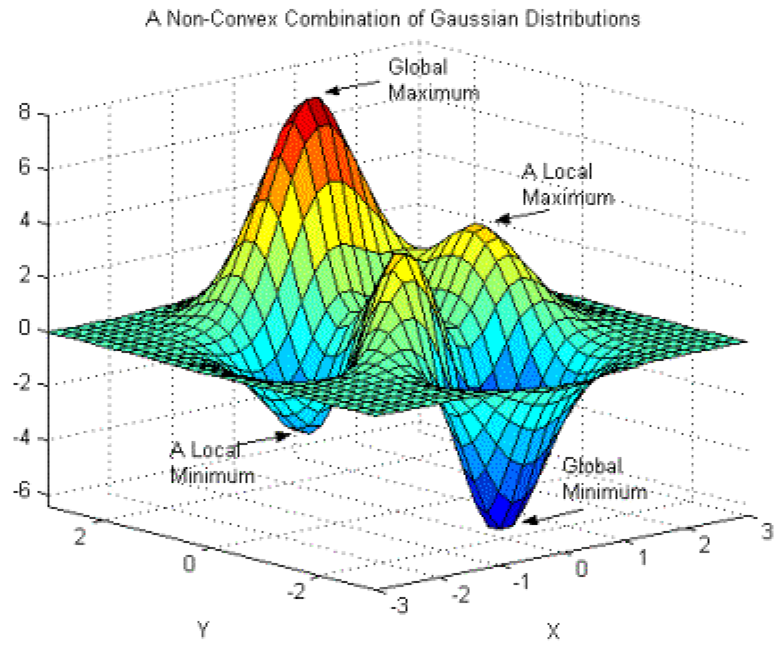
```python
import numpy as np
def df(f, h=1e-6):
    return lambda x: (f(x + h) - f(x - h))/(2*h)
```

Differentiation plays a key role in training and tuning machine learning models. Even LLM uses differentiation to compute the optimal weights.

Non-convex optimization

A non-convex optimization problem is an optimization problem where the objective function is non-convex, or the feasible set is non-convex. Non-convex optimization problems are difficult to solve because they have multiple local minima. The gradient descent algorithm is not guaranteed to converge to the global minimum if the objective function is non-convex or the feasible set is non-convex.

Non-convex optimization (2)



How to solve non-convex optimization problems? Escape local minima using random restarts.

Heuristics and metaheuristics

A heuristic is a technique that is used to solve a problem. A metaheuristic is a heuristic that is used to solve a class of problems.

Simplest metaheuristic: **random local search**. Start at a random point and move in a random direction until a local minimum is reached. Repeat the process multiple times and keep the best solution.

Coding a random local search in Python

```python
import numpy as np
def random_local_search(f, x0, alpha = 0.1, tol = 1e-6, max_iter = 1000):
    x = x0
    for _ in range(max_iter):
        x_new = x - alpha*np.random.randn(*x.shape)
        if np.max(np.abs(x_new - x)) < tol:
            break
        x = x_new
    return x
```