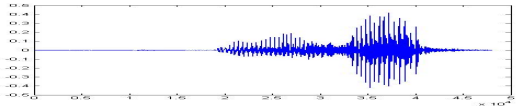
Optimization for Deep Learning

•Slides from Deep Learning Tutorial. Hung-yi Lee, NTU.

Machine Learning ≈ Looking for a Function

- Speech Recognition

$f($  $)=$ “How are you”

- Image Recognition

$f($  $)=$ “Cat”

- Playing Go

$f($  $)=$ “5-5” (next move)

- Dialogue System

$f($ “Hi” $)=$ “Hello”

(what the user said) (system response)

Framework

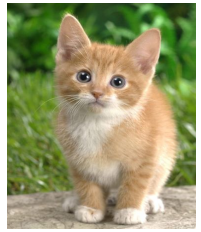
Image Recognition:

$f($  $)=$ "cat"

A set of function

Model

$f_1($  $)=$ "cat"

$f_2($  $)=$ "money"

$f_1($  $)=$ "dog"

$f_2($  $)=$ "snake"

Framework

Image Recognition:

$f($  $)=$ "cat"

A set of function

Model

Goodness of function f

Training Data

$f_1($  $)=$ "cat"

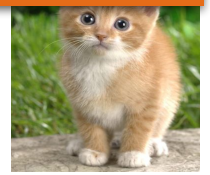
$f_2($  $)=$ "money"

Better!

$f_1($  $)=$ "snake"

$f_2($  $)=$ "snake"

Supervised Learning

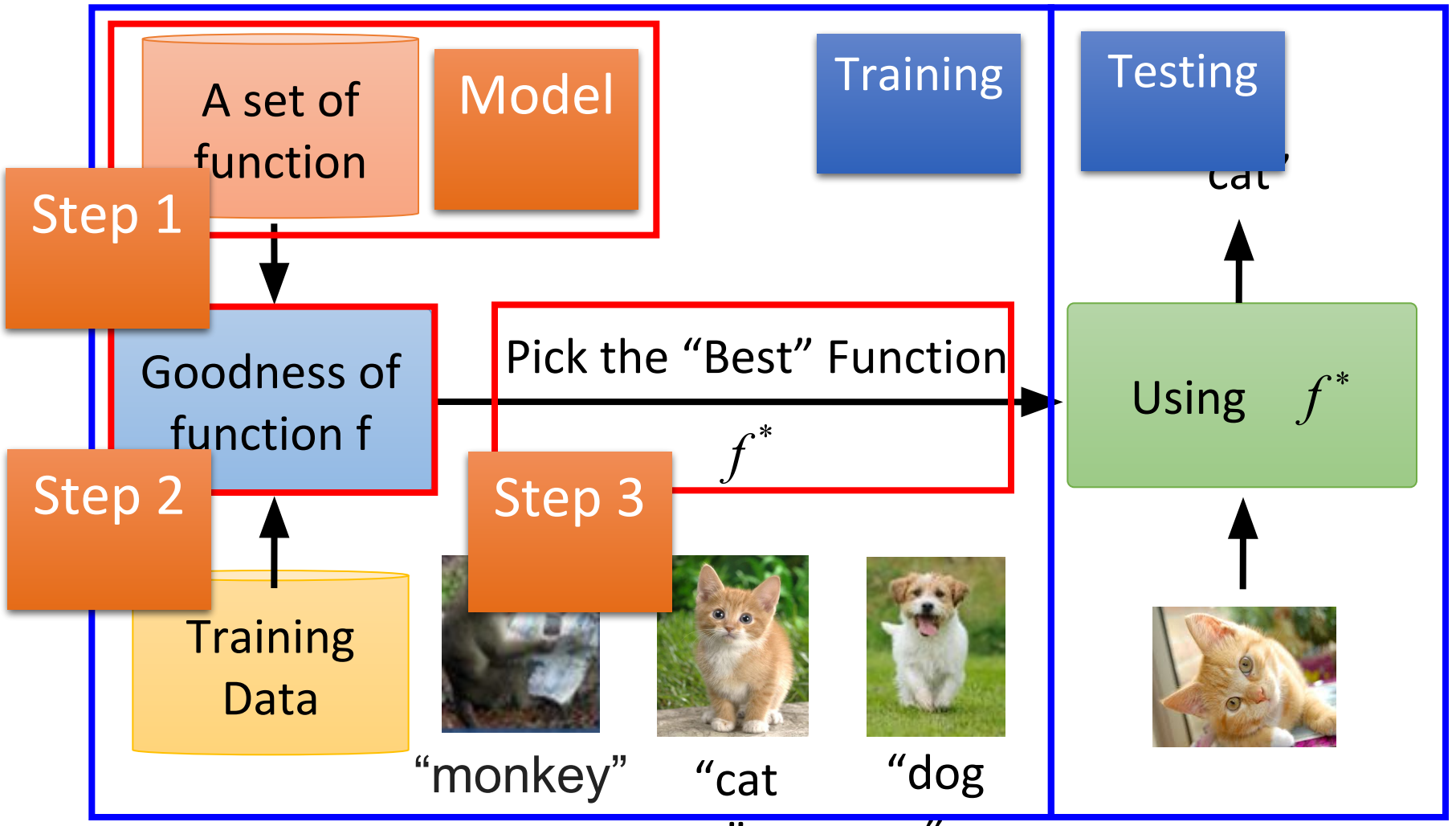
function input:   

function output: "monkey" "cat" "dog"

Framework

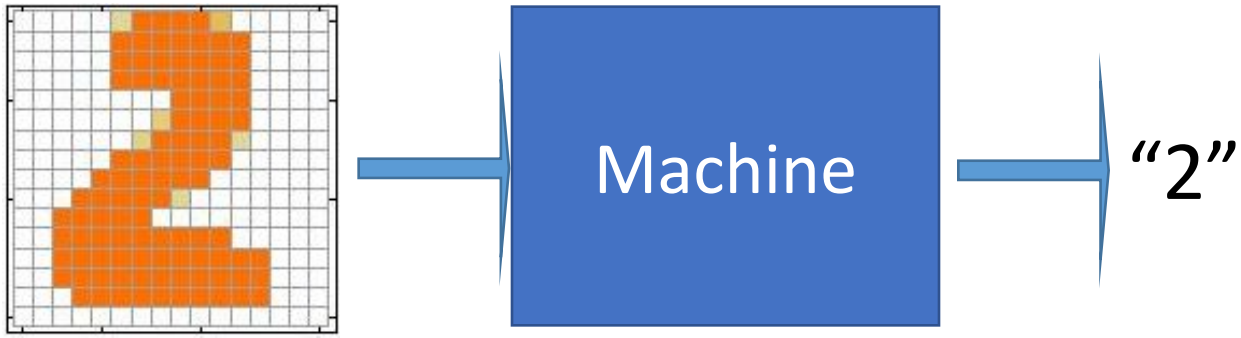
Image Recognition:

$f($  $) =$ "cat"



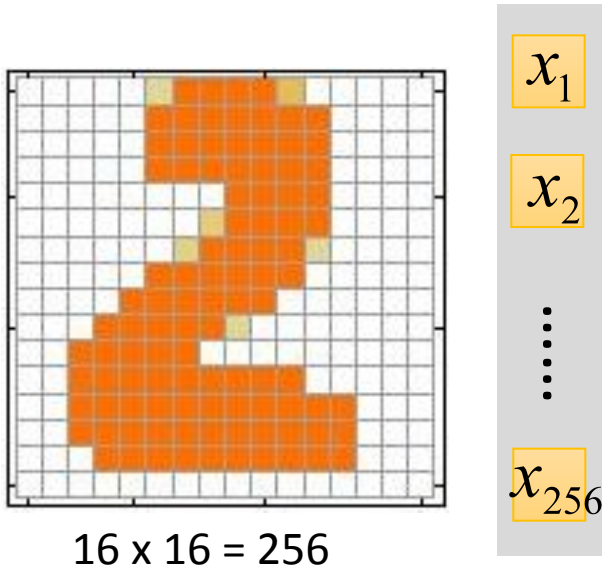
Example Application

- Handwriting Digit Recognition



Handwriting Digit Recognition

Input

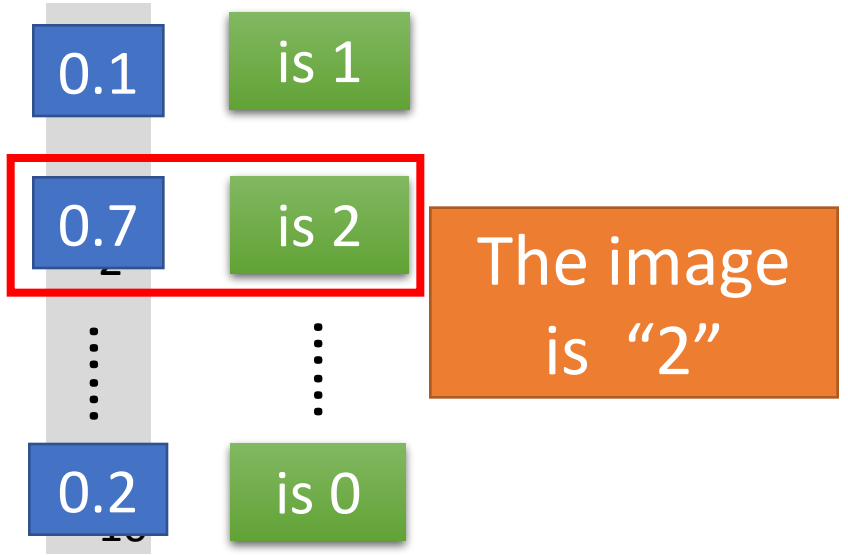


16 x 16 = 256

Ink → 1

No ink → 0

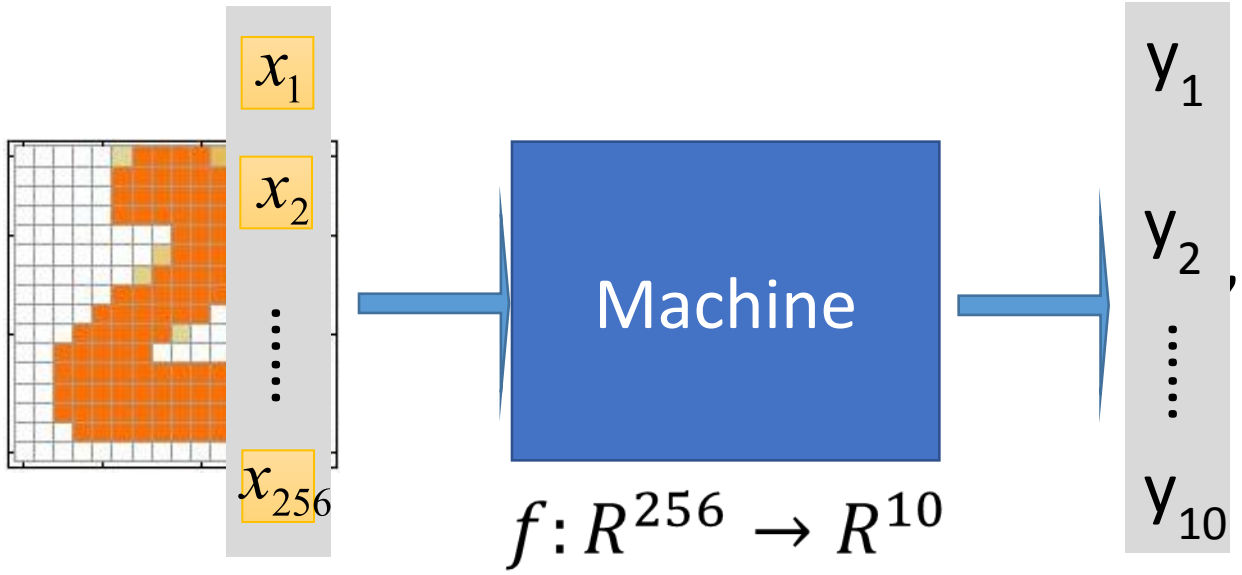
Output



Each dimension represents the confidence of a digit.

Example Application

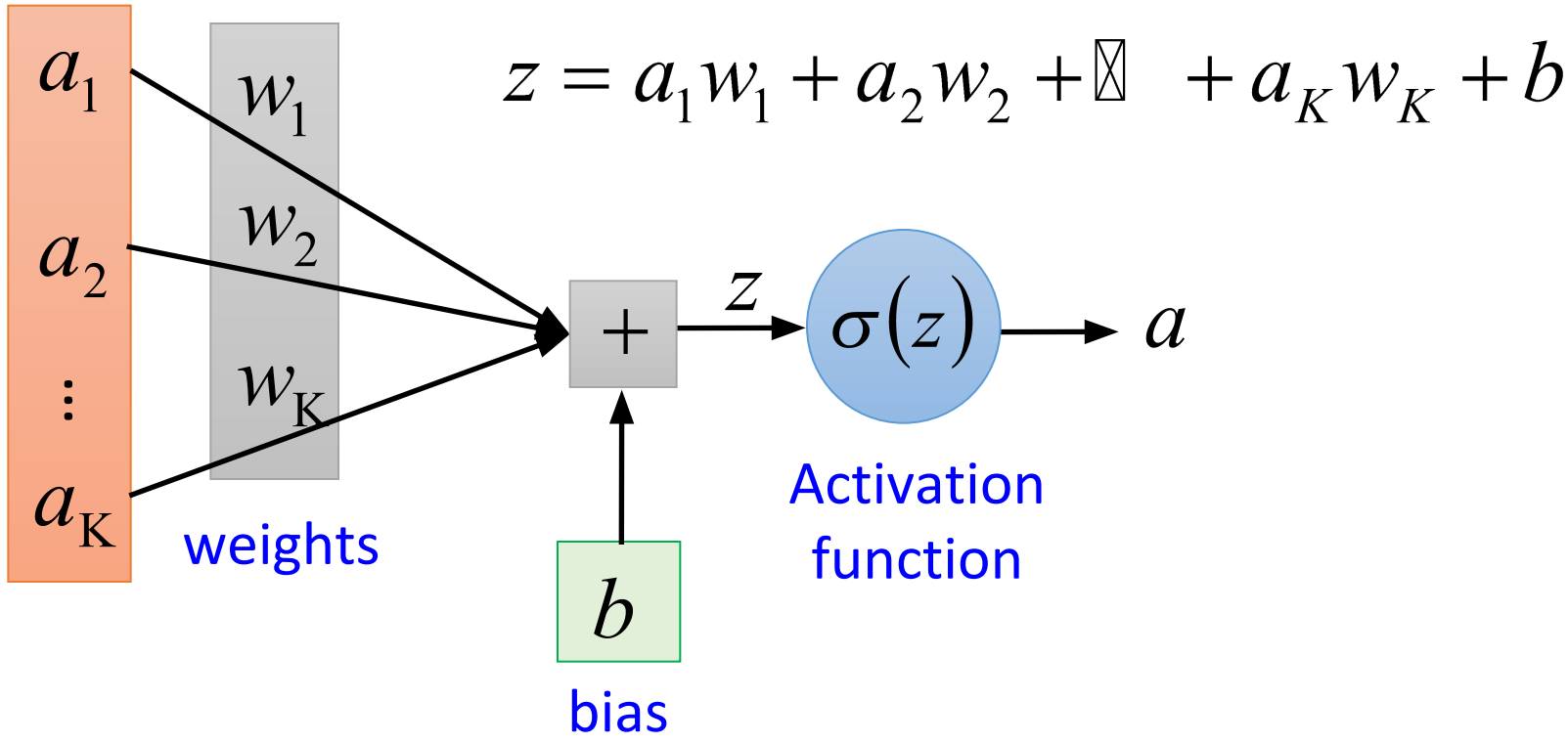
- Handwriting Digit Recognition



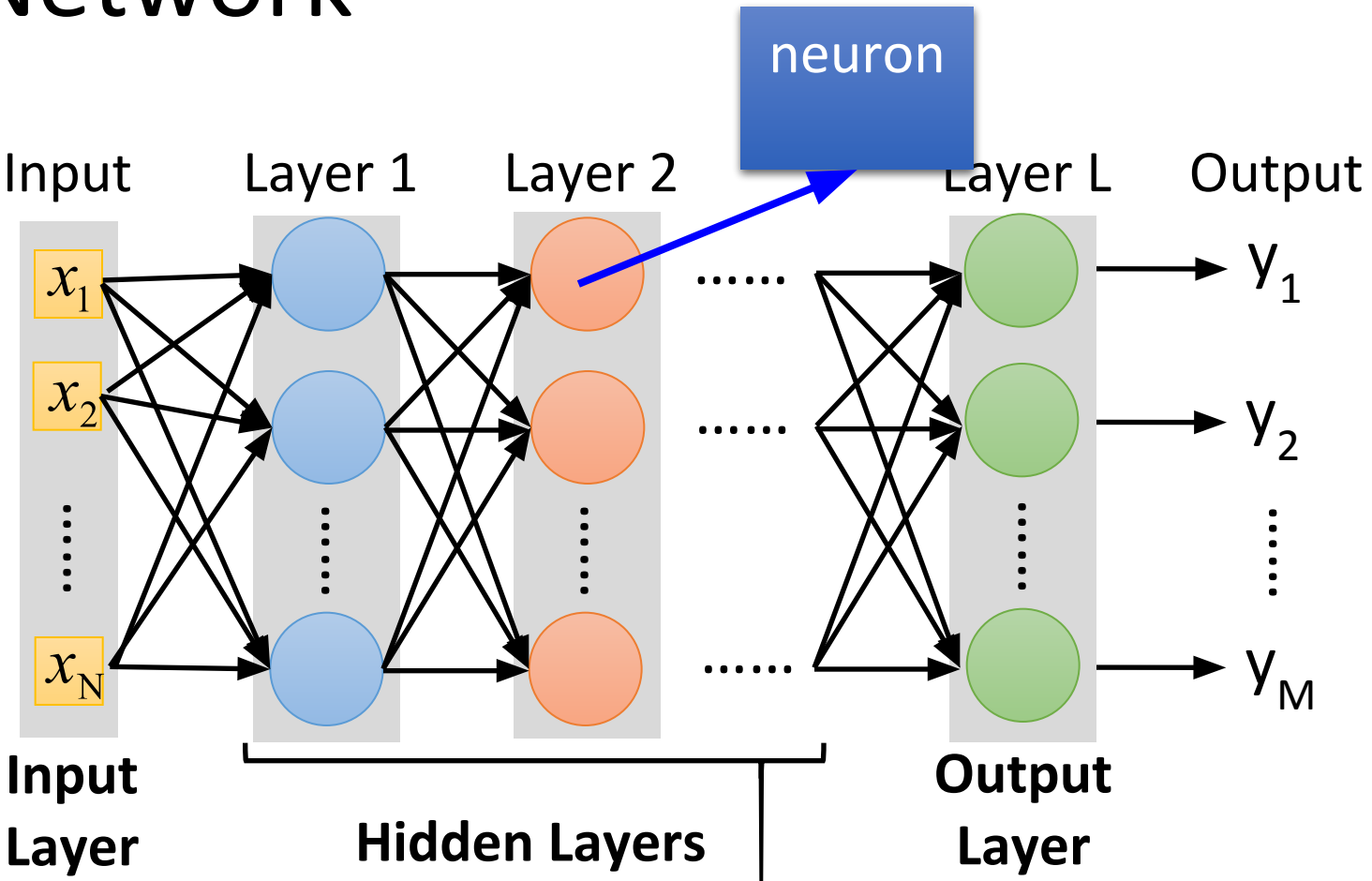
In deep learning, the function f is represented by neural network

Element of Neural Network

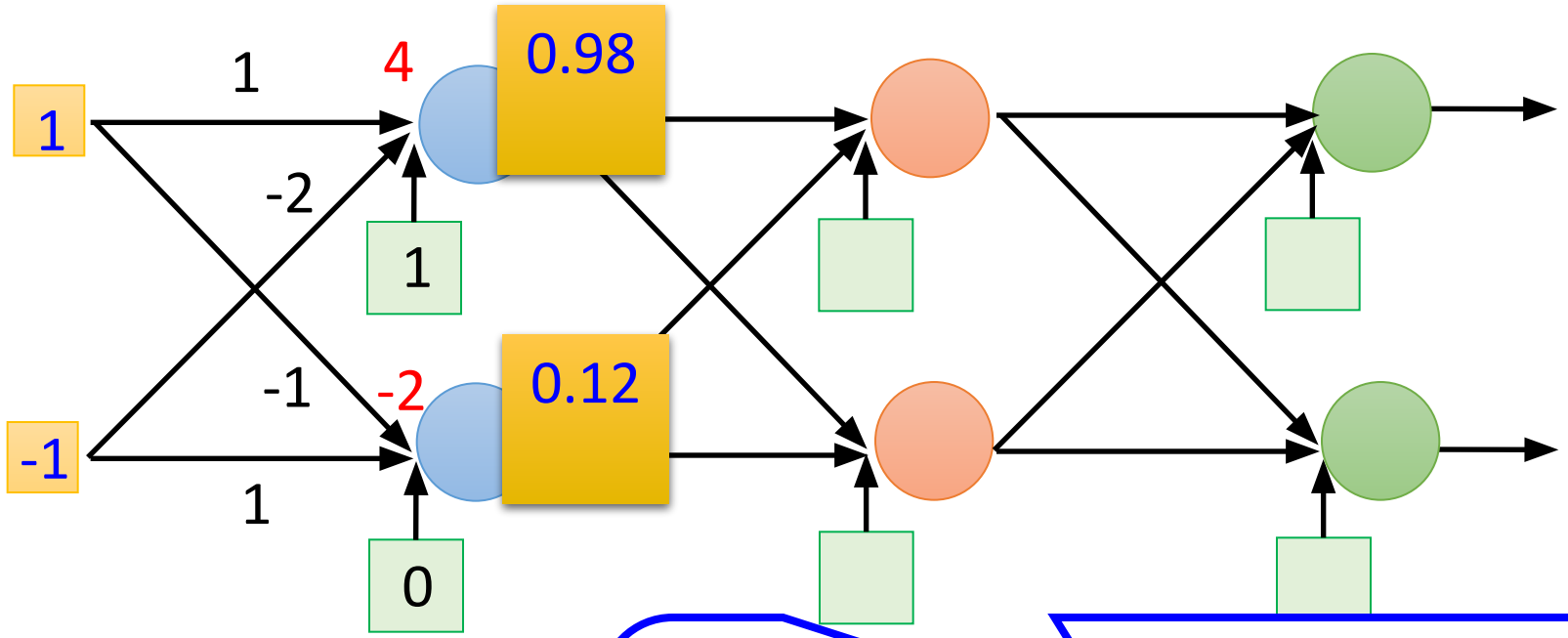
Neuron $f: R^K \rightarrow R$



Neural Network

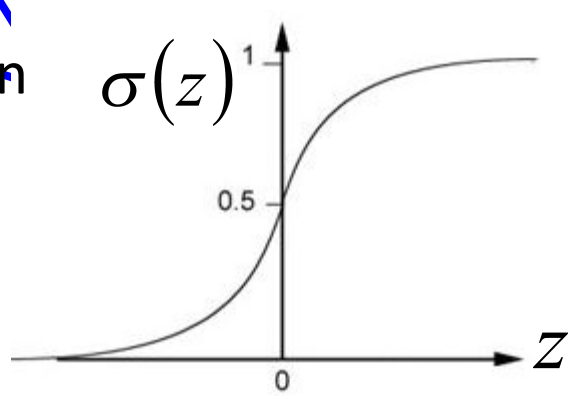


Example of Neural Network

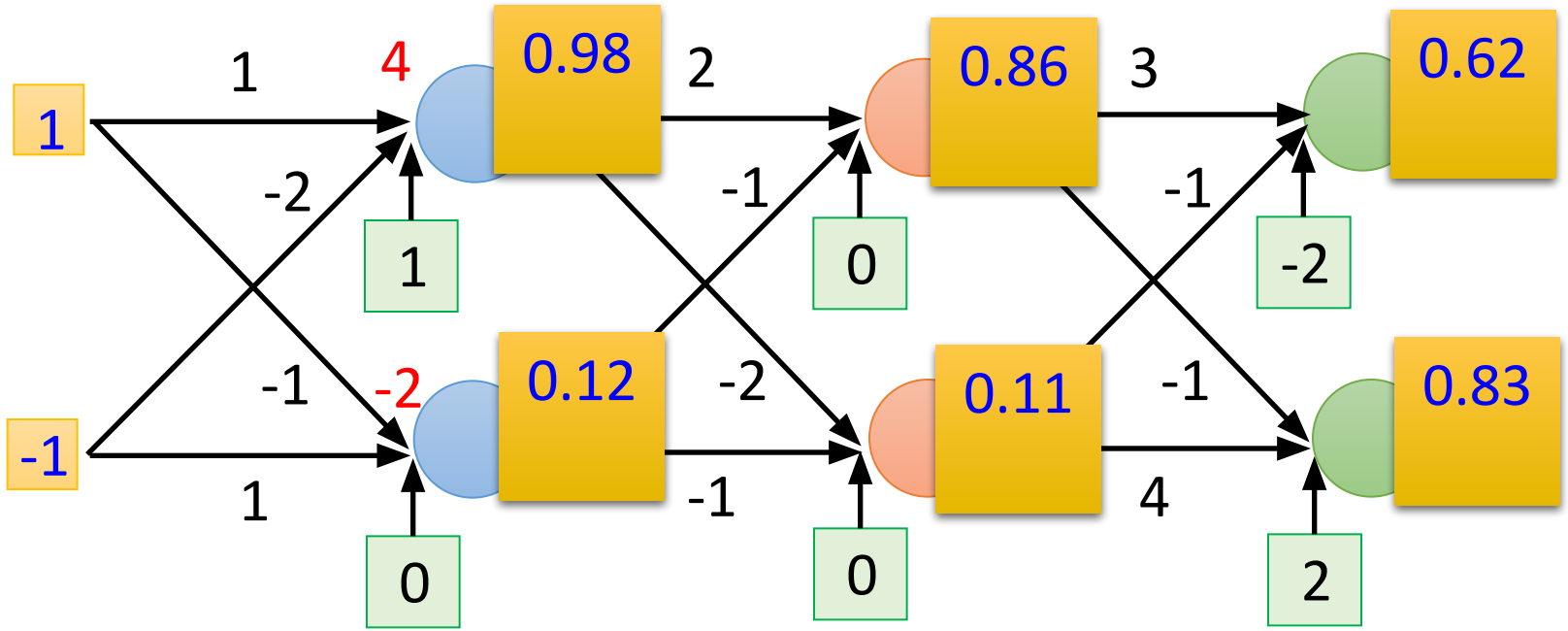


Sigmoid Function

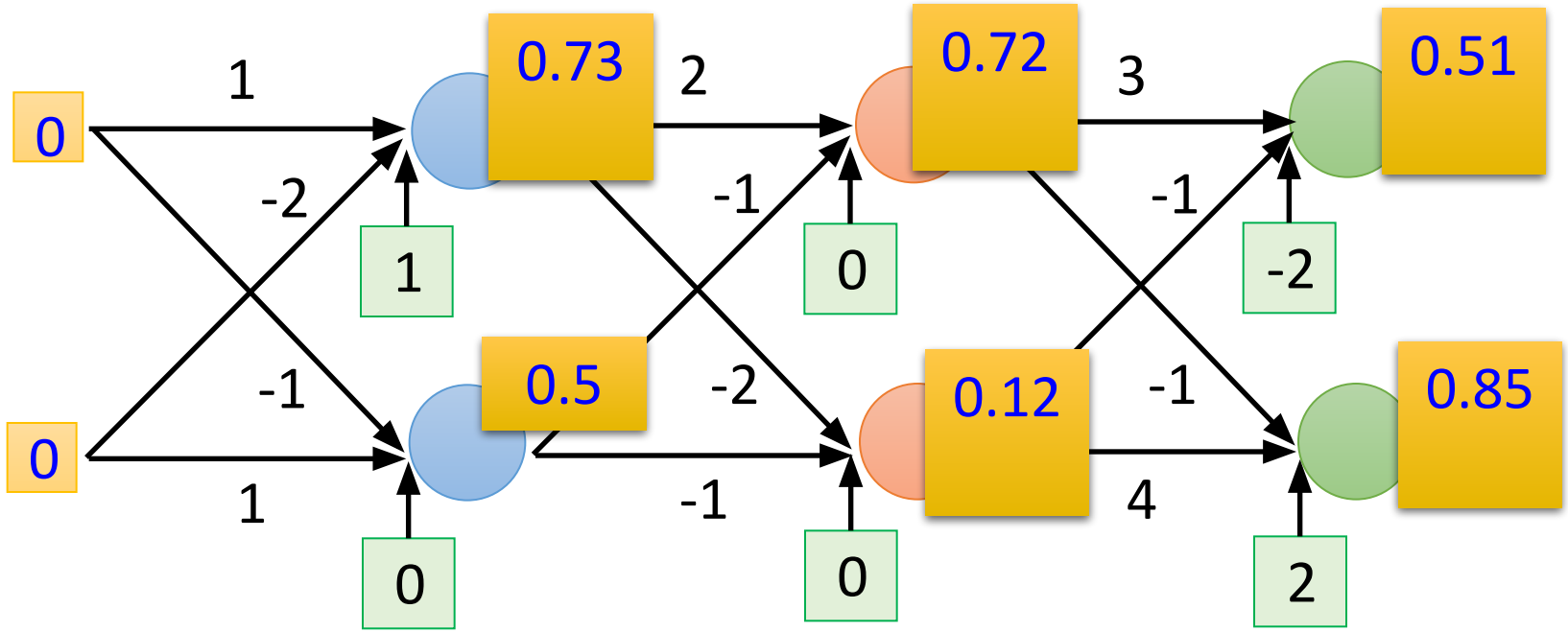
$$\sigma(z) = \frac{1}{1+e^{-z}}$$



Example of Neural Network



Example of Neural Network

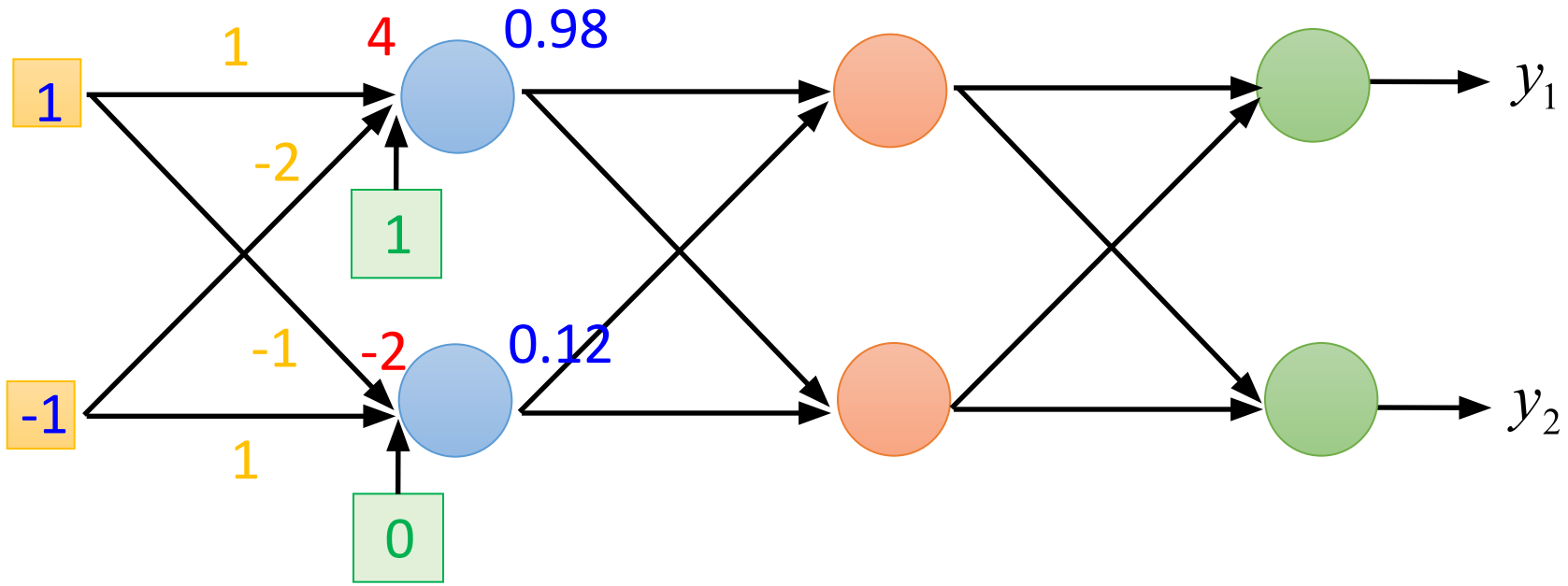


$f: R^2 \to R^2$

$$f\left(\begin{bmatrix}1\\-1\end{bmatrix}\right) = \begin{bmatrix}0.62\\0.83\end{bmatrix} \quad f\left(\begin{bmatrix}0\\0\end{bmatrix}\right) = \begin{bmatrix}0.51\\0.85\end{bmatrix}$$

Different parameters define different function

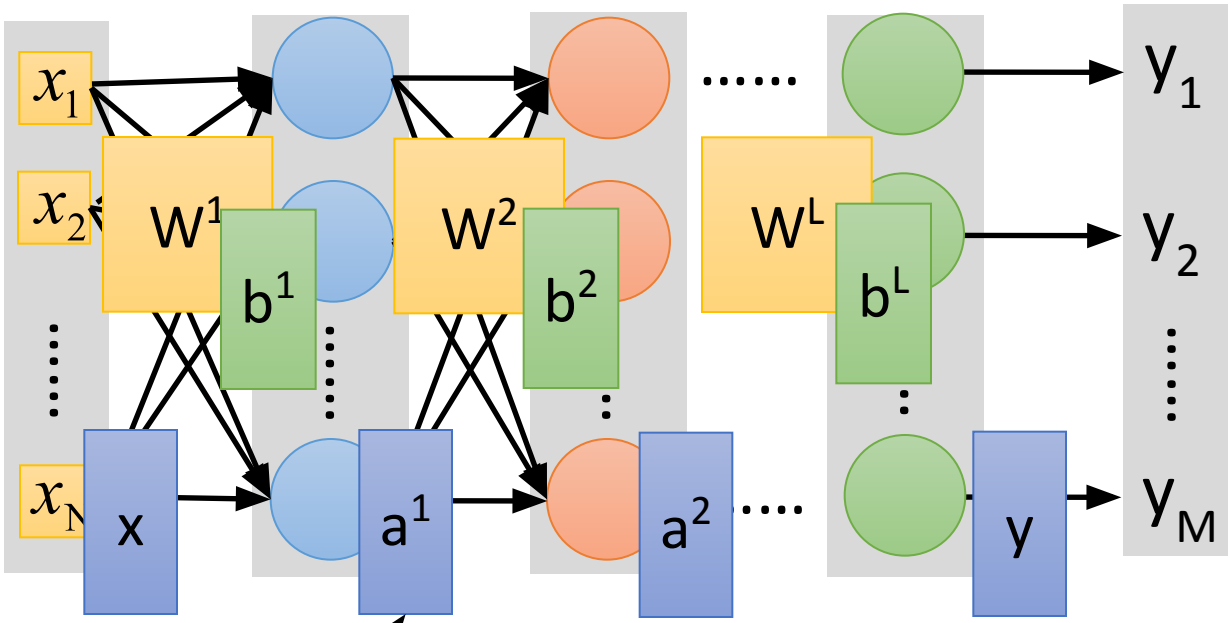
Matrix Operation



$$\sigma\left(\begin{bmatrix}1 & -2\\ -1 & 1\end{bmatrix}\begin{bmatrix}1\\ -1\end{bmatrix}+\begin{bmatrix}1\\ 0\end{bmatrix}\right)=\begin{bmatrix}0.98\\ 0.12\end{bmatrix}$$

$$\begin{bmatrix}4\\ -2\end{bmatrix}$$

Neural Network

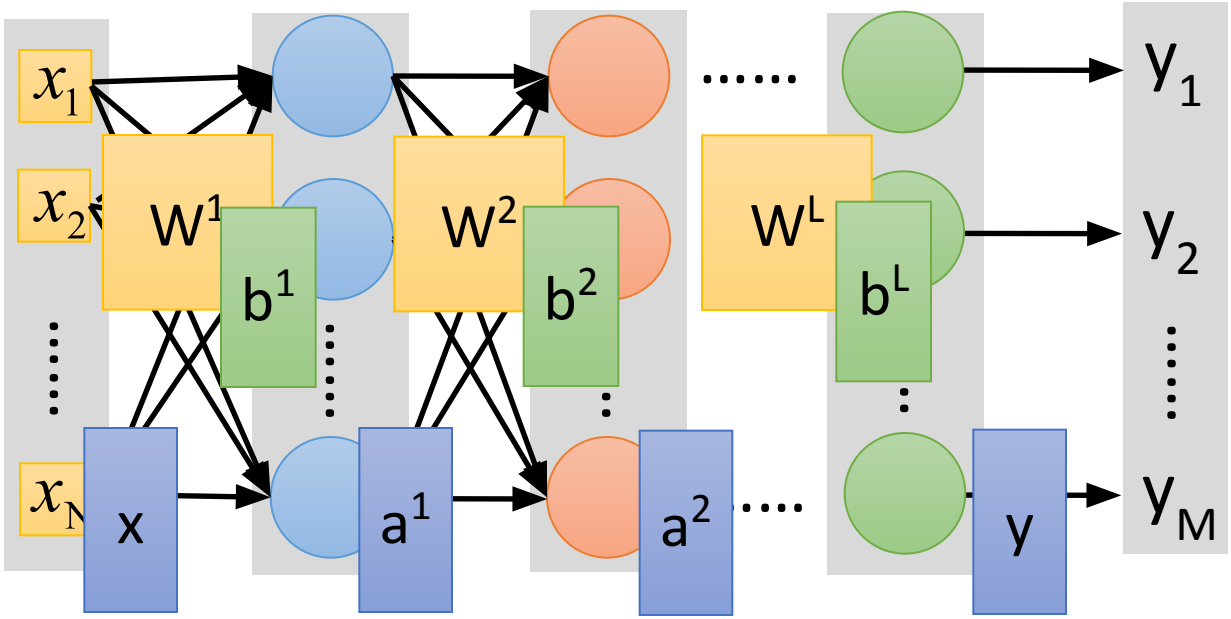


$$\sigma(\ W^1\ x\ +\ b^1\ )$$

$$\sigma(\ W^2\ a^1\ +\ b^2\ )$$

$$\sigma(\ W^L\ a^{L-1}\ +\ b^L\ )$$

Neural Network



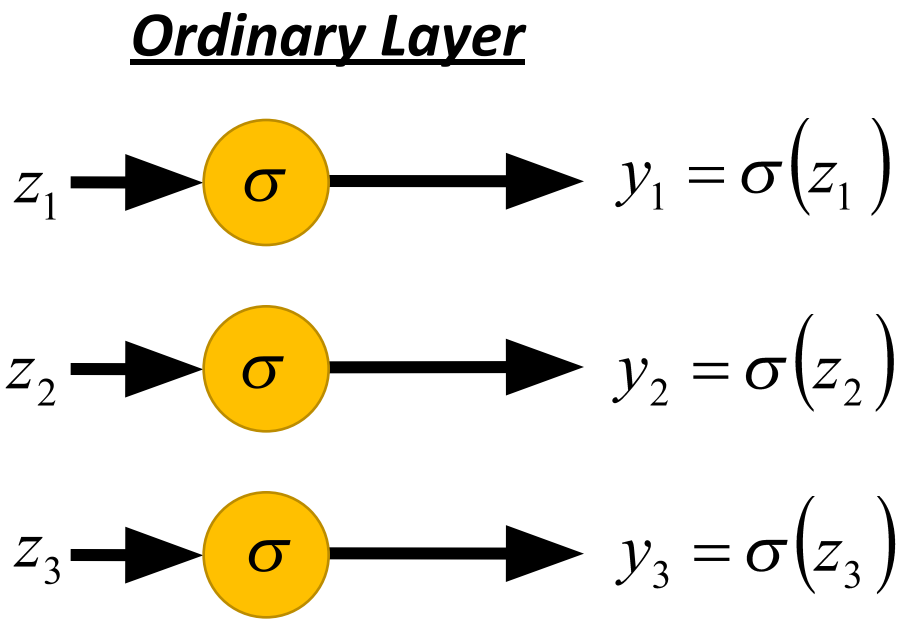
$$y = f(x)$$

Using parallel computing techniques to speed up matrix operation

$$= \sigma(W^L \cdots \sigma(W^2 \sigma(W^1 x + b^1) + b^2) \cdots + b^L)$$

Softmax

- Softmax layer as the output layer



In general, the output of network can be any value.

May not be easy to interpret

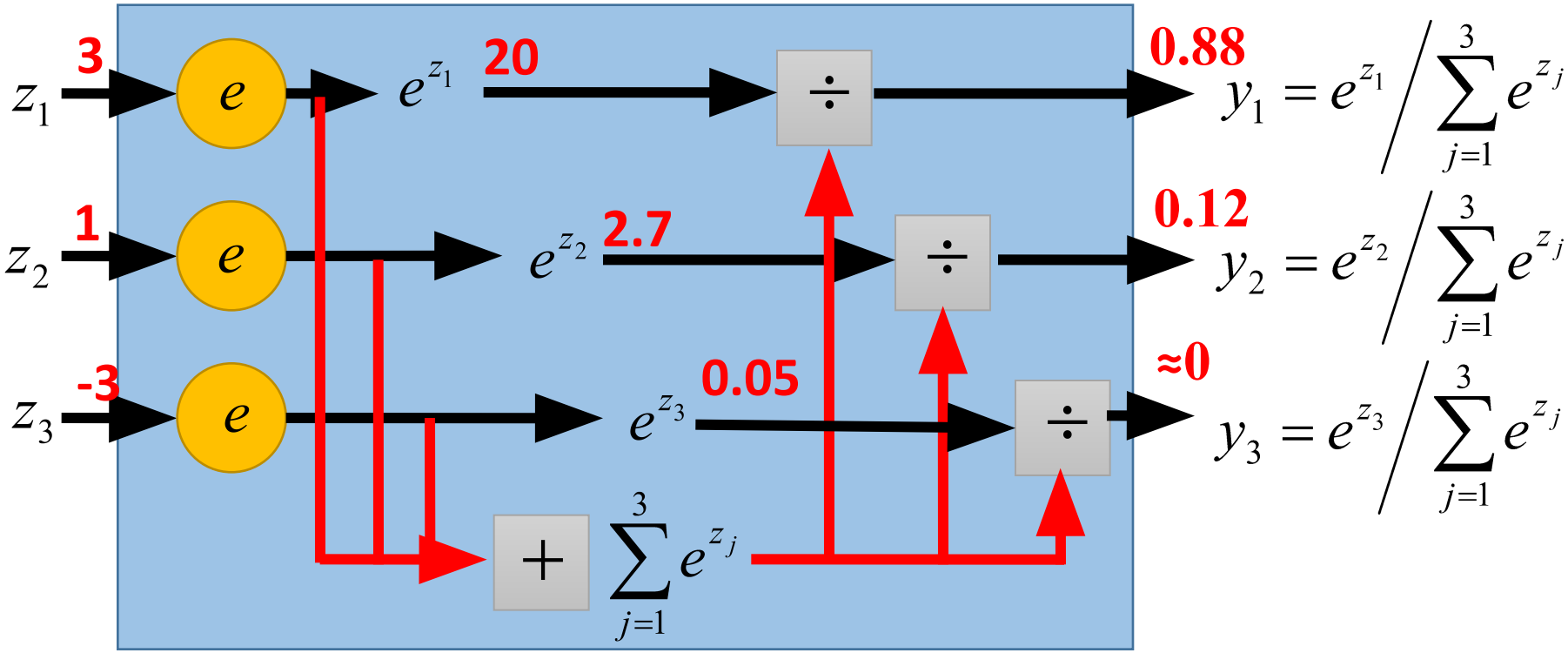
Softmax

- Softmax layer as the output layer

Probability:

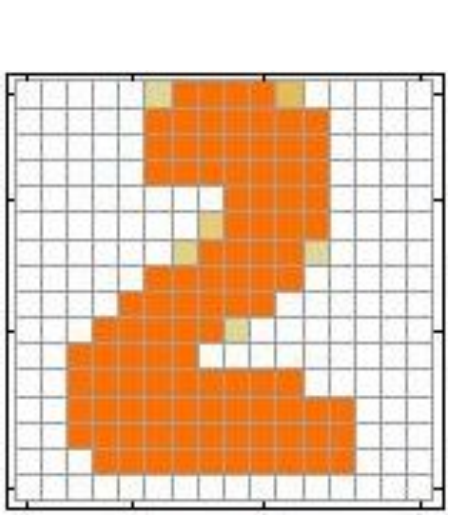
- $1 > y_i > 0$
- $\sum_i y_i = 1$

Softmax Layer



How to set network parameters

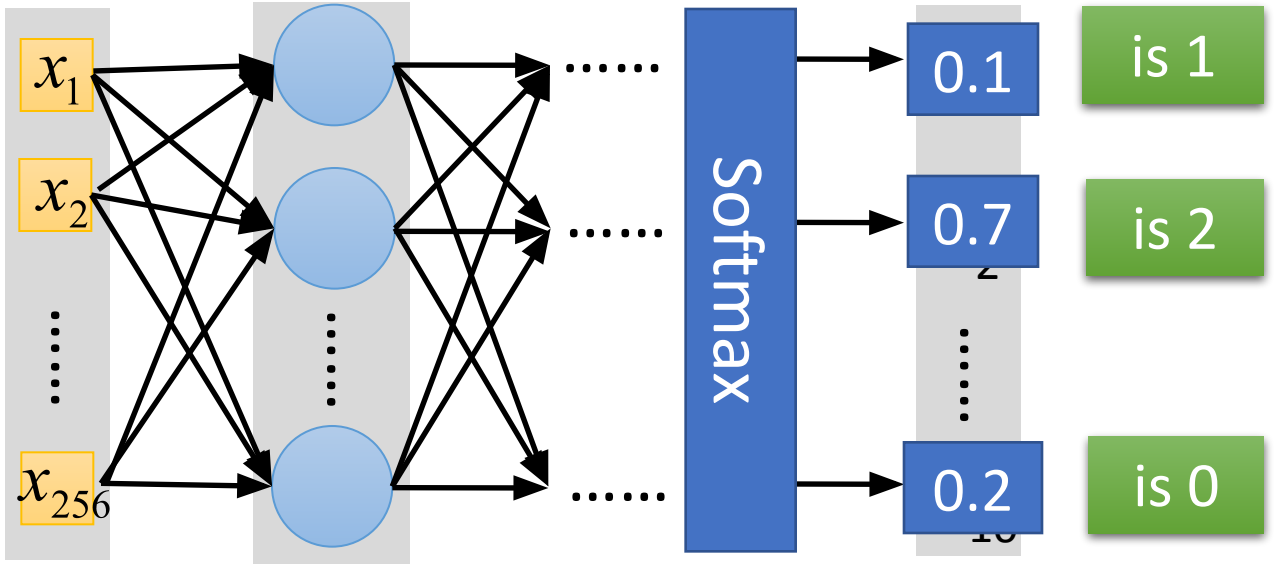
$$\theta = \{W^1, b^1, W^2, b^2, \cdots W^L, b^L\}$$



16 x 16 = 256

Ink → 1

No ink → 0



Set the network parameters θ such that ……

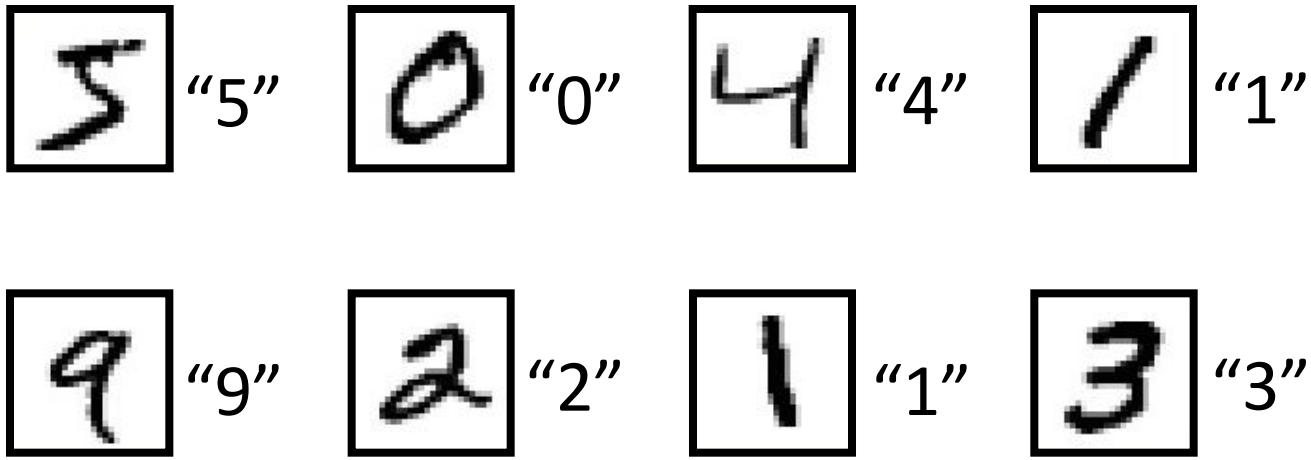
Input alue

Input: has the maximum value

How to let the neural network achieve this

Training Data

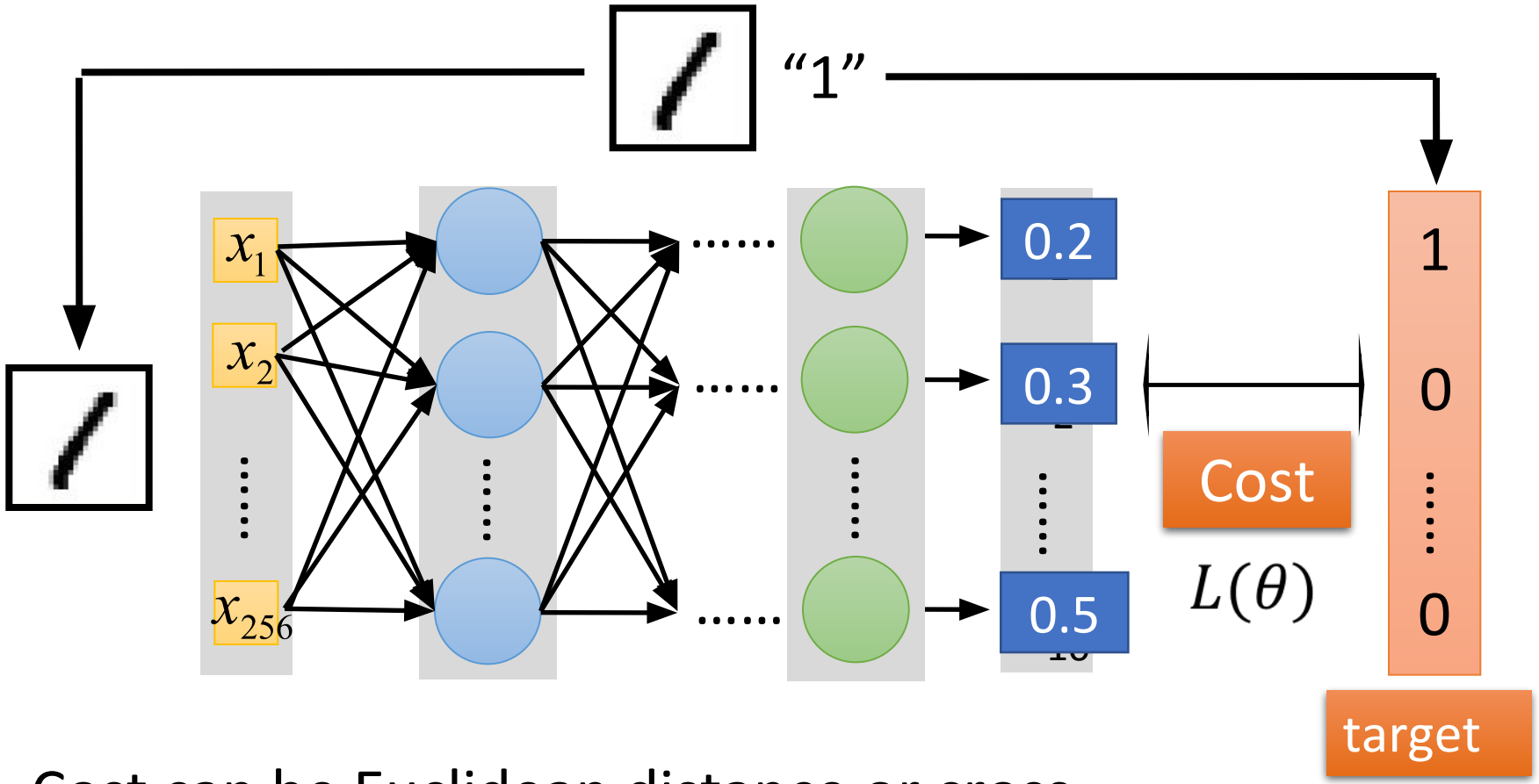
- Preparing training data: images and their labels



Using the training data to find the network parameters.

Cost

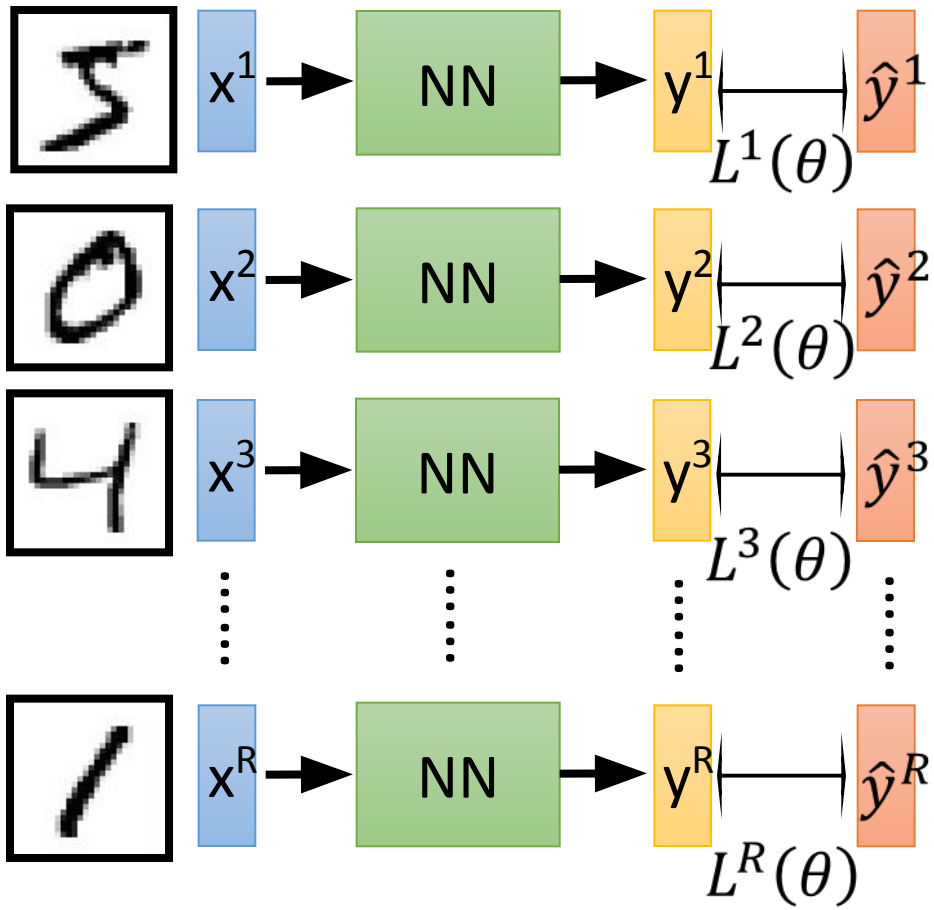
Given a set of network parameters θ, each example has a cost value.



Cost can be Euclidean distance or cross entropy of the network output and target

Total Cost

For all training data …



Total Cost:

$$C(\theta) = \sum_{r=1}^{R} L^r(\theta)$$

How bad the network parameters θ is on this task

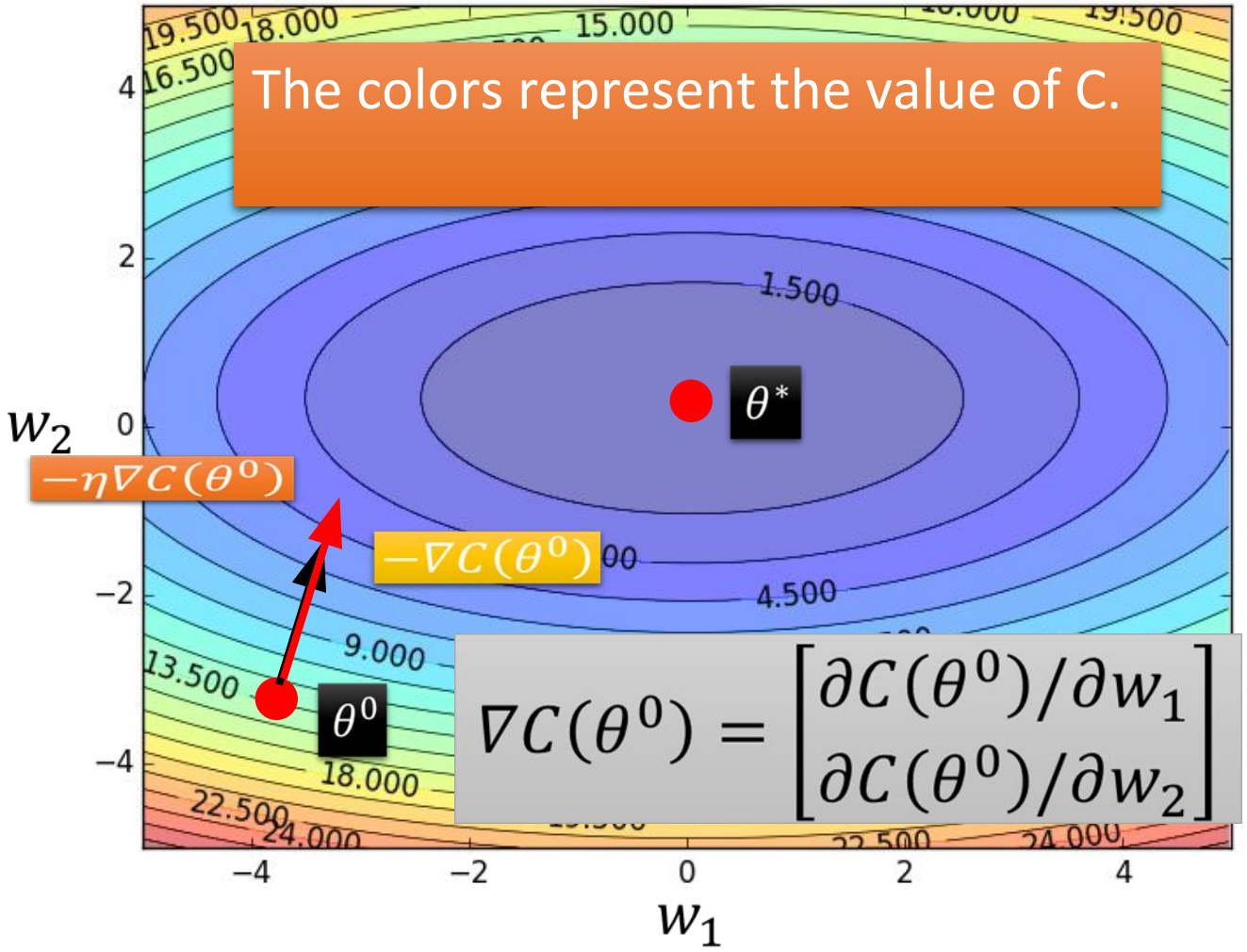
Find the network parameters θ^* that minimize this value

Gradient Descent

Assume there are only two parameters w_1 and w_2 in a network.

$$\theta = \{w_1, w_2\}$$

Error Surface



Randomly pick a starting point θ^0

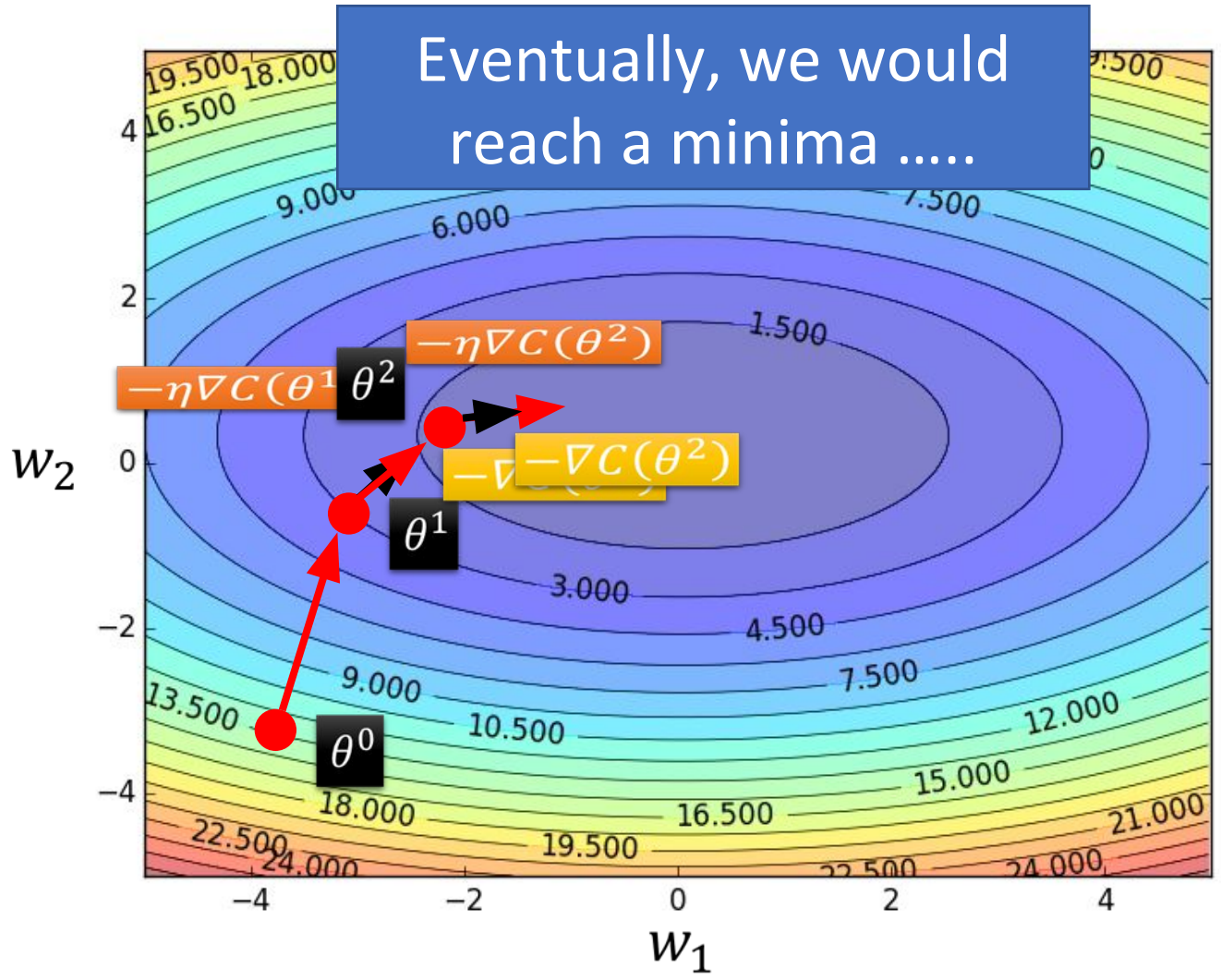
Compute the negative gradient at θ^0

$$-\nabla C(\theta^0)$$

Times the learning rate η

$$-\eta\nabla C(\theta^0)$$

Gradient Descent



Randomly pick a starting point θ^0

Compute the negative gradient at θ^0

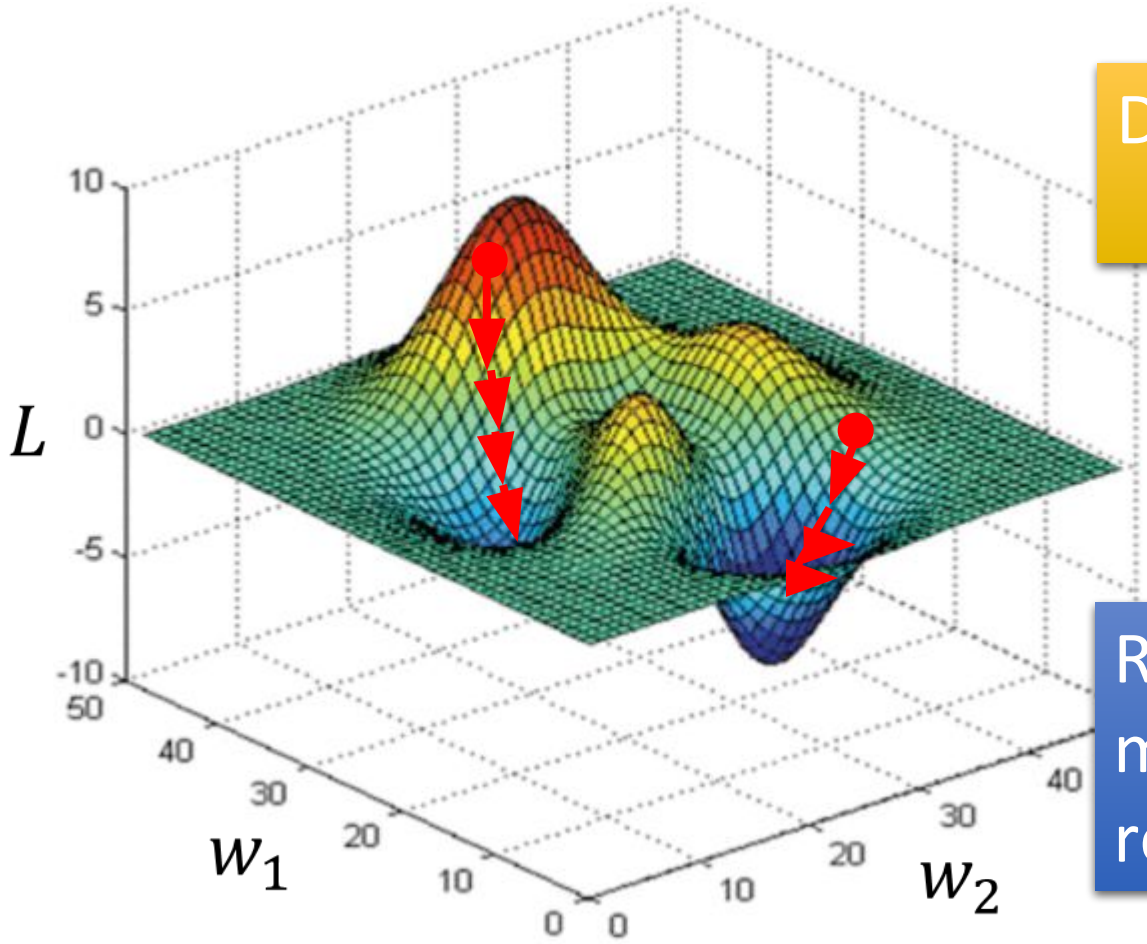
$-\nabla C(\theta^0)$

Times the learning rate η

$-\eta\nabla C(\theta^0)$

Local Minima

- Gradient descent never guarantee global minima



Different initial point

Reach different minima, so different results

The Problem with Gradient Descent

- Large-scale optimization

$$h(x) = \frac{1}{N} \sum_{i=1}^{N} f(x; y_i)$$

- Computing the gradient takes O(N) time

$$\nabla h(x) = \frac{1}{N} \sum_{i=1}^{N} \nabla f(x; y_i)$$

Gradient Descent with More Data

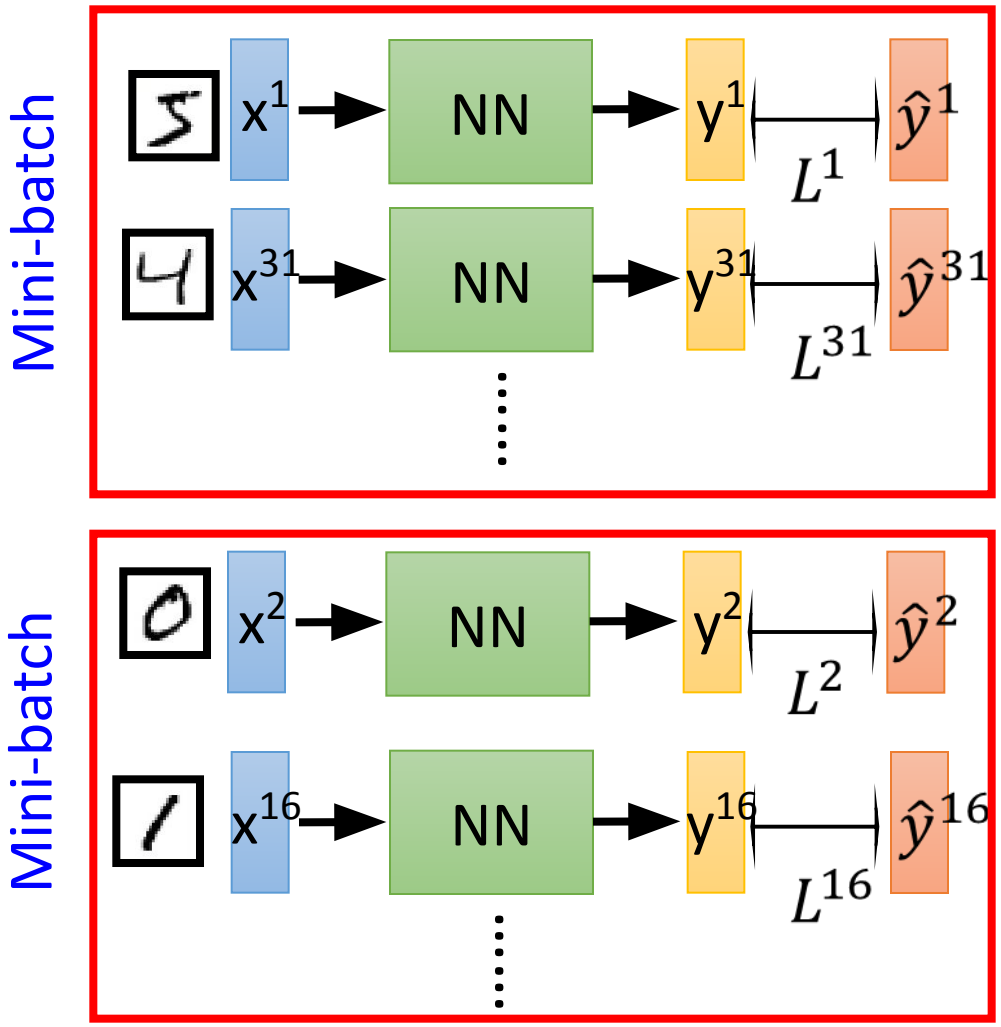
- Suppose we add more examples to our training set
  - For simplicity, imagine we just add an extra copy of every training example

$$\nabla h(x) = \frac{1}{2N}\sum_{i=1}^{N} \nabla f(x; y_i) + \frac{1}{2N}\sum_{i=1}^{N} \nabla f(x; y_i)$$

- **Same objective function**
  - But gradients take **2x the time to compute** (unless we cheat)

- We want to **scale up to huge datasets**, so how can we do this?

Mini-batch



- Randomly initialize θ^0
- Pick the 1st batch

$$C = L^1 + L^{31} + \cdots$$

$$\theta^1 \leftarrow \theta^0 - \eta \nabla C(\theta^0)$$

- Pick the 2nd batch

$$C = L^2 + L^{16} + \cdots$$

$$\theta^2 \leftarrow \theta^1 - \eta \nabla C(\theta^1)$$

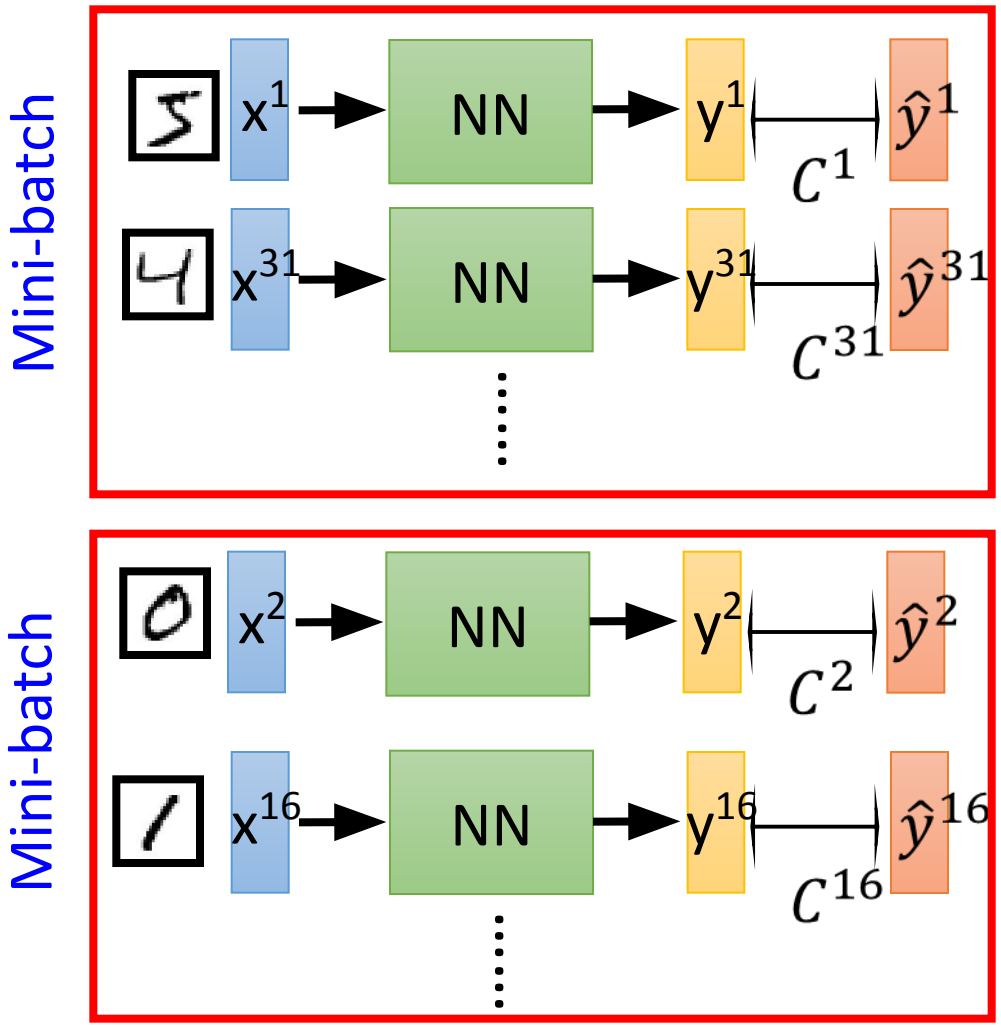
⋮

C is different each time when we update parameters!

Mini-batch

Faster

Better!



- Randomly initialize θ^0
- Pick the 1st batch

$$C = C^1 + C^{31} + \cdots$$

$$\theta^1 \leftarrow \theta^0 - \eta \nabla C(\theta^0)$$

- Pick the 2nd batch

$$C = C^2 + C^{16} + \cdots$$

$$\theta^2 \leftarrow \theta^1 - \eta \nabla C(\theta^1)$$

⋮

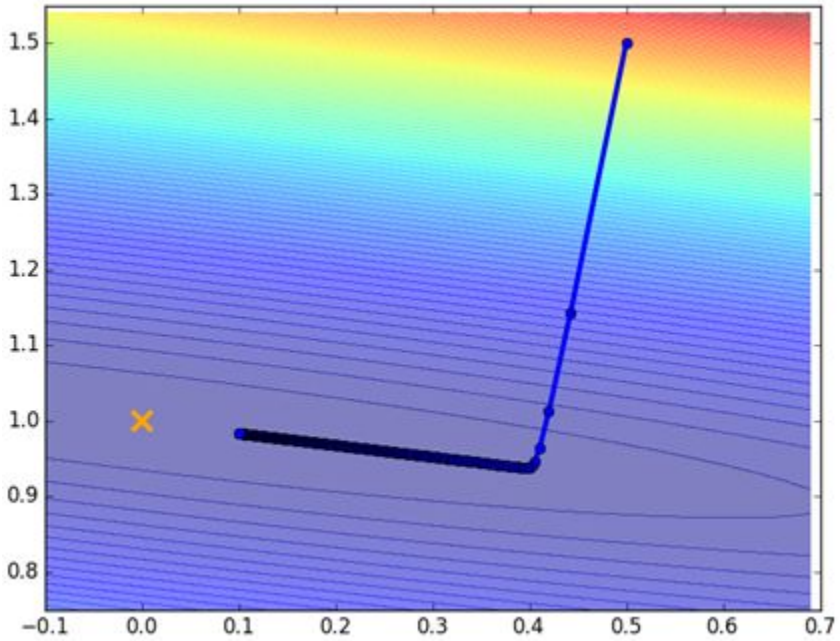
- Until all mini-batches have been picked

one epoch

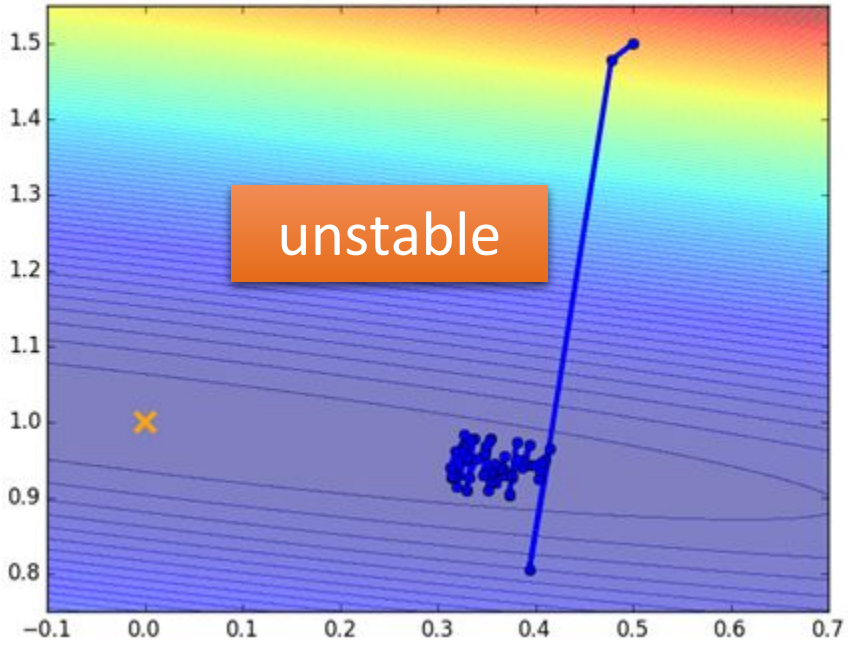
Repeat the above process

Mini-batch

Original Gradient Descent



With Mini-batch



The colors represent the total C on all training data.

Stochastic Gradient Descent (SGD)

- Idea: rather than using the full gradient, just use one training example
  - Super fast to compute

$$x_{t+1} = x_t - \alpha \nabla f(x_t; y_{\tilde{i}_t})$$

- In expectation, it's just gradient descent:

$$\mathbf{E}[x_{t+1}] = \mathbf{E}[x_t] - \alpha \mathbf{E}[\nabla f(x_t; y_{i_t})]$$

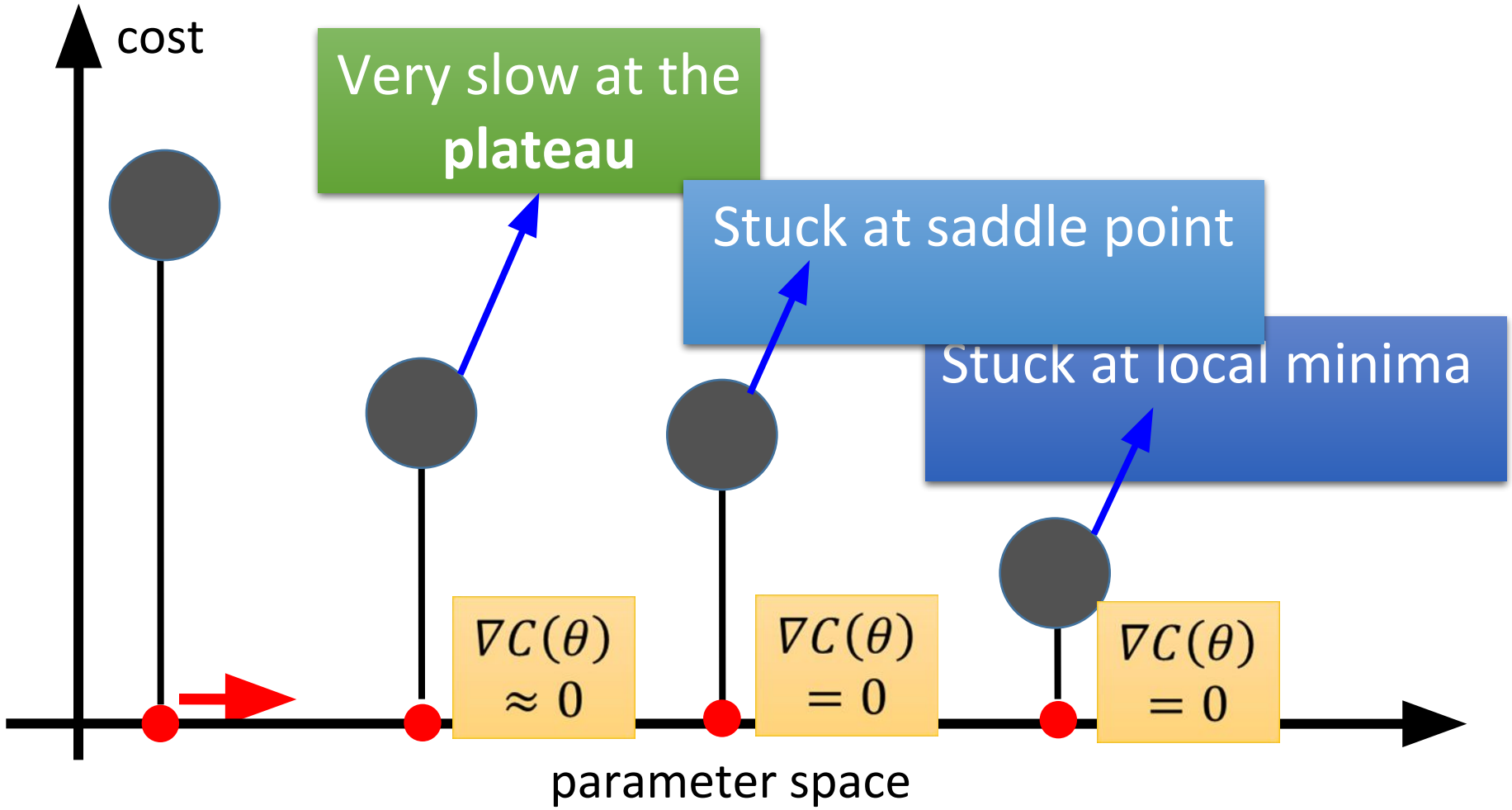
$$= \mathbf{E}[x_t] - \alpha \frac{1}{N} \sum_{i=1}^{N} \nabla f(x_t; y_i)$$

This is an example selected uniformly at random from the dataset.

- A special case of mini-batch gradient descent with batch size 1

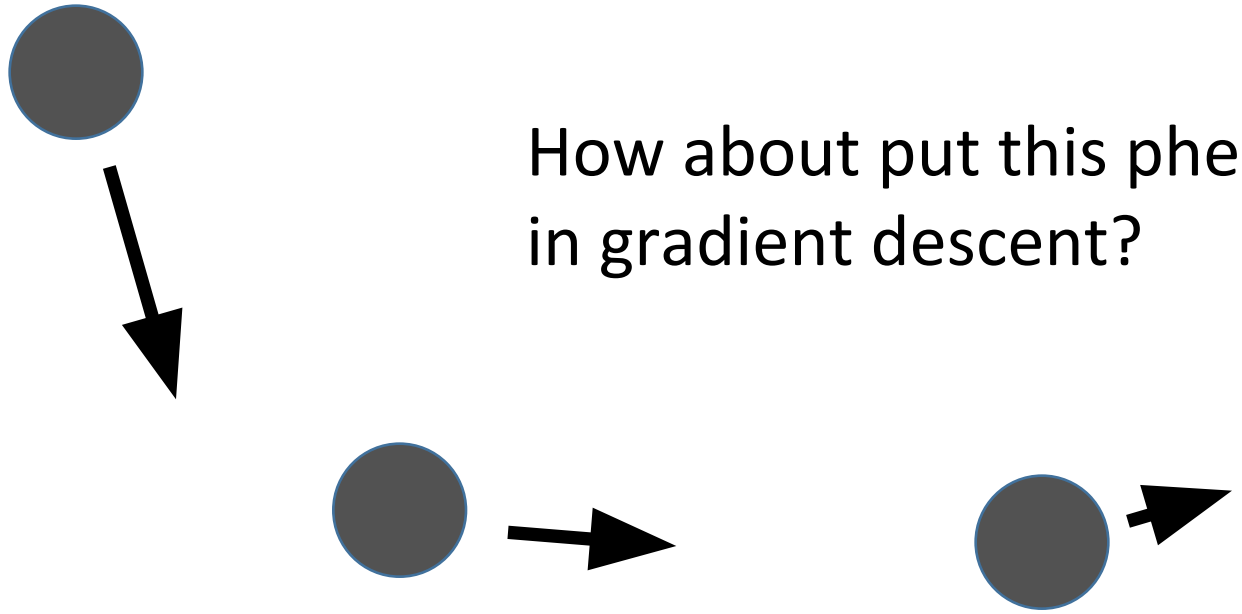
Herbert Robbins and Sutton Monro, A Stochastic Approximation *Method,The Annals of Mathematical Statistics*, Vol. 22, No. 3. (Sep., 1951), pp. 400-407.

Besides local minima ……



In physical world ……

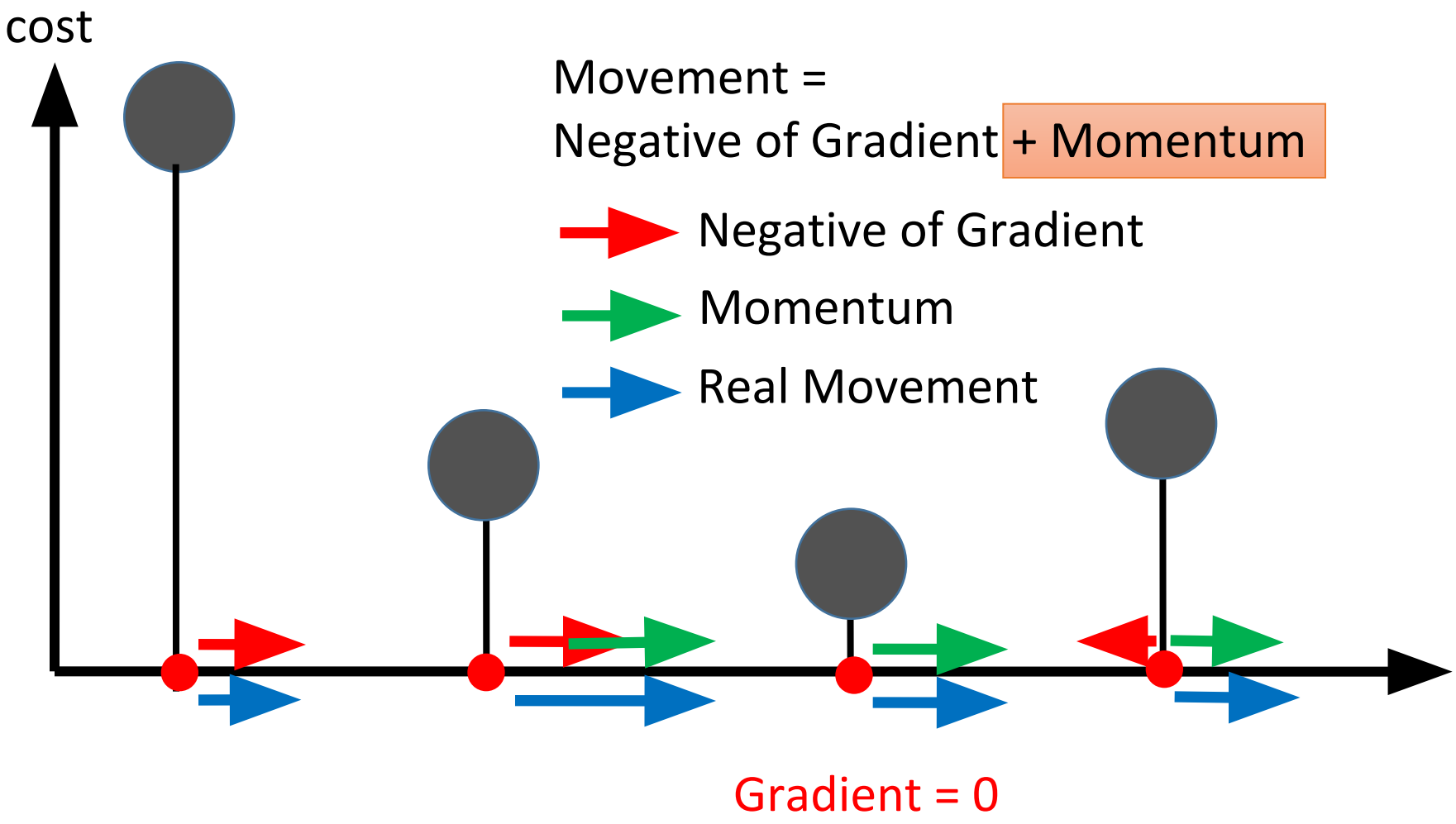
- Momentum



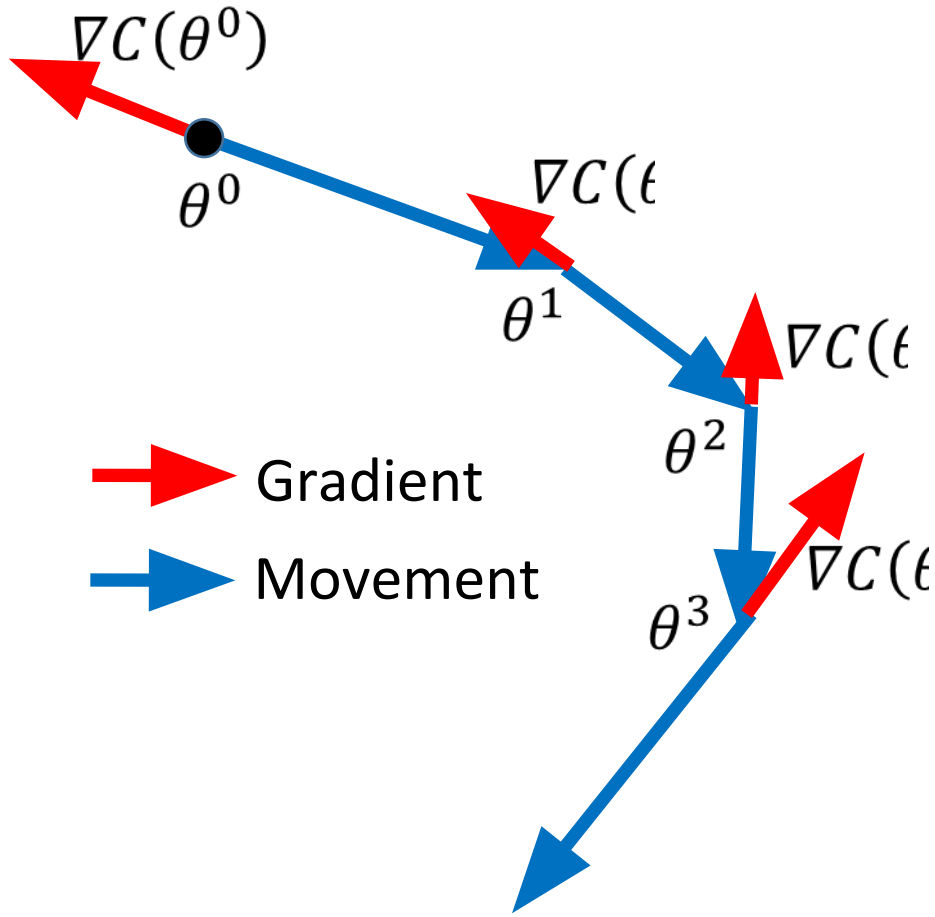
How about put this phenomenon in gradient descent?

Momentum

Still not guarantee reaching global minima, but give some hope ……



Original Gradient descent



Start at position θ^0

Compute gradient at θ^0

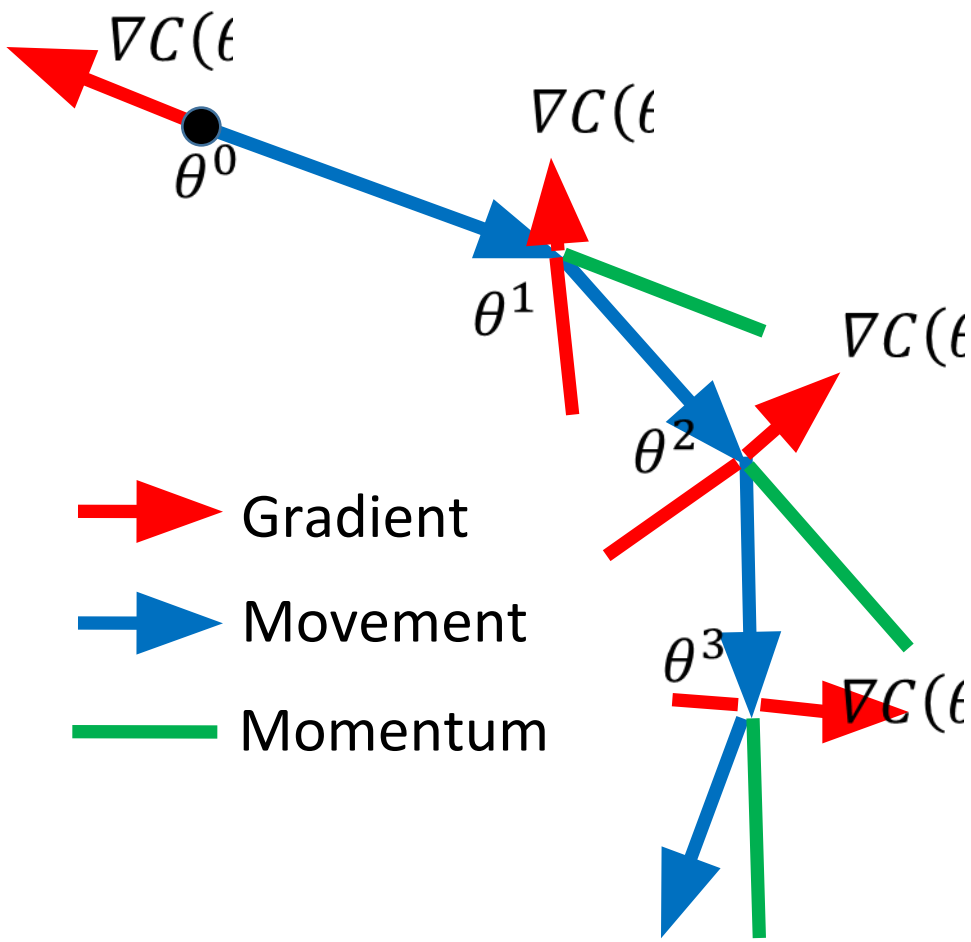
Move to $\theta^1 = \theta^0 - \mu\nabla C(\theta^0)$

Compute gradient at θ^1

Move to $\theta^2 = \theta^1 - \mu\nabla C(\theta^1)$

⋮

Gradient descent with Momentum



Start at point θ^0

Momentum $v^0=0$

(v^i : movement at the i-th update)

Compute gradient at θ^0

Momentum $v^1 = \lambda v^0 - \mu \nabla C(\theta^0)$

Move to $\theta^1 = \theta^0 + v^1$

Compute gradient at θ^1

Momentum $v^2 = \lambda v^1 - \mu \nabla C(\theta^1)$

Move to $\theta^2 = \theta^1 + v^2$

……

Gradient descent with Momentum

v^i is actually the weighted sum of all the previous gradient: $\nabla C(\theta^0), \nabla C(\theta^1), \ldots \nabla C(\theta^{i-1})$

$v^0 = 0$

$v^1 = -\mu \nabla C(\theta^0)$

$v^2 = -\lambda \mu \nabla C(\theta^0) - \mu \nabla C(\theta^1)$

……

Start at point θ^0

Momentum $v^0 = 0$

(v^i : movement at the i-th update)

Compute gradient at θ^0

Momentum $v^1 = \lambda v^0 - \mu \nabla C(\theta^0)$

Move to $\theta^1 = \theta^0 + v^1$

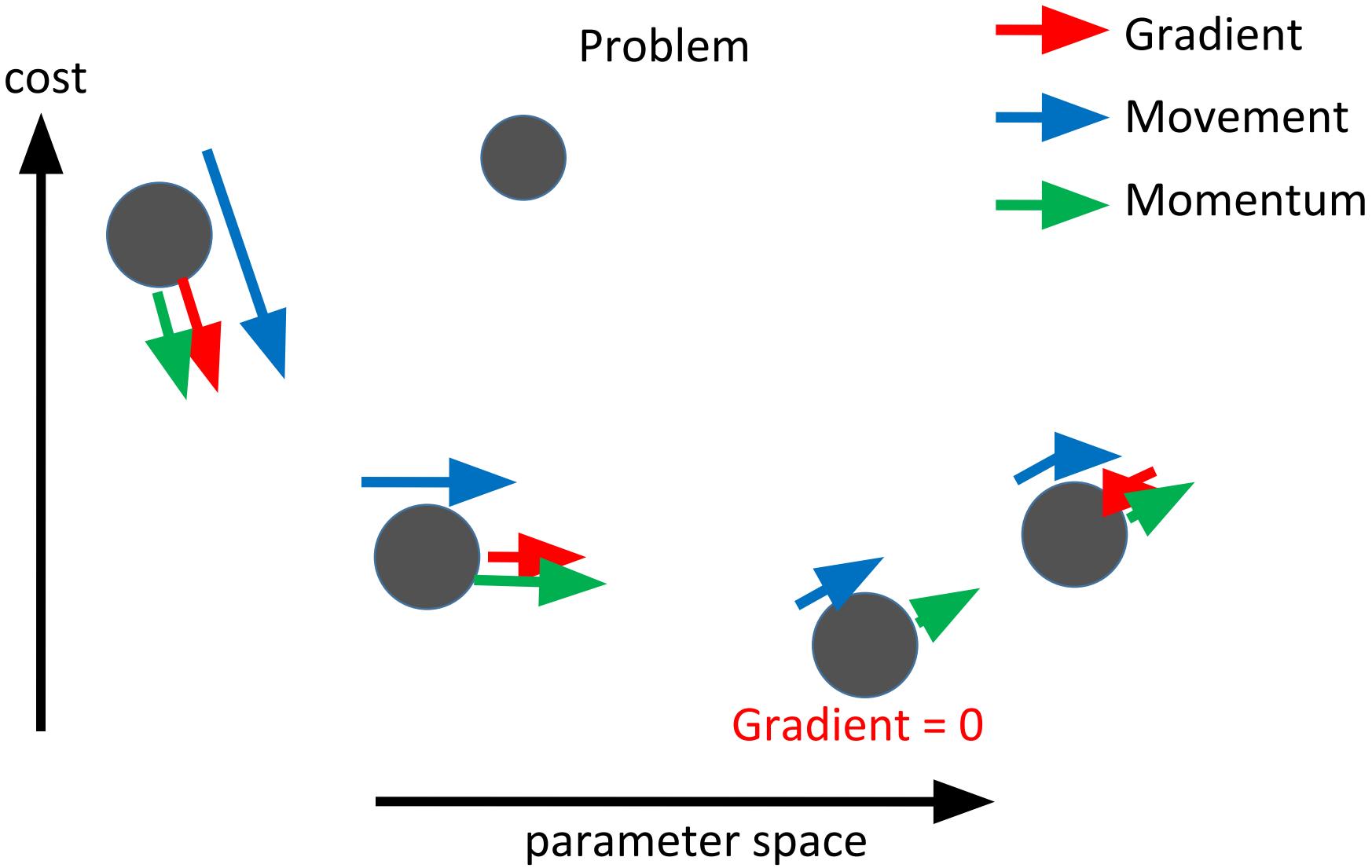
Compute gradient at θ^1

Momentum $v^2 = \lambda v^1 - \mu \nabla C(\theta^1)$

Move to $\theta^2 = \theta^1 + v^2$

……

Gradient descent with Momentum



Momentum update

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

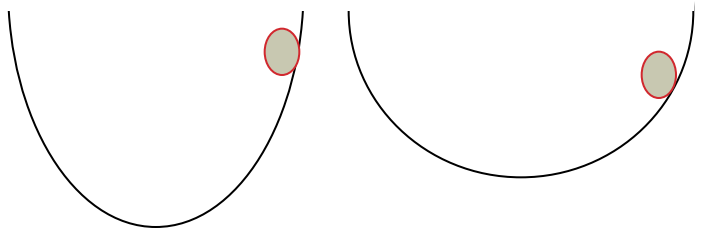


```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

- 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)

Momentum update

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



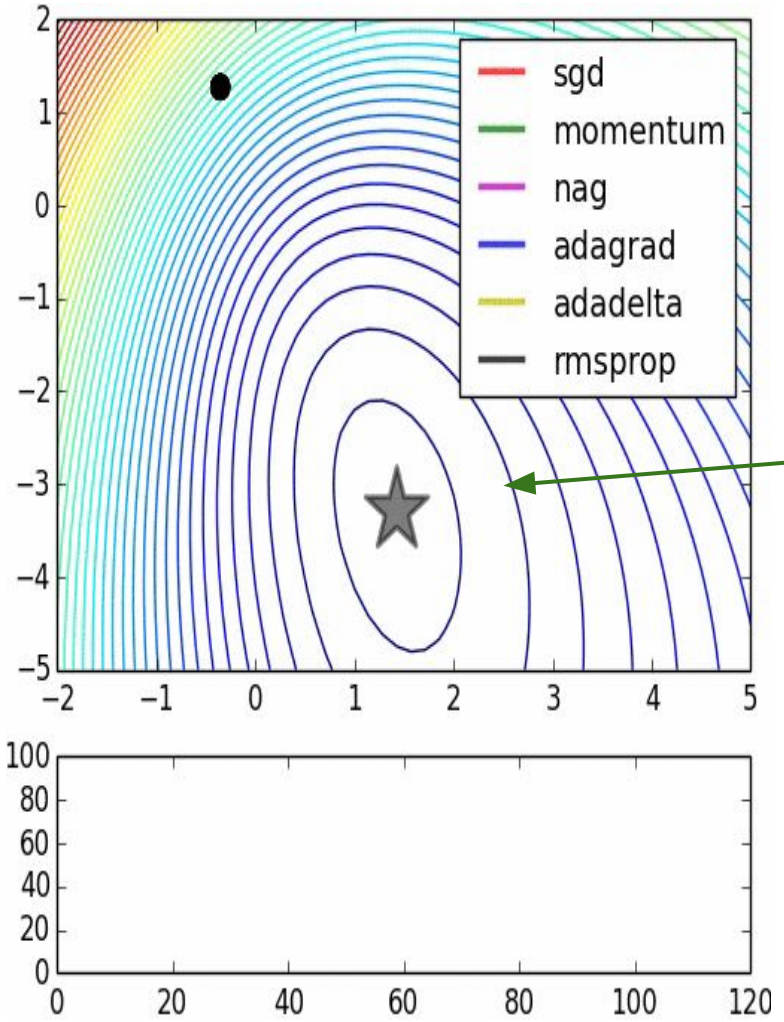
```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

- Allows a velocity to “build up” along shallow directions
- Velocity becomes damped in steep direction due to quickly changing sign

SGD

vs

Momentum

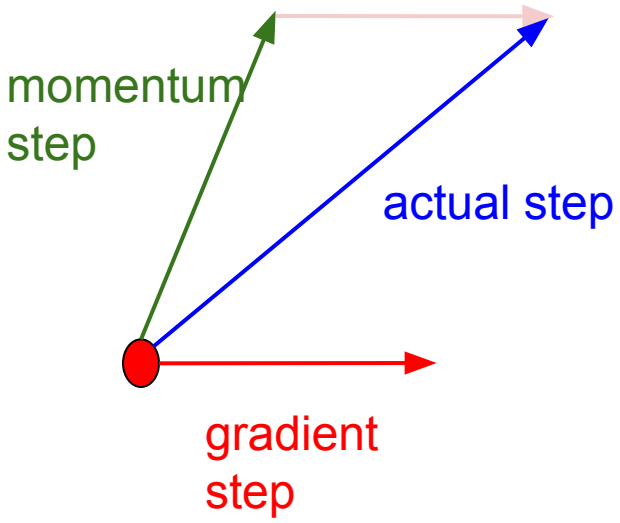


notice momentum overshooting the target, but overall getting to the minimum much faster than vanilla SGD.

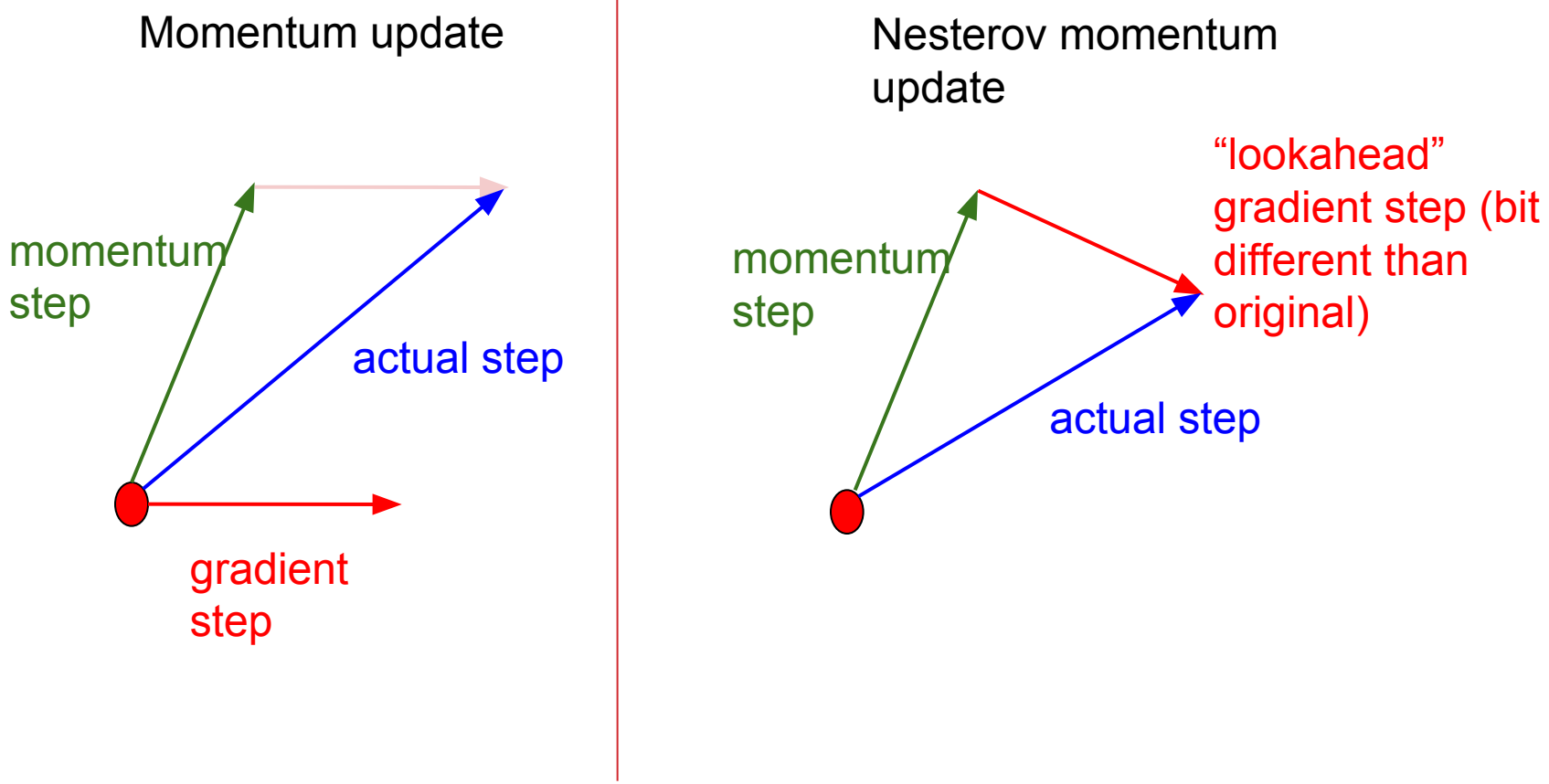
Nesterov Momentum update

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

Ordinary momentum update:

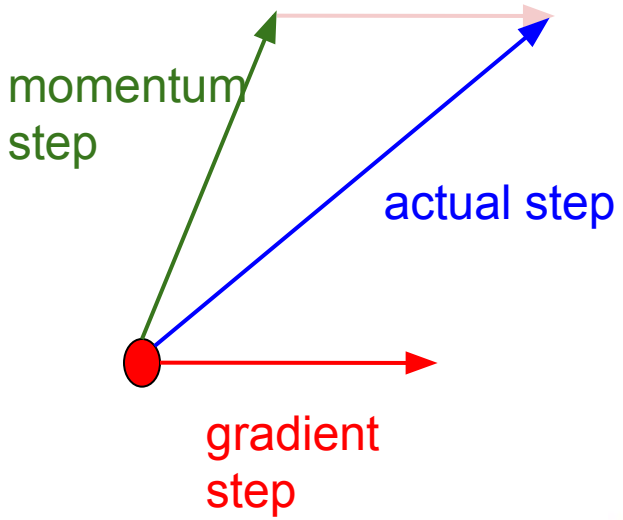


Nesterov Momentum update

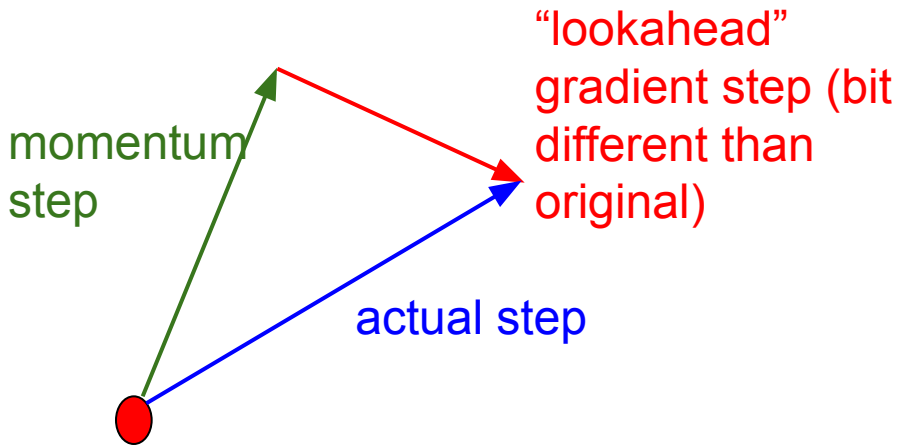


Nesterov Momentum update

Momentum update



Nesterov momentum update



Nesterov: the only difference...

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

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

Nesterov Momentum update

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

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

Slightly inconvenient…
usually we have :

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

Nesterov Momentum update

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

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

Slightly inconvenient…
usually we have :

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

Variable transform and rearranging saves the day:

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

Nesterov Momentum update

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

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

Slightly inconvenient…
usually we have :

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

Variable transform and rearranging saves the day:

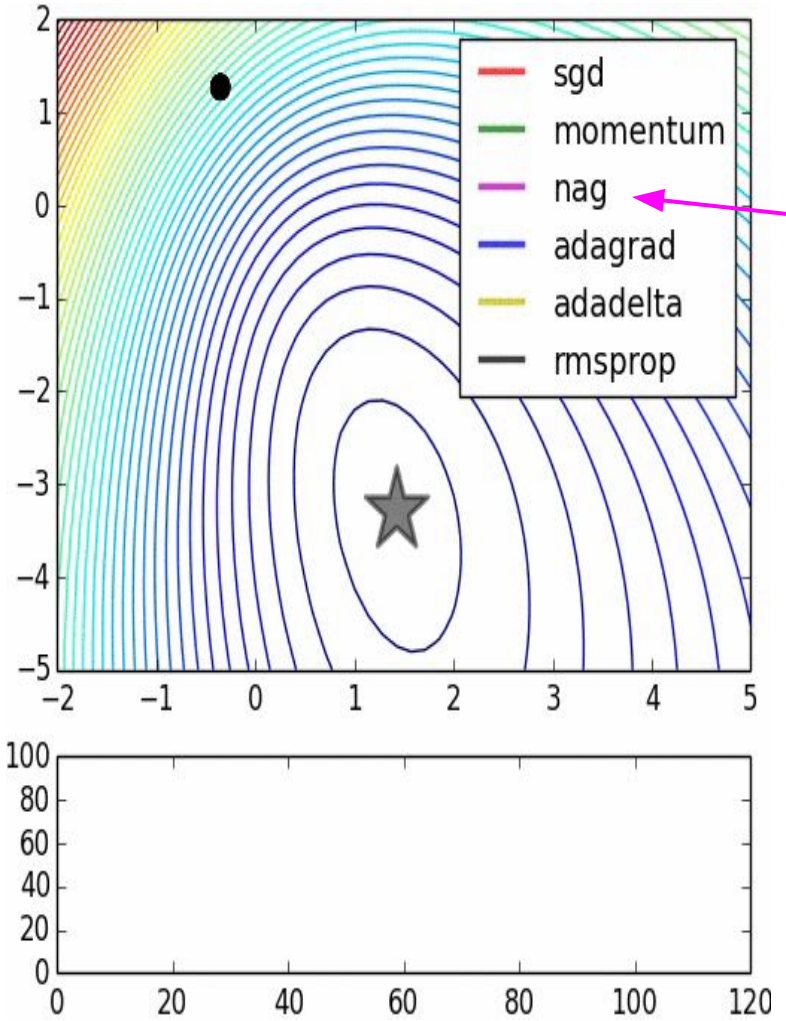
$$\phi_{t-1} = \theta_{t-1} + \mu v_{t-1}$$

Replace all thetas with phis, rearrange and obtain:

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

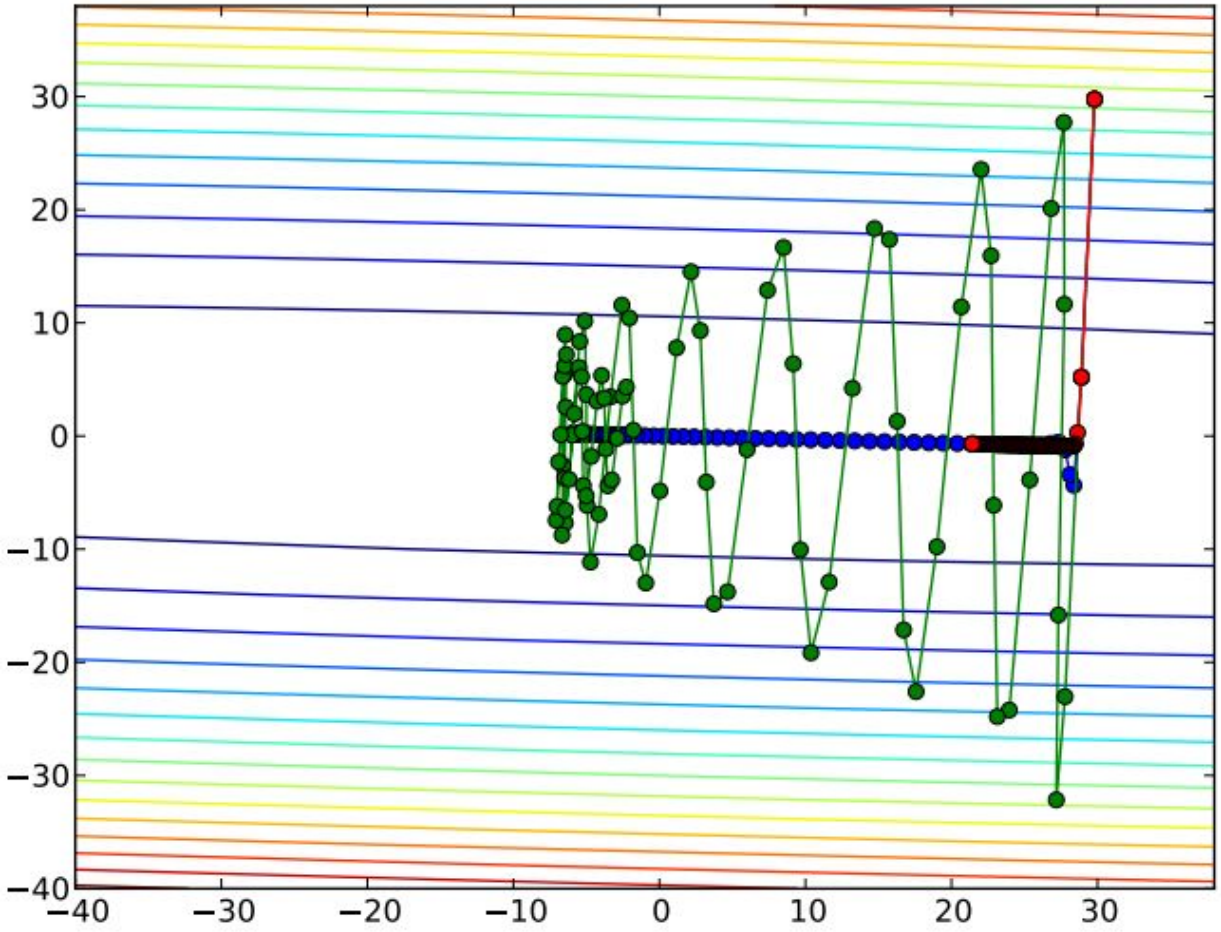
$$\phi_t = \phi_{t-1} - \mu v_{t-1} + (1 + \mu) v_t$$

```
# Nesterov momentum update rewrite
v_prev = v
v = mu * v - learning_rate * dx
x += -mu * v_prev + (1 + mu) * v
```



nag = Nesterov Accelerated Gradient

Gradient descent, Momentum, NAG



Methods:

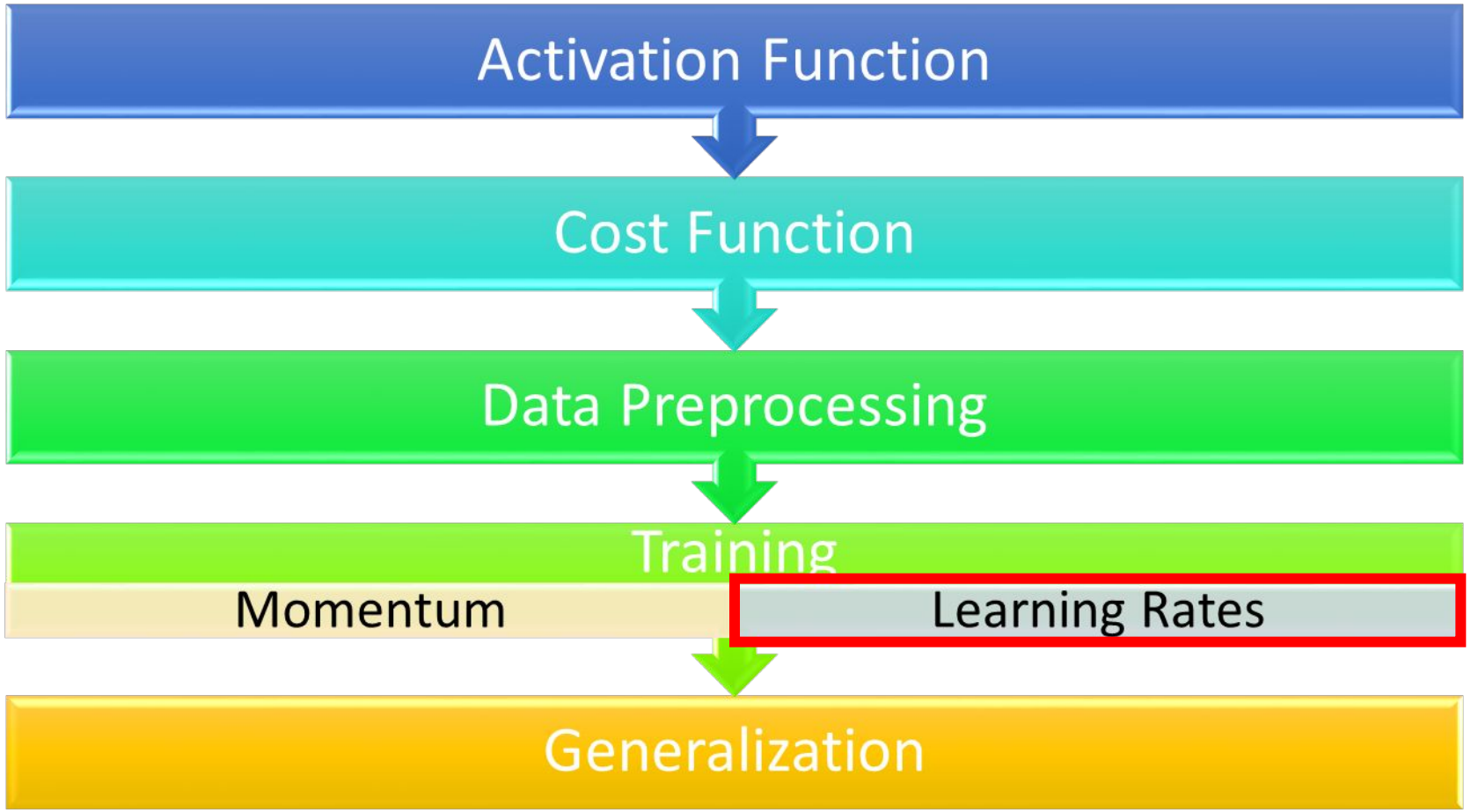
Gradient descent

Momentum

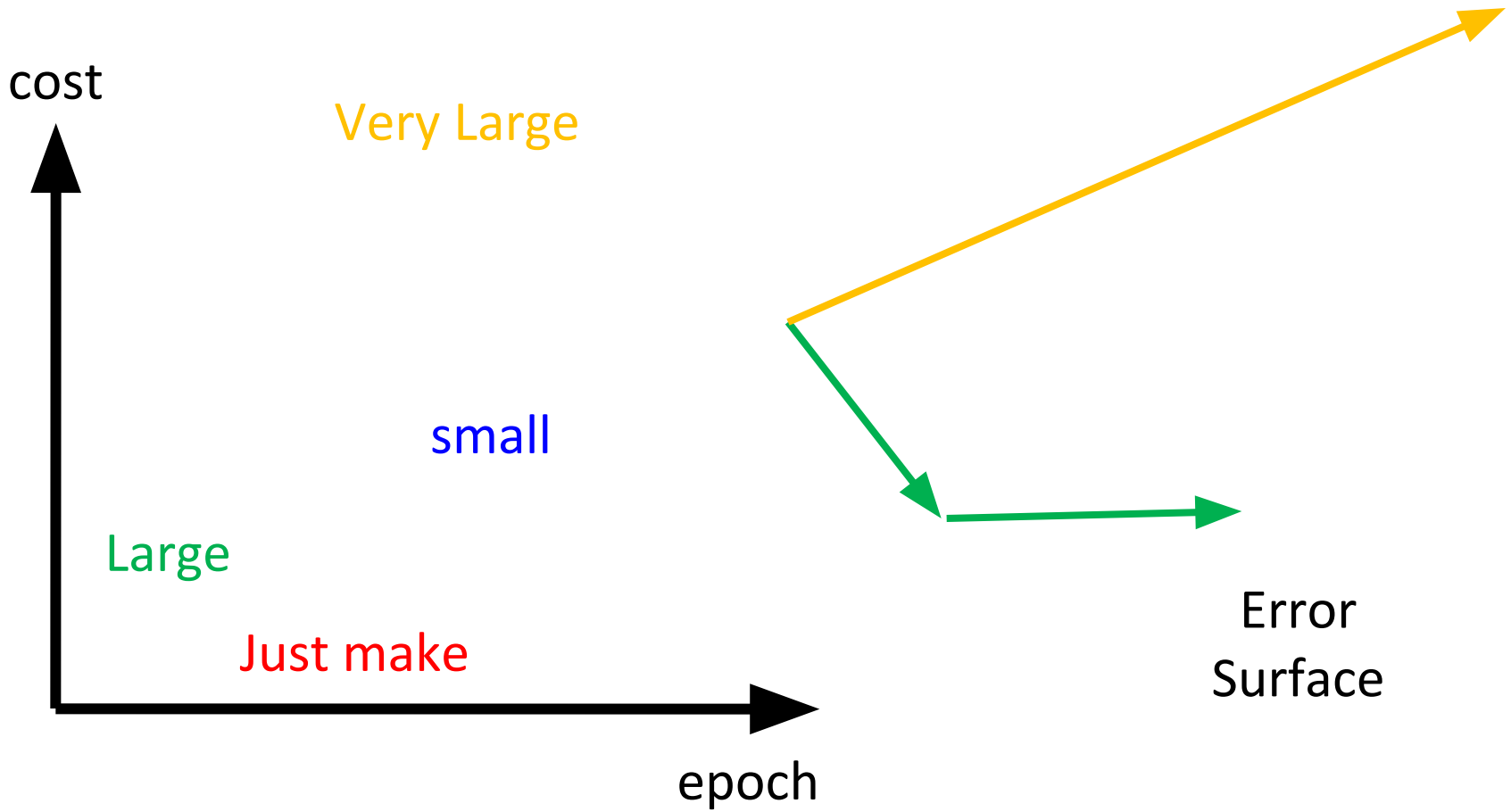
NAG

Source: http://www.cs.toronto.edu/~fritz/absps/momentum.pdf

Outline - Training



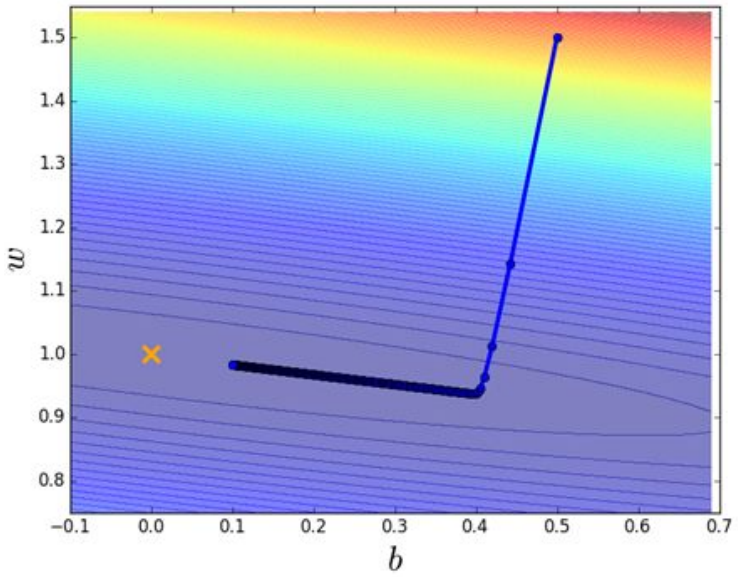
Learning Rates



Learning Rates

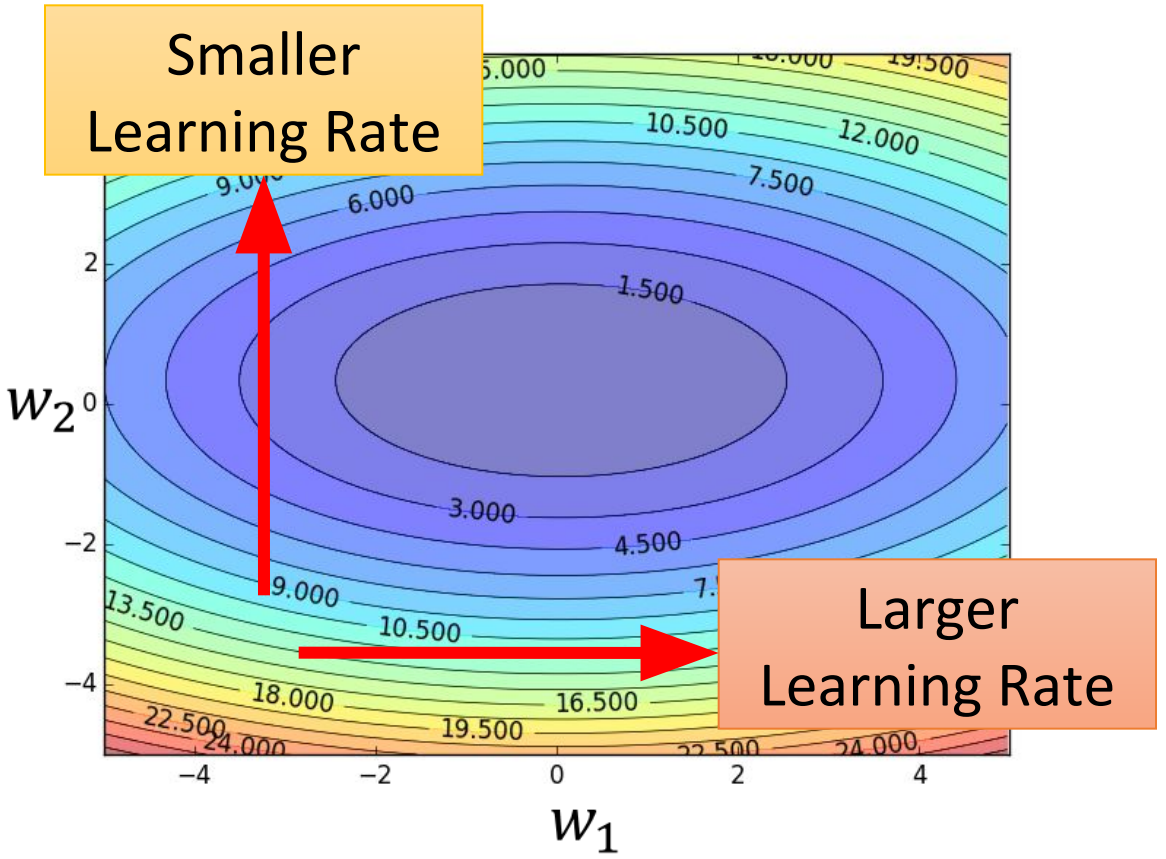
- Popular & Simple Idea: Reduce the learning rate by some factor every few epochs.
  - At the beginning, we are far from a minimum, so we use larger learning rate
  - After several epochs, we are close to a minimum, so we reduce the learning rate
  - E.g. 1/t decay: $\eta^t = \eta/(t+1)$

Not always true



Adaptive Learning Rates

- Each parameter should have different learning rates



RProp

g^t: gradient obtained with w^t

- Each parameter should have different learning rates

Gradient descent:

$$w^{t+1} \leftarrow w^t - \eta g^t$$

$$\begin{cases} 1 & \text{If } g^t > 0 \\ -1 & \text{If } g^t < 0 \end{cases}$$

Another Aspect:

The learning rate is $\frac{\eta}{|g^t|}$.

$$w^{t+1} \leftarrow w^t - \frac{\eta}{|g^t|} g^t$$

RProp - Problem

$$\frac{\partial C}{\partial w} = E[\frac{\partial C_x}{\partial w}]$$

- RProp is problematic for stochastic gradient descent

Gradient descent:

$$w^{t+1} \leftarrow w^t - \eta \frac{\partial C}{\partial w}$$

0.1

1

Stochastic Gradient descent:

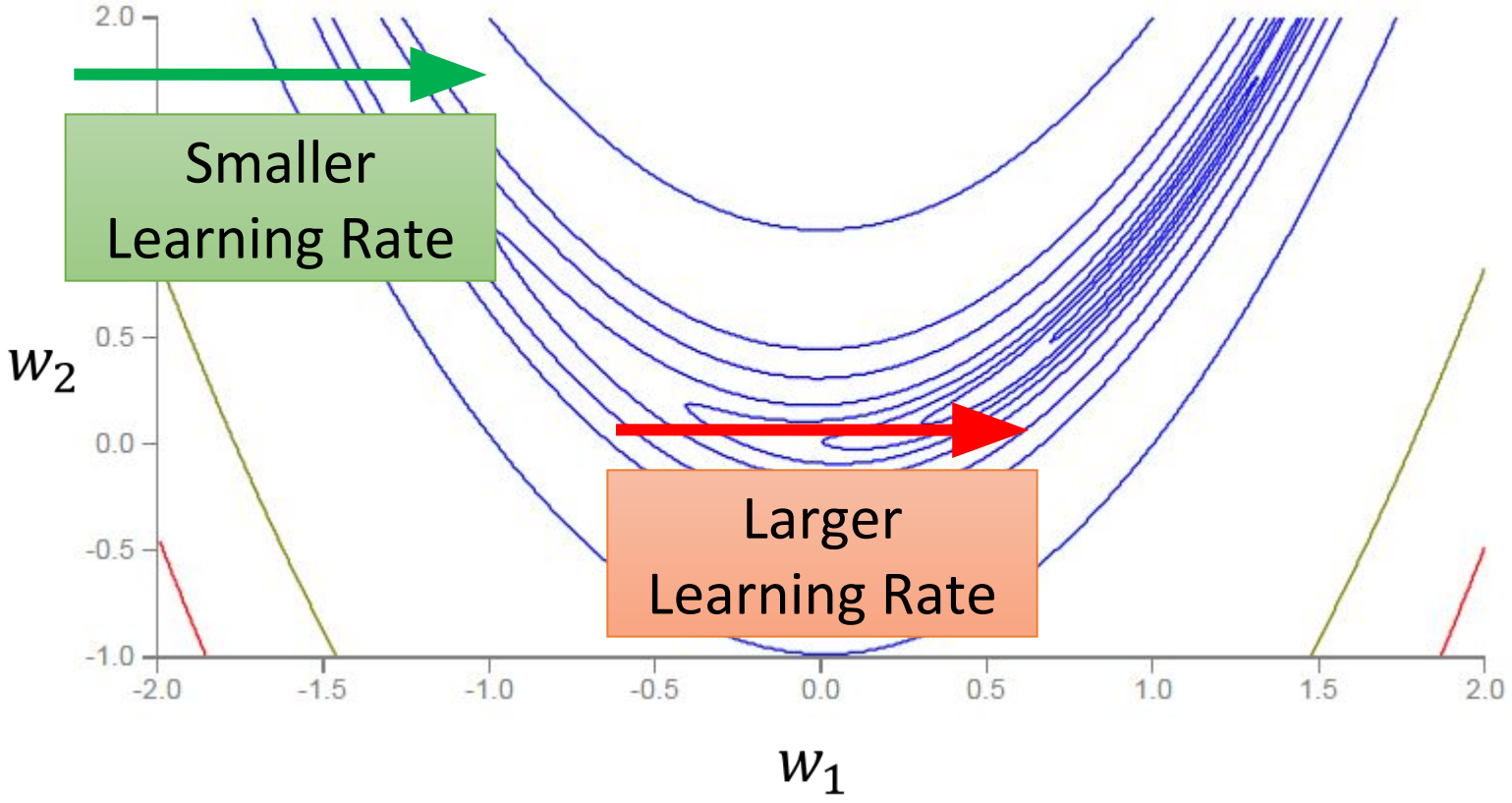
$$w^{t+1} \leftarrow w^t - \eta \frac{\partial C_x}{\partial w}$$

0.9 -0.1 -0.1 -0.1 -0.1

1 -1 -1 -1 -1

RMSProp

Error Surface can be complex in deep learning.



RMSProp

$$w^1 \leftarrow w^0 - \frac{\eta}{\sigma^0} g^0 \qquad \sigma^0 = g^0$$

$$w^2 \leftarrow w^1 - \frac{\eta}{\sigma^1} g^1 \qquad \sigma^1 = \sqrt{\alpha(\sigma^0)^2 + (1-\alpha)(g^1)^2}$$

$$w^3 \leftarrow w^2 - \frac{\eta}{\sigma^2} g^2 \qquad \sigma^2 = \sqrt{\alpha(\sigma^1)^2 + (1-\alpha)(g^2)^2}$$

$$\vdots$$

$$w^{t+1} \leftarrow w^t - \frac{\eta}{\sigma^t} g^t \qquad \sigma^t = \sqrt{\alpha(\sigma^{t-1})^2 + (1-\alpha)(g^t)^2}$$

Adagrad

Duchi *et al*., Adaptive subgradient methods for online learning and stochastic optimization, *JMLR* 2011

- Divide the learning rate by "***average" gradient***

$$w^{t+1} \leftarrow w^t - \frac{\eta}{\sigma} g^t$$

σ: Average gradient of parameter w

Estimated while updating the parameters

If w has small average gradient σ → Larger learning rate

If w has large average gradient σ → Smaller learning rate

Adagrad

$$w^1 \leftarrow w^0 - \frac{\eta}{\sigma^0} g^0 \qquad \sigma^0 = g^0$$

$$w^2 \leftarrow w^1 - \frac{\eta}{\sigma^1} g^1 \qquad \sigma^1 = \sqrt{\frac{1}{2}\left[(g^0)^2 + (g^1)^2\right]}$$

$$w^3 \leftarrow w^2 - \frac{\eta}{\sigma^2} g^2 \qquad \sigma^2 = \sqrt{\frac{1}{3}\left[(g^0)^2 + (g^1)^2 + (g^2)^2\right]}$$

$$\vdots$$

$$w^{t+1} \leftarrow w^t - \frac{\eta}{\sigma^t} g^t \qquad \sigma^t = \sqrt{\frac{1}{t+1}\sum_{i=0}^{t} (g^i)^2}$$

Adagrad

- Divide the learning rate by "***average" gradient***
  - The "average" gradient is obtained while updating the parameters

$$w^{t+1} \leftarrow w^t - \frac{\eta_t}{\sigma^t} g^t \qquad \sigma^t = \sqrt{\frac{1}{t+1}\sum_{i=0}^{t}(g^i)^2}$$

$$\eta^t = \frac{\eta}{\sqrt{t+1}} \qquad w^{t+1} \leftarrow w^t - \frac{\eta}{\sqrt{\sum_{i=0}^{t}(g^i)^2}} g^t$$

1/t decay

Adagrad

Original Gradient Descent

$$\theta^t \leftarrow \theta^{t-1} - \eta \nabla C(\theta^{t-1})$$

Each parameter w are considered separately

$$w^{t+1} \leftarrow w^t - \eta_w g^t \qquad g^t = \frac{\partial C(\theta^t)}{\partial w}$$

Parameter dependent learning rate

$$\eta_w = \frac{\eta}{\sqrt{\sum_{i=0}^{t} (g^i)^2}}$$

constant

Summation of the square of the previous derivatives

Adagrad

$$\eta_w = \frac{\eta}{\sqrt{\sum_{i=0}^{t}(g^i)^2}}$$

w_1	g^0
	0.1

w_2	g^0
	20.0

Learning rate:

$$\frac{\eta}{\sqrt{0.1^2}} = \frac{\eta}{0.1}$$

$$\frac{\eta}{\sqrt{0.1^2 + 0.2^2}} = \frac{\eta}{0.22}$$

Learning rate:

$$\frac{\eta}{\sqrt{20^2}} = \frac{\eta}{20}$$

$$\frac{\eta}{\sqrt{20^2 + 10^2}} = \frac{\eta}{22}$$

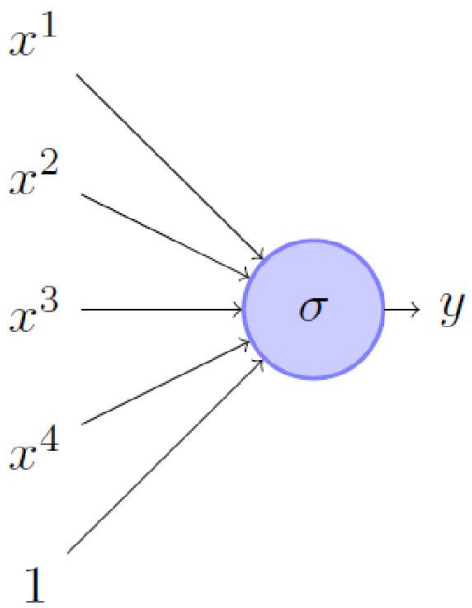
Observation: 1. Learning rate is smaller and smaller for all parameters

2. Smaller derivatives, larger learning rate, and vice versa

Why?

Not the whole story ……

- Adadelta
  - http://www.matthewzeiler.com/pubs/googleTR2012/googleTR2012.pdf
- Adam
  - http://arxiv.org/pdf/1412.6980.pdf
- AdaSecant
  - http://arxiv.org/pdf/1412.7419v4.pdf
- “No more pesky learning rates”
  - Using second order information
  - http://arxiv.org/pdf/1206.1106.pdf



$$y = f(x) = \frac{1}{1+e^{-(\mathbf{w}\cdot\mathbf{x}+b)}}$$

$$\mathbf{x} = \{x^1, x^2, x^3, x^4\}$$

$$\mathbf{w} = \{w^1, w^2, w^3, w^4\}$$

- Given this network, it should be easy to see that given a single point $(\mathbf{x}, y)$...
- $\nabla w^1 = (f(\mathbf{x}) - y) * f(\mathbf{x}) * (1 - f(\mathbf{x})) * x^1$
- $\nabla w^2 = (f(\mathbf{x}) - y) * f(\mathbf{x}) * (1 - f(\mathbf{x})) * x^2$ ... so on
- If there are n points, we can just sum the gradients over all the n points to get the total gradient
- What happens if the feature x^2 is very sparse? (*i.e.*, if its value is 0 for most inputs)
- ∇w^2 will be 0 for most inputs (see formula) and hence w^2 will not get enough updates
- If x^2 happens to be sparse as well as important we would want to take the updates to w^2 more seriously
- Can we have a different learning rate for each parameter which takes care of the frequency of features ?

Some Slides: Prof. Mitesh M. Khapra, IIT Madras

Intuition

- Decay the learning rate for parameters in proportion to their update history (more updates means more decay)

Update rule for Adagrad

$$v_t = v_{t-1} + (\nabla w_t)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t + \epsilon}} * \nabla w_t$$

... and a similar set of equations for b_t

Intuition

- Adagrad decays the learning rate very aggressively (as the denominator grows)
- As a result after a while the frequent parameters will start receiving very small updates because of the decayed learning rate
- To avoid this why not decay the denominator and prevent its rapid growth

Update rule for RMSProp

$$v_t = \beta * v_{t-1} + (1 - \beta)(\nabla w_t)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t + \epsilon}} * \nabla w_t$$

... and a similar set of equations for b_t

am — a most popular choice in deep learning optimization

- Adam can be looked at as a combination of RMSprop and Stochastic Gradient Descent with momentum.
- It uses the squared gradients to scale the learning rate like RMSprop and it takes advantage of momentum by using moving average of the gradient instead of gradient itself like SGD with momentum.

Adam is an adaptive learning rate method, which means, it computes individual learning rates for different parameters. Its name is derived from **adaptive** moment **estimation**, and the reason it's called that is because Adam uses estimations of first and second moments of gradient to adapt the learning rate for each weight of the neural network.

Now, what is moment ? *N*-th moment of a random variable is defined as the expected value of that variable to the power of n. More formally:

$$m_n = E[X^n]$$

Courtesy: Vitaly Bushaev, through www.towardsdatacience.com

Note, that gradient of the cost function of neural network can be considered a random variable, since it usually evaluated on some small random batch of data. The first moment is mean, and the second moment is **uncentered variance** (meaning we don't subtract the mean during variance calculation).

To estimates the moments, Adam utilizes exponentially moving averages, computed on the gradient evaluated on a current mini-batch:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

where m and v are moving averages, g is gradient on current mini-batch, and betas — new introduced hyper-parameters of the algorithm. They have really good default values of 0.9 and 0.999 respectively. Almost no one ever changes these values. The vectors of moving averages are initialized with zeros at the first iteration.

To see how these values correlate with the moment, defined as in first equation, let's take look at expected values of our moving averages. Since m and v are estimates of first and second moments, we want to have the following property:

$$E[m_t] = E[g_t]$$
$$E[v_t] = E[g_t^2]$$

Expected values of the estimators should equal the parameter we're trying to estimate, as it happens, the parameter in our case is also the expected value. If these properties held true, that would mean, that we have **unbiased estimators**.

Now, we will see that these do not hold true for the our moving averages. Because we initialize averages with zeros, the estimators are biased towards zero. Let's prove that for m (the proof for v would be analogous). To prove that we need to formula for m to the very first gradient. Let's try to unroll a couple values of m to see the pattern we're going to use:

$$m_0 = 0$$
$$m_1 = \beta_1 m_0 + (1-\beta_1) g_1 = (1-\beta_1) g_1$$
$$m_2 = \beta_1 m_1 + (1-\beta_1) g_2 = \beta_1 (1-\beta_1) g_1 + (1-\beta_1) g_2$$
$$m_3 = \beta_1 m_2 + (1-\beta_1) g_3 = \beta_1^2 (1-\beta_1) g_1 + \beta_1 (1-\beta_1) g_2 + (1-\beta_1) g_3$$

- For convenience we will denote ∇w_t as g_t and β_1 as β

$$
\begin{aligned}
m_t &= \beta * m_{t-1} + (1-\beta) * g_t \\
m_0 &= 0 \\
m_1 &= \beta m_0 + (1-\beta) g_1 \\
&= (1-\beta) g_1 \\
m_2 &= \beta m_1 + (1-\beta) g_2 \\
&= \beta(1-\beta) g_1 + (1-\beta) g_2 \\
m_3 &= \beta m_2 + (1-\beta) g_3 \\
&= \beta(\beta(1-\beta) g_1 + (1-\beta) g_2) + (1-\beta) g_3 \\
&= \beta^2(1-\beta) g_1 + \beta(1-\beta) g_2 + (1-\beta) g_3 \\
&= (1-\beta) \sum_{i=1}^{3} \beta^{3-i} g_i
\end{aligned}
$$

- In general,

$$
m_t = (1-\beta) \sum_{i=1}^{t} \beta^{t-i} g_i
$$

- So we have, $m_t = (1-\beta)\sum_{i=1}^{t}\beta^{t-i}g_i$
- Taking Expectation on both sides

$$E[m_t] = E[(1-\beta)\sum_{i=1}^{t}\beta^{t-i}g_i]$$

$$E[m_t] = (1-\beta)E[\sum_{i=1}^{t}\beta^{t-i}g_i]$$

$$E[m_t] = (1-\beta)\sum_{i=1}^{t}E[\beta^{t-i}g_i]$$

$$= (1-\beta)\sum_{i=1}^{t}\beta^{t-i}E[g_i]$$

- Assumption: All g_i's come from the same distribution i.e. $E[g_i] = E[g]\ \forall i$

$$E[m_t] = (1-\beta)\sum_{i=1}^{t}(\beta)^{t-i}E[g_i]$$

$$= E[g](1-\beta)\sum_{i=1}^{t}(\beta)^{t-i}$$

$$= E[g](1-\beta)(\beta^{t-1}+\beta^{t-2}+\cdots+\beta^0)$$

$$= E[g](1-\beta)\frac{1-\beta^t}{1-\beta}$$

the last fraction is the sum of a GP with common ratio $= \beta$

$$E[m_t] = E[g](1-\beta^t)$$

$$E[\frac{m_t}{1-\beta^t}] = E[g]$$

$$E[\hat{m}_t] = E[g](\because \frac{m_t}{1-\beta^t} = \hat{m}_t)$$

Hence we apply the bias correction because then the expected value of $\hat{m}_t$ is the same as the expected value of g_t

Now we need to correct the estimator, so that the expected value is the one we want. This step is usually referred to as bias correction. The final formulas for our estimator will be as follows:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

The only thing left to do is to use those moving averages to scale learning rate individually for each parameter. The way it's done in Adam is very simple, to perform weight update we do the following:

$$w_t = w_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

where w is model weights, eta (look like the letter n) is the step size (it can depend on iteration). And that's it, that's the update rule for Adam.

utting it all together…..

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1^{st} moment vector)
$v_0 \leftarrow 0$ (Initialize 2^{nd} moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

Kingma, D and Ba, J . (2015) *Adam: A method for Stochastic Optimization*. ICLR 2015, Available at: https://arxiv.org/pdf/1412.6980.pdf