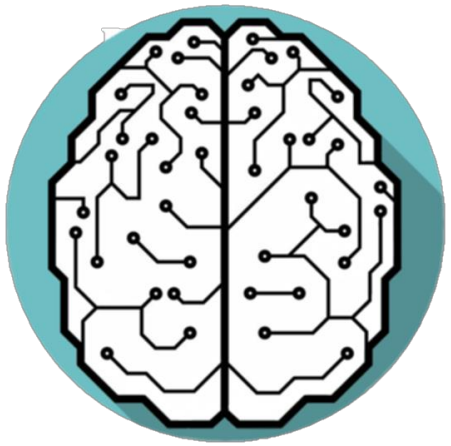
PyTorch Tutorial

03. Gradient Descent

Revision

- What would be the best model for the data?
- Linear model?

x (hours)	y (points)
1	2
2	4
3	6
4	?



Linear Model

$$\hat{y} = x * \omega$$

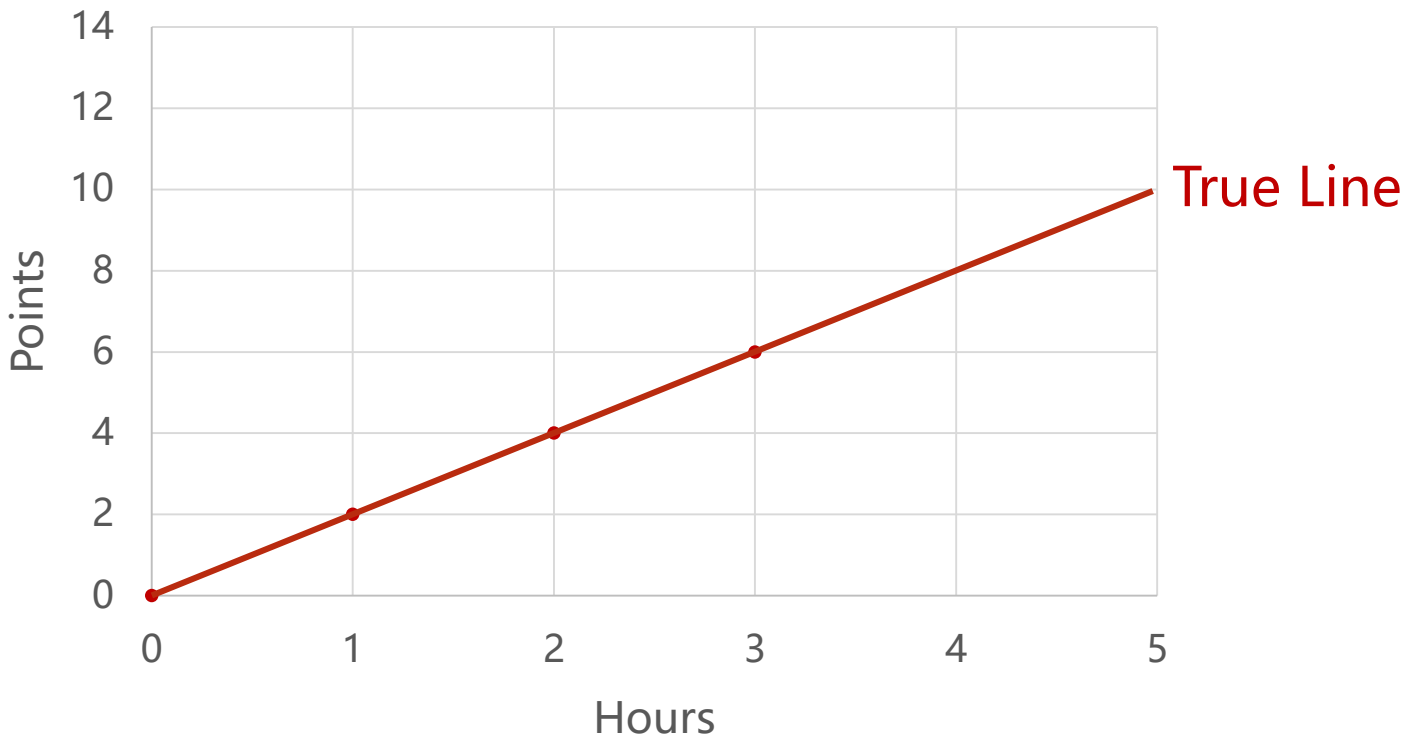
To simplify the model

Revision

Linear Model

$$\hat{y} = x * \omega$$

x (hours)	y (points)
1	2
2	4
3	6



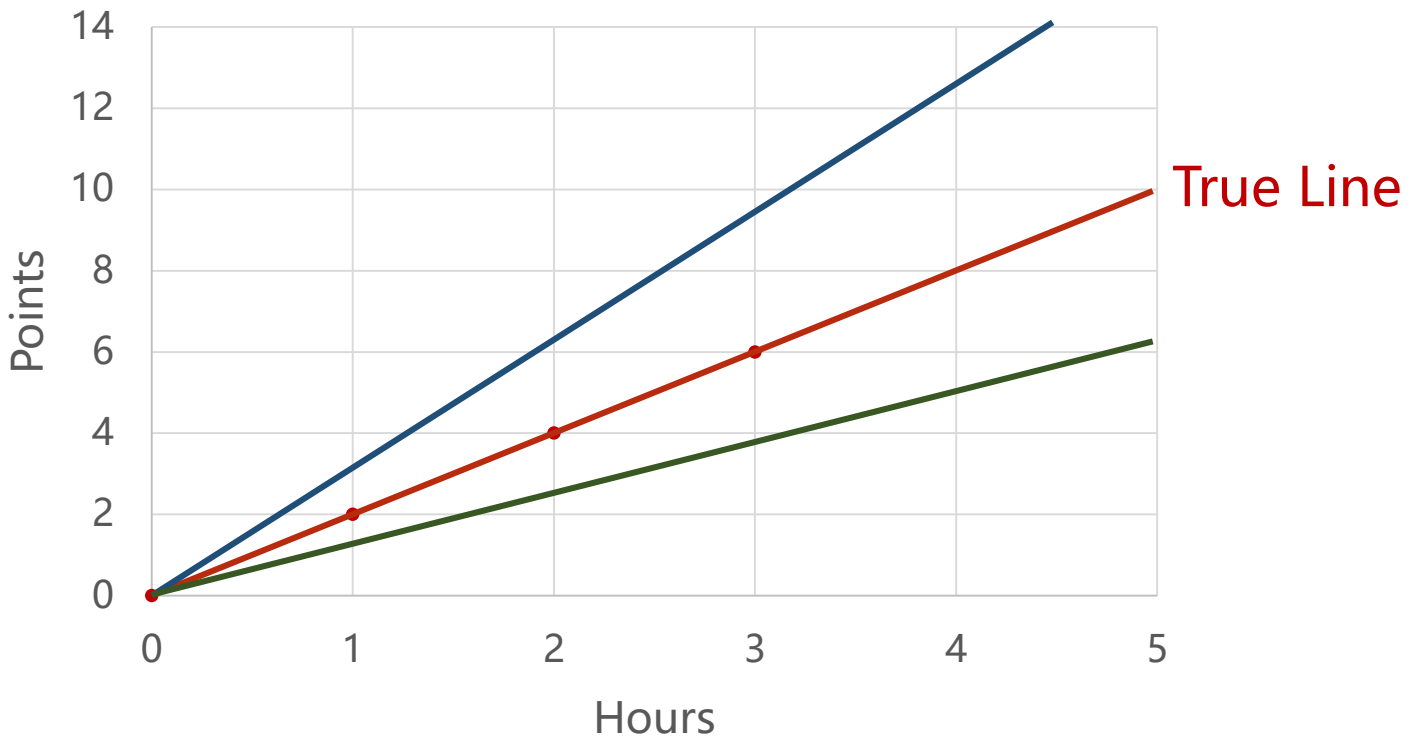
Revision

Linear Model

$$\hat{y} = x * \omega$$

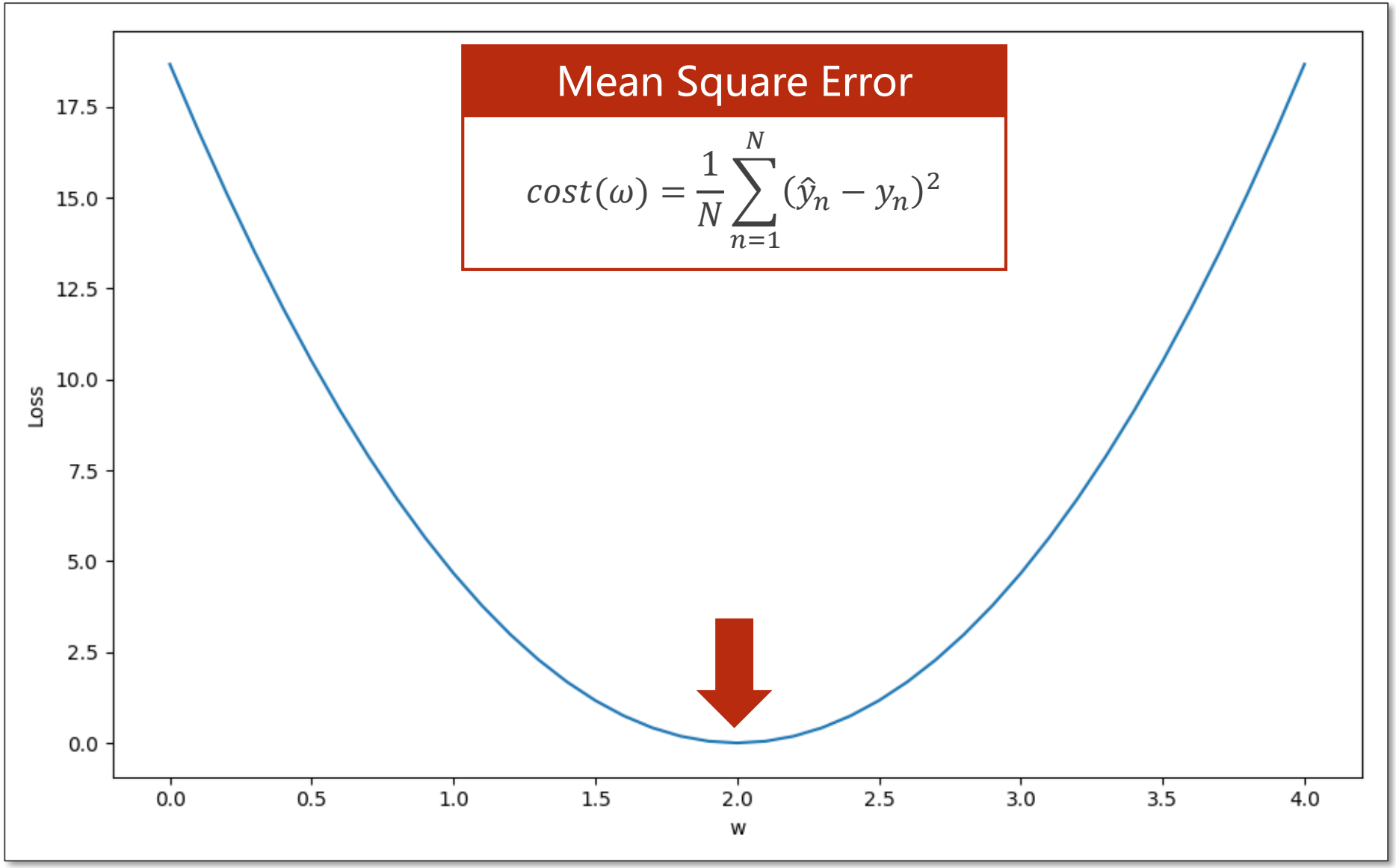
x (hours)	y (points)
1	2
2	4
3	6

The machine starts with **a random guess**, ω = random value

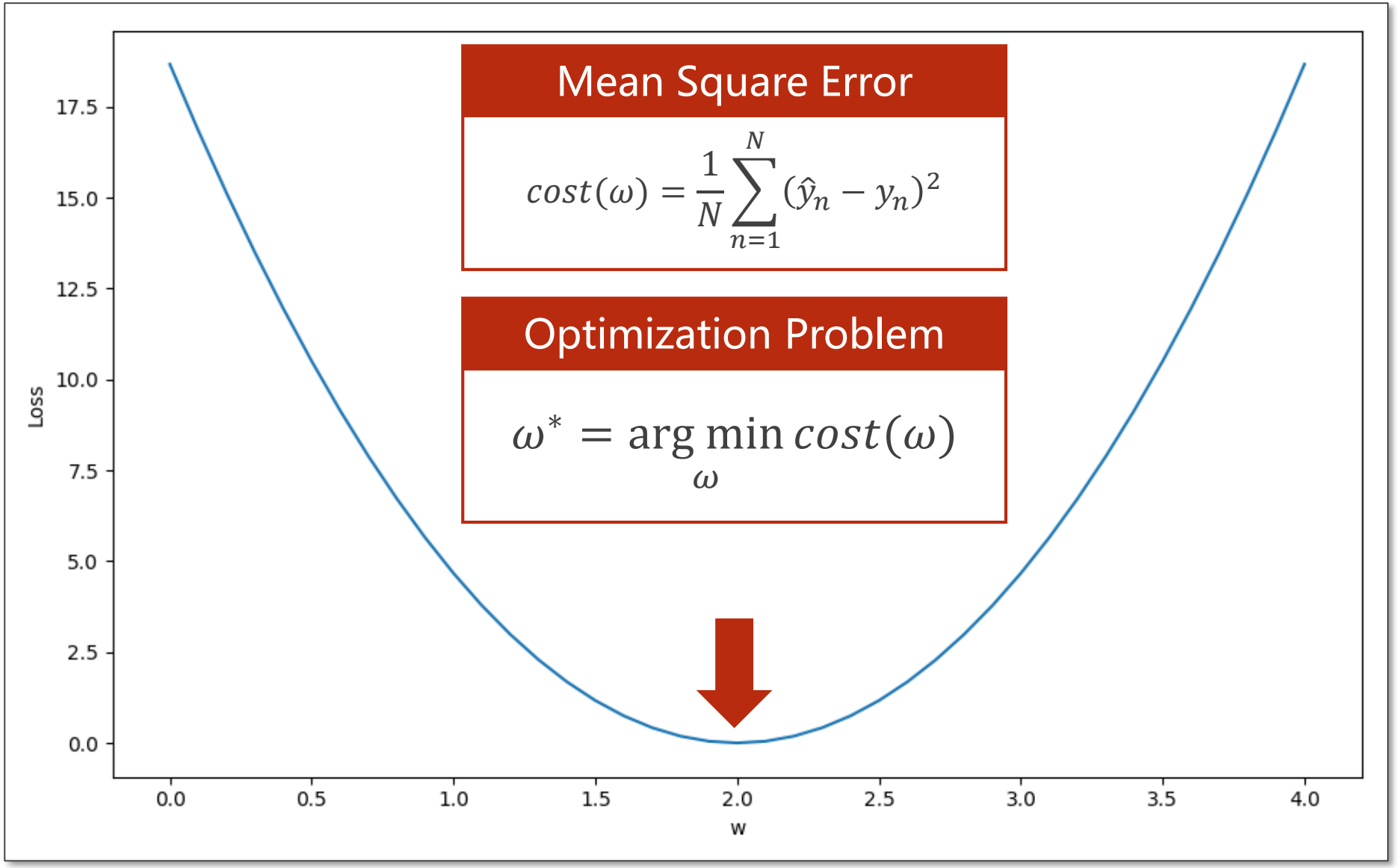


Revision

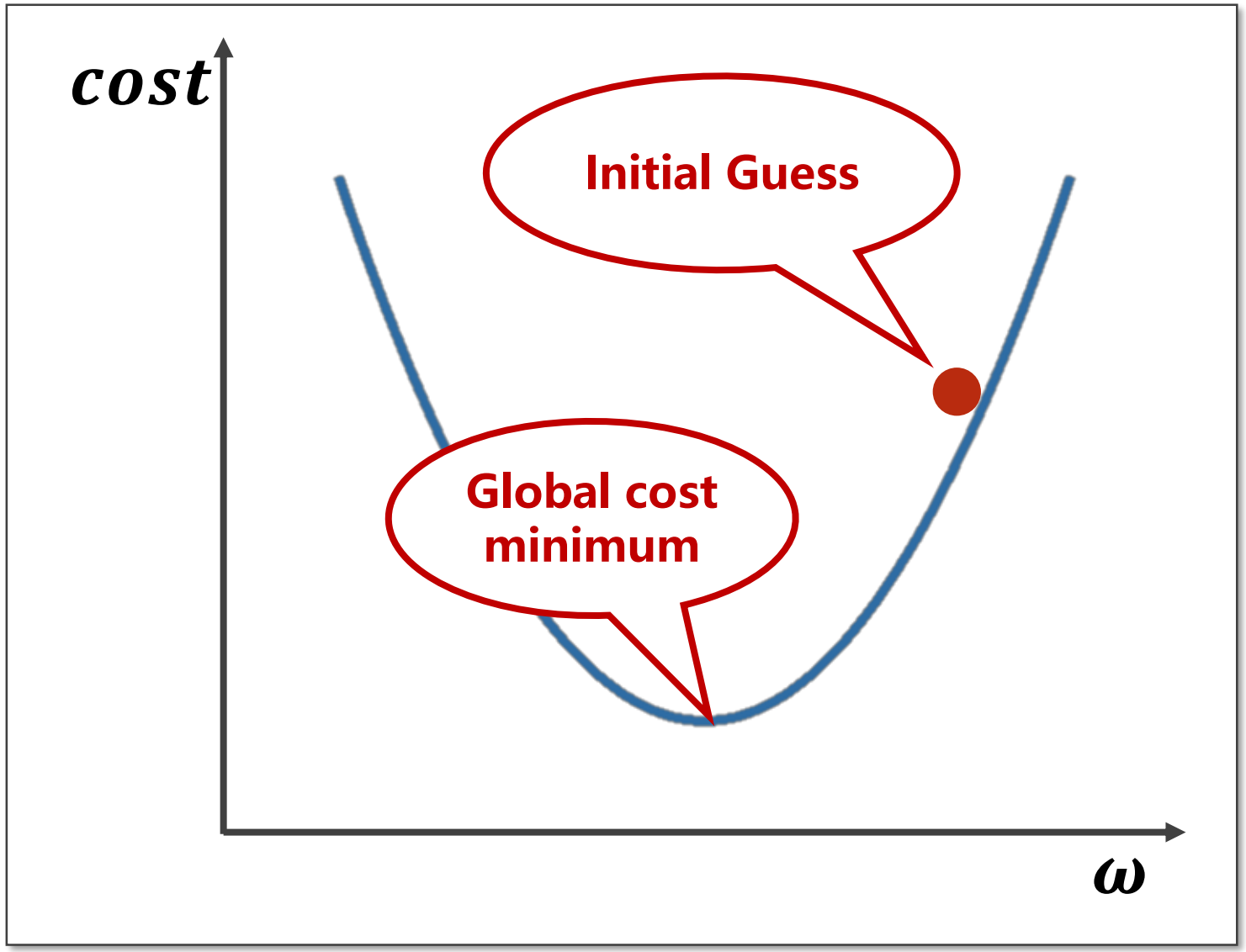
It can be found that when $\omega = 2$, the cost will be minimal.



Optimization Problem



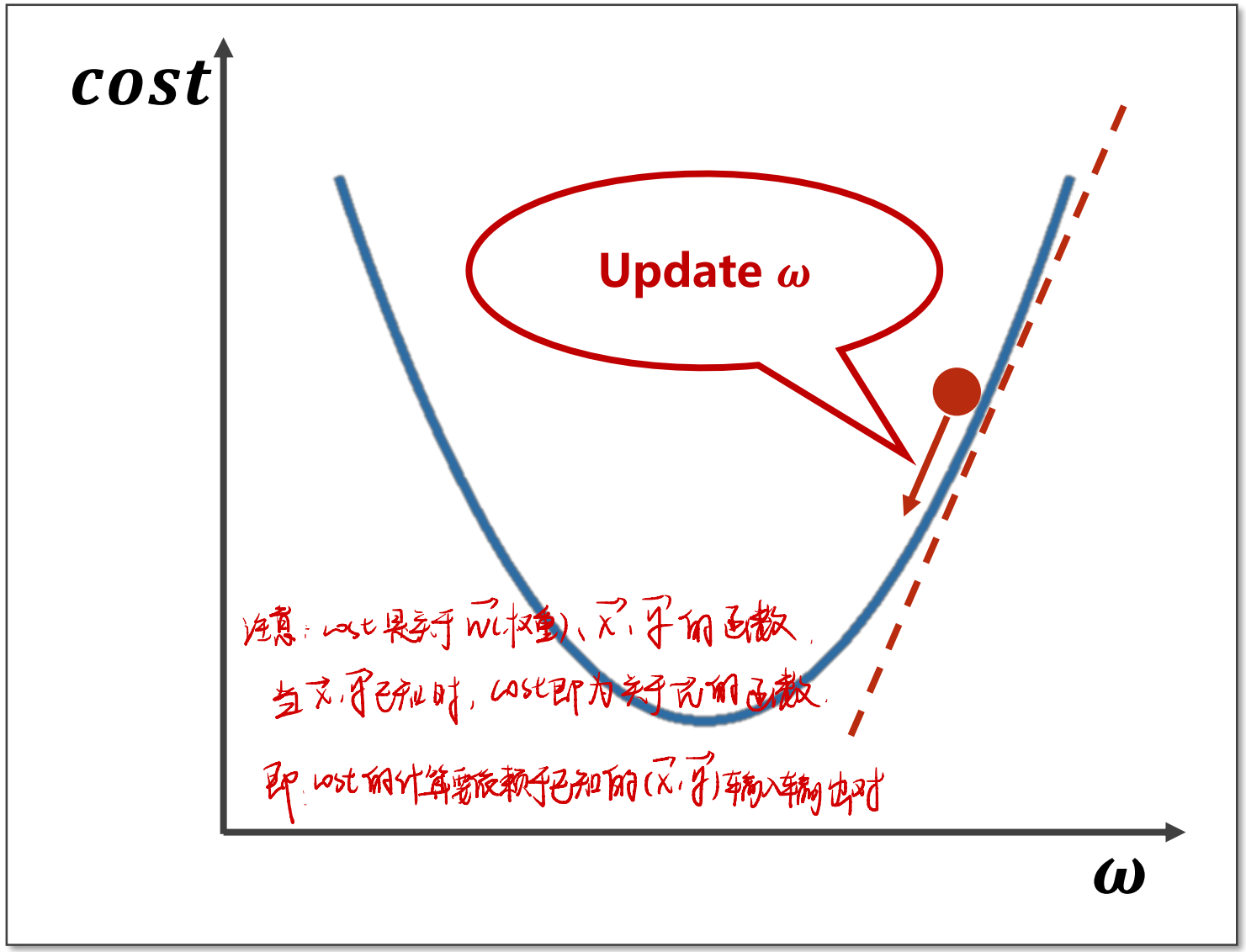
Gradient Descent Algorithm



梯度下降：

下降为何是负梯度

Gradient Descent Algorithm

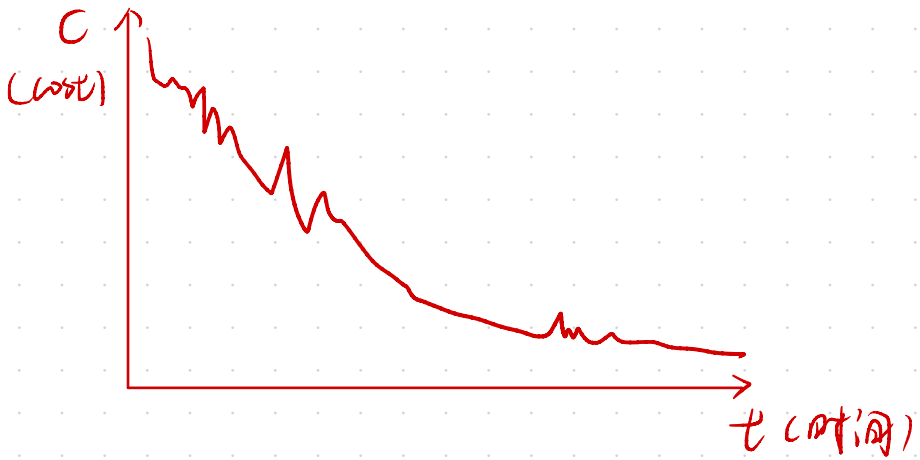


Gradient

$$\frac{\partial cost}{\partial \omega}$$

Update

$$\omega = \omega - \alpha \frac{\partial cost}{\partial \omega}$$



一般都不平滑，所以要做平滑化：指数加权均值.

如 原数据 C_0 C_1 C_2 ···

加权后 C_0 C_1' C_2' ···

有 $C_0' = C_0$，$C_t' = \beta C_t + (1-\beta) C_{t-1}$，$\beta$ 是权重值.

这样可以更好观察变化趋势.

Gradient Descent Algorithm

Derivative

$$\frac{\partial cost(\omega)}{\partial \omega} = \frac{\partial}{\partial \omega}\frac{1}{N}\sum_{n=1}^{N}(x_n \cdot \omega - y_n)^2$$

$$= \frac{1}{N}\sum_{n=1}^{N}\frac{\partial}{\partial \omega}(x_n \cdot \omega - y_n)^2$$

$$= \frac{1}{N}\sum_{n=1}^{N}2 \cdot (x_n \cdot \omega - y_n)\frac{\partial (x_n \cdot \omega - y_n)}{\partial \omega}$$

$$= \frac{1}{N}\sum_{n=1}^{N}2 \cdot x_n \cdot (x_n \cdot \omega - y_n)$$

Gradient

$$\frac{\partial cost}{\partial \omega}$$

Update

$$\omega = \omega - \alpha \frac{\partial cost}{\partial \omega}$$

Update

$$\omega = \omega - \alpha \frac{1}{N}\sum_{n=1}^{N}2 \cdot x_n \cdot (x_n \cdot \omega - y_n)$$

Implementation

```
x_data = [1.0, 2.0, 3.0]
y_data = [2.0, 4.0, 6.0]

w = 1.0

def forward(x):
    return x * w

def cost(xs, ys):
    cost = 0
    for x, y in zip(xs, ys):
        y_pred = forward(x)
        cost += (y_pred - y) ** 2
    return cost / len(xs)

def gradient(xs, ys):
    grad = 0
    for x, y in zip(xs, ys):
        grad += 2 * x * (x * w - y)
    return grad / len(xs)

print('Predict (before training)', 4, forward(4))
for epoch in range(100):
    cost_val = cost(x_data, y_data)
    grad_val = gradient(x_data, y_data)
    w -= 0.01 * grad_val
    print('Epoch:', epoch, 'w=', w, 'loss=', cost_val)
print('Predict (after training)', 4, forward(4))
```

```
x_data = [1.0, 2.0, 3.0]
y_data = [2.0, 4.0, 6.0]
```

Prepare the training set.

Implementation

```
x_data = [1.0, 2.0, 3.0]
y_data = [2.0, 4.0, 6.0]

w = 1.0

def forward(x):
    return x * w

def cost(xs, ys):
    cost = 0
    for x, y in zip(xs, ys):
        y_pred = forward(x)
        cost += (y_pred - y) ** 2
    return cost / len(xs)

def gradient(xs, ys):
    grad = 0
    for x, y in zip(xs, ys):
        grad += 2 * x * (x * w - y)
    return grad / len(xs)

print('Predict (before training)', 4, forward(4))
for epoch in range(100):
    cost_val = cost(x_data, y_data)
    grad_val = gradient(x_data, y_data)
    w -= 0.01 * grad_val
    print('Epoch:', epoch, 'w=', w, 'loss=', cost_val)
print('Predict (after training)', 4, forward(4))
```

```
w = 1.0
```

Initial guess of weight.

Implementation

```
x_data = [1.0, 2.0, 3.0]
y_data = [2.0, 4.0, 6.0]

w = 1.0

def forward(x):
    return x * w

def cost(xs, ys):
    cost = 0
    for x, y in zip(xs, ys):
        y_pred = forward(x)
        cost += (y_pred - y) ** 2
    return cost / len(xs)

def gradient(xs, ys):
    grad = 0
    for x, y in zip(xs, ys):
        grad += 2 * x * (x * w - y)
    return grad / len(xs)

print('Predict (before training)', 4, forward(4))
for epoch in range(100):
    cost_val = cost(x_data, y_data)
    grad_val = gradient(x_data, y_data)
    w -= 0.01 * grad_val
    print('Epoch:', epoch, 'w=', w, 'loss=', cost_val)
print('Predict (after training)', 4, forward(4))
```

```
def forward(x):
    return x * w
```

Define the model:

Linear Model

$$\hat{y} = x * \omega$$

Implementation

```
x_data = [1.0, 2.0, 3.0]
y_data = [2.0, 4.0, 6.0]

w = 1.0

def forward(x):
    return x * w

def cost(xs, ys):
    cost = 0
    for x, y in zip(xs, ys):
        y_pred = forward(x)
        cost += (y_pred - y) ** 2
    return cost / len(xs)

def gradient(xs, ys):
    grad = 0
    for x, y in zip(xs, ys):
        grad += 2 * x * (x * w - y)
    return grad / len(xs)

print('Predict (before training)', 4, forward(4))
for epoch in range(100):
    cost_val = cost(x_data, y_data)
    grad_val = gradient(x_data, y_data)
    w -= 0.01 * grad_val
    print('Epoch:', epoch, 'w=', w, 'loss=', cost_val)
print('Predict (after training)', 4, forward(4))
```

```
def cost(xs, ys):
    cost = 0
    for x, y in zip(xs, ys):
        y_pred = forward(x)
        cost += (y_pred - y) ** 2
    return cost / len(xs)
```

Define the cost function

Mean Square Error

$$cost(\omega) = \frac{1}{N}\sum_{n=1}^{N}(\hat{y}_n - y_n)^2$$

Implementation

```
x_data = [1.0, 2.0, 3.0]
y_data = [2.0, 4.0, 6.0]

w = 1.0

def forward(x):
    return x * w

def cost(xs, ys):
    cost = 0
    for x, y in zip(xs, ys):
        y_pred = forward(x)
        cost += (y_pred - y) ** 2
    return cost / len(xs)

def gradient(xs, ys):
    grad = 0
    for x, y in zip(xs, ys):
        grad += 2 * x * (x * w - y)
    return grad / len(xs)

print('Predict (before training)', 4, forward(4))
for epoch in range(100):
    cost_val = cost(x_data, y_data)
    grad_val = gradient(x_data, y_data)
    w -= 0.01 * grad_val
    print('Epoch:', epoch, 'w=', w, 'loss=', cost_val)
print('Predict (after training)', 4, forward(4))
```

```
def gradient(xs, ys):
    grad = 0
    for x, y in zip(xs, ys):
        grad += 2 * x * (x * w - y)
    return grad / len(xs)
```

Define the gradient function

Gradient

$$\frac{\partial cost}{\partial \omega} = \frac{1}{N}\sum_{n=1}^{N} 2 \cdot x_n \cdot (x_n \cdot \omega - y_n)$$

Implementation

```
x_data = [1.0, 2.0, 3.0]
y_data = [2.0, 4.0, 6.0]

w = 1.0

def forward(x):
    return x * w

def cost(xs, ys):
    cost = 0
    for x, y in zip(xs, ys):
        y_pred = forward(x)
        cost += (y_pred - y) ** 2
    return cost / len(xs)

def gradient(xs, ys):
    grad = 0
    for x, y in zip(xs, ys):
        grad += 2 * x * (x * w - y)
    return grad / len(xs)

print('Predict (before training)', 4, forward(4))
for epoch in range(100):
    cost_val = cost(x_data, y_data)
    grad_val = gradient(x_data, y_data)
    w -= 0.01 * grad_val
    print('Epoch:', epoch, 'w=', w, 'loss=', cost_val)
print('Predict (after training)', 4, forward(4))
```

```
for epoch in range(100):
    cost_val = cost(x_data, y_data)
    grad_val = gradient(x_data, y_data)
    w -= 0.01 * grad_val
```

Do the update

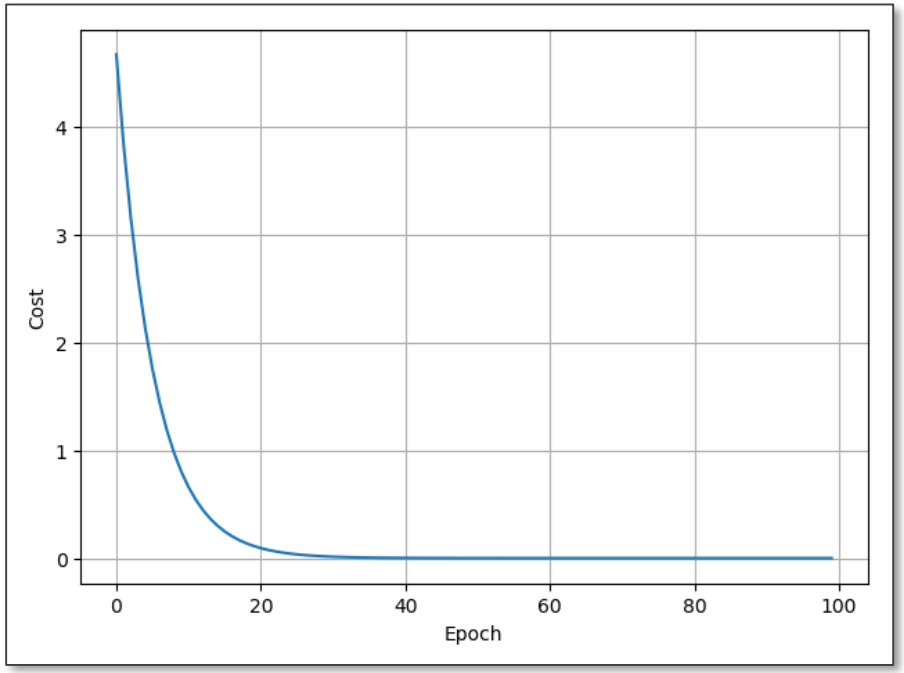
Update

$$\omega = \omega - \alpha \frac{\partial cost}{\partial \omega}$$

Implementation

```
x_data = [1.0, 2.0, 3.0]
y_data = [2.0, 4.0, 6.0]

w = 1.0

def forward(x):
    return x * w

def cost(xs, ys):
    cost = 0
    for x, y in zip(xs, ys):
        y_pred = forward(x)
        cost += (y_pred - y) ** 2
    return cost / len(xs)

def gradient(xs, ys):
    grad = 0
    for x, y in zip(xs, ys):
        grad += 2 * x * (x * w - y)
    return grad / len(xs)

print('Predict (before training)', 4, forward(4))
for epoch in range(100):
    cost_val = cost(x_data, y_data)
    grad_val = gradient(x_data, y_data)
    w -= 0.01 * grad_val
    print('Epoch:', epoch, 'w=', w, 'loss=', cost_val)
print('Predict (after training)', 4, forward(4))
```

```
Predict (before training) 4 4.0
Epoch:   0 w= 1.09 cost= 4.67
Epoch:   1 w= 1.18 cost= 3.84
Epoch:   2 w= 1.25 cost= 3.15
Epoch:   3 w= 1.32 cost= 2.59
Epoch:   4 w= 1.39 cost= 2.13
Epoch:   5 w= 1.44 cost= 1.75
Epoch:   6 w= 1.50 cost= 1.44
Epoch:   7 w= 1.54 cost= 1.18
Epoch:   8 w= 1.59 cost= 0.97
Epoch:   9 w= 1.62 cost= 0.80
Epoch:  10 w= 1.66 cost= 0.66
.....
Epoch:  90 w= 2.00 cost= 0.00
Epoch:  91 w= 2.00 cost= 0.00
Epoch:  92 w= 2.00 cost= 0.00
Epoch:  93 w= 2.00 cost= 0.00
Epoch:  94 w= 2.00 cost= 0.00
Epoch:  95 w= 2.00 cost= 0.00
Epoch:  96 w= 2.00 cost= 0.00
Epoch:  97 w= 2.00 cost= 0.00
Epoch:  98 w= 2.00 cost= 0.00
Epoch:  99 w= 2.00 cost= 0.00
Predict (after training)  4  8.00
```



Cost in each epoch

Stochastic Gradient Descent

随机梯度下降——更常用（梯度下降变种），即对每一个样本（输入输出对）单独求梯度

普通梯度下降：

Gradient Descent

$$\omega = \omega - \alpha \frac{\partial cost}{\partial \omega}$$

Derivative of Cost Function

$$\frac{\partial cost}{\partial \omega} = \frac{1}{N} \sum_{n=1}^{N} 2 \cdot x_n \cdot (x_n \cdot \omega - y_n)$$

随机：

Stochastic Gradient Descent

$$\omega = \omega - \alpha \frac{\partial loss}{\partial \omega}$$

不用再求均值，因为只使用了一个样本

Derivative of Loss Function

$$\frac{\partial loss_n}{\partial \omega} = 2 \cdot x_n \cdot (x_n \cdot \omega - y_n)$$

① 随机梯度下降：　依次对每个样本求梯度：　g是cost(损失)

即：$w_0^1 = w_1^1 - \alpha \frac{\partial g(x^1, y^1)}{\partial w}$

$w_1^1 = w_2^1 - \alpha \frac{\partial g(x^{1001}, y^{1001})}{\partial w}$

⋮

一次没有平均值

优点：更精确　缺点：无法并行运算，时间复杂度高

w迭代一次需要用的多次梯度计算

这里 $g(x, y, w) = cost = (\hat{y}_1 - y_1)^2 = (\hat{y}_1 - wx_1)^2$（没有平均值）

② 梯度下降：

$w_2 = w_{21} - \alpha \frac{\partial g}{\partial w}$

这里 $g(x, y, w) = cost = \frac{1}{N}\sum_{i=0}^{N}(\hat{y}_1 - y_1)^2 = \frac{1}{N}\sum_{i=0}^{N}(\hat{y}_1 - wx_1)^2$（有平均值）

↓

w迭代一次只需进行一次计算

即 □□□□

↑

所以在实际应用中会将二者混合（折中）使用（mini batch），即进行批量的随机梯度下降

↓

即将样本分为多个组，组内随机~，组之间梯度下降

Implementation of SGD

```
x_data = [1.0, 2.0, 3.0]
y_data = [2.0, 4.0, 6.0]

w = 1.0

def forward(x):
    return x * w

def loss(x, y):
    y_pred = forward(x)
    return (y_pred - y) ** 2

def gradient(x, y):
    return 2 * x * (x * w - y)

print('Predict (before training)', 4, forward(4))

for epoch in range(100):
    for x, y in zip(x_data, y_data):
        grad = gradient(x, y)
        w = w - 0.01 * grad
        print("\tgrad: ", x, y, grad)
        l = loss(x, y)

    print("progress:", epoch, "w=", w, "loss=", l)

print('Predict (after training)', 4, forward(4))
```

```
def loss(x, y):
    y_pred = forward(x)
    return (y_pred - y) ** 2
```

Calculate loss function:

Loss Function

$$loss = (\hat{y} - y)^2 = (x * \omega - y)^2$$

Implementation of SGD

```
x_data = [1.0, 2.0, 3.0]
y_data = [2.0, 4.0, 6.0]

w = 1.0

def forward(x):
    return x * w

def loss(x, y):
    y_pred = forward(x)
    return (y_pred - y) ** 2

def gradient(x, y):
    return 2 * x * (x * w - y)

print('Predict (before training)', 4, forward(4))

for epoch in range(100):
    for x, y in zip(x_data, y_data):
        grad = gradient(x, y)
        w = w - 0.01 * grad
        print("\tgrad: ", x, y, grad)
        l = loss(x, y)

    print("progress:", epoch, "w=", w, "loss=", l)

print('Predict (after training)', 4, forward(4))
```

```
def gradient(x, y):
    return 2 * x * (x * w - y)
```

Calculate loss function:

Derivative of Loss Function

$$\frac{\partial loss_n}{\partial \omega} = 2 \cdot x_n \cdot (x_n \cdot \omega - y_n)$$

Implementation of SGD

```
x_data = [1.0, 2.0, 3.0]
y_data = [2.0, 4.0, 6.0]

w = 1.0

def forward(x):
    return x * w

def loss(x, y):
    y_pred = forward(x)
    return (y_pred - y) ** 2

def gradient(x, y):
    return 2 * x * (x * w - y)

print('Predict (before training)', 4, forward(4))

for epoch in range(100):
    for x, y in zip(x_data, y_data):
        grad = gradient(x, y)
        w = w - 0.01 * grad
        print("\tgrad: ", x, y, grad)
        l = loss(x, y)

    print("progress:", epoch, "w=", w, "loss=", l)

print('Predict (after training)', 4, forward(4))
```

```
for epoch in range(100):
    for x, y in zip(x_data, y_data):
        grad = gradient(x, y)
        w = w - 0.01 * grad
        print("\tgrad: ", x, y, grad)
        l = loss(x, y)
```

Update weight by every grad of sample of train set.

PyTorch Tutorial

03. Gradient Descent