Neural Network

Chapter 4 Optimization

西安电子科技大学
XIDIAN UNIVERSITY

Outline

01 Gradient Descent

02 Batch-size Optimization SGD

03 Category optimization

04 Back-propagation Algorithm

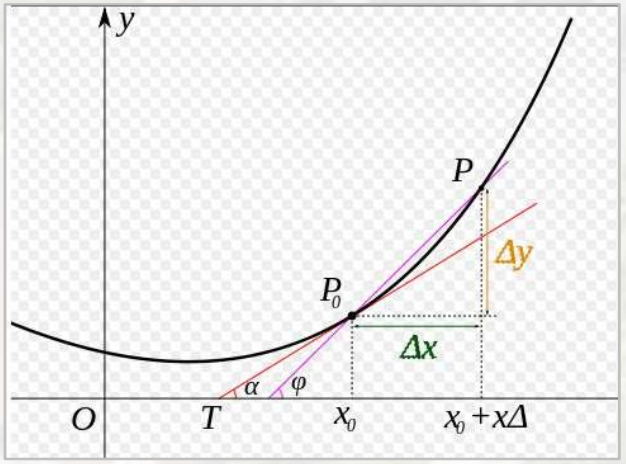
01

Gradient Descent

rivative

函数曲线上的切线斜率；函数在某点的变化率。

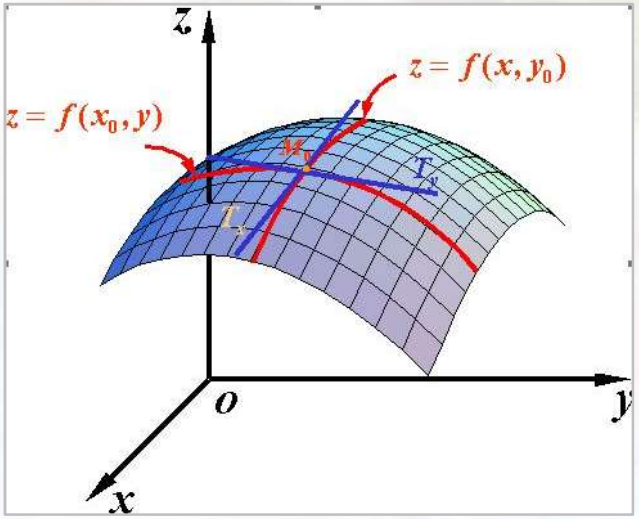
$$f'(x_0) = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} = \lim_{\Delta x \to 0} \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x}$$



rtial derivative

多元函数在坐标轴正方向上的变化率

$$f_x(x,y) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x, y) - f(x,y)}{\Delta x}$$



- 偏导数$f_x(x,y)$就是曲面被平面$y = y_0$所截得的曲线在点M_0处的切线M_0T_x对x轴的斜率
- 偏导数$f_y(x,y)$就是曲面被平面$x = x_0$所截得的曲线在点M_0处的切线M_0T_y对y轴的斜率

Gradient Descent

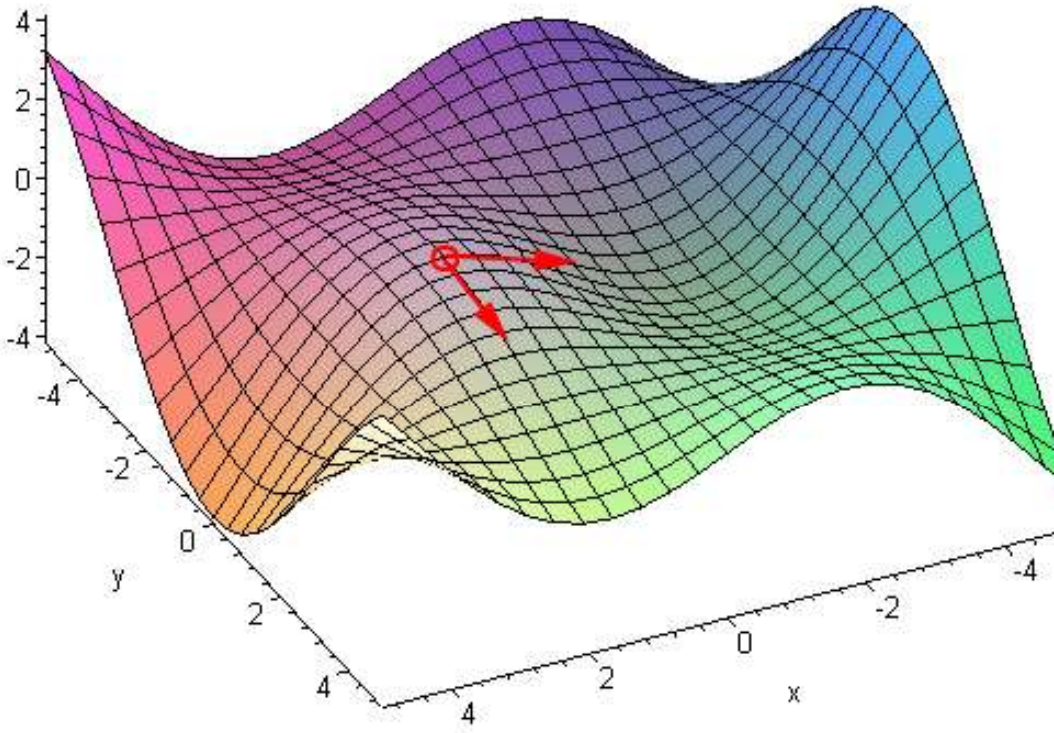
irectional
derivative

irectional Derivatives : Along the
xes…

方向导数用于研究函数沿各个不同方向时函数的变化率，是标量。

$$\frac{\partial f(x,y)}{\partial y}$$

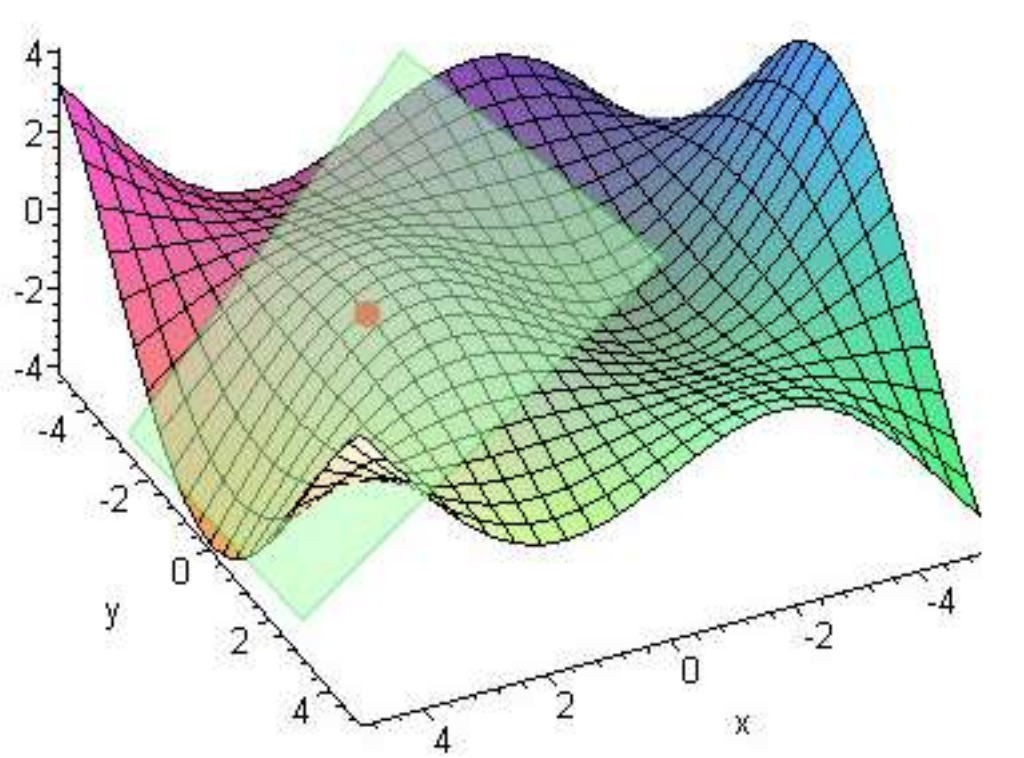
$$\frac{\partial f(x,y)}{\partial x}$$



$$\frac{\partial f}{\partial l}|_{(x_0,y_0)} = \lim_{\substack{\Delta x\to 0\\ \Delta y\to 0}} \frac{f(x_0+\Delta x, y_0+\Delta y)-f(x_0,y_0)}{\sqrt{\Delta x^2+\Delta y^2}}$$

$$\frac{\partial f}{\partial l}|_{(x_0,y_0)} = f_x'(x_0,y_0)cos\alpha + f_y'(x_0,y_0)cos\beta$$

- The gradient defines (hyper) plane approximating the function infinitesimally



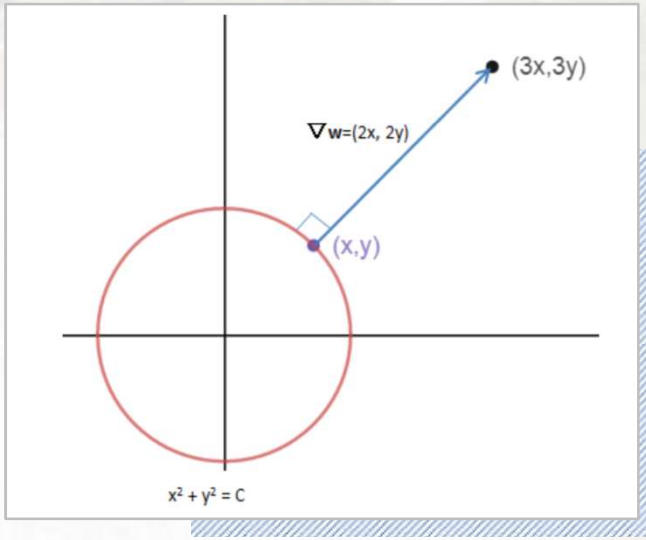
$$\Delta z = \frac{\partial f}{\partial x} \cdot \Delta x + \frac{\partial f}{\partial y} \cdot \Delta y$$

Gradient

矢量。梯度的方向是方向导数中取到最大值的方向，梯度的模是最大方向导数

$$gradf(x_0, x_1, \ldots, x_n) = \left(\frac{\partial f}{\partial x_0}, \ldots, \frac{\partial f}{\partial x_j}, \ldots, \frac{\partial f}{\partial x_n}\right)$$

- 在单变量的函数中，梯度其实就是函数的微分，代表着函数在某个给定点的切线的斜率
- 在多变量函数中，梯度是一个向量，向量有方向，梯度的方向就指出了函数在给定点的上升最快的方向
- 具有一阶连续偏导数才有梯度
- 梯度垂直于原函数的等值面



Gradient Descent

In 1-dimension, the derivative of a function:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

In multiple dimensions, the **gradient** is the vector of (partial derivatives).

Gradient Descent

- Method to find local optima of differentiable a function f
  - Intuition: gradient tells us direction of greatest increase, negative gradient gives us direction of greatest decrease
    - Take steps in directions that reduce the function value
  - Definition of derivative guarantees that if we take a small enough step in the direction of the negative gradient, the function will decrease in value
    - How small is small enough?

Gradient Descent Algorithm:

- Pick an initial point x_0
- Iterate until convergence

$$x_{t+1} = x_t - \gamma_t \nabla f(x_t)$$

where γ_t is the t^{th} step size (sometimes called learning rate)

Gradient Descent Algorithm:

- Pick an initial point x_0
- Iterate until convergence

$$x_{t+1} = x_t - \gamma_t \nabla f(x_t)$$

where γ_t is the t^{th} step size (sometimes called learning rate)

When do we stop?

Gradient Descent Algorithm:

- Pick an initial point x_0
- Iterate until convergence

$$x_{t+1} = x_t - \gamma_t \nabla f(x_t)$$

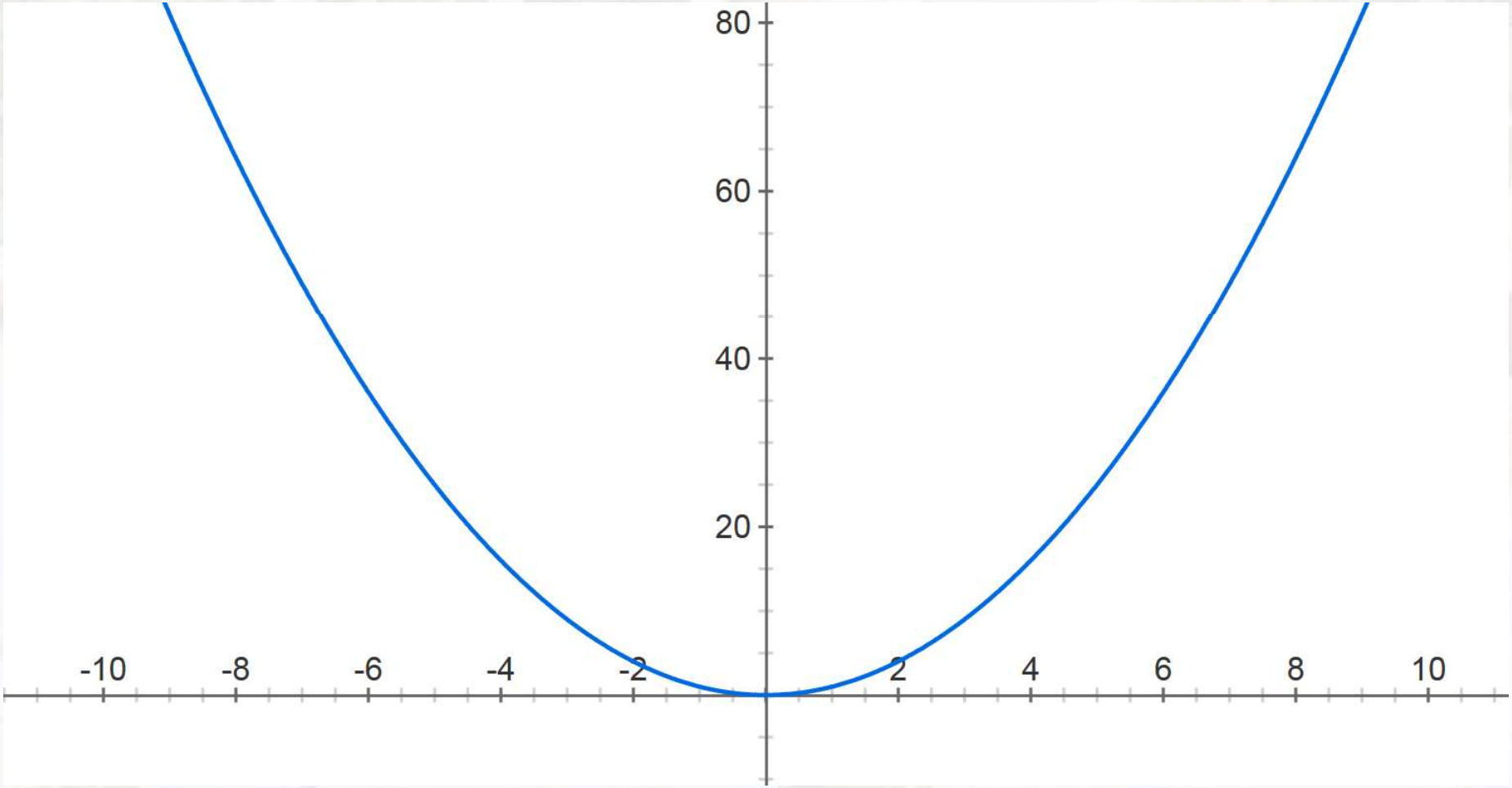
where γ_t is the t^{th} step size (sometimes called learning rate)

Possible Stopping Criteria: iterate until $\|\nabla f(x_t)\| \leq \epsilon$ for some $\epsilon > 0$

How small should ϵ be?

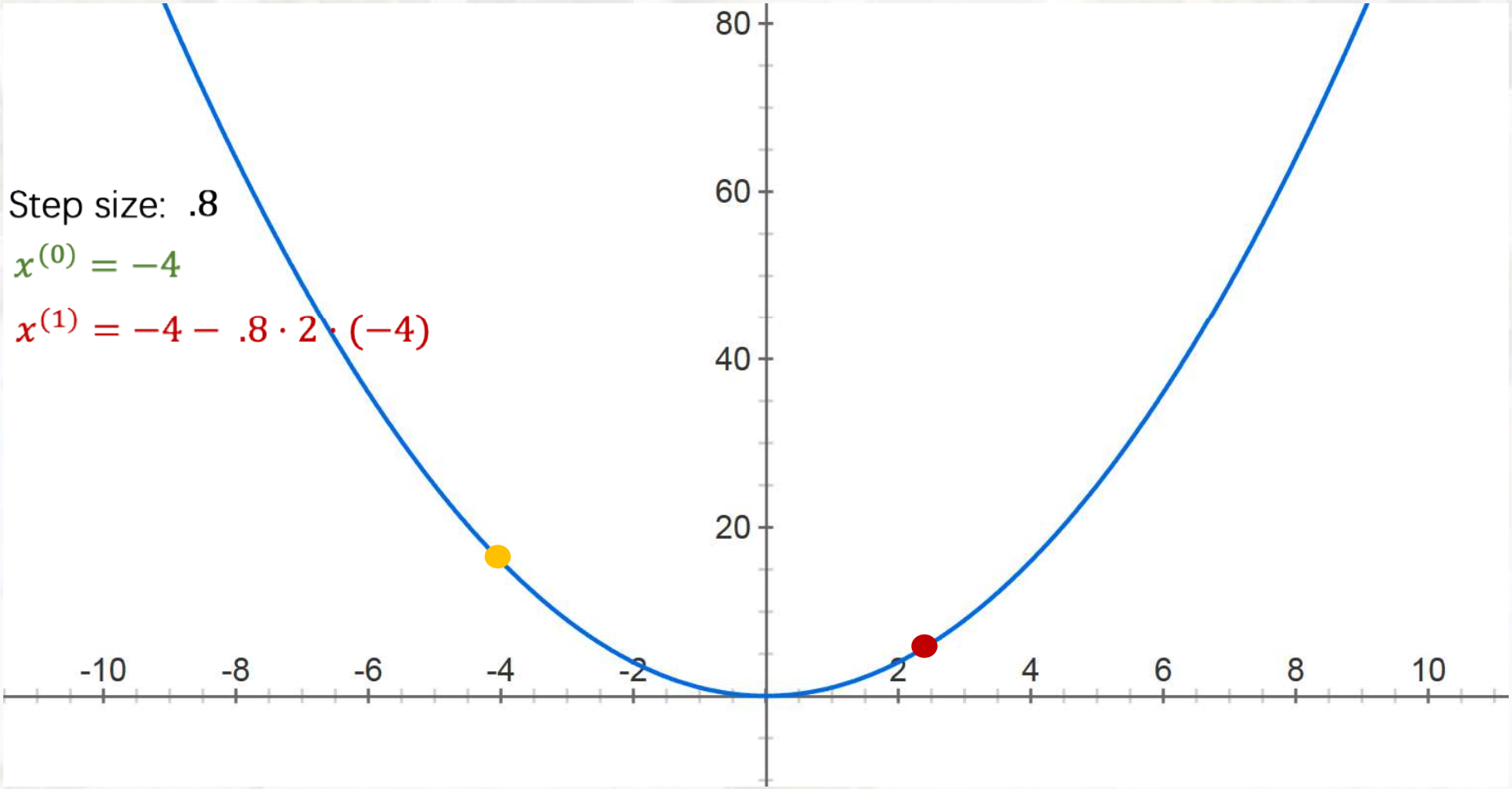
Gradient Descent

$f(x) = x^2$



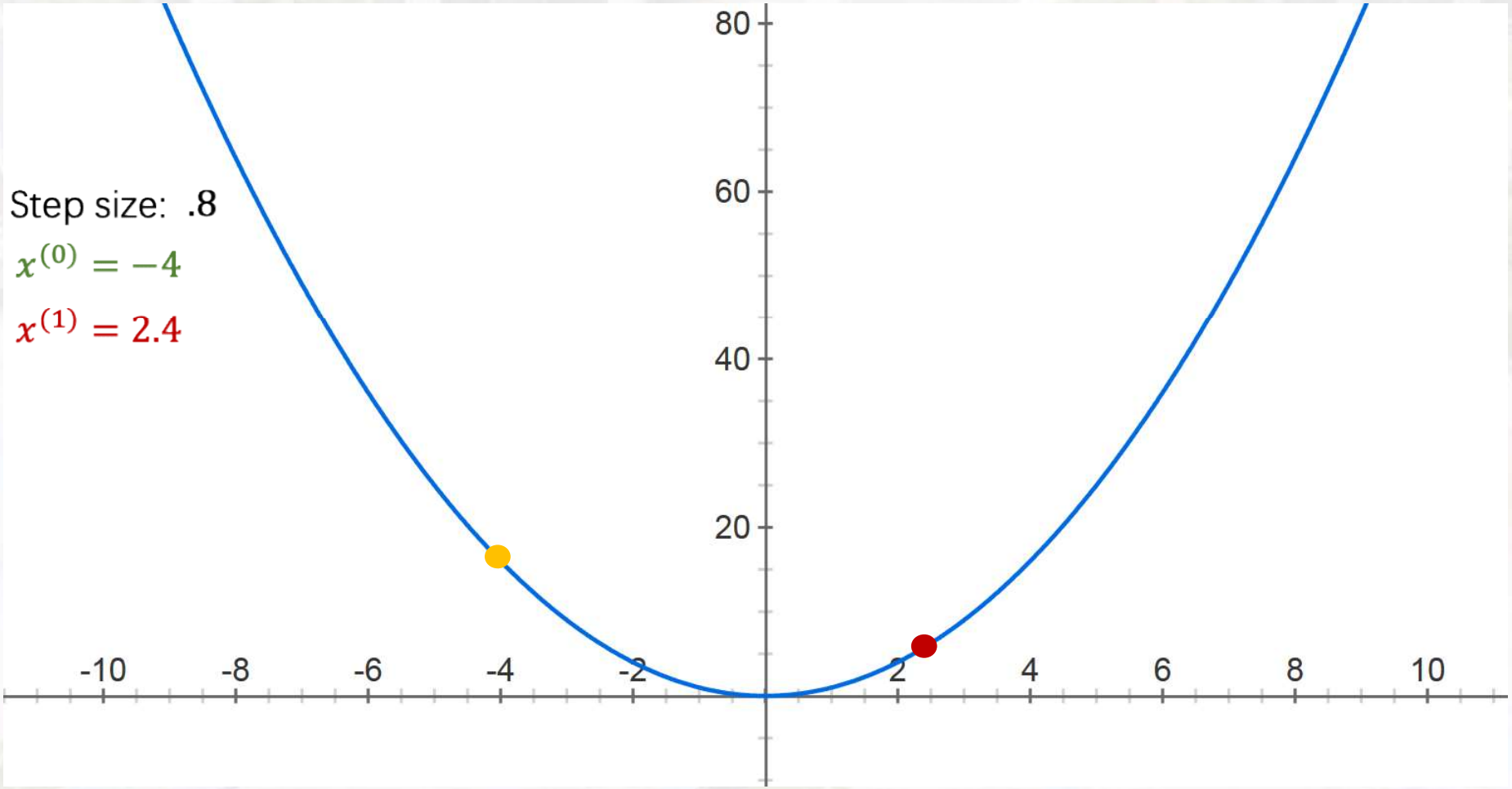
Gradient Descent

$f(x) = x^2$



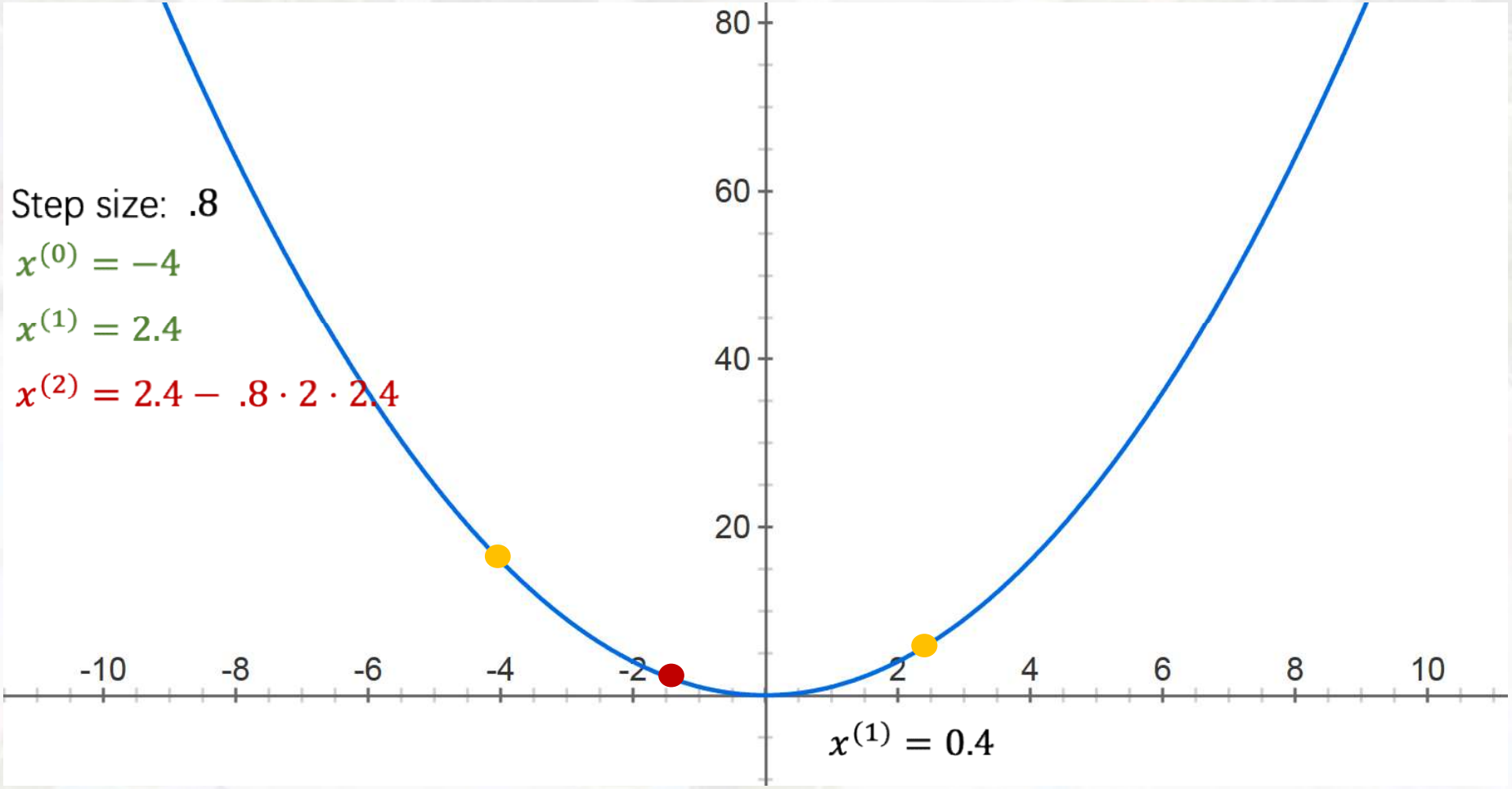
Gradient Descent

$f(x) = x^2$

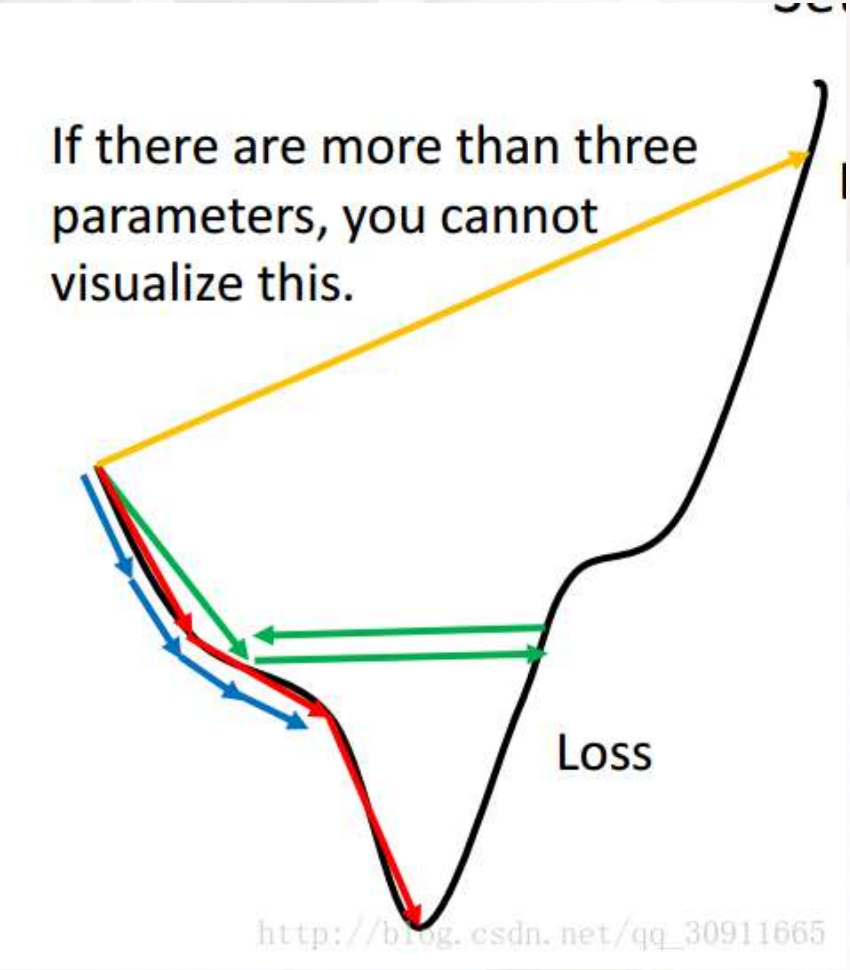


Gradient Descent

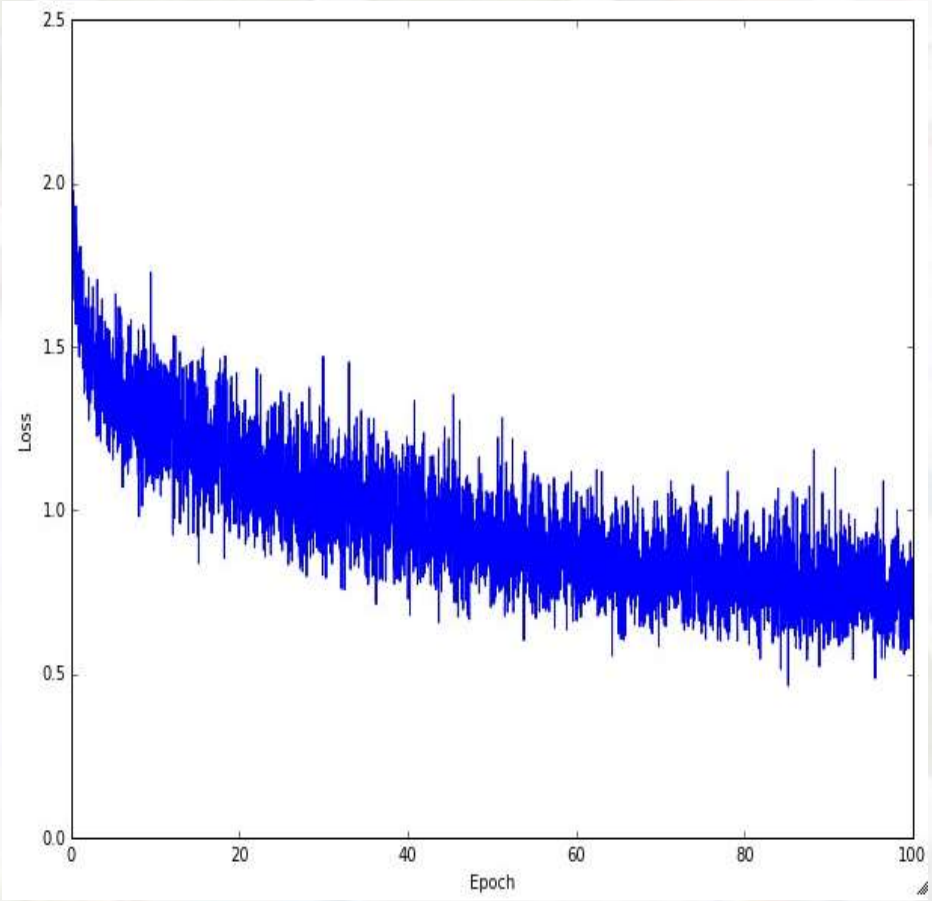
$f(x) = x^2$

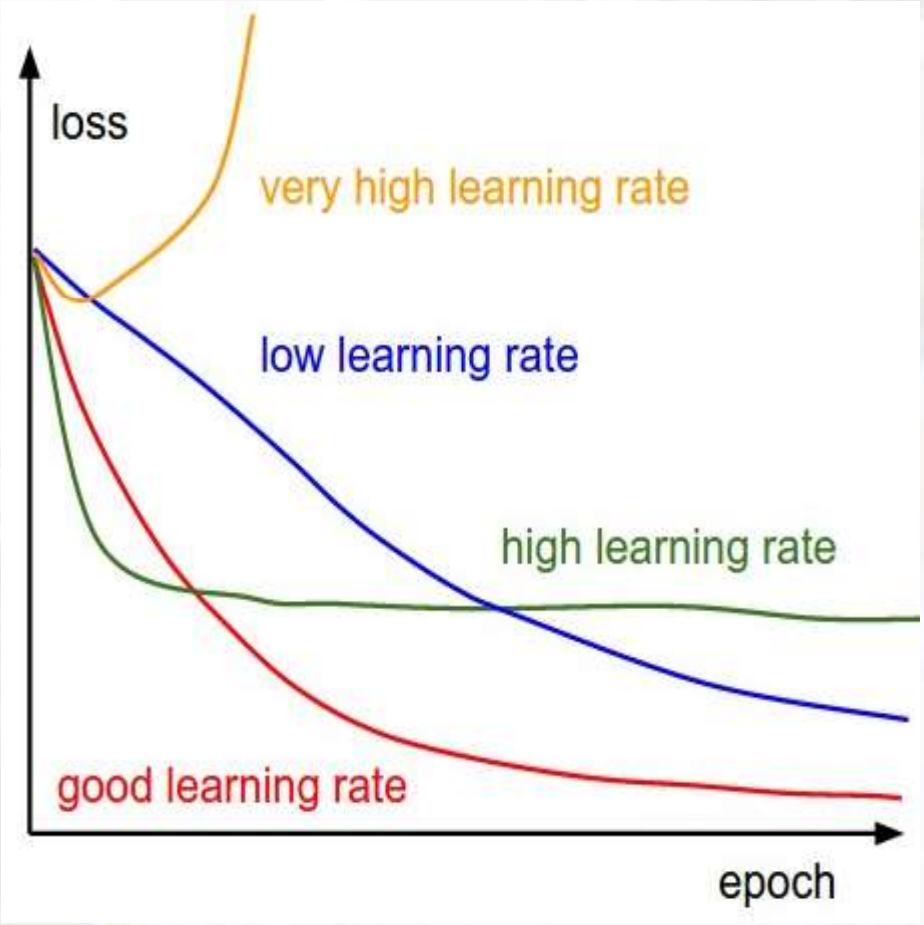


The effects of step size (or “learning rate”)

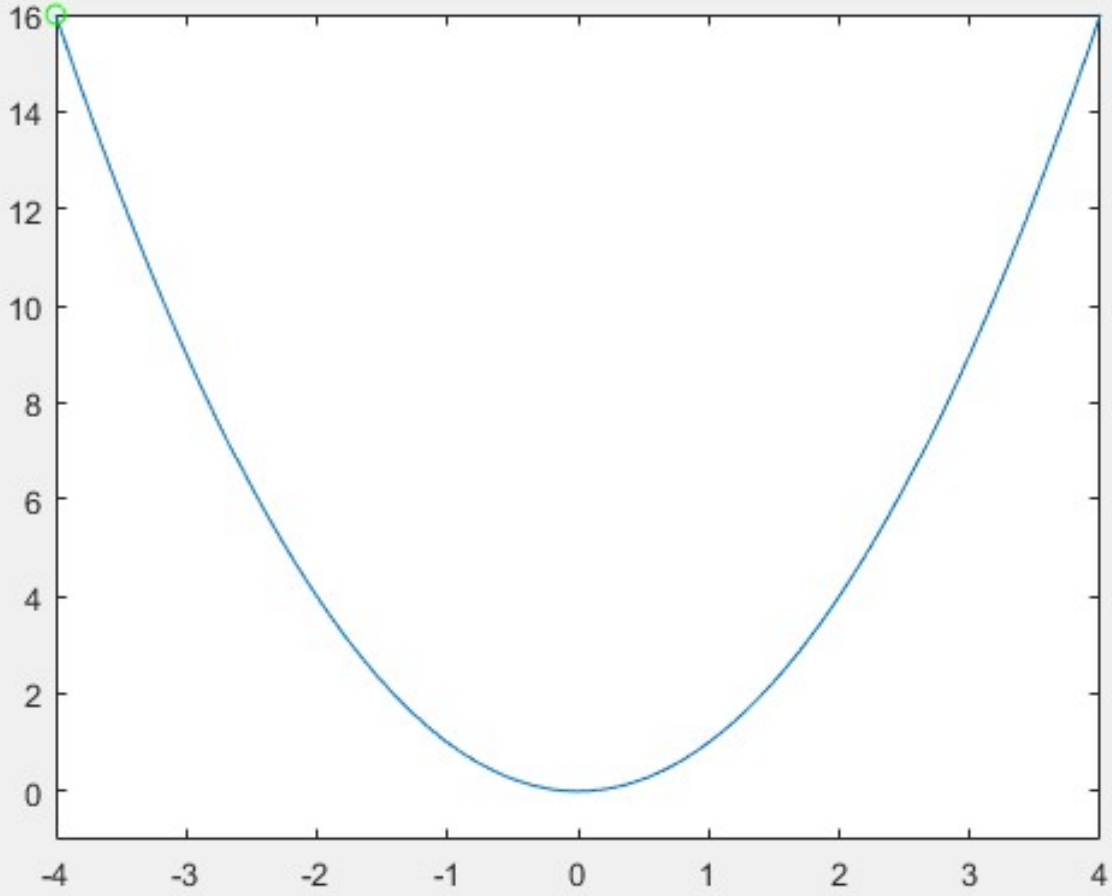


The effects of step size (or “learning rate”)

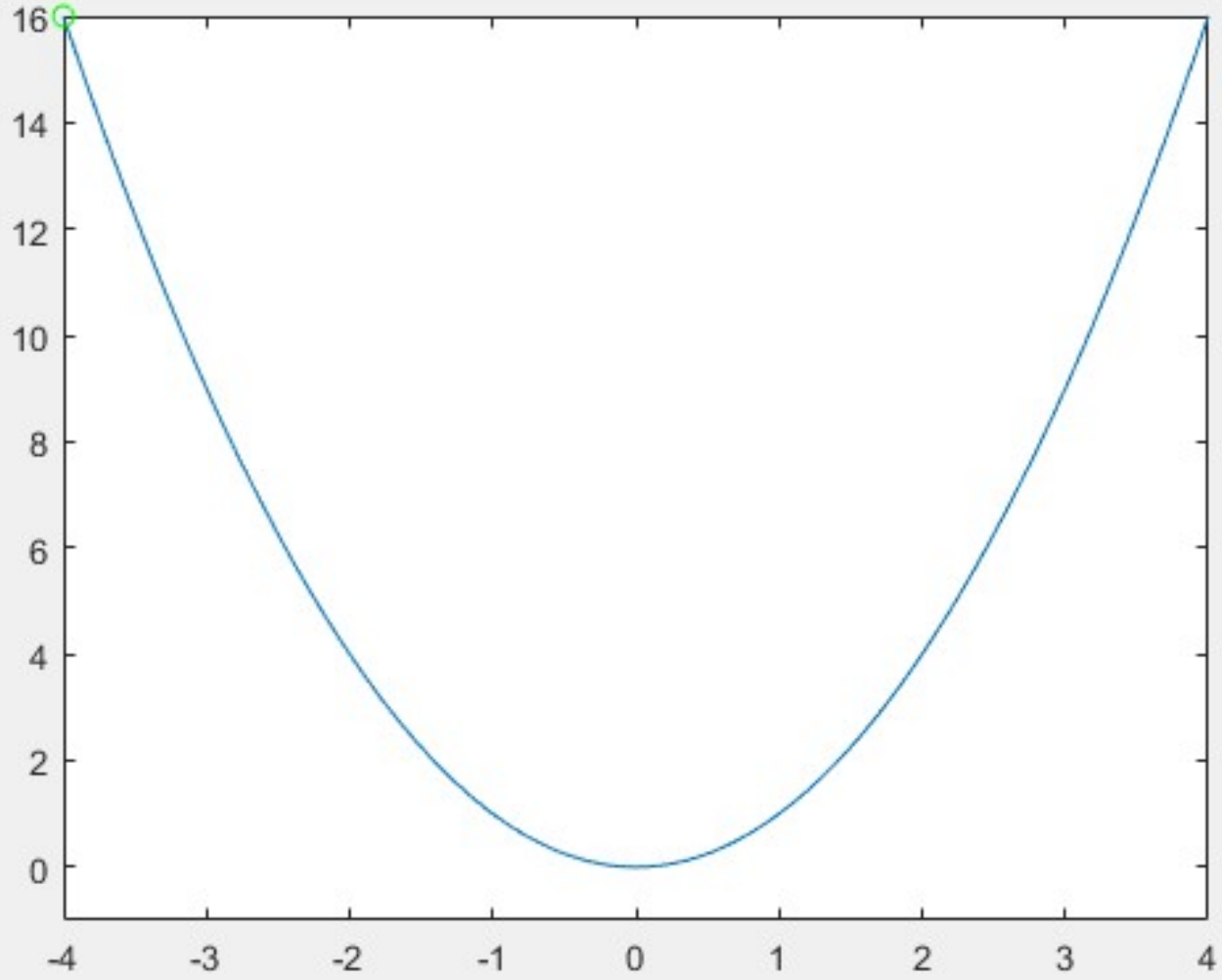




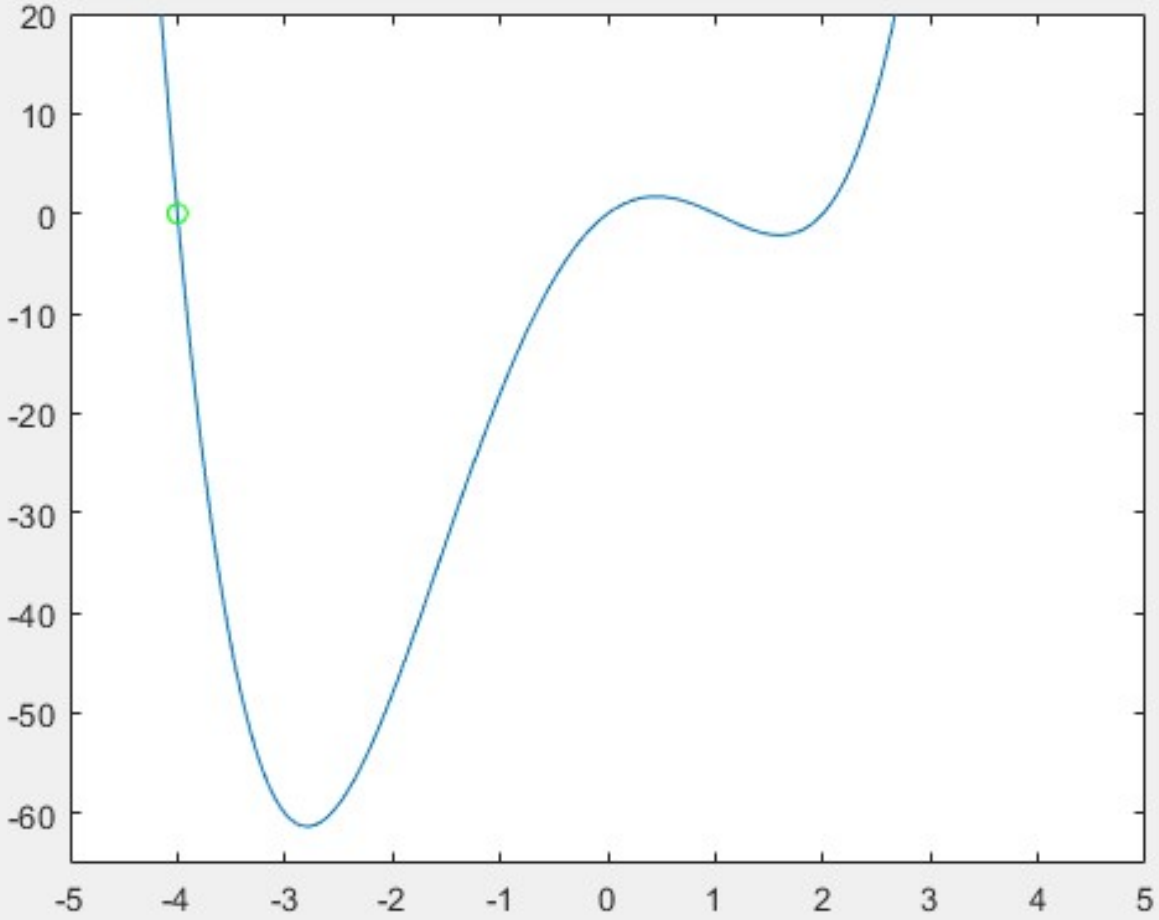
Gradient Descent



Gradient Descent



Gradient Descent



Gradient Descent

o find an optimum of a function $f(x)$ for high-dimensional
we want zeros of its gradient: $\nabla f(x) = 0$

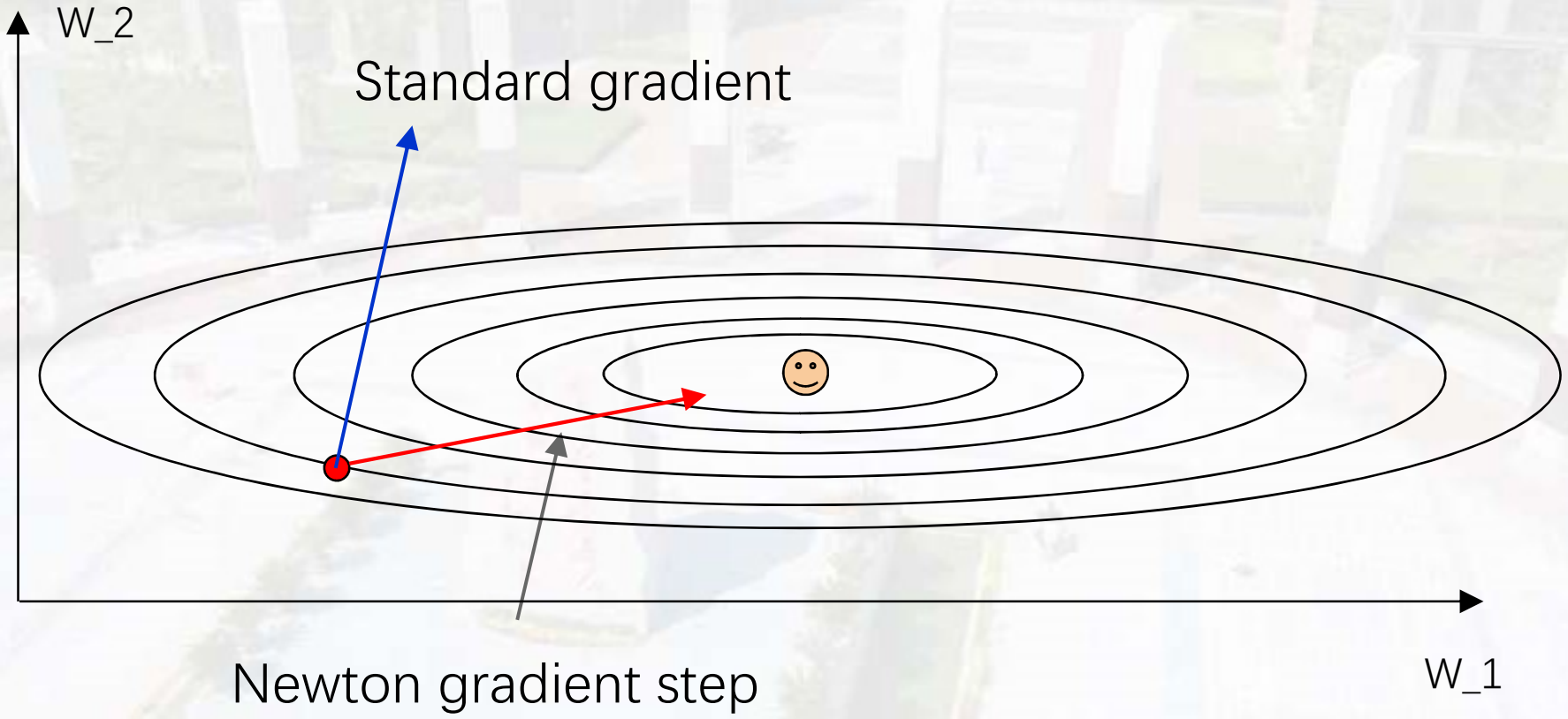
or zeros of $\nabla f(x)$ with a vector displacement h, Taylor's
xpansion is:

$$\nabla f(x+h) = \nabla f(x) + h^T H_f(x) + O(||h||^2)$$

here H_f is the Hessian matrix of second derivatives of f.
he update is:

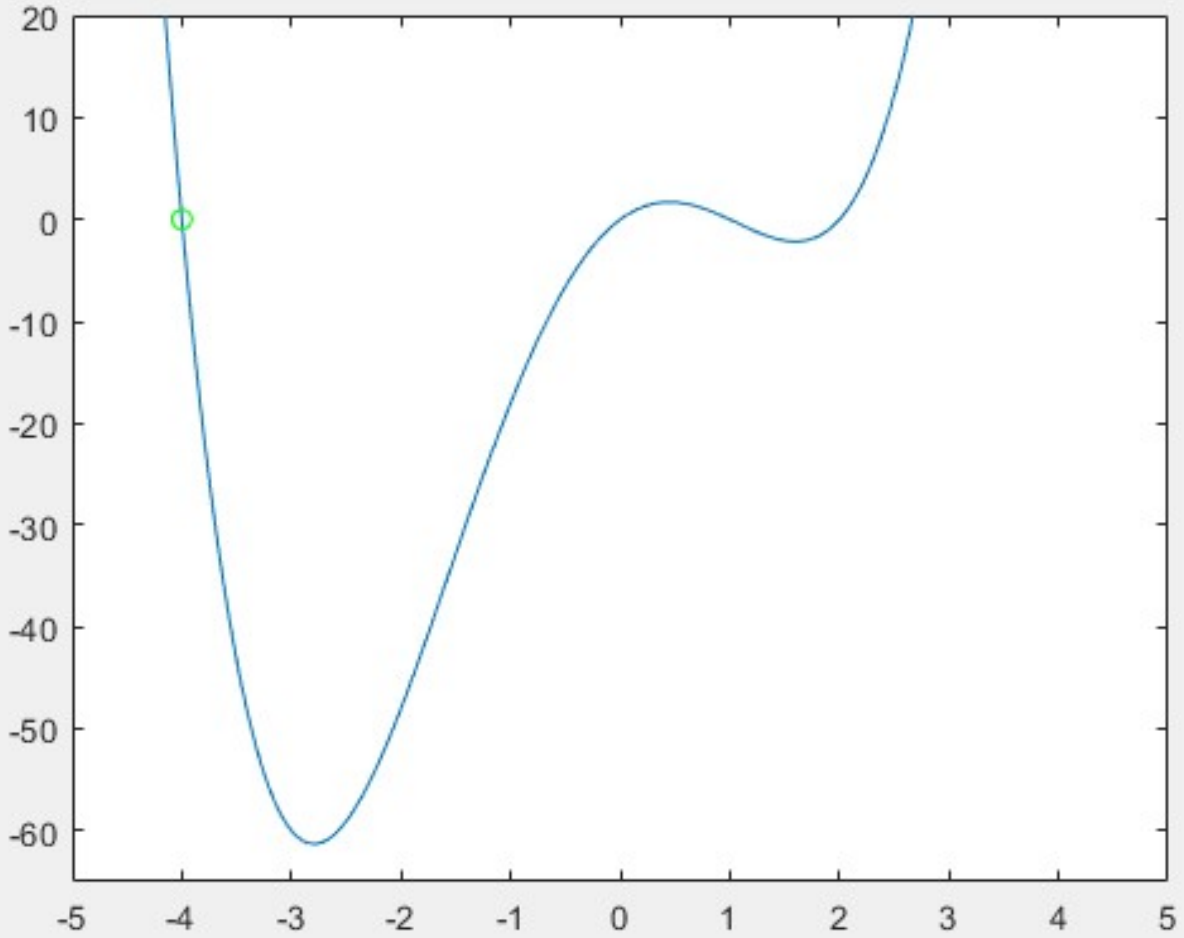
$$x_{n+1} = x_n - H_f(x_n)^{-1} \nabla f(x_n)$$

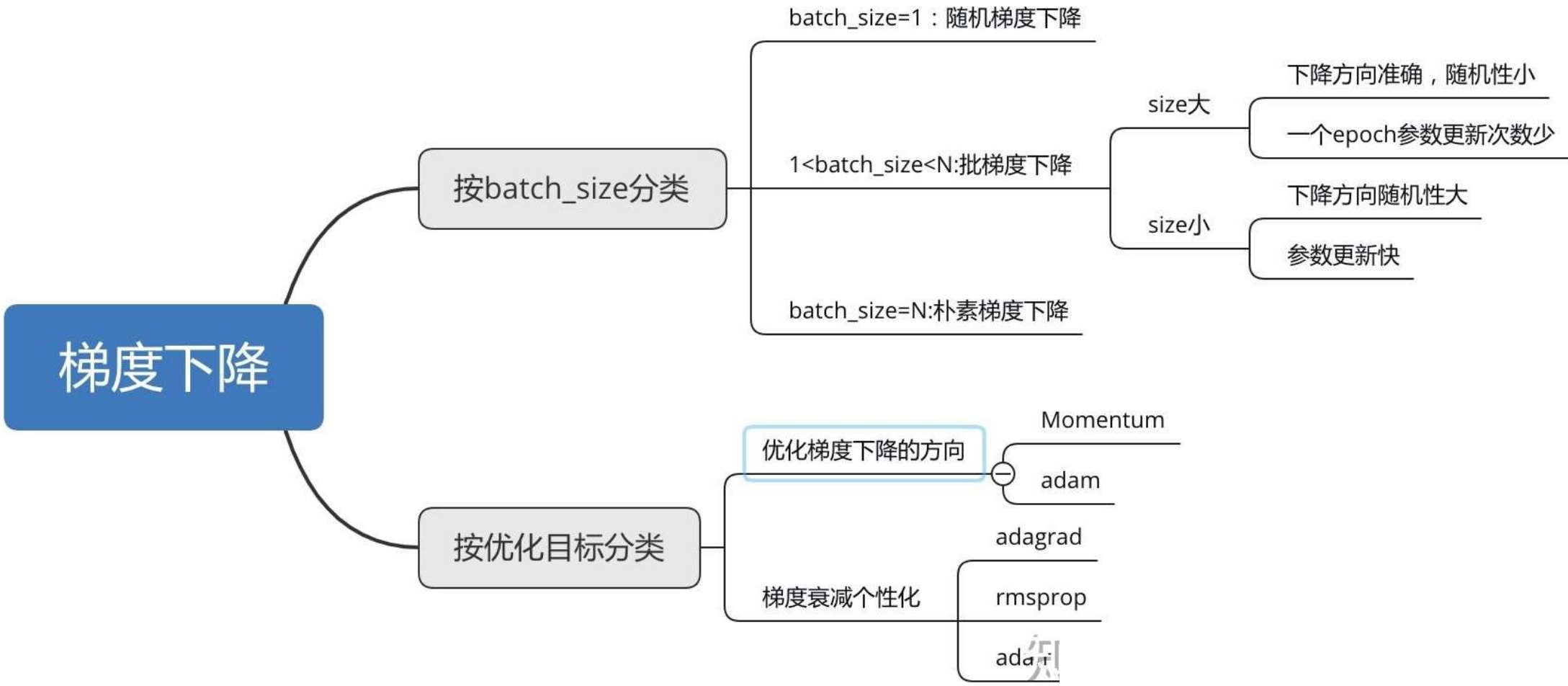
Gradient Descent



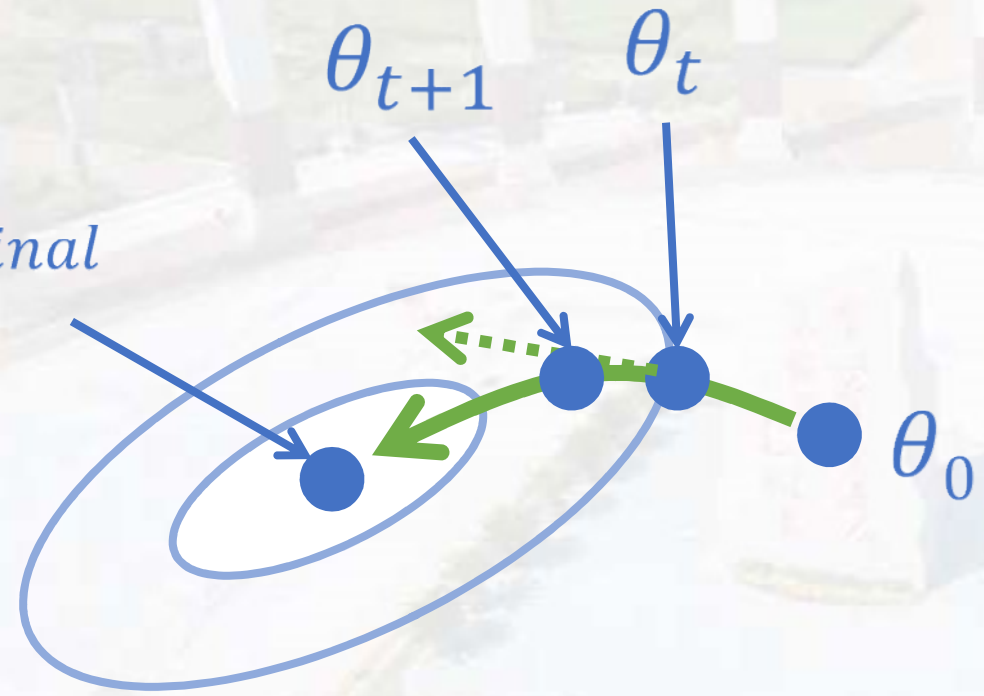
02

Batch-size Optimization


20
10
0
-10
-20
-30
-40
-50
-60
-5
-4
-3
-2
-1
0
1
2
3
4
5


梯度下降
按batch_size分类
batch_size=1：随机梯度下降
1<batch_size<N:批梯度下降
size大
下降方向准确，随机性小
一个epoch参数更新次数少
size小
下降方向随机性大
参数更新快
batch_size=N:朴素梯度下降
按优化目标分类
优化梯度下降的方向
Momentum
adam
梯度衰减个性化
adagrad
rmsprop
adam

Gradient descent



- Initialize θ_0 randomly
- For t in 0,…, T_{maxiter}

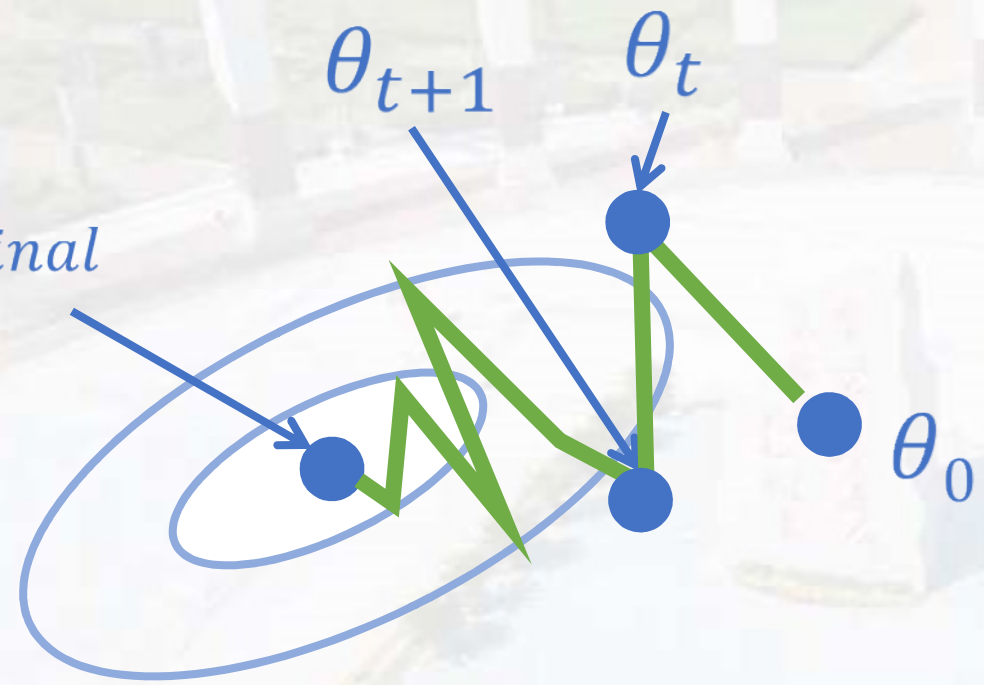
$$\theta^{t+1} = \theta^t - \eta_t \cdot \nabla L(\theta^t)$$

Gradient of the objectiv

Learning rate (step size)

- computation of $\nabla L(\theta^t)$ requires a full sweep over the training dat
- Per-iteration comp. cost = O(n)

Stochastic gradient descent (SGD)



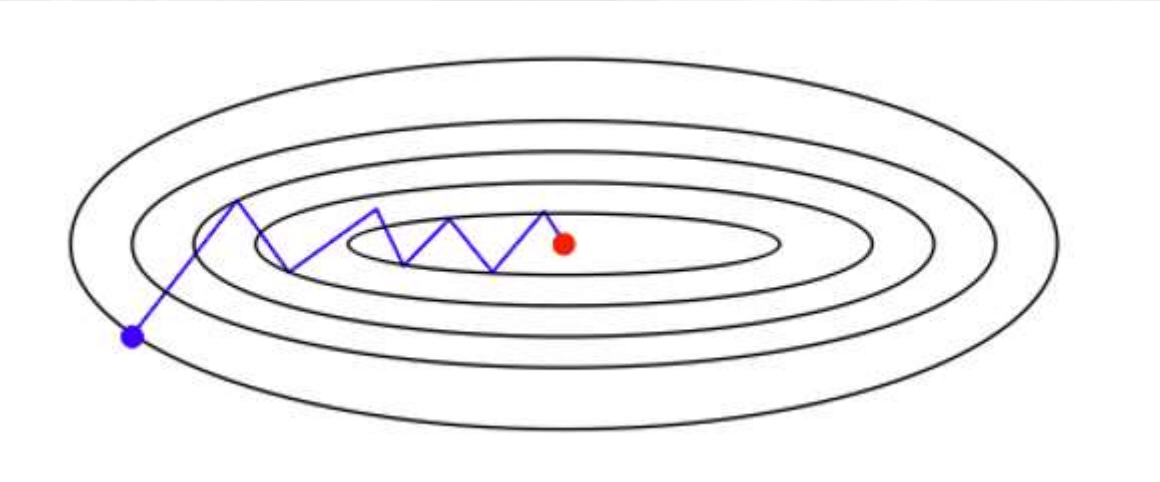
- Initialize θ_0 randomly
- For t in 0,..., T_{maxiter}

$$\theta^{t+1} = \theta^t - \eta_t \cdot \nabla Loss(f_\theta(x_i), y$$

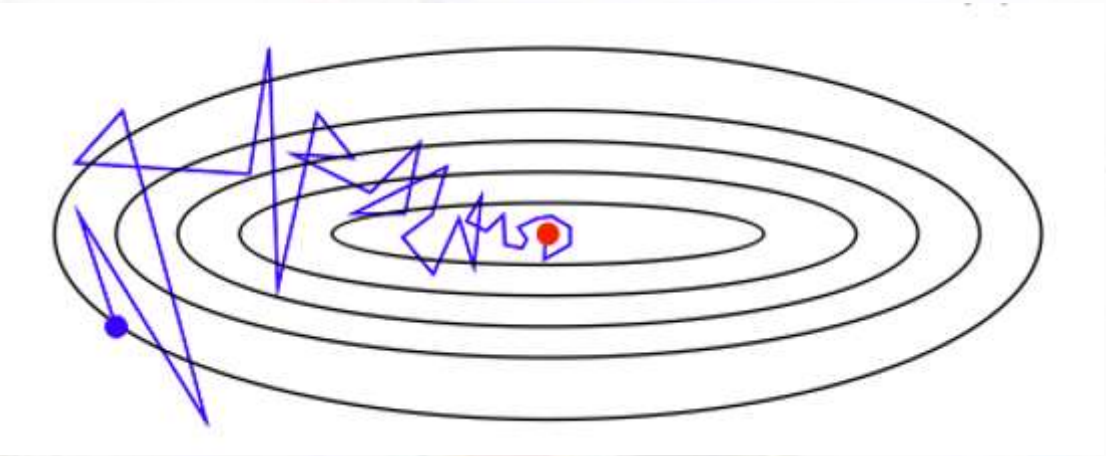
Stochastic gradient

where index i is chosen randomly

- computation of $\nabla Loss(\dots)$ requires only one training example
- Per-iteration comp. cost = O(1)



Gradient descent



Stochastic gradient descent (SGD)

随机梯度下降法（Stochastic Gradient Descent, SGD）

随机梯度下降算法每次从训练集中随机选择一个样本来进行学习；

优点：每次只随机选择一个样本来更新模型参数，因此每次的学习是非常快速的，并且可以进行在线更新；

SGD波动带来的好处，在类似盆地区域，即很多局部极小值点，那么这个波动的点可能会使得优化的方向从当前的局部极小值点调到另一个更好的局限极小值点，这样便可能对于非凹函数，最终收敛于一个较好的局部极值点，甚至全局极值点。

缺点：每次更新可能并不会按照正确的方向进行，因此会带来优化波动，使得迭次数增多，即收敛速度变慢。

◆梯度下降法计算推导：

- 判别函数：$f_w(x^{(i)}) = w^T x^{(i)}$
- 1.随机梯度下降（Stochastic Gradient Descent, SGD）：
- 目标函数：$$L^{(i)}(w) = \frac{1}{2}(f_w(x^{(i)}) - y^{(i)})^2$$

$$\frac{\nabla L^{(i)}(w)}{\nabla w_j} = (f_w(x^{(i)}) - y^{(i)})x_j^{(i)}$$

$$w_j := w_j - \eta(f_w(x^{(i)}) - y^{(i)})x_j^{(i)}$$

◆批量梯度下降法（Batch Gradient Descent, BGD）

- 每次使用全部的训练样本来更新模型参数/学习；
- 优点：每次更新都会朝着正确的方向进行，最后能够保证收敛于极值点；
- 缺点：每次学习时间过长，并且如果训练集很大以至于需要消耗大量的内存，不能进行在线模型参数更新。

梯度下降法计算推导：

判别函数：

2. 批量梯度下降(Batch Gradient Descent, BGD)：

目标函数：$$L(w)=\frac{1}{M}\sum_{i=1}^{M}(f_w(x^{(i)})-y^{(i)})^2$$

$$\frac{\nabla L(w)}{\nabla w_j}=\frac{1}{M}\sum_{i=1}^{m}(f_w(x^{(i)})-y^{(i)})x_j^{(i)}$$

$$w_j := w_j-\eta\frac{1}{M}\sum_{i=1}^{m}(f_w(x^{(i)})-y^{(i)})x_j^{(i)}$$

小批量梯度下降法（Mini-batch Gradient Descent, MBGD）

小批量梯度下降综合了batch梯度下降与stochastic梯度下降，在每次更新速度与更新次数中间一个平衡，其每次更新从训练集中随机选择k (k<m)个样本进行学习；

优点：

相对于随机梯度下降，Mini-batch梯度下降降低了收敛波动性，即降低了参数更新的方差，使得更新更加稳定；

相对于批量梯度下降，其提高了每次学习的速度；

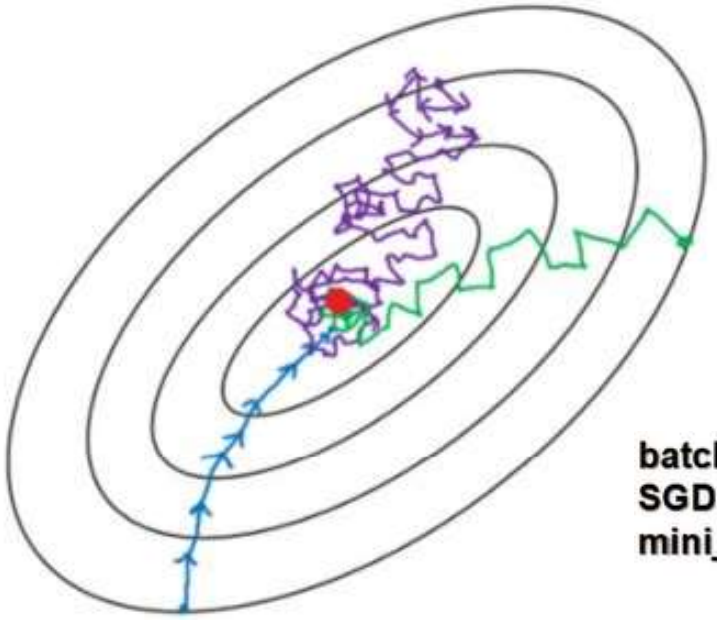
MBGD不用担心内存瓶颈从而可以利用矩阵运算进行高效计算；

◆小批量梯度下降法推导：

- 判别函数：$f_w(x^{(i)}) = w^T x^{(i)}$
- 3.小批量梯度下降(Mini-Batch Gradient Descent, MBGD)
- 目标函数：$L(w) = \frac{1}{k}\sum_{i=1}^{k}(f_w(x^{(i)}) - y^{(i)})^2 \quad k < m$

$$\frac{\nabla L(w)}{\nabla w_j} = \frac{1}{k}\sum_{i=1}^{k}(f_w(x^{(i)}) - y^{(i)})x_j^{(i)} \quad k < m$$

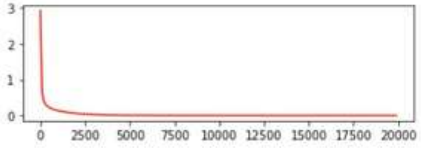
$$w_j := w_j - \eta\frac{1}{k}\sum_{i=1}^{k}(f_w(x^{(i)}) - y^{(i)})x_j^{(i)}$$



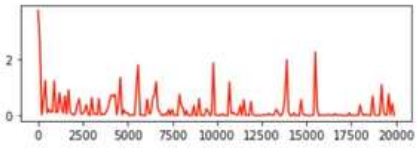
Batch gradient descent
Mini-batch gradient Descent
Stochastic gradient descent

batch：loss稳定的减少，曲线非常平滑，但是训练时间长。
SGD：曲线振幅大，不稳定
mini_batch :相对稳定些，是我们现在常用的方法

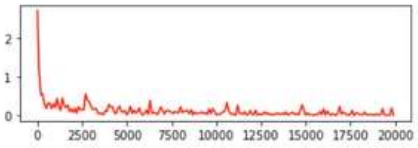
纵坐标为损失值
横坐标为训练轮数



batch



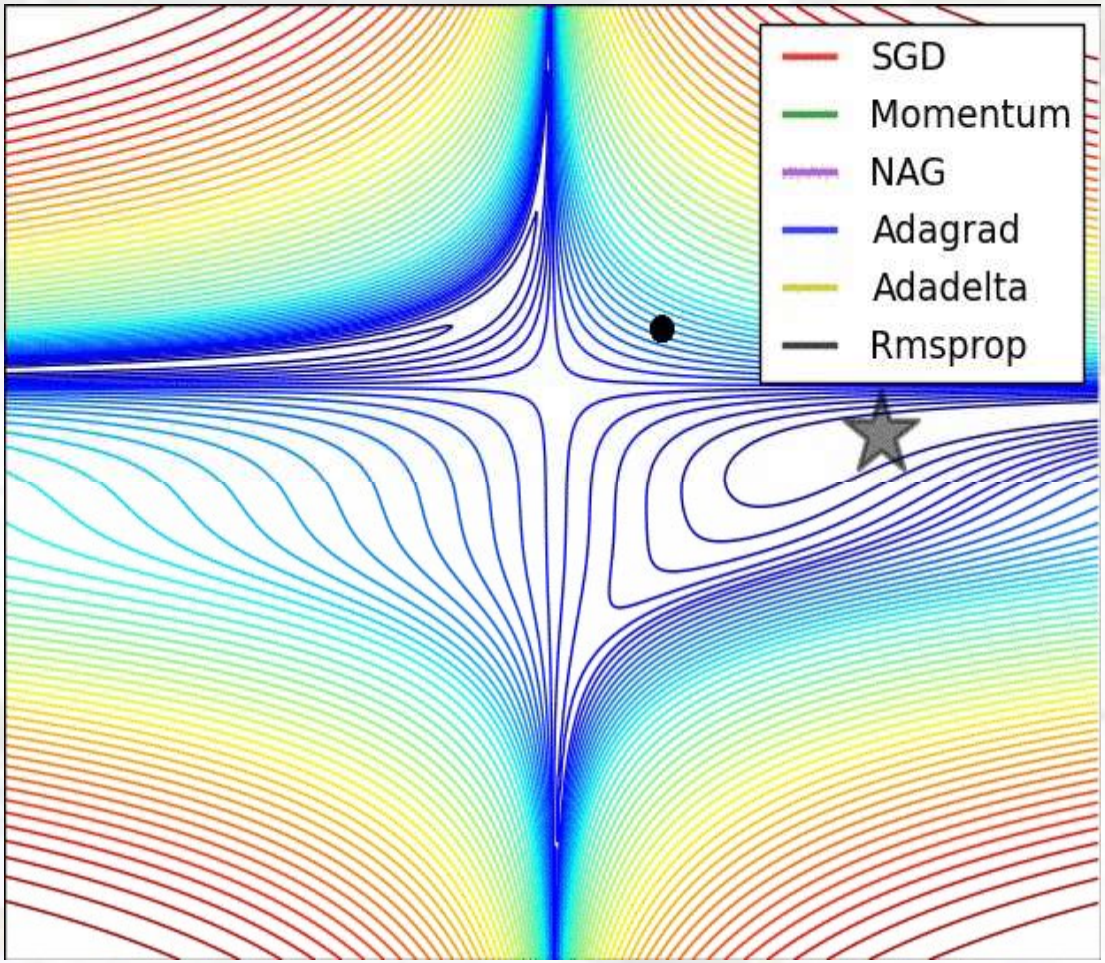
SGD



mini_batch


03
Category Optimization

Category Optimization



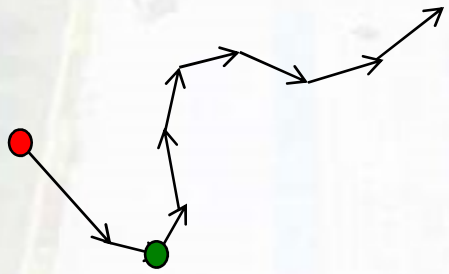
(image credits to Alec Radford)

The intuition behind the momentum method

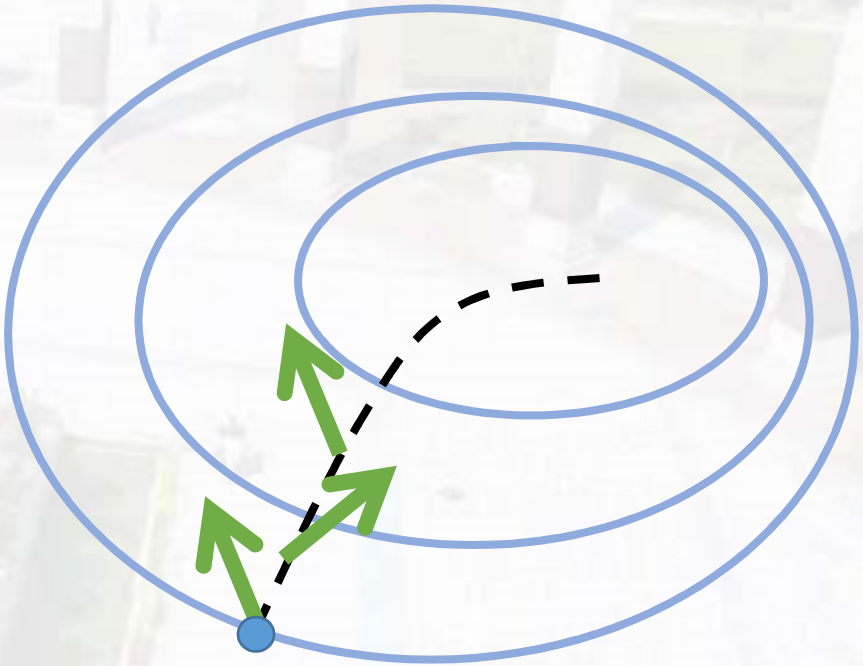
Imagine a ball on the error surface. The location of the ball in the horizontal plane represents the weight vector.

- The ball starts off by following the gradient, but once it has velocity, it no longer does steepest descent.
- Its momentum makes it keep going in the previous direction.

- It damps oscillations in directions of high curvature by combining gradients with opposite signs.
- It builds up speed in directions with a gentle but consistent gradient.



- Momentum SGD: improves SGD by incorporating “momentum”



Category Optimization

```
Gradient descent update
+= - learning_rate * dx
```

```
omentum update
 mu * v - learning_rate * dx # integrate velocity
= v # integrate position
```

$$\mathbf{v}_t = \alpha \mathbf{v}_{t-1} - \eta \nabla f(x_{t-1})$$

$$x_t = x_{t-1} + \mathbf{v}_t$$

- Physical interpretation as ball rolling down the loss function + friction (mu coefficient).
- mu = usually ~0.5, 0.9, or 0.99 (Sometimes annealed over time, e.g. from 0.5 -> 0.99)

The equations of the momentum method

$$\mathbf{v}(t) = \alpha \mathbf{v}(t-1) - \eta \frac{\partial E}{\partial \mathbf{w}}(t)$$

The effect of the gradient is to increment the previous velocity. The velocity also decays by a which is slightly less then 1.

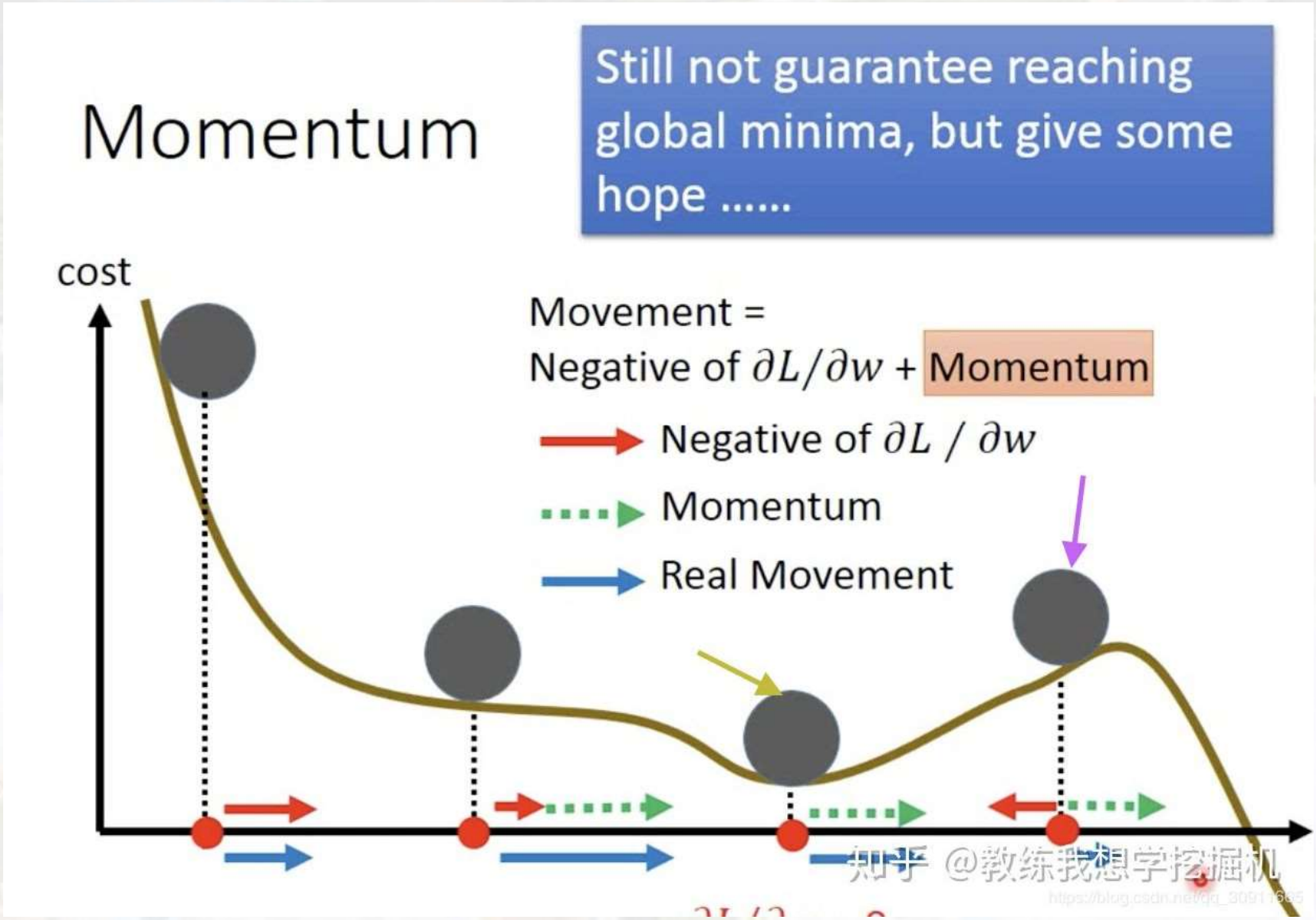
$$\Delta \mathbf{w}(t) = \mathbf{v}(t)$$

The weight change is equal to the current velocity.

$$= \alpha \mathbf{v}(t-1) - \eta \frac{\partial E}{\partial \mathbf{w}}(t)$$

$$= \alpha \Delta \mathbf{w}(t-1) - \eta \frac{\partial E}{\partial \mathbf{w}}(t)$$

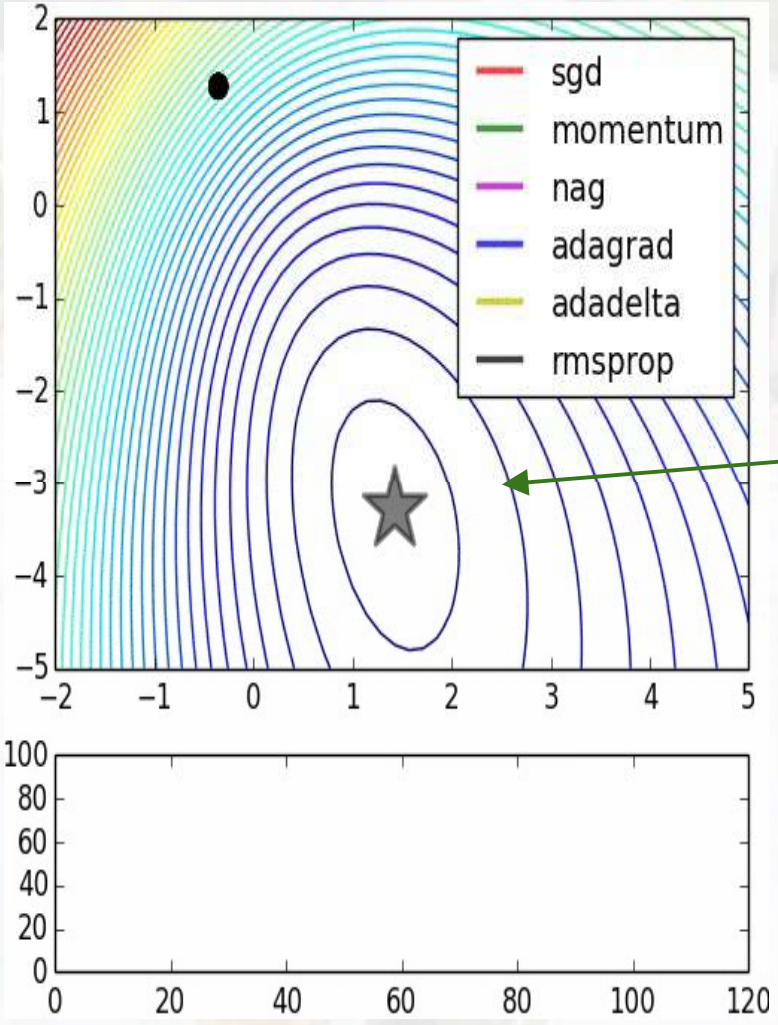
The weight change can be expressed in terms of the previous weight change and the current gradient.


Momentum
Still not guarantee reaching global minima, but give some hope ……
cost
Movement =
Negative of $\partial L/\partial w$ + Momentum
Negative of $\partial L / \partial w$
Momentum
Real Movement
知乎 @教练我想学挖掘机

SGD

vs

Momentum

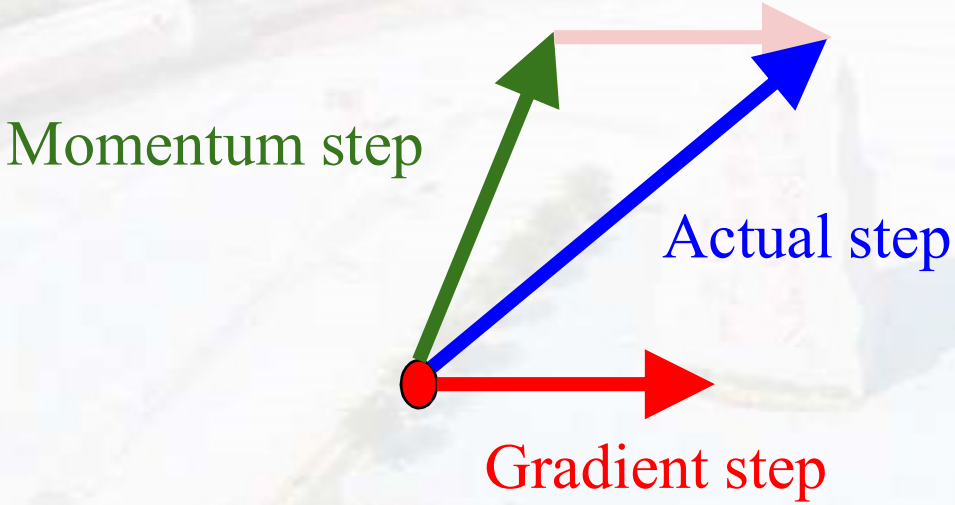


notice momentum overshooting the target, but overall getting to the minimum much faster than vanilla SGD.

Category Optimization

```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

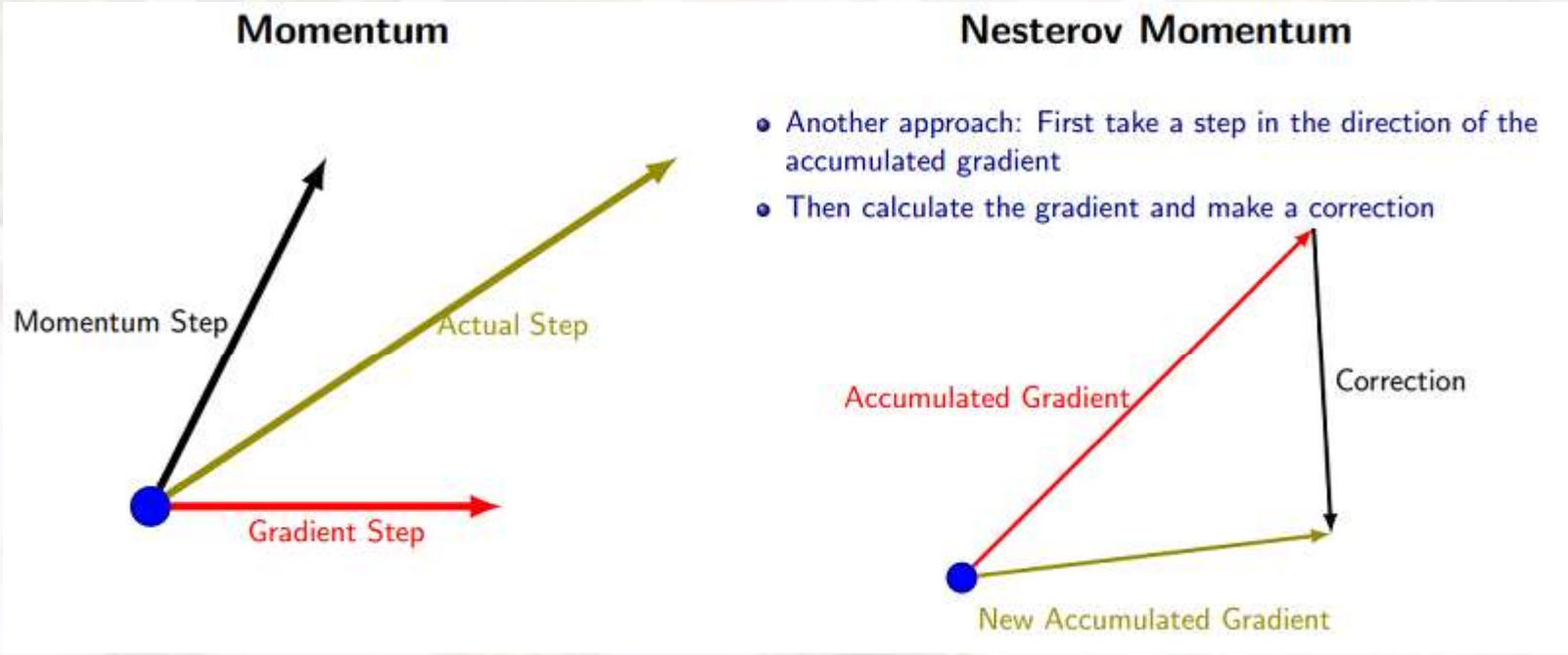
Ordinary momentum update:



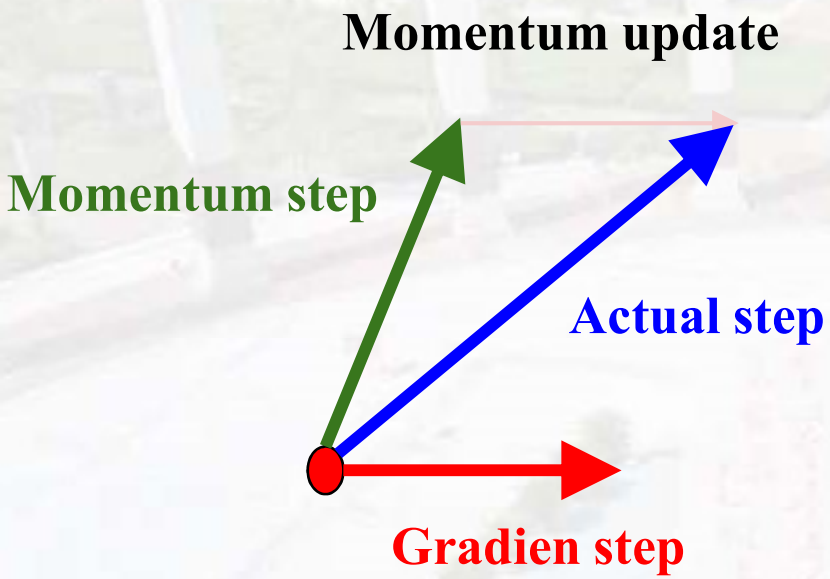
A better type of momentum (Nesterov 1983)

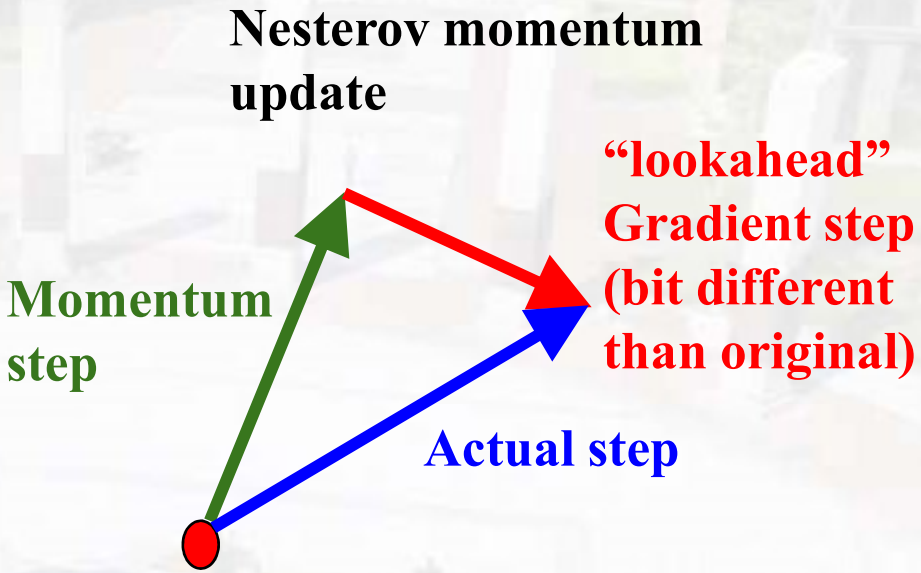
- The standard momentum method first computes the gradient at the current location and then takes a big jump in the direction of the updated accumulated gradient.
- Ilya Sutskever (2012 unpublished) suggested a new form of momentum that often works better.
  - Inspired by the Nesterov method for optimizing convex functions.
- First make a big jump in the direction of the previous accumulated gradient.
- Then measure the gradient where you end up and make a correction.
  - Its better to correct a mistake after you have made it!

A picture of the Nesterov method



Nesterov Momentum是对Momentum的改进，可以理解为nesterov动量在标准动量方法中添加了一个校正因子。





Nesterov: the only difference...

$$\mathbf{v}_t = \alpha \mathbf{v}_{t-1} - \eta \nabla f(x_{t-1} + \alpha \mathbf{v}_{t-1})$$

$$x_t = x_{t-1} + \mathbf{v}_t$$

More optimization

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\theta_{t-1} + \mu v_{t-1})$$

$$\theta_t = \theta_{t-1} + v_t$$

Slightly inconvenient…
usually we have :

$$\theta_{t-1}, \nabla f(\theta_{t-1})$$

Variable transform and rearranging saves the day:

$$\phi_{t-1} = \theta_{t-1} + \mu v_{t-1}$$

AdaGrad

```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

$$S_t = S_{t-1} + \nabla f^2(x_t)$$

$$x_{t+1} = x_t - \frac{\eta}{\sqrt{S_t + \varepsilon}} \nabla f(x_t)$$

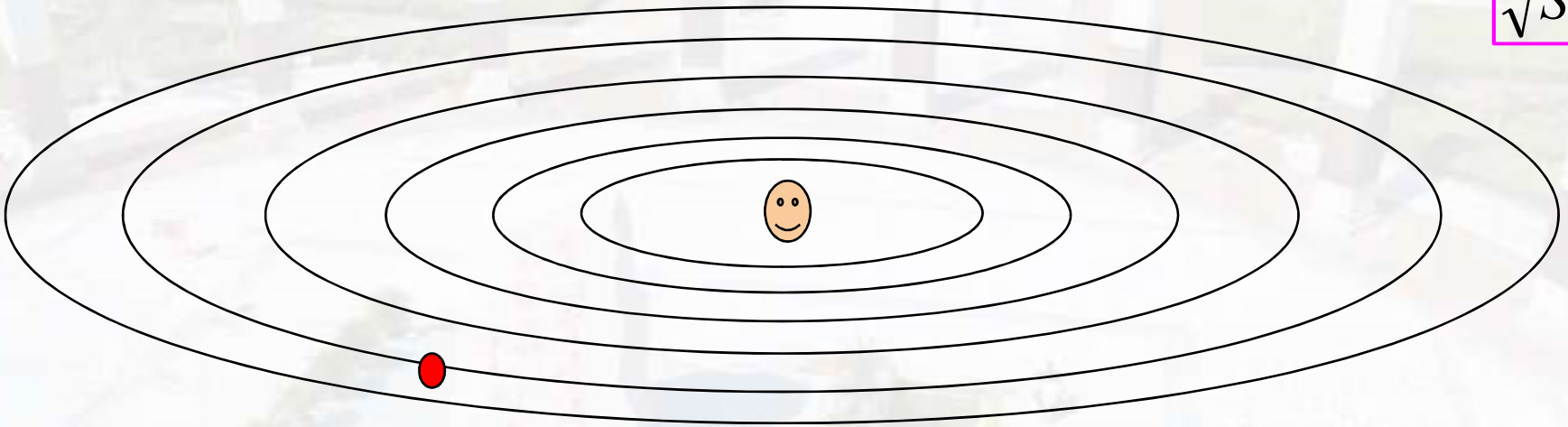
Added element-wise scaling of the gradient based on the historical sum of squares in eac
dimension.

通常，我们在每一次更新参数时，对于所有的参数使用相同的学习率。而AdaGrad
算法的思想是：每一次更新参数时（一次迭代），不同的参数使用不同的学习率

```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

$$S_t = S_{t-1} + \nabla f^2(x_t)$$

$$x_{t+1} = x_t - \frac{\eta}{\sqrt{S_t + \varepsilon}} \nabla f(x_t)$$



优点：对于梯度较大的参数，意味着学习率会变得较小。而对于梯度较小的参数，则效果相反。这样就可以使得参数在平缓的地方下降的稍微快些，不至于徘徊不前。

缺点：由于是累积梯度的平方，到后面累积的比较大，会导致导致梯度消失。

Gradient Vanishing

$$S_t \to +\infty \longrightarrow \frac{\eta}{\sqrt{S_t + \varepsilon}} \to 0$$

rmsprop: A mini-batch version of rprop

- rprop is equivalent to using the gradient but also dividing by the size of the gradient.
  - The problem with mini-batch rprop is that we divide by a different number for each mini-batch. So why not force the number we divide by to be very similar for adjacent mini-batches?
- rmsprop: Keep a moving average of the squared gradient for each weight

$$MeanSquare(w, t) = 0.9\, MeanSquare(w, t-1) + 0.1 \left(\partial E / \partial w(t)\right)^2$$

- Dividing the gradient by $\sqrt{MeanSquare(w, t)}$ makes the learning work much better (Tijmen Tieleman, unpublished).

rmsprop: A mini-batch version of rprop

$$x_{t+1} = x_t - \frac{\eta}{\sqrt{S_t + \varepsilon}} \nabla f(x_t) \qquad \varepsilon = 10^{-6}$$

$$S_t = \beta S_{t-1} + (1-\beta)\nabla f^2(x_t) \qquad \beta = 0.999$$

什么用： 1.对历史梯度与当前梯度进行加权平均　　2. 减小久远梯度信息的影响力

$$\begin{aligned} s_t &= \beta(\beta s_{t-2} + (1-\beta)g_{t-1}^2) + (1-\beta)g_t^2 \\ &= \beta^2 s_{t-2} + (1-\beta)(g_t^2 + \beta g_{t-1}^2) \\ &\cdots\cdots \\ &= (1-\beta)(g_t^2 + \beta g_{t-1}^2 + \beta^2 g_{t-2}^2 + \cdots) \end{aligned}$$

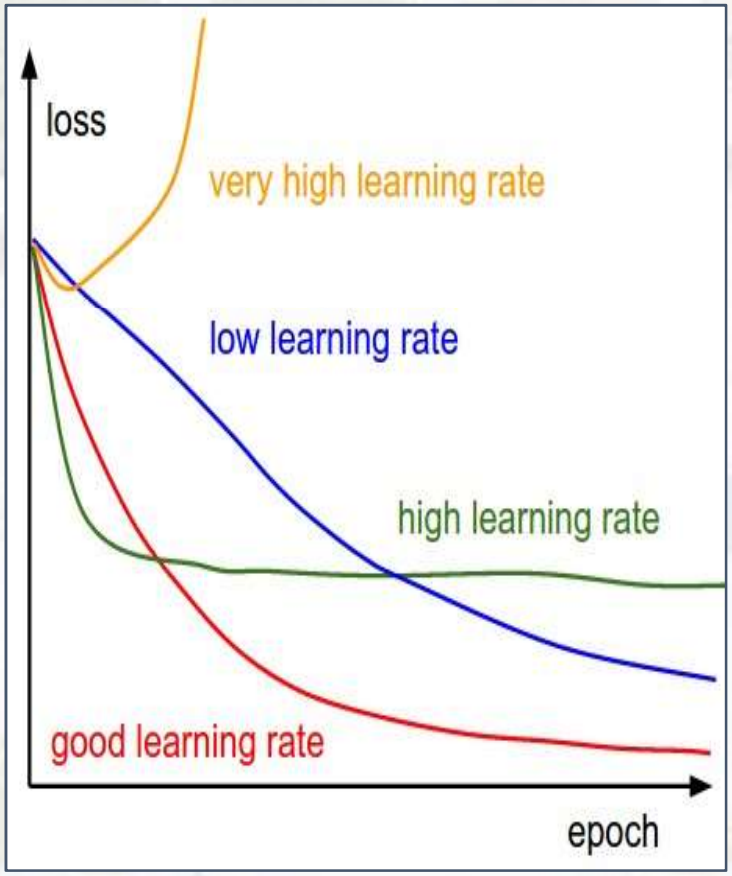
$$\because \sum_{i=0}^{\infty} \beta^i = \frac{1}{1-\beta}$$

∴ 权重和为1

指数加权移动平均法

Category Optimization

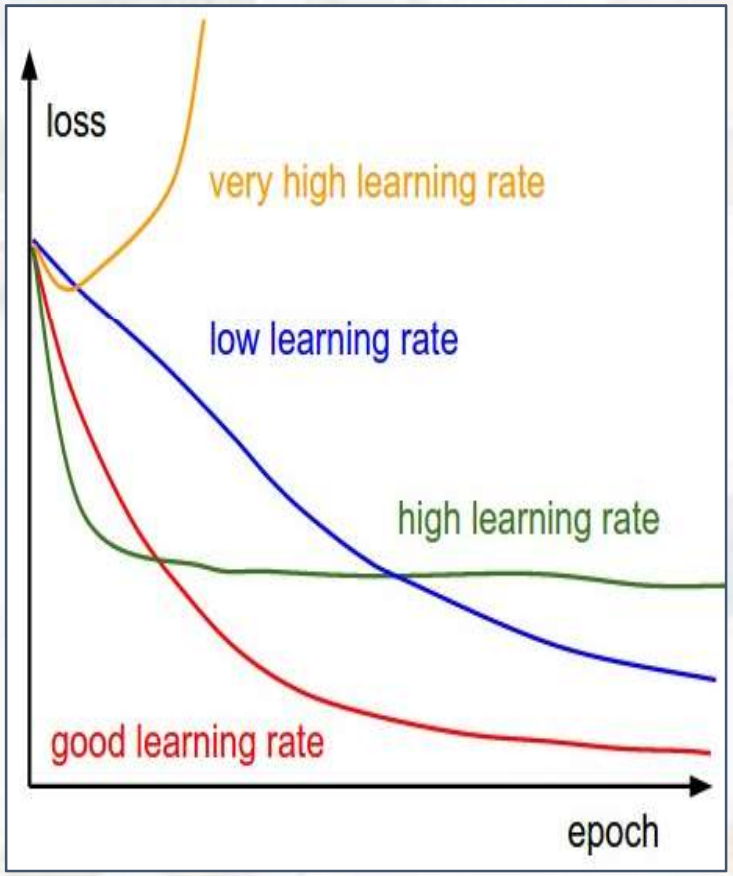
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



Q: Which one of these learning rates is best to use?

Category Optimization

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



=> Learning rate decay over time!

step decay:
e.g. decay learning rate by half every few epochs.

exponential decay:

$$\alpha = \alpha_0 e^{-kt}$$

1/t decay:

$$\alpha = \alpha_0 / (1 + kt)$$

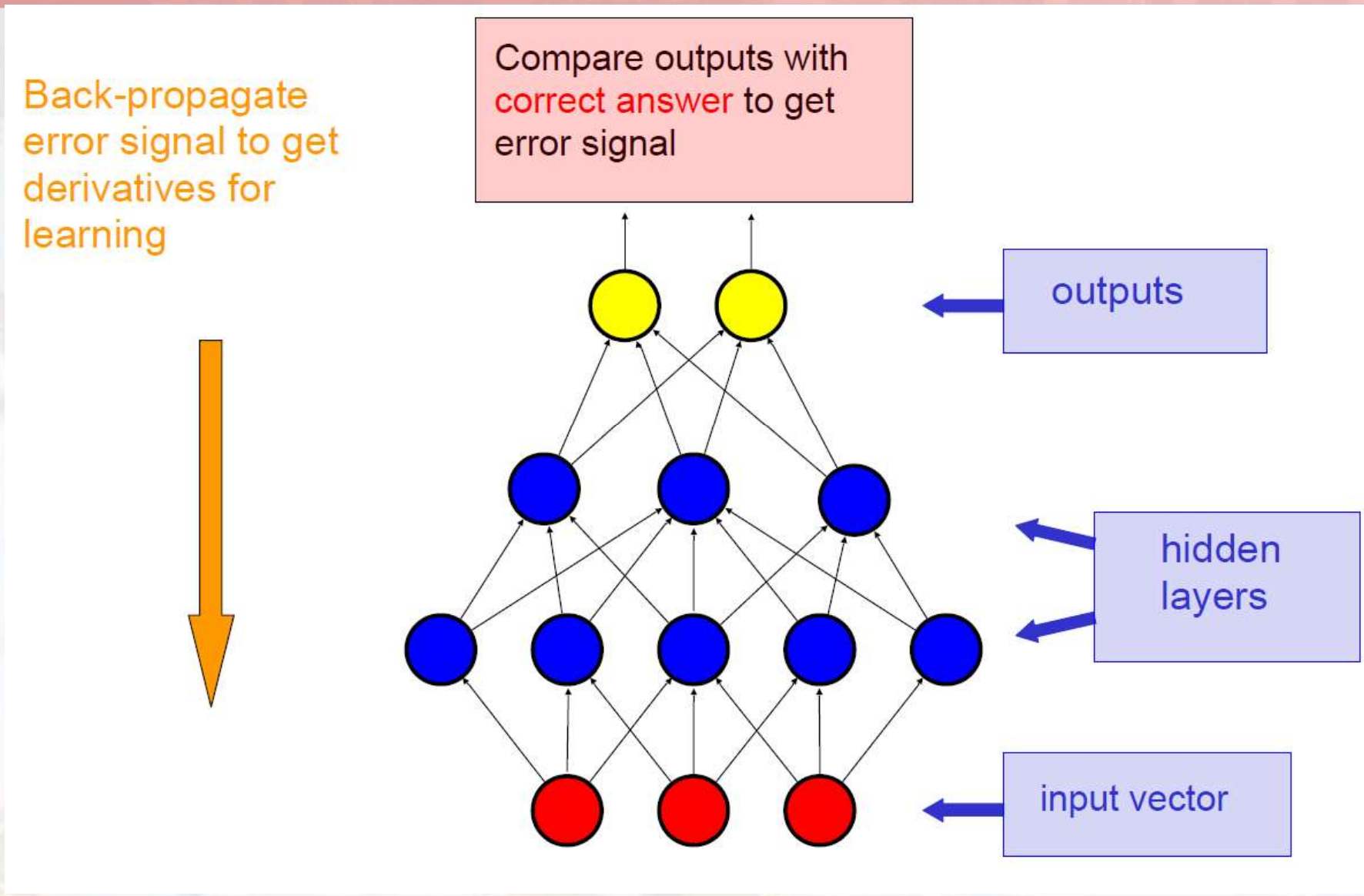
04

Back-propagation Algorithm

The Back-propagation Algorithm

- **1986**: the solution to multi-layer ANN weight update rediscovered
- Conceptually simple - the global error is backward propagated to network nodes, weights are modified proportional to their contribution
- Most important ANN learning algorithm
- Become known as ***back-propagation*** because the error is send back through the network to correct all weights

Back-propagation Algorithm



The Back-propagation Algorithm

- Like the Perceptron - calculation of error is based on difference between target and actual output:

$$E = \frac{1}{2}\sum (y_p - y_a)^2$$

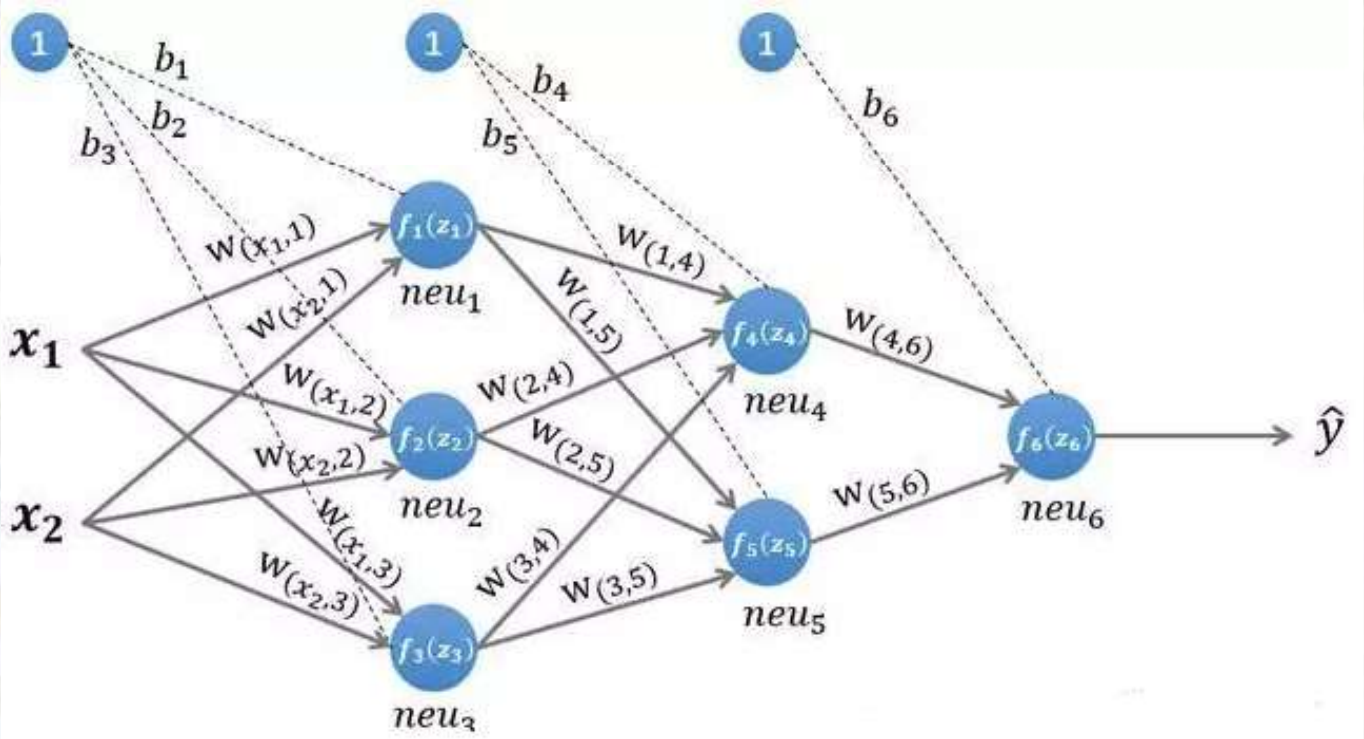
- However in BP it is the rate of change of the error which is the important feedback through the network

generalized delta rule $\Delta w_{ij} = -\eta \frac{\delta E}{\delta w_{ij}}$

- Relies on the sigmoid activation function for communication

The Back-propagation Algorithm

- How to calculate the following BP algorithm in the feed-forward procedure.



The Back-propagation Algorithm

The weight of all layers:

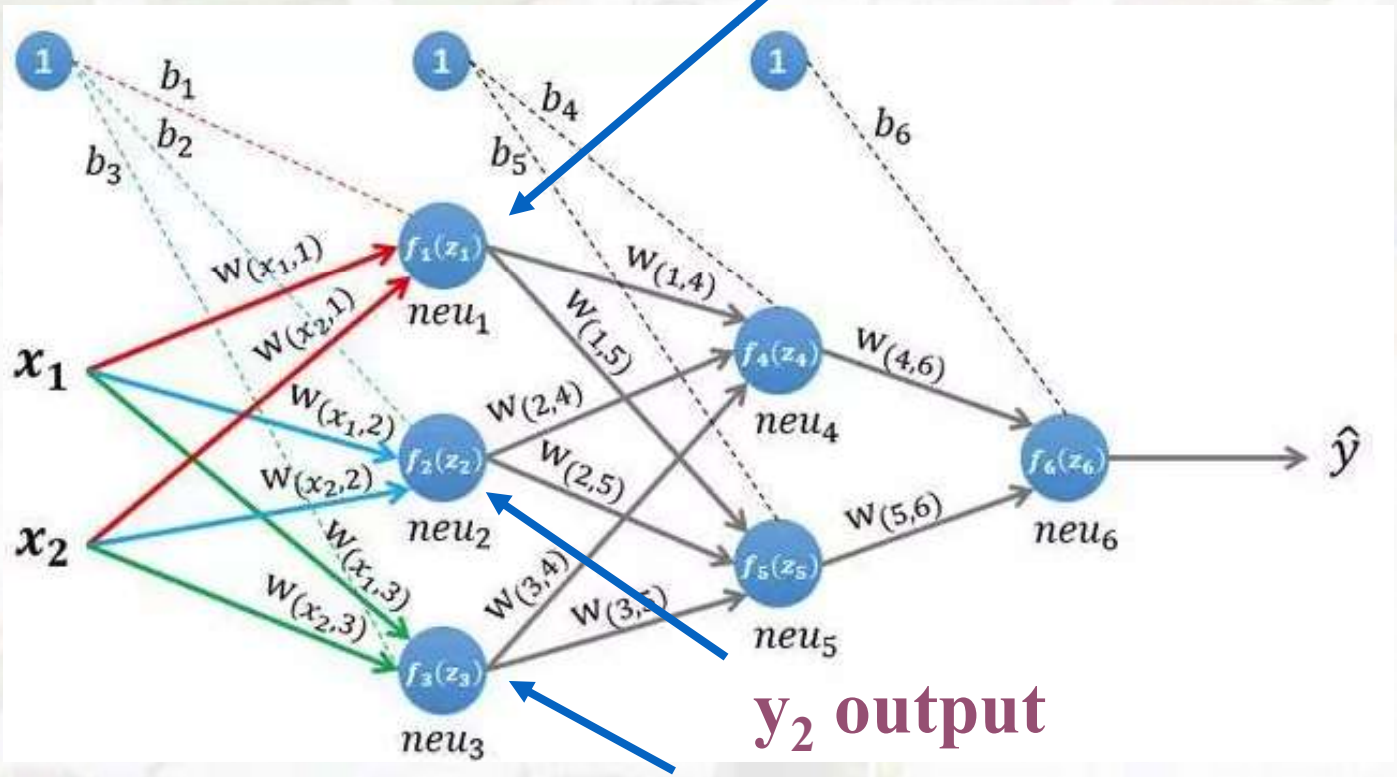
$$W^{(1)} = \begin{bmatrix} w_{(x_1,1)}, w_{(x_2,1)} \\ w_{(x_1,2)}, w_{(x_2,2)} \\ w_{(x_1,3)}, w_{(x_2,3)} \end{bmatrix}, \quad b^{(1)} = [b_1, b_2, b_3]$$

$$W^{(2)} = \begin{bmatrix} w_{(1,4)}, w_{(2,4)}, w_{(3,4)} \\ w_{(1,5)}, w_{(2,5)}, w_{(3,5)} \end{bmatrix}, \quad b^{(2)} = [b_4, b_5]$$

$$W^{3} = \begin{bmatrix} w_{(4,6)}, w_{(5,6)} \end{bmatrix}, \quad b^{(3)} = [b_6]$$

The Back-propagation Algorithm

The First hidden layer



The Back-propagation Algorithm

The First hidden layer

$$Z_1 = W_{(x_1,1)} * x_1 + W_{(x_2,1)} * x_1 + b_1$$

$$Z_2 = W_{(x_1,2)} * x_1 + W_{(x_2,2)} * x_2 + b_2$$

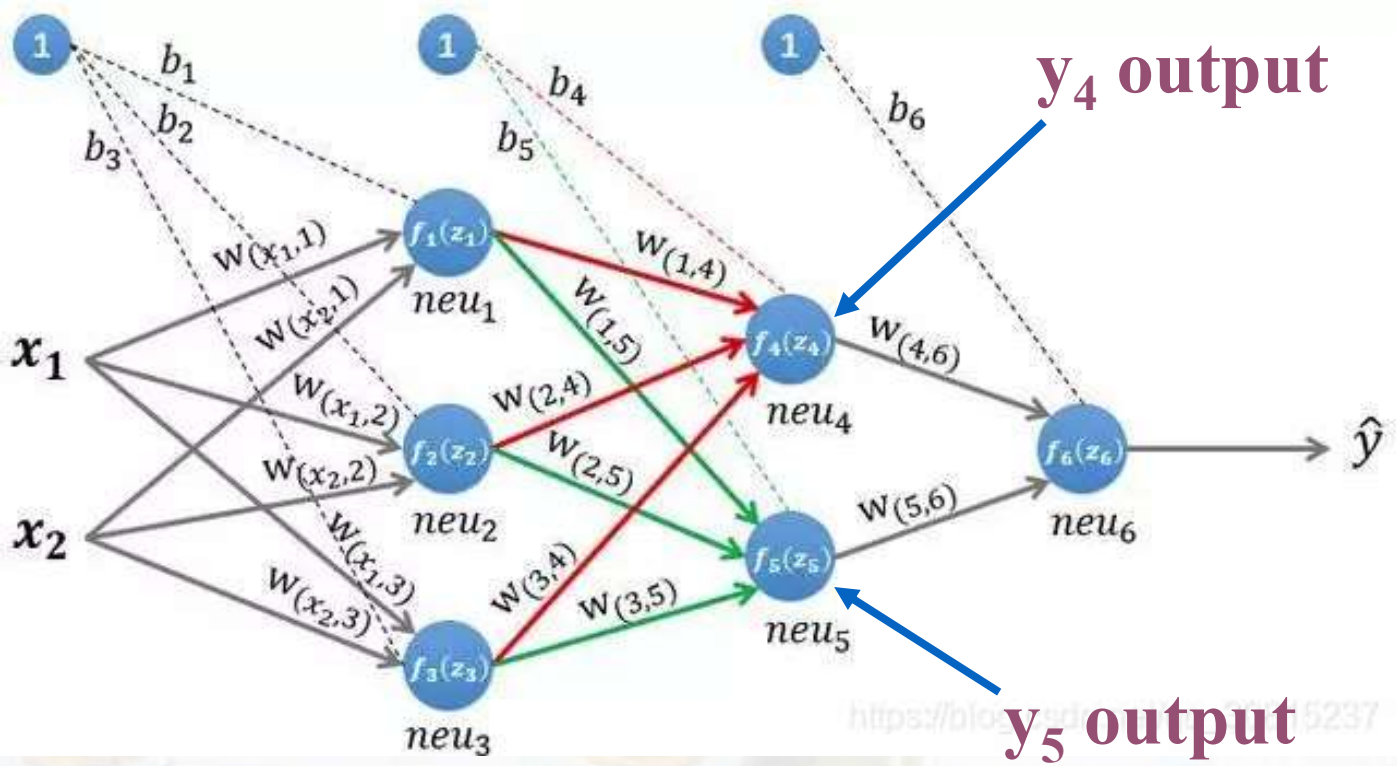
$$Z_3 = W_{(x_1,3)} * x_1 + W_{(x_2,3)} * x_2 + b_3$$

Then, the output will be:

$$y_1 = f(z_1),\ y_2 = f(z_2),\ y_3 = f(z_1),$$

The Back-propagation Algorithm

The Second layer



The Back-propagation Algorithm

The Second hidden layer

$$Z_4 = W_{(1,4)} * y_1 + W_{(2,4)} * y_2 + W_{(3,4)} * y_3 + b_4$$

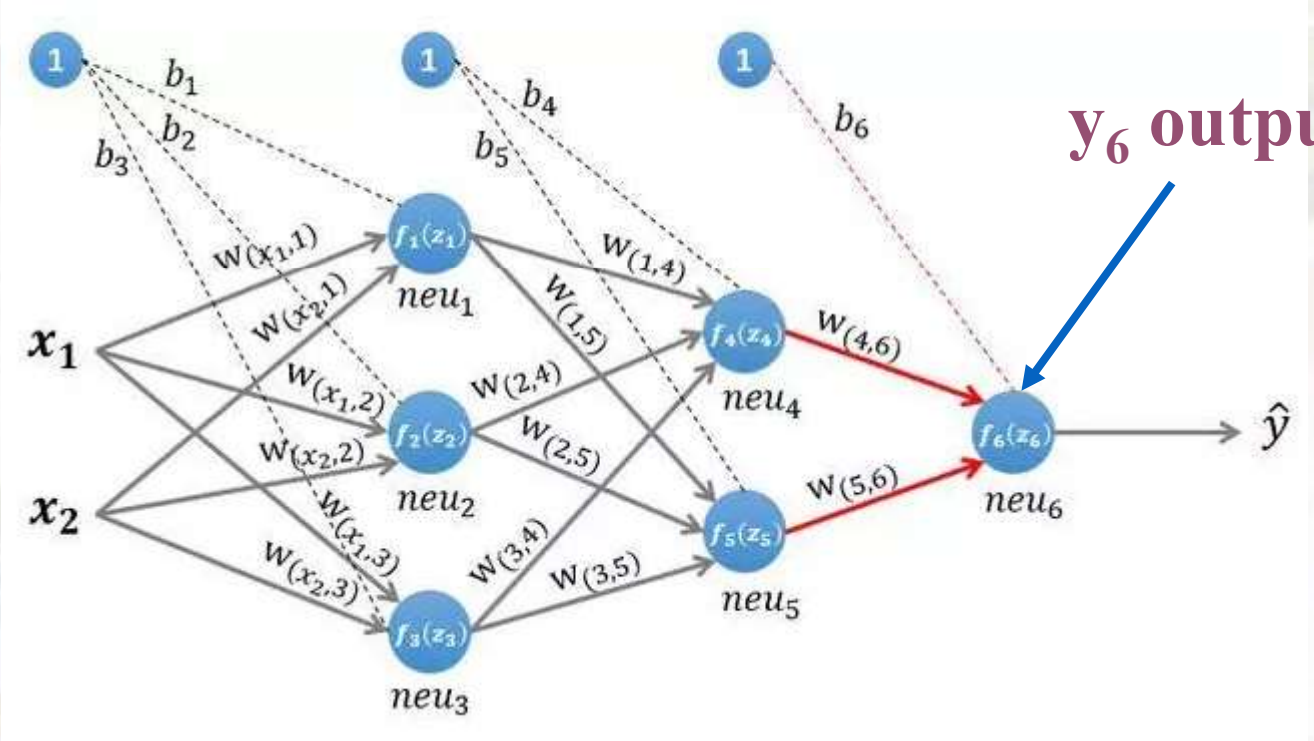
$$Z_5 = W_{(1,5)} * y_1 + W_{(2,5)} * y_2 + W_{(3,5)} * y_3 + b_5$$

Then, the output will be:

$$y_4 = f(z_4),\ y_5 = f(z_5)$$

The Back-propagation Algorithm

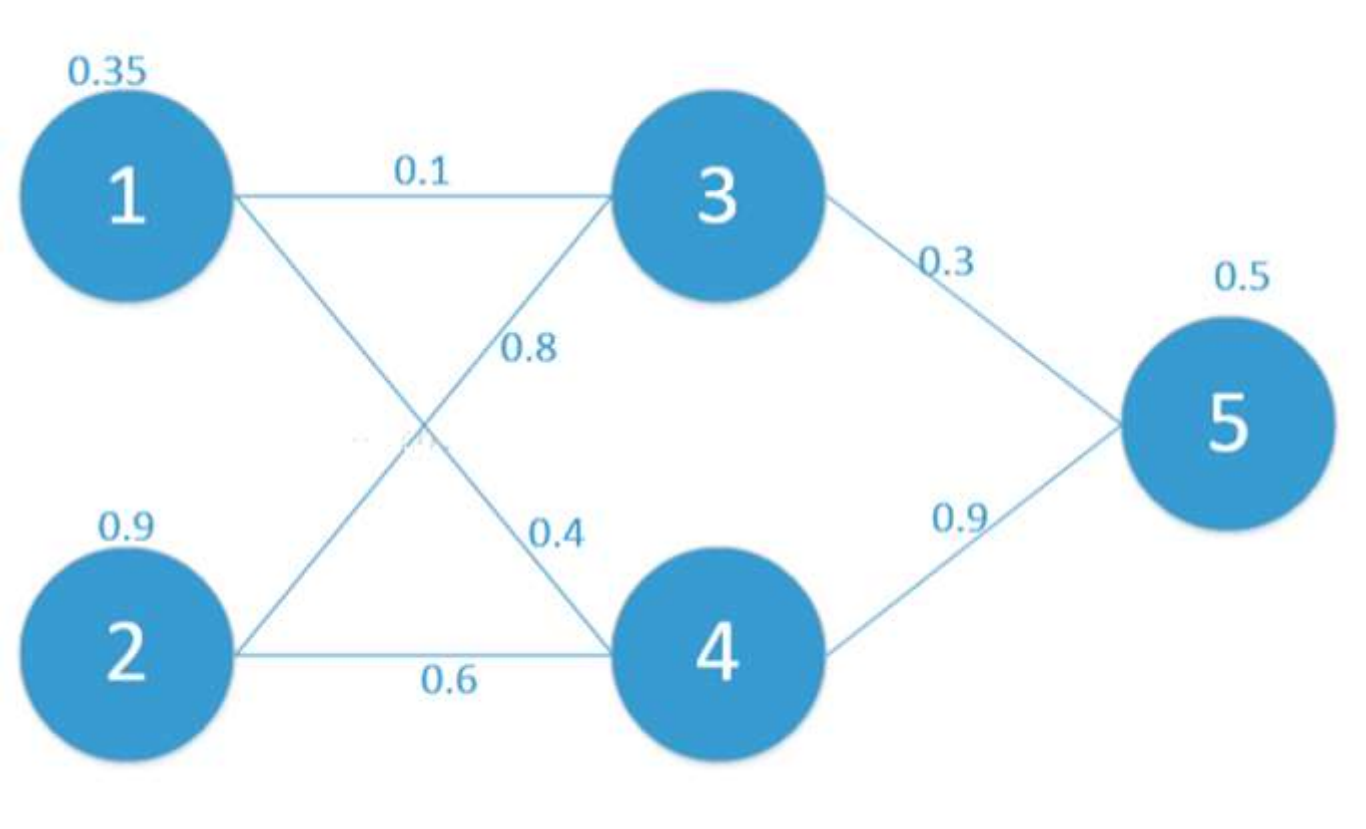
The final layer



$$y_6 = f(W_{(4,6)} * y_4 + W_{(5,6)} * y_5)$$

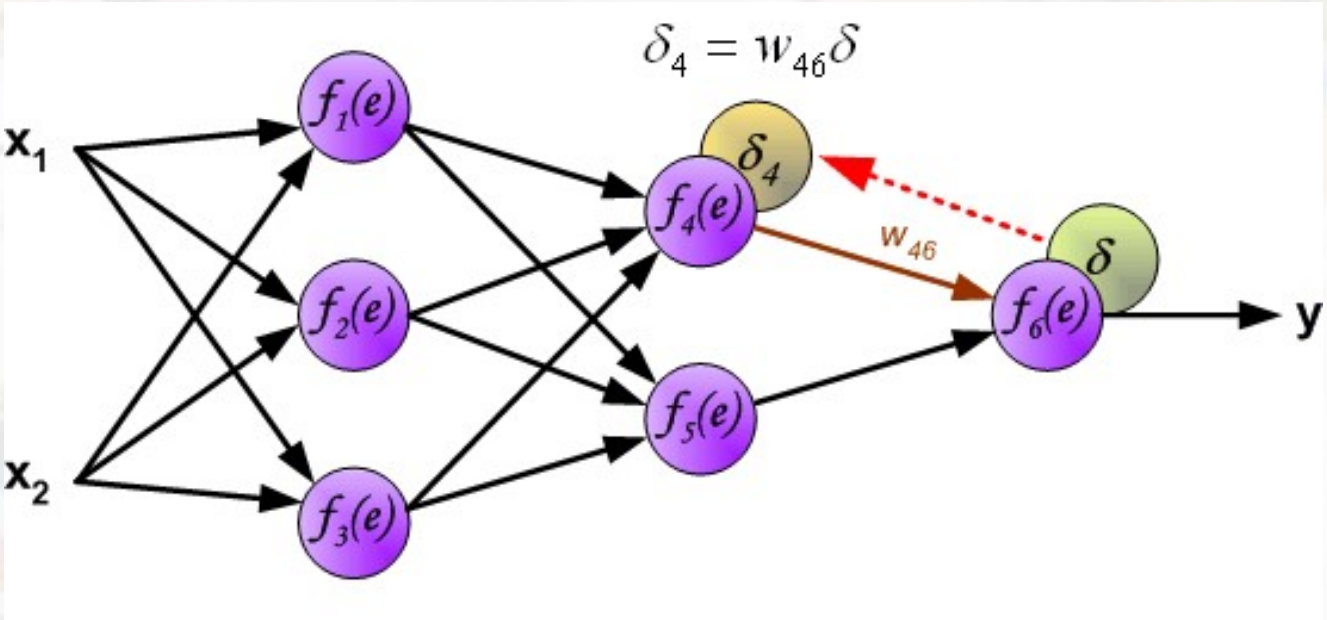
The Back-propagation Algorithm

Example II – Calculate the y5 Error Computing(误差计算)

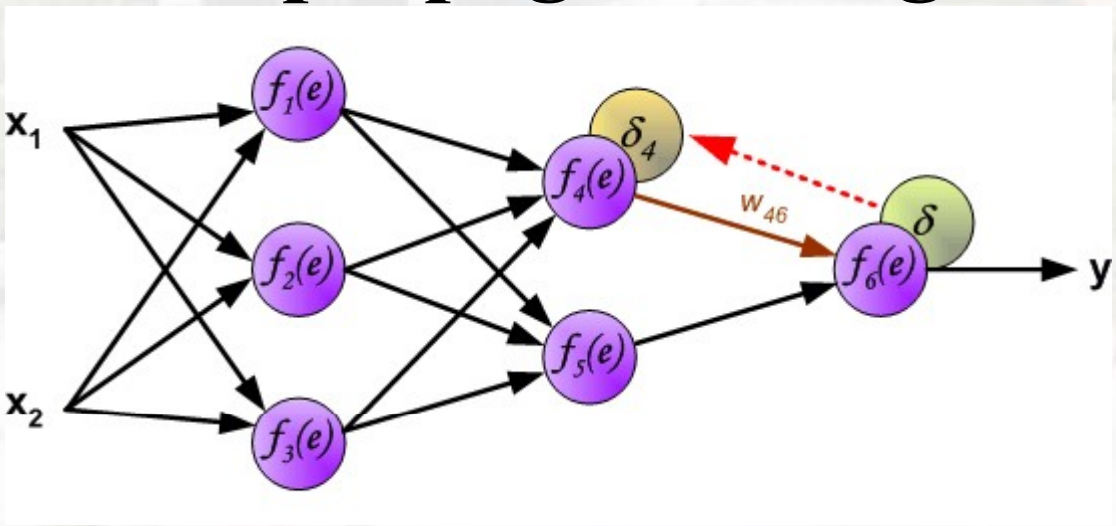


The Back-propagation Algorithm

- How to calculate the updated weight.



The Back-propagation Algorithm



$$y_j = \sum_i \omega_{ij} x_i \qquad y_j = f(z_j) = \frac{1}{1+e^{-z_j}}$$

$\hat{y}_j = O_j$ ***Actual Value*** y_j ***Expected Value***

MSE $E = \frac{1}{2}\sum_j (y_j - \hat{y}_j)^2$

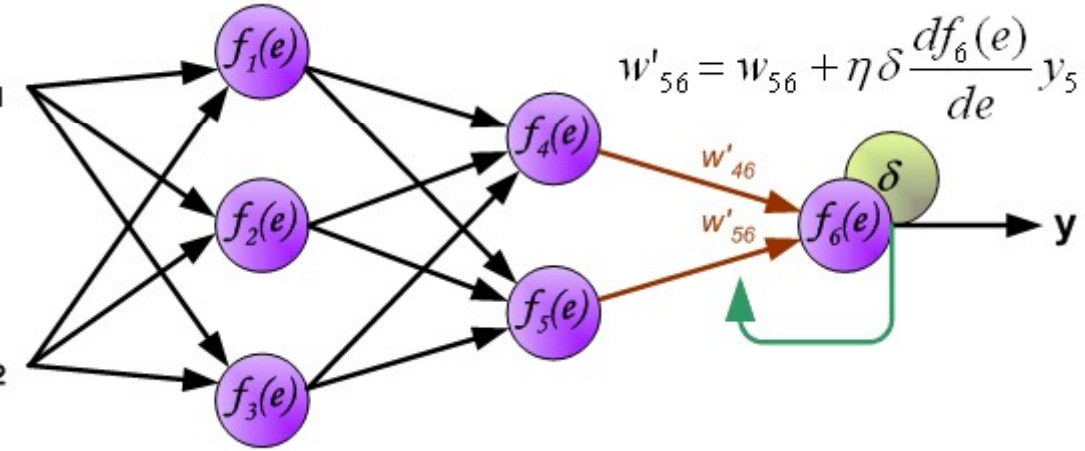
The Back-propagation Algorithm

- It is impossible to compute error signal for internal neurons directly, because output values of these neurons are unknown. For many years the effective method for training multiplayer networks has been unknown. Only in the middle eighties the backpropagation algorithm has been worked out. The idea is to propagate error signal d (computed in single teaching step) back to all neurons, which output signals were input for discussed neuron.

The Back-propagation Algorithm

$$w'_{46} = w_{46} + \eta\,\delta\,\frac{df_6(e)}{de}\,y_4$$

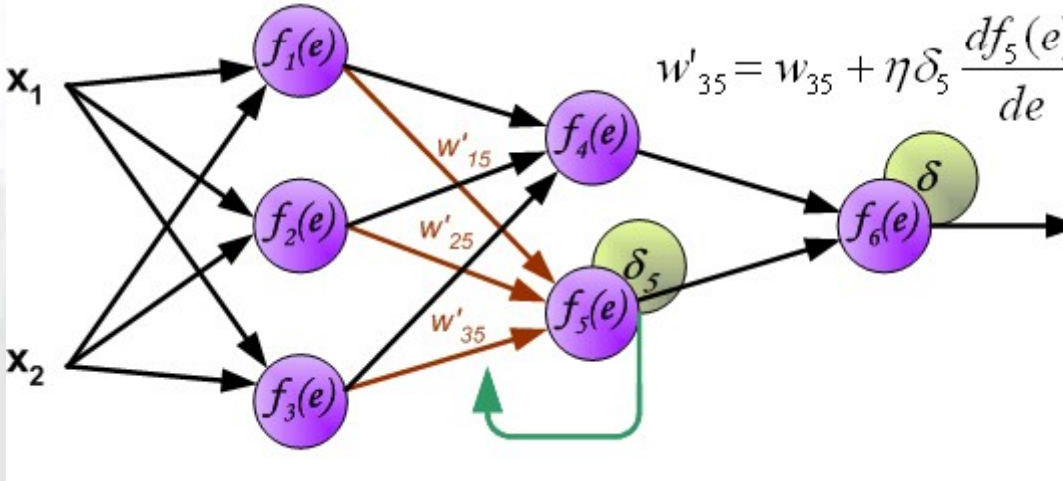
$$w'_{56} = w_{56} + \eta\,\delta\,\frac{df_6(e)}{de}\,y_5$$



$$w'_{15} = w_{15} + \eta\delta_5 \frac{df_5(e)}{de}$$

$$w'_{25} = w_{25} + \eta\delta_5 \frac{df_5(e)}{de}$$

$$w'_{35} = w_{35} + \eta\delta_5 \frac{df_5(e)}{de}$$



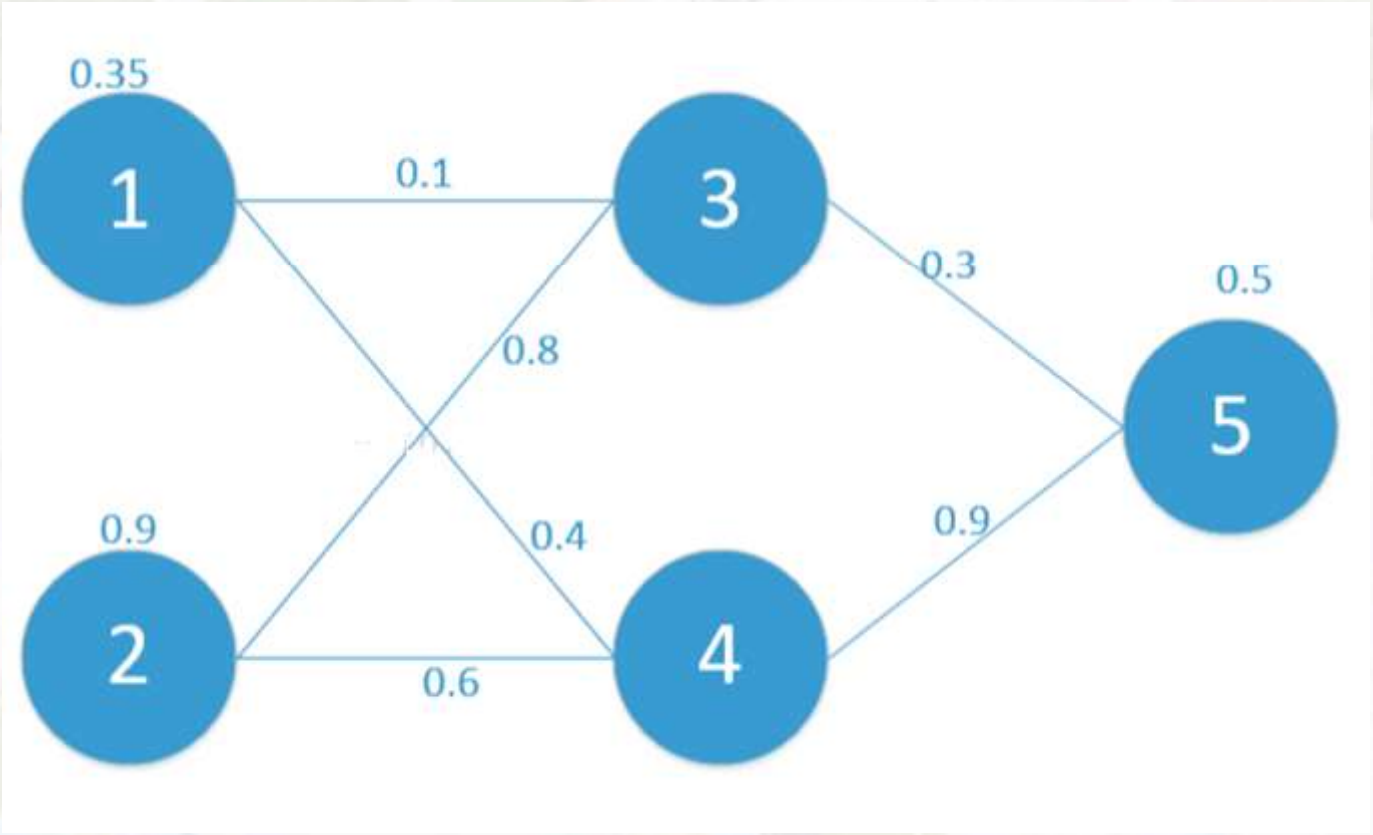
The Back-propagation Algorithm

Example II – W_{53} Weight Update(权值修正)

w_{ij}	结点i到结点j的权重
z_i	结点i的输入
y_i	结点i的输出
$E = \frac{1}{2}(y_p - y_a)^2$	损失函数
$f(x) = \frac{1}{1+e^{-x}}$	激活函数

The Back-propagation Algorithm

Example II – W_{53} Weight Update(权值修正)



$w_{53} = ?$

The Back-propagation Algorithm

Example II – W_{53} Weight Update(权值修正)

Matrix Representation:

$$X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} 0.35 \\ 0.9 \end{bmatrix}$$

$$W_0 = \begin{bmatrix} w_{31} & w_{32} \\ w_{41} & w_{42} \end{bmatrix} = \begin{bmatrix} 0.1 & 0.8 \\ 0.4 & 0.6 \end{bmatrix}$$

$$y_a = 0.5$$

$$W_1 = \begin{bmatrix} w_{53} & w_{54} \end{bmatrix} = \begin{bmatrix} 0.3 & 0.9 \end{bmatrix}$$

$$E = \frac{1}{2}(y_p - y_a)^2$$

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

The Back-propagation Algorithm

Example II – W_{53} Weight Update(权值修正)

Feed-forward calculation:

$$z1 = \begin{bmatrix} z_3 \\ z_4 \end{bmatrix} = w_0 * X = \begin{bmatrix} w_{31} & w_{32} \\ w_{41} & w_{42} \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$= \begin{bmatrix} w_{31} * x_1 & w_{32} * x_2 \\ w_{41} * x_1 & w_{42} * x_2 \end{bmatrix}$$

$$= \begin{bmatrix} 0.1 * 0.35 & 0.8 * 0.9 \\ 0.4 * 0.35 & 0.6 * 0.9 \end{bmatrix} = \begin{bmatrix} 0.755 \\ 0.68 \end{bmatrix}$$

$$y1 = \begin{bmatrix} y_3 \\ y_4 \end{bmatrix} = f(z1) = f(w_0 * X$$

$$= f(\begin{bmatrix} 0.755 \\ 0.68 \end{bmatrix})$$

$$= \begin{bmatrix} 0.682 \\ 0.663 \end{bmatrix}$$

The Back-propagation Algorithm

Example II – W_{53} Weight Update(权值修正)

Feed-forward calculation:

$$z_5 = w1 * y1 = \begin{bmatrix} w_{53} & w_{54} \end{bmatrix} * \begin{bmatrix} y_3 \\ y_4 \end{bmatrix}$$

$$= \begin{bmatrix} w_{53} * y_3 & w_{54} * y_4 \end{bmatrix}$$

$$= \begin{bmatrix} 0.801 \end{bmatrix}$$

$$y_5 = f(z_5) = f(\begin{bmatrix} w_{53} & w_{54} \end{bmatrix} * \begin{bmatrix} y_3 \\ y_4 \end{bmatrix}$$

$$= f(\begin{bmatrix} 0.801 \end{bmatrix})$$

$$= 0.690$$

$$E = \frac{1}{2}(y_p - y_a)^2 = \frac{1}{2}(0.690 - 0.5)^2 = 0.01805$$

Example II – W_{53} Weight Update(权值修正)

Back-propagation calculation:

$$\Delta w_{53} = \frac{\partial E}{\partial w_{53}} = ?$$

$$\begin{cases} E = \frac{1}{2}(y_5 - y_a)^2 \\ y_5 = f(z_5) \\ z_5 = w_{53} * y_3 + w_{54} * y_4 \end{cases}$$

$$\Delta w_{53} = \frac{\partial E}{\partial w_{53}} = \boxed{\frac{\partial E}{\partial y_5}\frac{\partial y_5}{\partial z_5}\frac{\partial z_5}{\partial w_{53}}}$$

Chain Rule (链式法则)

The Back-propagation Algorithm

Example II – W_{53} Weight Update(权值修正)

Derivative of Sigmoid Function

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

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

$$= \frac{1}{1+e^{-x}} * \frac{e^{-x}}{1+e^{-x}}$$

$$= \frac{1}{1+e^{-x}} * \frac{e^{-x}+1-1}{1+e^{-x}}$$

$$= \frac{1}{1+e^{-x}} * (\frac{1+e^{-x}}{1+e^{-x}} + \frac{-1}{1+e^{-x}})$$

$$= f(x) * (1 - f(x))$$

Example II – W_{53} Weight Update(权值修正)

Back-propagation calculation:

$$\Delta w_{53} = \frac{\partial E}{\partial w_{53}} = \frac{\partial E}{\partial y_5}\frac{\partial y_5}{\partial z_5}\frac{\partial z_5}{\partial w_{53}}$$

$$\begin{cases} E = \frac{1}{2}(y_5 - y_a)^2 \\ y_5 = f(z_5) \\ z_5 = w_{53} * y_3 + w_{54} * y_4 \end{cases}$$

$$= (y_5 - y_a) * (f(z_5) * (1 - z_5)) * y_3$$

$$= (0.69 - 0.5) * (0.69 * (1 - 0.69)) * 0.663 = 0.02711$$

Example II – W_{53} Weight Update(权值修正)

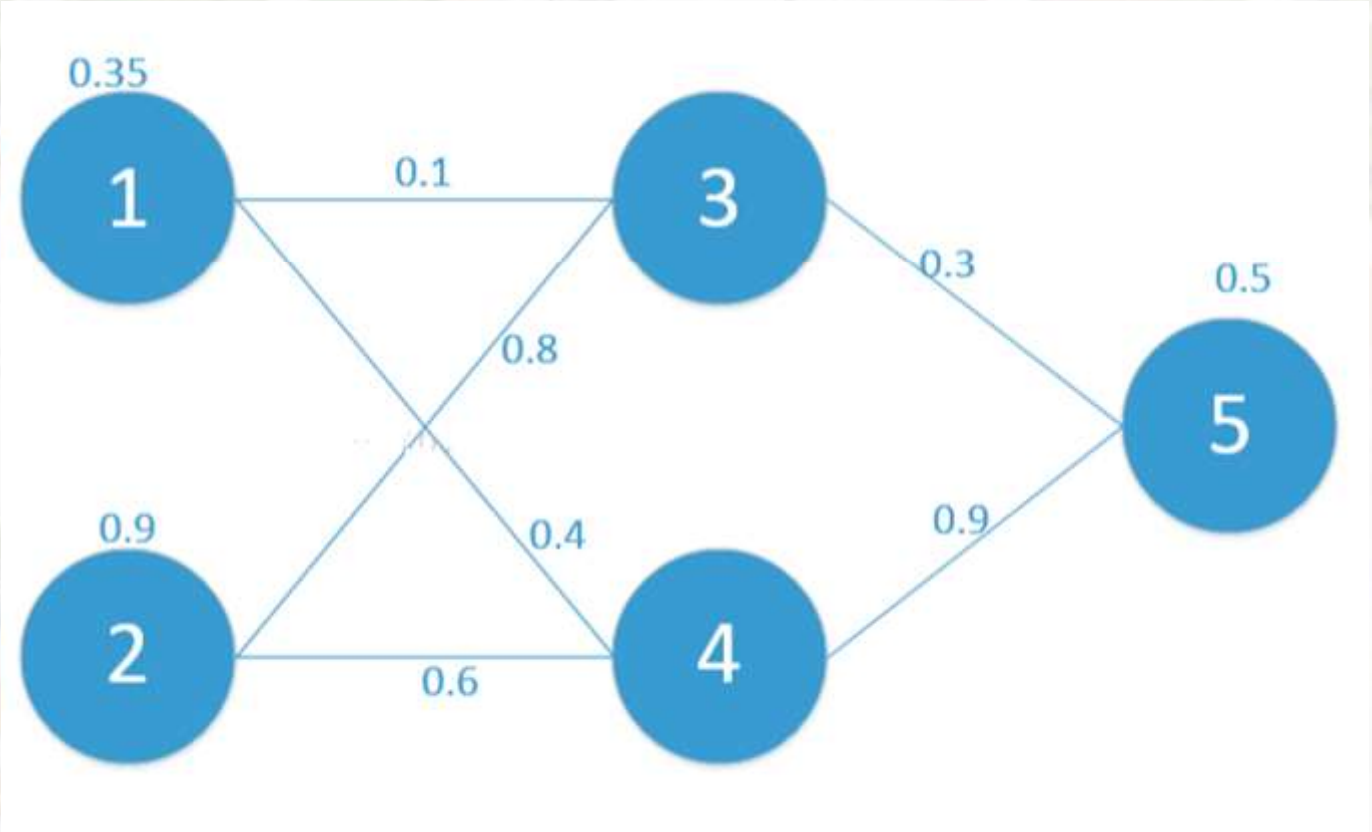
Back-propagation calculation:

$$\Delta w_{53} = 0.02711$$

$$w_{53\_new} = w_{53\_old} - \Delta w_{53} = 0.3 - 0.02711 = 0.27299$$

The Back-propagation Algorithm

Example II – W_{31} W_{41} W_{32} W_{42} Weight Update(权值修正)



Example II – W_{31} Weight Update(权值修正)

Back-propagation calculation:

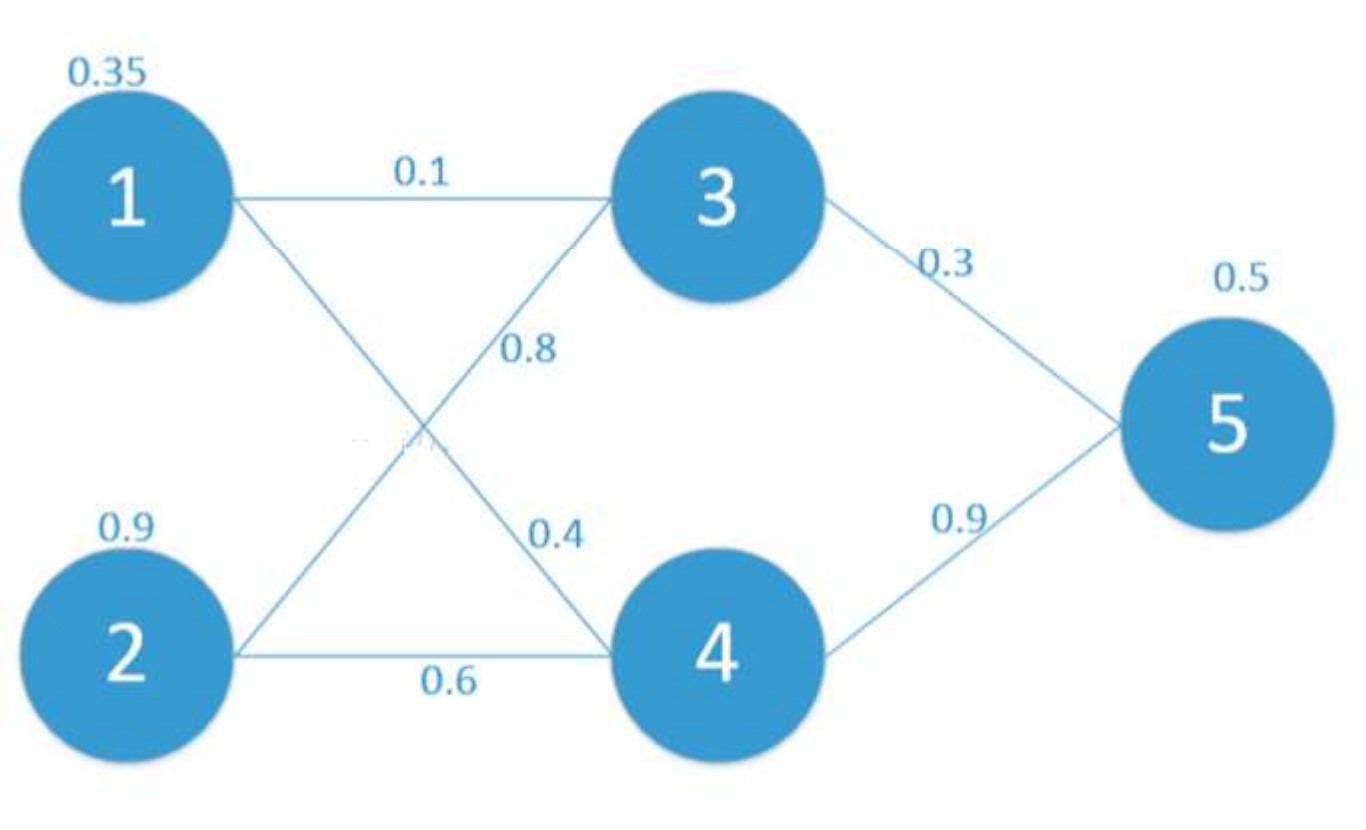
$$\Delta w_{31} = \frac{\partial E}{\partial w_{31}} = ?$$

$$\Delta w_{31} = \frac{\partial E}{\partial w_{31}} = \frac{\partial E}{\partial y_5}\frac{\partial y_5}{\partial z_5}\frac{\partial z_5}{\partial y_3}\frac{\partial y_3}{\partial z_3}\frac{\partial z_3}{\partial w_{31}}$$

$$\begin{cases} E = \frac{1}{2}(y_5 - y_a)^2 \\ y_5 = f(z_5) \\ z_5 = w_{53} * y_3 + w_{54} * y_4 \\ y_3 = f(z_3) \\ z_3 = w_{31} * x_1 + w_{32} * x_1 \end{cases}$$

The Back-propagation Algorithm

Example II – Calculate the output based on the updated weight (权值修正)



The Back-propagation Algorithm

Example II – Calculate the output based on the updated weight (权值修正)

$$\begin{cases} w_{31\_new} = w_{31\_old} - \dfrac{\partial E}{\partial w_{31}} = 0.09661944 \\ w_{32\_new} = w_{32\_old} - \dfrac{\partial E}{\partial w_{32}} = 0.78985831 \\ w_{41\_new} = w_{41\_old} - \dfrac{\partial E}{\partial w_{41}} = 0.39661944 \\ w_{42\_new} = w_{42\_old} - \dfrac{\partial E}{\partial w_{42}} = 0.58985831 \end{cases}$$

$$E = \frac{1}{2}(y_p - y_a)^2 = 0.016$$

The Back-propagation Algorithm

- Like the Perceptron - calculation of error is based on difference between target and actual output:

$$E = \frac{1}{2}\sum_j (t_j - o_j)^2$$

- However in BP it is the rate of change of the error which is the important feedback through the network

generalized delta rule $\Delta w_{ij} = -\eta \frac{\delta E}{\delta w_{ij}}$

- Relies on the sigmoid activation function for communication


THANK
YOU
谢谢！