How to minimize a function

Gradient descent methods

Recap: minimize empirical risk

Recall that in machine learning, we aim to minimize the empirical risk function:

$$\hat{h} = \arg\min_{h \in \mathscr{H}} \frac{1}{n} \sum_{i=1}^{n} \left[L\left(\mathbf{y}_i, h(\mathbf{x}_i)\right) \right] \tag{1}$$

- $\mathbf{x}_i \in \mathbb{R}^p$: (training) features
- $\mathbf{y}_i \in \mathbb{R}^q$: (training) response
- $h(\cdot)$: decision function
- $\mathscr{H}$: a pre–defined functional space (e.g. Hilbert space)

Parameterization of decision functions

In practice, we usually approximate h by some parameterized function h_θ, and equation (1) becomes

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \boxed{\frac{1}{n} \sum_{i=1}^{n} \left[L\left(\mathbf{y}_i, h_\theta(\mathbf{x}_i)\right) \right]} \tag{2}$$

objective function

Problem definition

- In a general minimization problem, we want to find

$$\hat{\mathbf{x}} = \arg\min_{\mathbf{x}} f(\mathbf{x})$$

- f is also referred to the **objective function**
- Machine learning is one of the most important application of minimization problems
- 詳見**數值最佳化(陳鵬文)**

Analytic solution

Sometimes we may have a closed–form solution for $\hat{\mathbf{x}}$ with some specific $f(\mathbf{x})$, e.g.

- quadratic functions (linear regression)

Agenda

- Iterative algorithms
- Gradient descent algorithms
  - Batch gradient descent (BGD)
  - Mini–batch gradient descent (mini–BGD)
  - Stochastic gradient descent (SGD)
  - Mini–batch SGD

J216

Iterative algorithms

Iterative minimization algorithms

- Generate a sequence $\mathbf{x}_0, \mathbf{x}_1, \ldots$ such that

$$f\left(\mathbf{x}_{k+1}\right) \leq f\left(\mathbf{x}_k\right)$$

- If we generate the sequence by $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha \Delta \mathbf{x}_k$ for some $\alpha > 0$, then $\Delta \mathbf{x}_k$ is called a descent direction

- The sequence usually converges to some local minimum (in fact, a stationary point) of f

$-\nabla f$ is a descent direction

From Taylor's expansion,

$$f\left(\mathbf{x}-\alpha\nabla f(\mathbf{x})\right) = f(\mathbf{x}) - \alpha\nabla f(\mathbf{x})^{\mathsf{T}}\nabla f(\mathbf{x}) + o\left(\alpha^2\right)$$
$$= f(\mathbf{x}) - \alpha\|\nabla f(\mathbf{x})\|^2 + o\left(\alpha^2\right)$$

Hence, when α is sufficiently small (so that the remainder term will not exceed $\alpha\|\nabla f(\mathbf{x})\|^2$) we have

$$f\left(\mathbf{x}-\alpha\nabla f(\mathbf{x})\right) \leq f(\mathbf{x})$$

Thus, $-\nabla f(\mathbf{x})$ is a descent direction for all $\mathbf{x}$

Gradient descent algorithms

Gradient descent

- make an initial guess $\mathbf{x}_0$

- for $k = 0,1,2,\ldots$:

learning rate

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \nabla f\left(\mathbf{x}_k\right)$$

- stop when $f\left(\mathbf{x}_k\right) - f\left(\mathbf{x}_{k+1}\right) <$ tol

How to obtain gradients

- By hand ... (error prone!)
- Numerical differentiation
- **Automatic differentiation**

Example: simple linear regression

Consider $h(x) = \beta_0 + \beta_1 x$ and $L\left(y, h(x)\right) = \left[y - h(x)\right]^2$, then

$$-\nabla f = \begin{bmatrix} \frac{2}{n}\sum_{i=1}^{n}\left(y_i - \beta_0 - \beta_1 x_i\right) \\ \frac{2}{n}\sum_{i=1}^{n} x_i\left(y_i - \beta_0 - \beta_1 x_i\right) \end{bmatrix}$$

Example: simple linear regression

```
import numpy as np
n = 1000
x = np.random.uniform(0, 1, n)
y = 1 + 2*x + np.random.normal(0,1,n)
```

```
def loss_fun(y_true, y_pred):
    res = y_true - y_pred
    return res**2
```

```
def empirical_risk(theta, x, y):
    X = np.column_stack((np.ones_like(y), x))
    y_pred = X @ theta.T
    return np.mean(loss_fun(y, y_pred))
```

```
def grad_fun(theta, x, y):
    a = theta[0]
    b = theta[1]
    n = np.size(y)
    df = np.empty_like(theta)
    res = y-a-b*x
    df[0] = -2*np.mean(res)
    df[1] =-2*np.mean(x*res)
    return df
```

```
from scipy.optimize import minimize
theta_0 = np.array([0,0])
result = minimize(empirical_risk, theta_0, (x, y), method='CG', jac=grad_fun)
```

```
result.x
```

```
array([1.07345752, 1.84049417])
```

Automatic differentiation

- Evaluate the **exact** $\nabla f(\mathbf{x}_k)$ automatically without the explicit form of $\nabla f(\mathbf{x})$ (we'll cover it in the lecture of back propagation)

- Software implementations:

  - TensorFlow

  - PyTorch

  - Autograd and JAX

Automatic differentiation by Autograd

```
import autograd.numpy as np

def loss_fun(y_true, y_pred):
    res = y_true - y_pred
    return res**2

def empirical_risk(theta, x, y):
    X = np.column_stack((np.ones_like(y), x))
    y_pred = X @ theta.T
    return np.mean(loss_fun(y, y_pred))
```

```
from autograd import grad
grad_fun2 = grad(empirical_risk)
result2 = minimize(empirical_risk, theta_0, (x, y), method='CG', jac=grad_fun2)
result2
```

```
     fun: 0.9798626075565533
     jac: array([-8.31785702e-07,  2.21956456e-07])
 message: 'Optimization terminated successfully.'
    nfev: 24
     nit: 13
    njev: 24
  status: 0
 success: True
       x: array([1.07345752, 1.84049417])
```

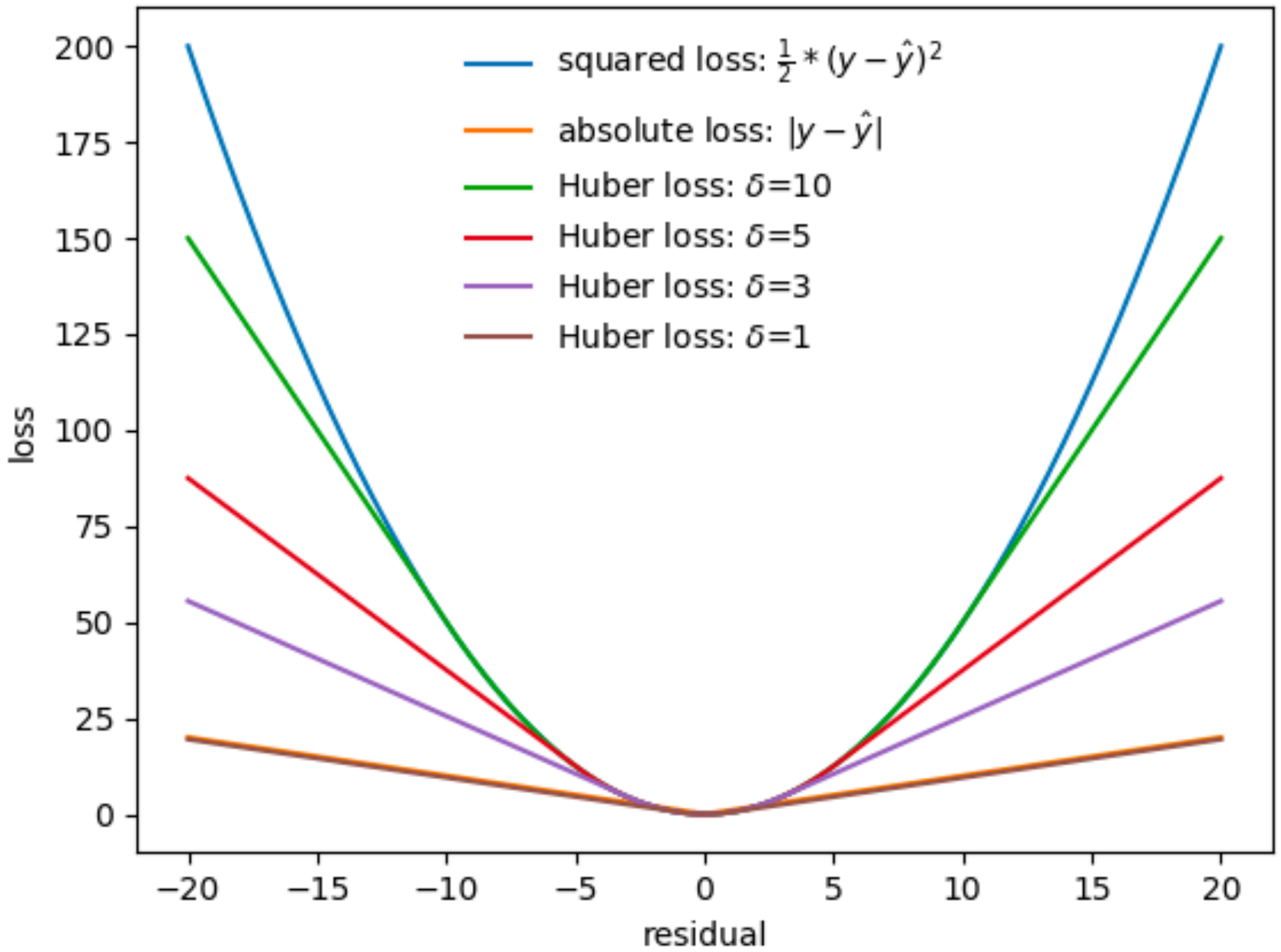
Robust regression

If we define the loss function in regression as the Huber loss

$$L_\delta\left(y,\hat{y}\right)=\begin{cases}\frac{1}{2}\left(y-\hat{y}\right)^2 & \text{if } \left|y-\hat{y}\right|\le\delta\\ \delta\left(\left|y-\hat{y}\right|-\frac{1}{2}\delta\right) & \text{otherwise,}\end{cases}$$

$\hat{h}(\mathbf{x})$ will be more robust to outliers.

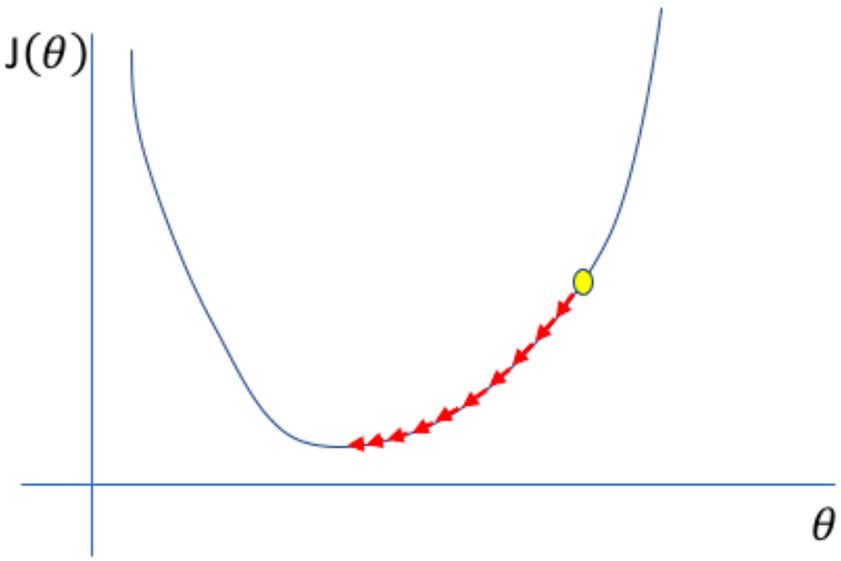
Huber loss



https://bit.ly/37MWwv8

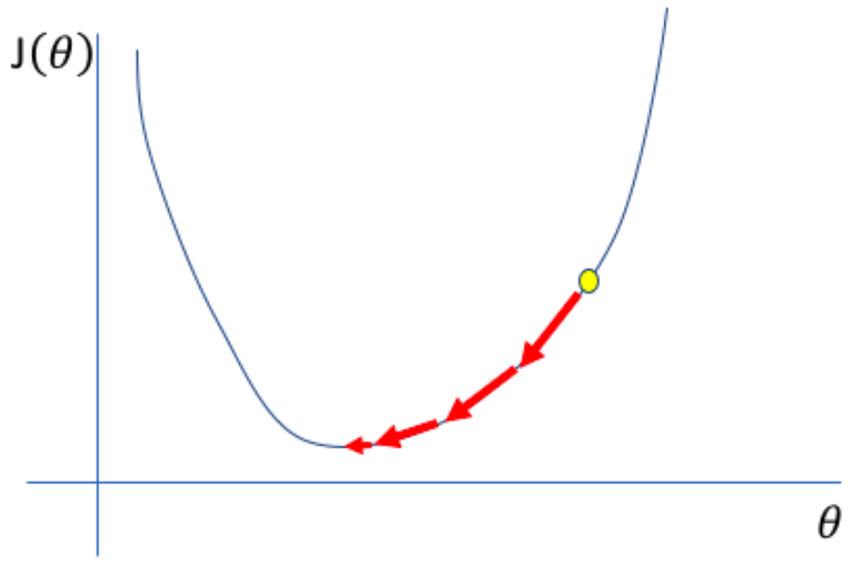
Learning rate

Too low



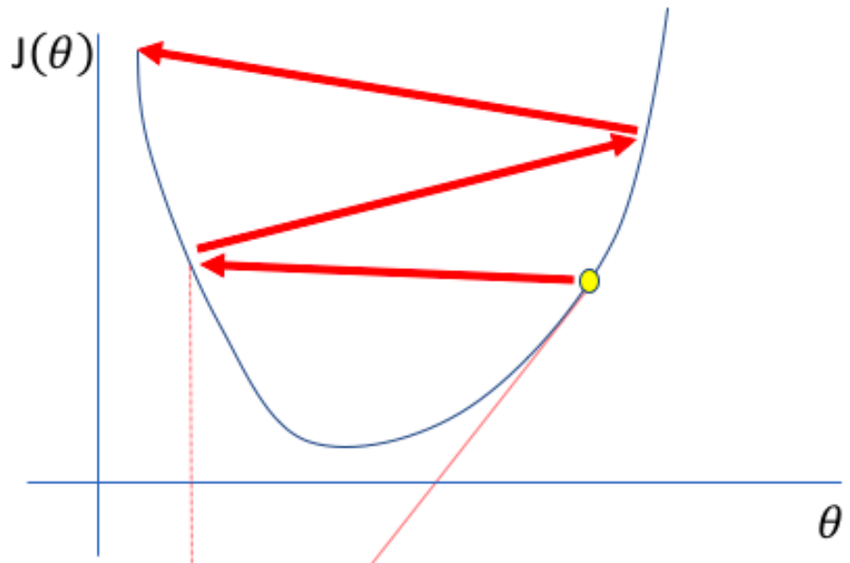
A small learning rate requires many updates before reaching the minimum point

Just right



The optimal learning rate swiftly reaches the minimum point

Too high



Too large of a learning rate causes drastic updates which lead to divergent behaviors

https://www.jeremyjordan.me/nn-learning-rate/

Extensions of gradient descent

Batch gradient descent

Recall that in machine learning, our objective function is usually

$$\frac{1}{n}\sum_{i=1}^{n} L\left(\mathbf{y}_i, h_\theta(\mathbf{x}_i)\right) \tag{3}$$

- The gradient of the equation (3) becomes

$$\frac{1}{n}\sum_{i=1}^{n}\nabla L\left(\mathbf{y}_i, h_\theta(\mathbf{x}_i)\right)$$

- The gradient descent method becomes

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \alpha_k \cdot \frac{1}{n}\sum_{i=1}^{n}\nabla L\left(\mathbf{y}_i, h_{\theta_k}(\mathbf{x}_i)\right)$$

- Known as batch gradient descent since all the training data is used in a batch to calculate the gradient

Stochastic gradient descent

- When n is large, computational issues may occur for BGD

- In SGD, we update the parameter by

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \alpha_k \cdot \nabla L\left(\mathbf{y}_i, h_{\theta_k}(\mathbf{x}_i)\right) \tag{4}$$

for some randomly selected i

Pseudo code for SGD

make an initial guess of $\boldsymbol{\theta}$ and set $k = 0$;

for **epoch** in $1,2,\cdots$:

    shuffle the dataset randomly

    for i in $1,2,\ldots,n$:

        update the parameter by equation (4)

        k += 1

Mini-batch SGD

- (Randomly) split the training data into mini-batches $\mathscr{B}$ of size $|\mathscr{B}|$

- For each step in an epoch, update the parameter by

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \frac{\alpha_k}{|\mathscr{B}|} \sum_{i \in \mathscr{B}} \nabla L \left(\mathbf{y}_i, h_{\theta_k}(\mathbf{x}_i) \right) \tag{5}$$

- $|\mathscr{B}|$ is also known as "batch size"

Pseudo code for mini–batch SGD

make an initial guess of $\boldsymbol{\theta}$ and set $k = 0$;

for epoch in $1,2,\cdots$:

    shuffle the dataset randomly

    split the dataset into B mini–batches

    for b in $1,2,\ldots,B$:

        update the parameter by equation (5)

        k += 1

Further readings

- Chapter 11 of Principles and Techniques of Data Science

- Chapter 8 of Data Science from Scratch: First Principles with Python

Homework: robust linear regression

1. Fit a robust MLR model with the Huber loss for the diabetes data by the (conjugated) gradient descent algorithm.

2. Fit an another robust MLR model by HuberRegressor with `alpha=0`.

3. Identify an appropriate δ by cross-validation (optional).