Lecture 6:
"Backpropagation Algorithm"

Dr. Ehab Essa

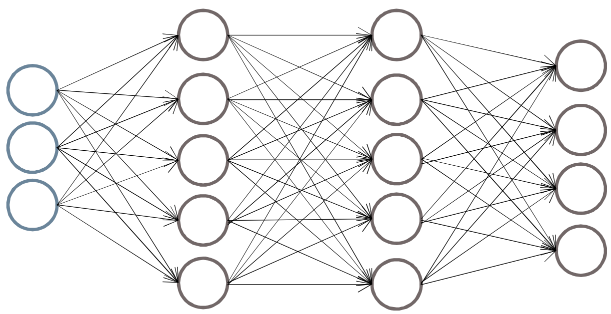
Computer Science Department, Mansoura University

Today

- Backpropagation Algorithm

- Lecture Notes:
  - **Chapter 6**, Fundamentals of Neural Networks: Architectures, Algorithms, and Applications by Laurene Fausett

Neural Network (Classification)



$$\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$$

$L =$ total no. of layers in network

$s_l =$ no. of units (not counting bias unit) in layer l

<u>Binary classification</u>

$y = 0 \text{ or } 1$

1 output unit

<u>Multi-class classification</u> (K classes)

$y \in \mathbb{R}^K$ E.g. $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$, $\begin{bmatrix}0\\1\\0\\0\end{bmatrix}$, $\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$ $\begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

pedestrian car motorcycle truck

K output units

Learning Algorithm

- Assume that a set of examples $\Im$={**x**(n),**t**(n)}, n=1,…,N is given. **x**(n) is the *input vector* of dimension m_0 and t(n) is the *desired response* vector of dimension M

- Thus an *error signal*, $e_{j(n)} = y_{j(n)} - t_j(n)$ can be defined for the output neuron j.

- We can derive a learning algorithm for an MLP by assuming an optimization approach which is based on the **steepest descent direction**, I.e.

$$\Delta \boldsymbol{w}(n) = -\alpha \boldsymbol{g}(n)$$

Where **g**(n) is the gradient vector of the cost function and α is the *learning rate*.

Learning Algorithm II

- The algorithm that it is derived from the steepest descent direction is called **back-propagation**
- Assume that we define a SSE instantaneous cost function (I.e. per example) as follows:

$$\mathrm{E}(n) = \frac{1}{2} \sum_{j \in C} e_j^2(n)$$

- Where C is the set of all *output neurons*.
- If we assume that there are N examples in the set $\Im$ then the *average squared error* is:

$$\mathrm{E}_{av} = \frac{1}{N} \sum_{n=1}^{N} \mathrm{E}(n)$$

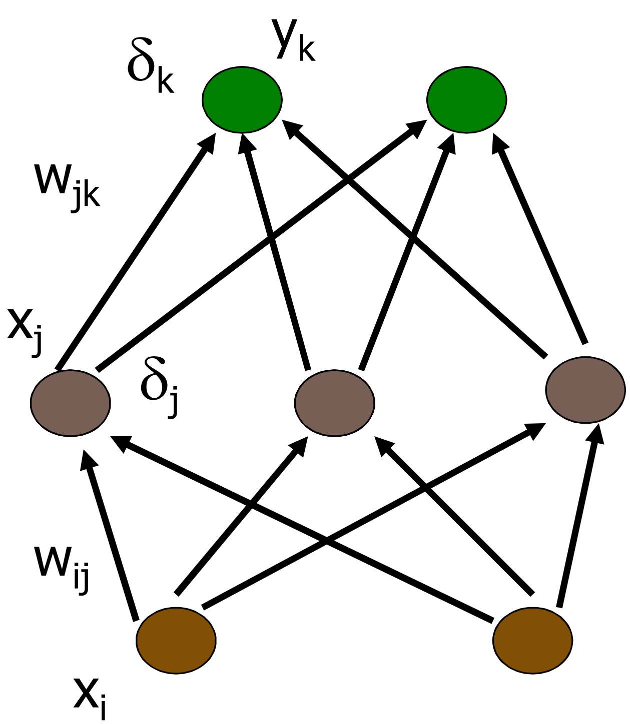
Learning Algorithm III

- We have two ways to calculate the gradient with respect to E_{av} or E(n).
  - **Batch** mode : In the first case we calculate the gradient per *epoch* (i.e. in all patterns N). E_{av} is used. (average gradient over the examples in the batch)
  - **Online** or **Stochastic** mode: updates model parameters according to the gradient calculated from one *pattern*. E(n) is used

- Assume that we use the online mode for the rest of the calculation. The gradient is defined as:

$$\nabla g(n) = \frac{\partial \mathrm{E}(n)}{\partial w_{ji}(n)}$$

Backpropagation



Backward step:
propagate errors from output to hidden layer

Forward step:
Propagate activation from input to output layer

The derivative of the sigmoid function

$$\frac{d}{dx}g(x) = \frac{d}{dx}\left(\frac{1}{1+e^{-x}}\right)$$

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

$$= -(1+e^{-x})^{-2}(-e^{-x})$$

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

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

$$= \frac{1+e^{-x}}{(1+e^{-x})^2} - \left(\frac{1}{1+e^{-x}}\right)^2$$

$$= g(x) - g(x)^2$$

$$g(x)' = g(x)(1-g(x))$$

Chain Rule

- In calculus, the chain rule is a formula for computing the derivative of the composition of two or more functions
- Consider z to be a function of the variable y, which is itself a function of x (y and z are therefore dependent variables), and so, z becomes a function of x as well, The chain rule may be written:

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}.$$

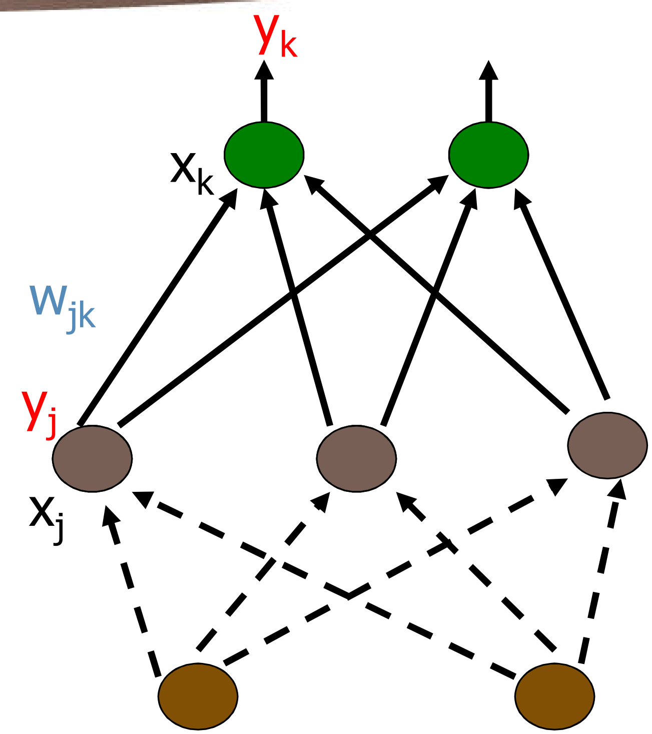
- If $z(y) = e^y$ and y = 2x, so that $z(y(x)) = e^{2x}$.

$$\frac{dz(y)}{dx} = \frac{dz(y)}{dy} \times \frac{dy(x)}{dx} = e^y \cdot 2$$

Derivation of learning rules

Output layer node

- $\frac{\partial E}{\partial W_{jk}} = \frac{\partial E}{\partial y_k} \cdot \frac{\partial y_k}{\partial x_k} \cdot \frac{\partial x_k}{\partial W_{jk}}$
- $\frac{\partial E}{\partial W_{jk}} = \frac{\partial}{\partial W_{jk}} \frac{1}{2} \sum_{k \in K} (y_k - t_k)^2$
- $\frac{\partial E}{\partial W_{jk}} = (y_k - t_k) \frac{\partial}{\partial W_{jk}} y_k$
- $\frac{\partial E}{\partial W_{jk}} = (y_k - t_k) \frac{\partial}{\partial W_{jk}} g(x_k)$
- $\frac{\partial E}{\partial W_{jk}} = (y_k - t_k) g(x_k)(1 - g(x_k)) \frac{\partial}{\partial W_{jk}} x_k$
- $\frac{\partial E}{\partial W_{jk}} = (y_k - t_k) y_k (1 - y_k) y_j$



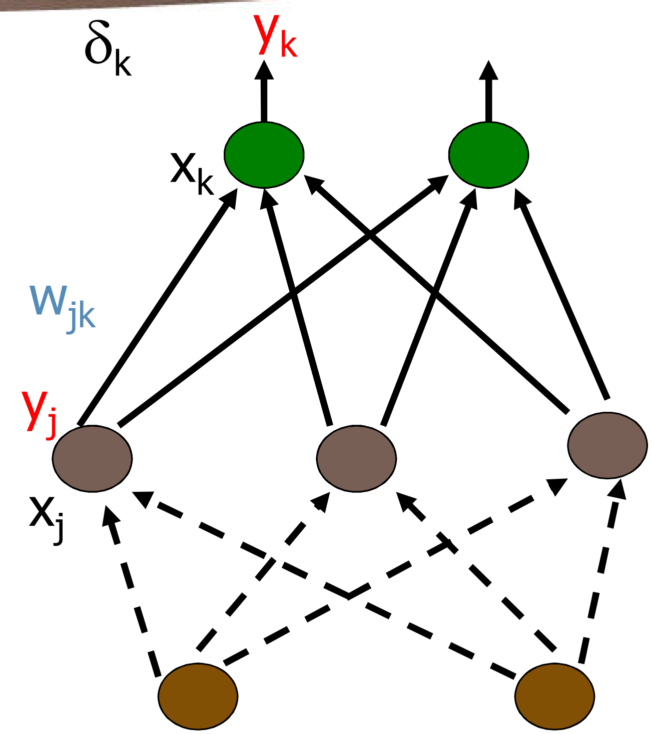
x pre-activation
y after-activation

- For notation purposes I will define δ_k to be the expression $(y_k - t_k) y_k (1 - y_k)$, so we can rewrite the equation above as

Output layer node

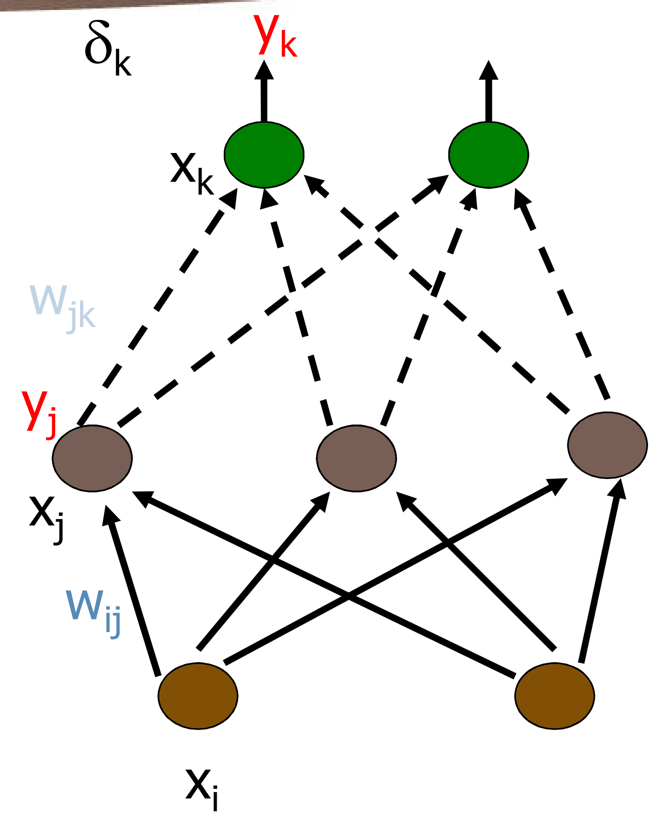
- $\frac{\partial E}{\partial W_{jk}} = (y_k - t_k)y_k(1 - y_k)y_j$
- $\frac{\partial E}{\partial W_{jk}} = \delta_k \; y_j$

Where $\delta_k = (y_k - t_k)y_k(1 - y_k)$,



Hidden Layer Node

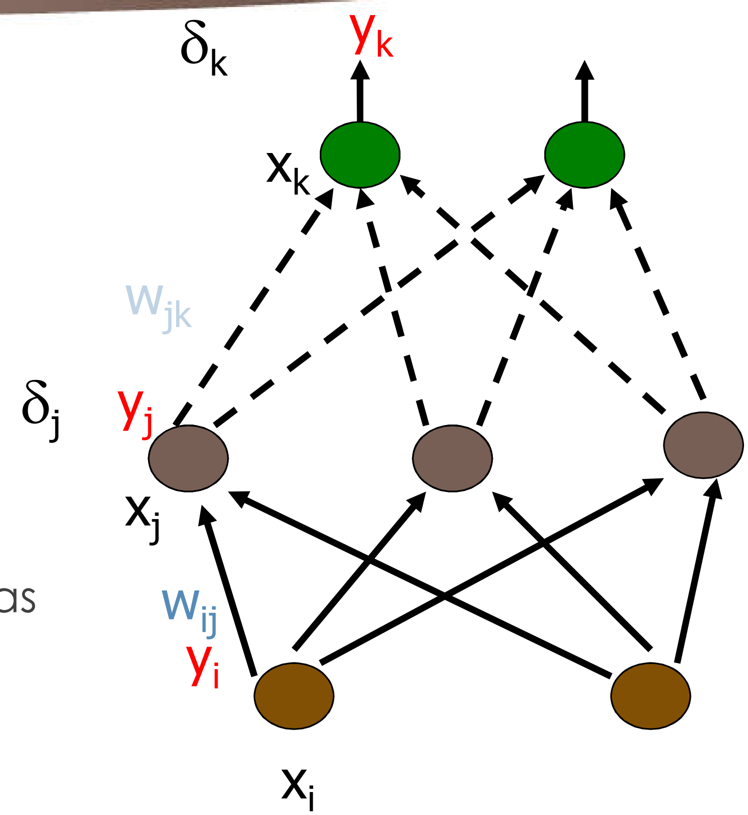
- $\frac{\partial E}{\partial W_{ij}} = \frac{\partial E}{\partial y_k} \cdot \frac{\partial y_k}{\partial x_k} \cdot \frac{\partial x_k}{\partial y_j} \cdot \frac{\partial y_j}{\partial x_j} \cdot \frac{\partial x_j}{\partial W_{ij}}$
- $\frac{\partial E}{\partial W_{ij}} = \frac{\partial}{\partial W_{ij}} \frac{1}{2} \sum_{k \in K} (y_k - t_k)^2$
- $\frac{\partial E}{\partial W_{ij}} = \sum_{k \in K} (y_k - t_k) \frac{\partial}{\partial W_{ij}} y_k$
- $\frac{\partial E}{\partial W_{ij}} = \sum_{k \in K} (y_k - t_k) \frac{\partial}{\partial W_{ij}} g(x_k)$
- $\frac{\partial E}{\partial W_{ij}} = \sum_{k \in K} (y_k - t_k)\, g(x_k)(1 - g(x_k)) \frac{\partial}{\partial W_{ij}} x_k$
- $\frac{\partial E}{\partial W_{ij}} = \sum_{k \in K} (y_k - t_k)\, y_k(1 - y_k) \frac{\partial x_k}{\partial y_j} \frac{\partial y_j}{\partial W_{ij}}$
- $\frac{\partial E}{\partial W_{ij}} = \sum_{k \in K} (y_k - t_k)\, y_k(1 - y_k) W_{jk} \frac{\partial y_j}{\partial W_{ij}}$



Hidden Layer Node

- $\frac{\partial E}{\partial W_{ij}} = \sum_{k \in K}(y_k - t_k)\, y_k(1 - y_k) W_{jk} \frac{\partial y_j}{\partial W_{ij}}$
- $\frac{\partial E}{\partial W_{ij}} = \frac{\partial y_j}{\partial W_{ij}} \sum_{k \in K}(y_k - t_k)\, y_k(1 - y_k) W_{jk}$
- $\frac{\partial E}{\partial W_{ij}} = y_j(1 - y_j) \frac{\partial x_j}{\partial W_{ij}} \sum_{k \in K}(y_k - t_k)\, y_k(1 - y_k) W_{jk}$
- $\frac{\partial E}{\partial W_{ij}} = y_j(1 - y_j) y_i \sum_{k \in K}(y_k - t_k)\, y_k(1 - y_k) W_{jk}$
- But, recalling our definition of δ_k we can write this as

$$\frac{\partial E}{\partial W_{ij}} = y_i y_j (1 - y_j) \sum_{k \in K} \delta_k W_{jk}$$

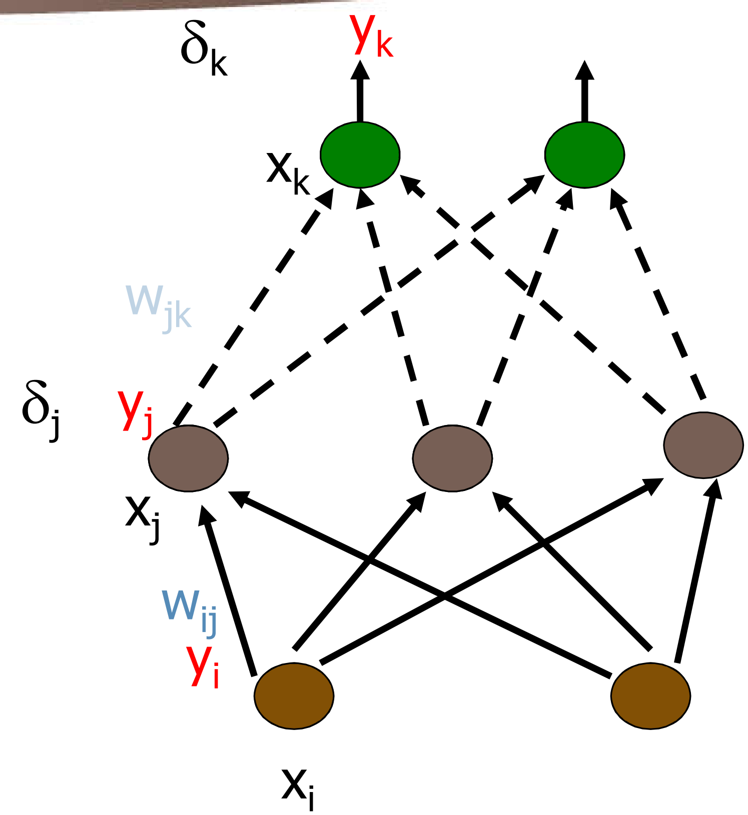


Hidden Layer Node

- Similar to before we will now define all terms besides the y_i to be δ_j , so we have

$$\frac{\partial E}{\partial W_{ij}} = y_i \delta_j$$

Where $\delta_j = y_j(1 - y_j) \sum_{k \in K} \delta_k W_{jk}$



Backpropagation Algorithm

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - For each training example $<(x_1,\ldots x_n),t>$ Do

Forward phase

- Input the instance $(x_1,\ldots,x_n)$ to the network and compute the network outputs y_k

Backward phase

- For each output unit k
  - $\delta_k = y_k(1-y_k)(y_k - t_k)$
- For each hidden unit j
  - $\delta_j = y_j(1-y_j) \sum_k w_{j,k} \delta_k$
- For each network weight $w_{i,j}$ Do

In case, weight are from input layer to hidden layer: $y_i=x_i$

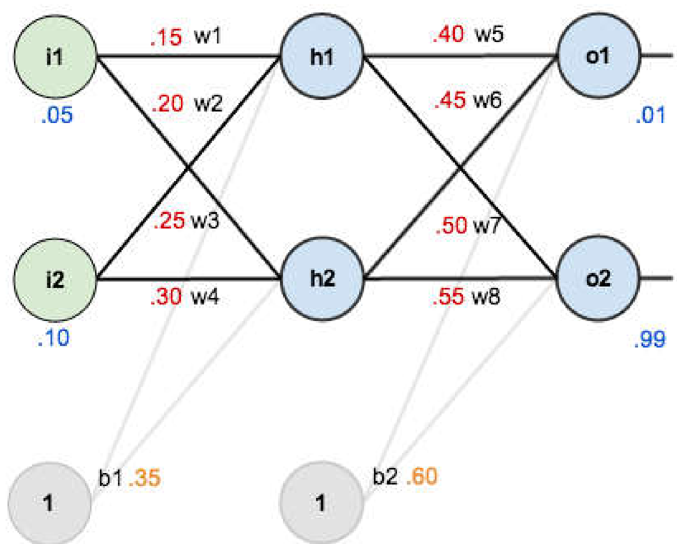
Learning rules

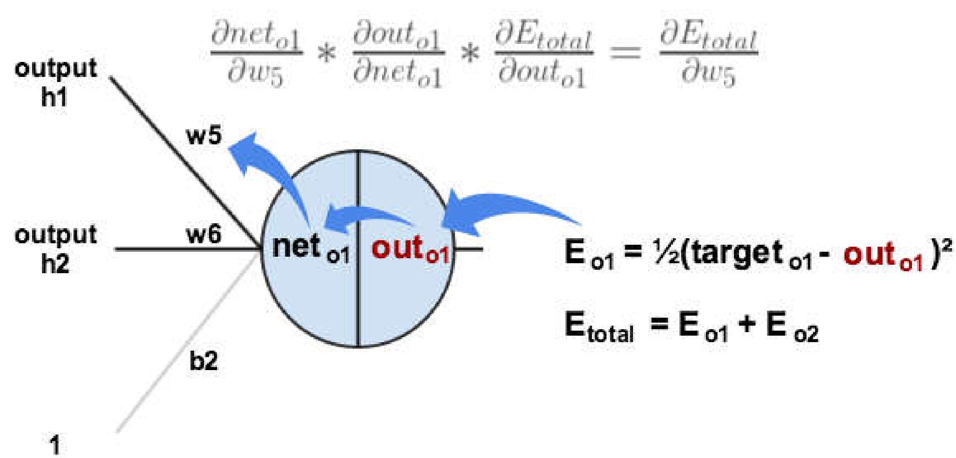
$w_{i,j}=w_{i,j}+\Delta w_{i,j}$ where $\Delta wij = -\alpha \delta_j y_i$

(Bias) $w_{0,j}=w_{0,j}+\Delta w_{0,j}$ where $\Delta w_{0j} = -\alpha \delta_j$

Numerical Example

- Check the example
  - https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

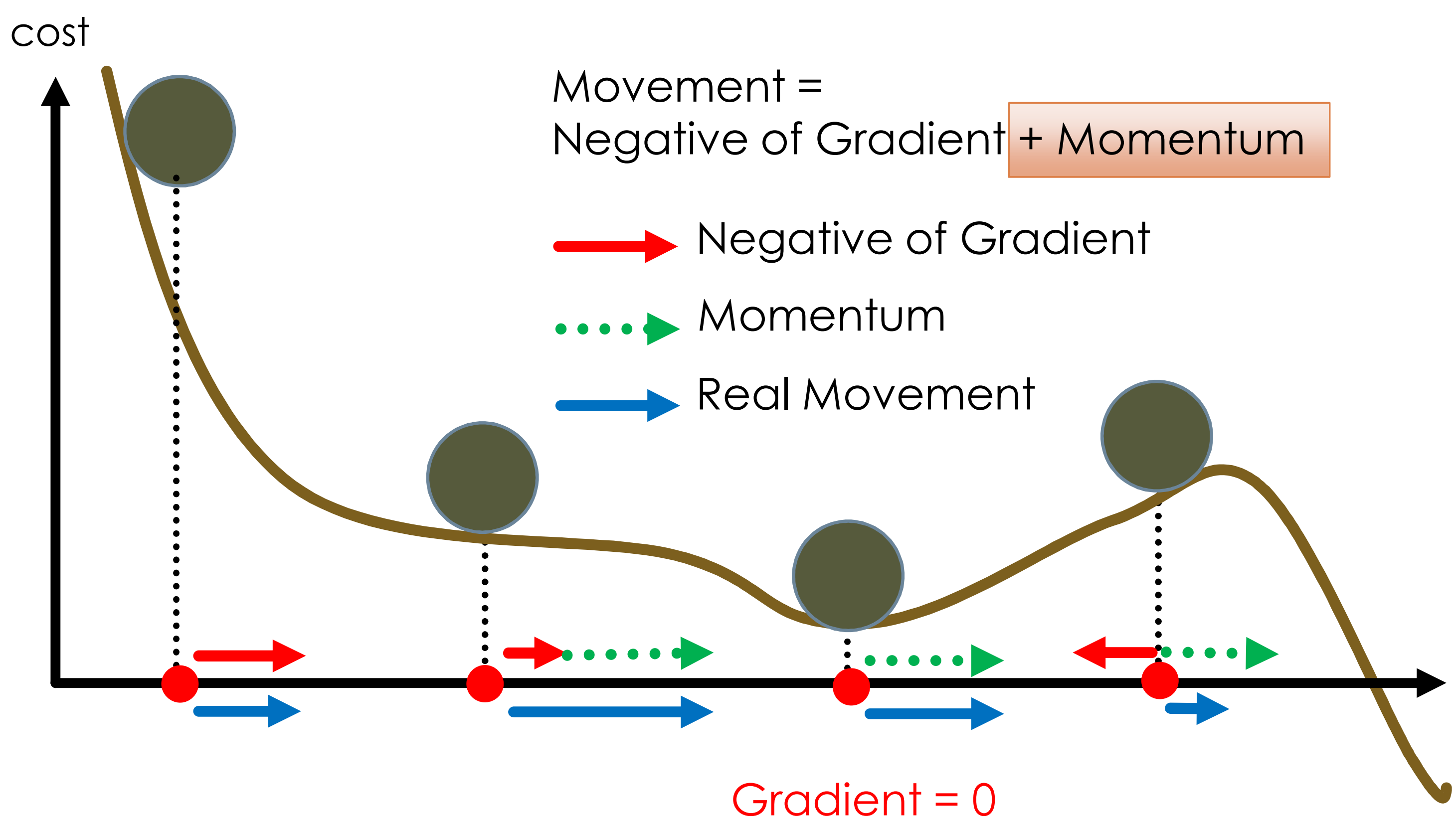




Variations of BP nets

Adding **momentum term**

- Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations.
- It does this by adding a fraction of the update vector of the past time step to the current update vector.



Variations of BP nets

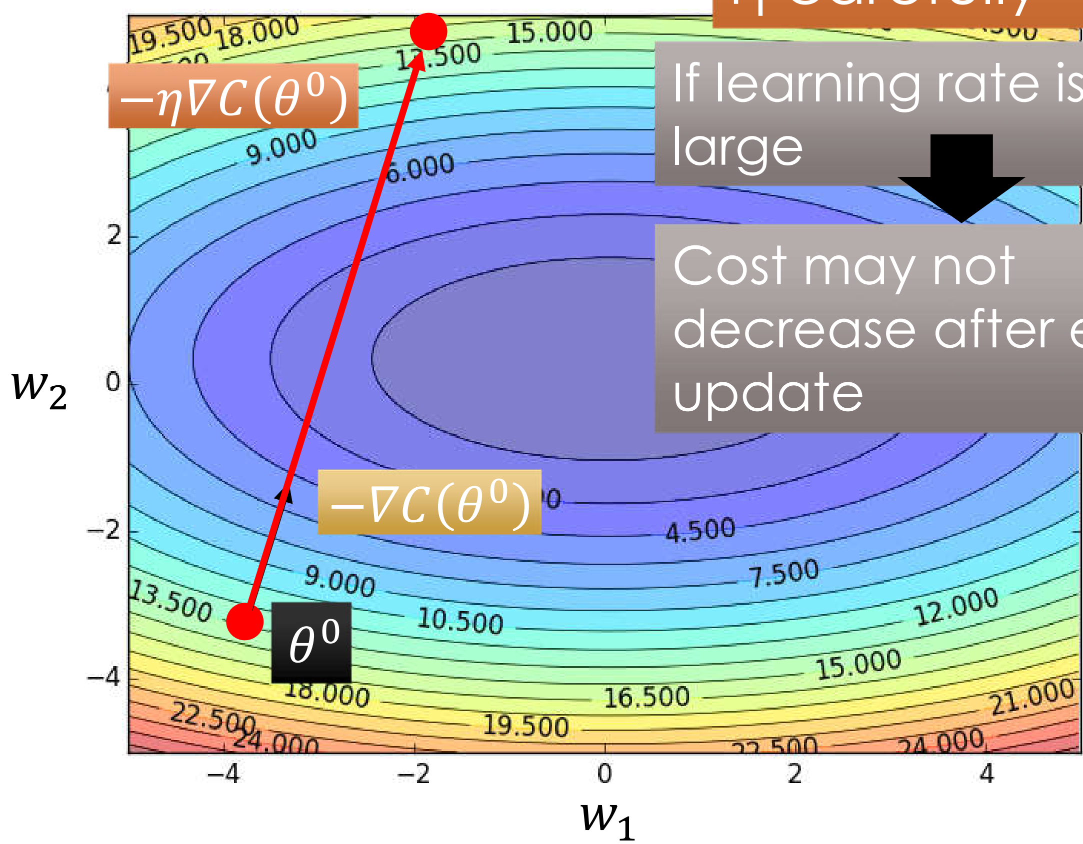
Adding **momentum term**

- Weights update at time t+1 contains the momentum of the previous updates, e.g.,
- $$\Delta w_{ij}(t+1) = \alpha \cdot \delta_j \cdot y_i + \mu \cdot \Delta w_{ij}(t), \text{ where } \ 0 < \mu < \alpha << 1$$

  $$\text{then } \Delta w_{ij}(t+1) = \sum_{s=1}^{t} \mu^{t-s} \alpha \cdot \delta_j(s) \cdot y_i(s)$$
- an exponentially weighted sum of all previous updates
- Avoid sudden change of directions of weight update (smoothing the learning process)
- Error is no longer monotonically decreasing

Learning Rate

Set the learning rate η carefully

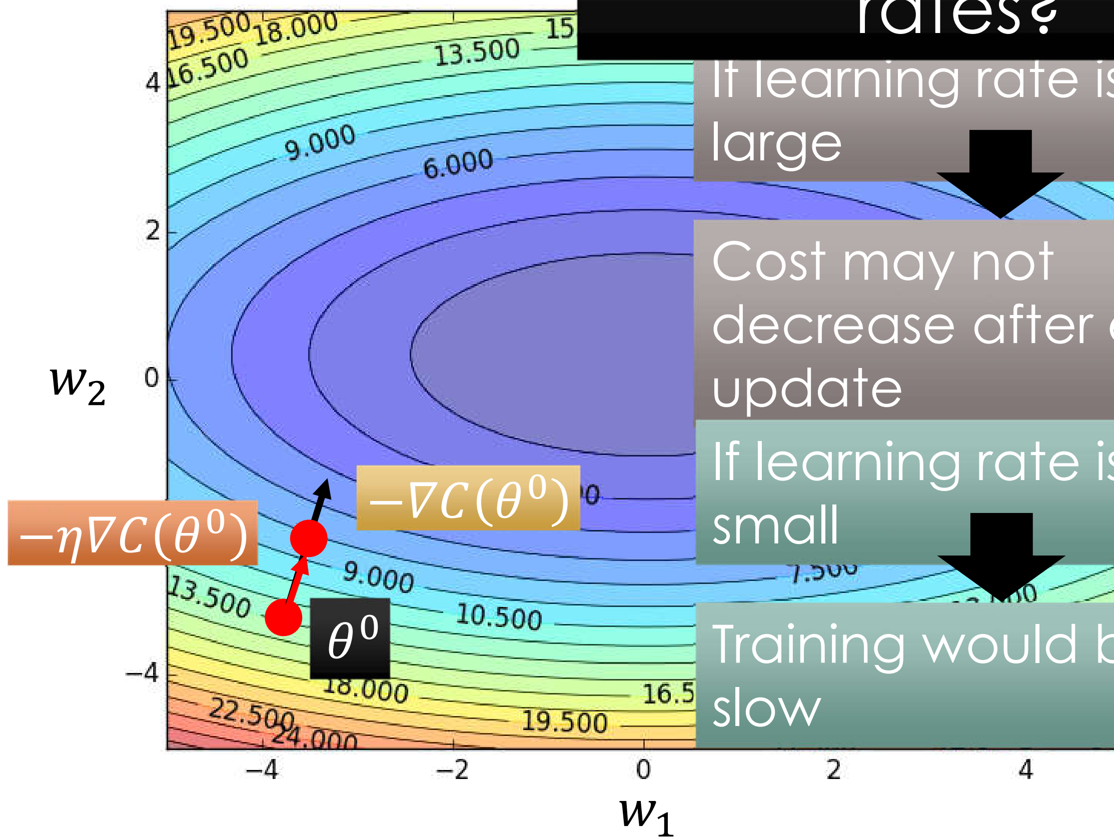


If learning rate is too large

Cost may not decrease after each update

Learning Rate

Can we give different parameters different learning rates?



If learning rate is too large

Cost may not decrease after each update

If learning rate is too small

Training would be too slow

Adagrad

Original Gradient Descent

$$w^t \leftarrow w^{t-1} - \eta \nabla E(w^{t-1})$$

Each parameter w are considered separately

$$w^{t+1} \leftarrow w^t - \boxed{\eta_w} \underline{g^t} \qquad \underline{g^t} = \frac{\partial E(w^t)}{\partial w}$$

Parameter dependent learning rate

constant

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

Summation of the square of the previous derivatives

Adagrad

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

w_1

g^0
0.1

Learning rate:

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

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

w_2

g^0
20.0

Learning rate:

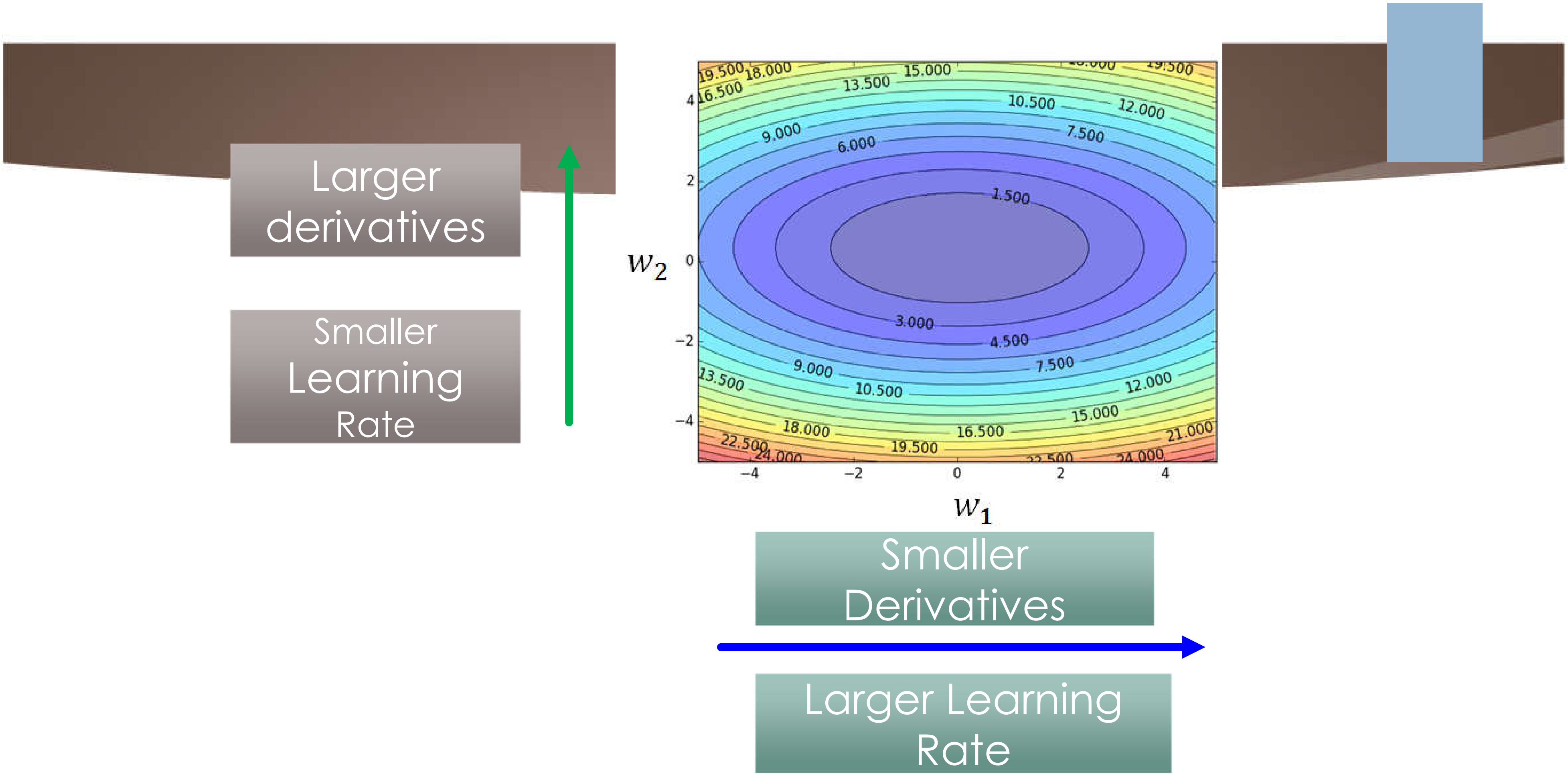
$$\frac{\eta}{\sqrt{20^2}} = \frac{\eta}{20}$$

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

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

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

Why?



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

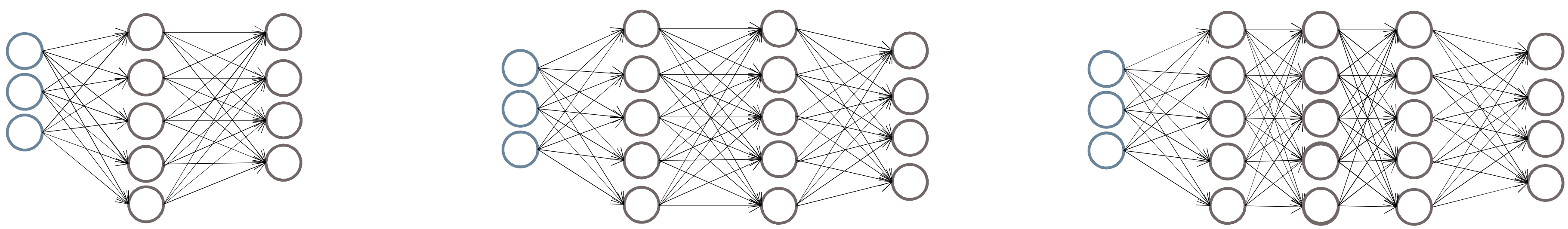
Why?

Not the whole story ……

- Adagrad [John Duchi, JMLR'11]
- RMSprop
  - https://www.youtube.com/watch?v=O3sxAc4hxZU
- Adadelta [Matthew D. Zeiler, arXiv'12]
- Adam [Diederik P. Kingma, ICLR'15]
- AdaSecant [Caglar Gulcehre, arXiv'14]
- "No more pesky learning rates" [Tom Schaul, arXiv'12]

Training a neural network

Pick a network architecture (connectivity pattern between neurons)



No. of input units: Dimension of features $x^{(i)}$
No. output units: Number of classes
Reasonable default: 1 hidden layer, or if >1 hidden layer, have same no. of hidden units in every layer (usually the more the better)

Cost function

Logistic regression:

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

Regularization term

Neural network:

$$h_\Theta(x) \in \mathbb{R}^K \quad (h_\Theta(x))_i = i^{th} \text{ output}$$

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_\Theta(x^{(i)}))_k)\right]$$
$$+\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2$$