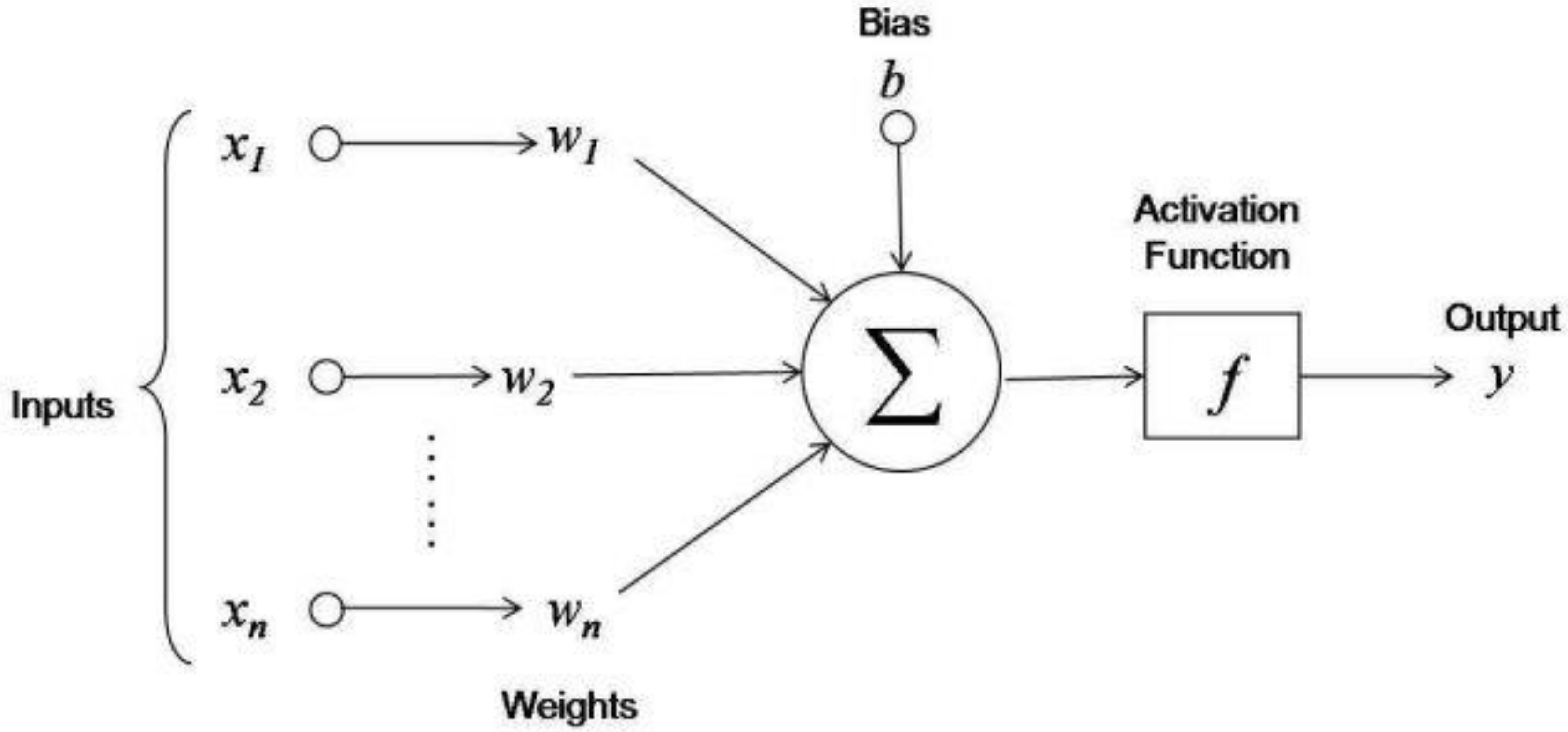
Course Outline

- TOPICS
  1. What is Machine Learning and Image Processing
  2. Traditional Features, K-NN classifier
  3. Linear Classification
  4. Perceptron Algorithm, Sigmoid Activation Function, Gradient Descent
  5. Stochastic Gradient Descent, Back-Propagation
  6. Multi-Layer Neural Network
  7. Convolution and Pooling
  8. Mid-Term Examination
  9. Mid-Term Examination
  10. Convolutional Neural Networks.
  11. Training Convolutional Neural Networks: Hyper-Parameters, Activation functions, initialization, dropout, batch normalization
  12. Recurrent Neural Networks
  13. Applications of Convolutional Neural Networks for Image Segmentation and Object Classification
  14. Project Presentations

One Layer Neural Network



Gradient Descent

Gradient-Descent(*training_examples*, η)

Each training example is a pair of the form <(x_1,…x_n),t> where (x_1,…,x_n) is the vector of input values, and t is the target output value, η is the learning rate (e.g. 0.1)

- Initialize each w_i to some small random value
- Until the termination condition is met,
- Do
  - Initialize each Δw_i to zero
  - For each <(x_1,…x_n),t> in *training_examples*
  - Do
    - Input the instance (x_1,…,x_n) to the linear unit and compute the output o
    - For each linear unit weight w_i
    - Do
      - $w_i = w_i + \eta\ (t-o)\ x_i$
  - For each linear unit weight w_i
  - Do
    - $w_i=w_i+\Delta w_i$

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d) x_{id}$$

Why Gradient Descent

The Gradient descent looks like to estimate roots of equations

Recall from the Numerical Methods(a.k.a Taylor Series, Newton Raphon, etc…)

Any smooth function can be approximated as a polynomial.

Take $\boldsymbol{x = x_{i+1}}$ Then $\boldsymbol{f(x) \approx f(x_i)}$ ***zero order*** approximation

$$f(x) \cong f(x_i) + f'(x_i)(x - x_i)$$

first order approximation

Second order approximation:

$$f(x) \cong f(x_i) + \frac{f'(x_i)}{1!}(x - x_i) + \frac{f''(x_i)}{2!}(x - x_i)^2$$

n

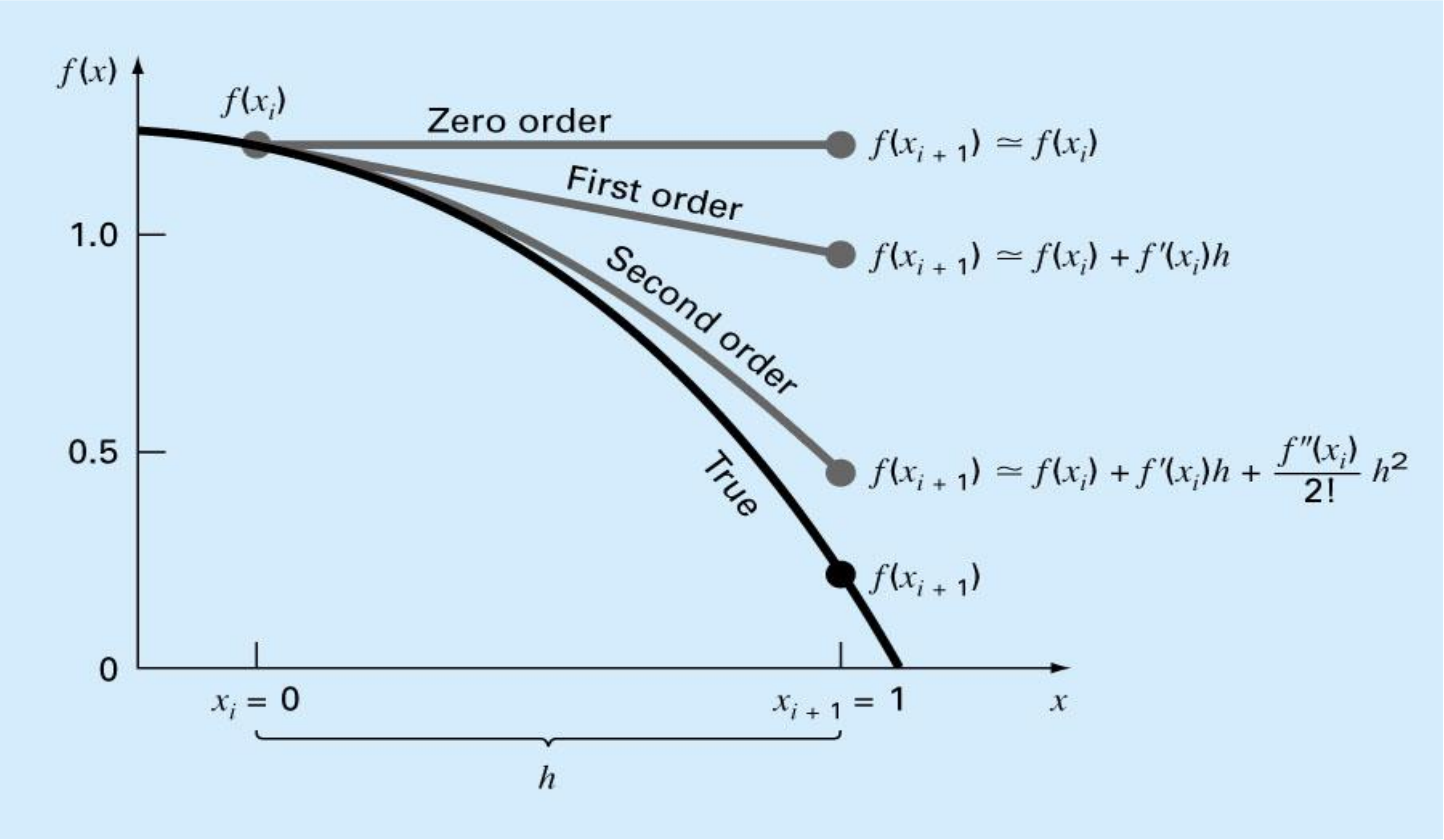
$\boldsymbol{n^{th}}$ ***order*** approximation:

$$f(x) \cong f(x_i) + \frac{f'(x_i)}{1!}(x - x_i) + \frac{f''(x_i)}{2!}(x - x_i)^2 + \ldots + \frac{f^{(n)}(x_i)}{n!}(x - x_i)^n + R_n$$

- Each additional term will contribute some improvement to the approximation. Only if an infinite number of terms are added will the series yield an exact result.
- In most cases, only a few terms will result in an approximation that is close enough to the true value for practical purposes

Example

Approximate the function $f(x) = 1.2 - 0.25x - 0.5x^2 - 0.15x^3 - 0.1x^4$ from $x_i = 0$ with $h = 1$ and **predict** $f(x)$ at $x_{i+1} = 1$.



Why Gradient Descent

Taylor Approximation : $f(x) \cong f(x_i) + f^{'}(x_i)(x - x_i)$

Newton-Raphson: $x_{i+1} = x_i - \dfrac{f(x_i)}{f'(x_i)}$

Gradient Descent :

$$w_i = w_i + \Delta w_i$$

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

$$w_i = w_i + \Delta w_i$$

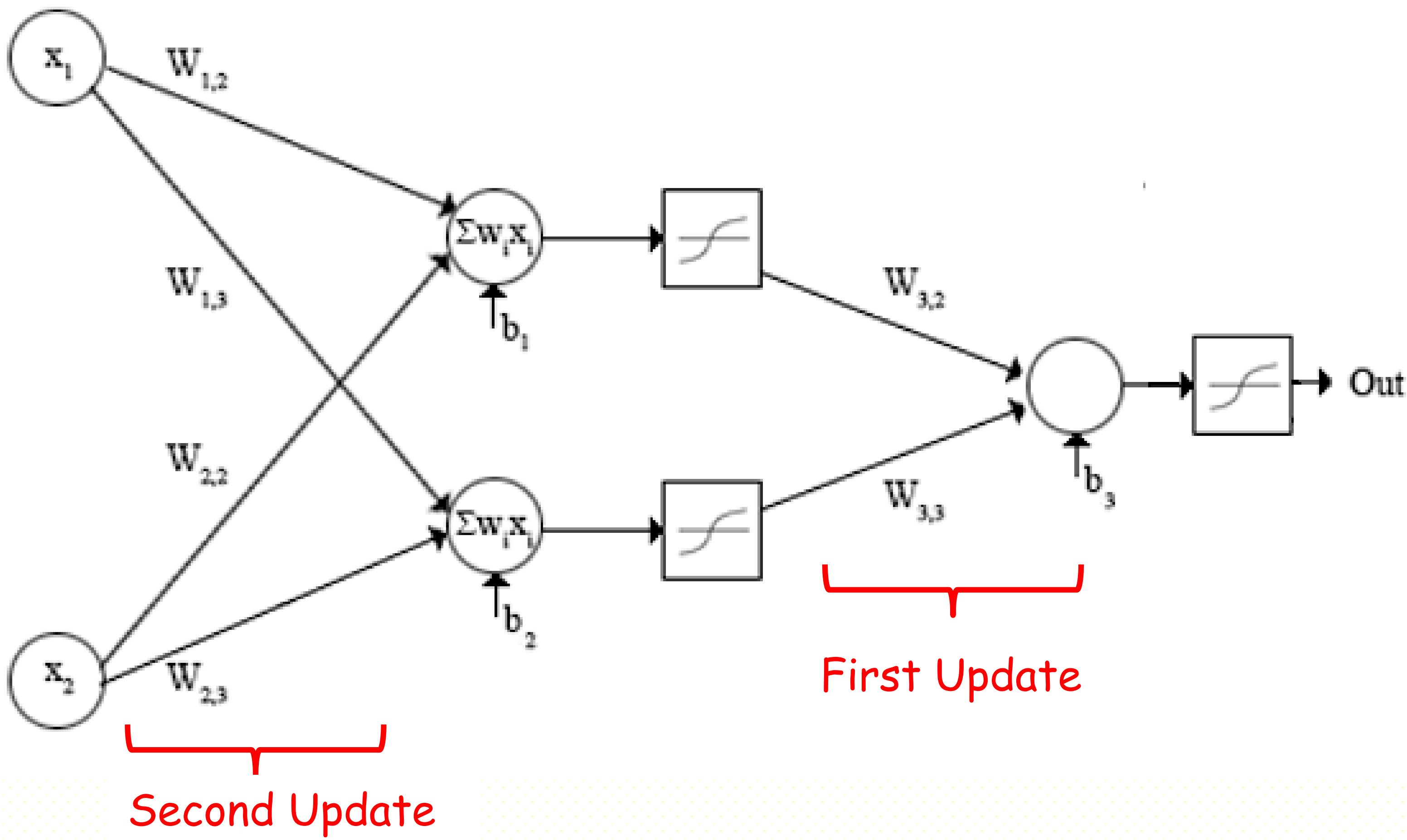
$$w_i = w_i + \eta(\mathrm{t} - \mathrm{o})\,\mathrm{x}_i$$

Core Robbins – Monro algorithm

for unconstrained root – finding

$\hat{\theta}_{k+1} = \hat{\theta}_k - a_k \boldsymbol{Y}_k(\hat{\theta}_k)$, where $a_k > 0$

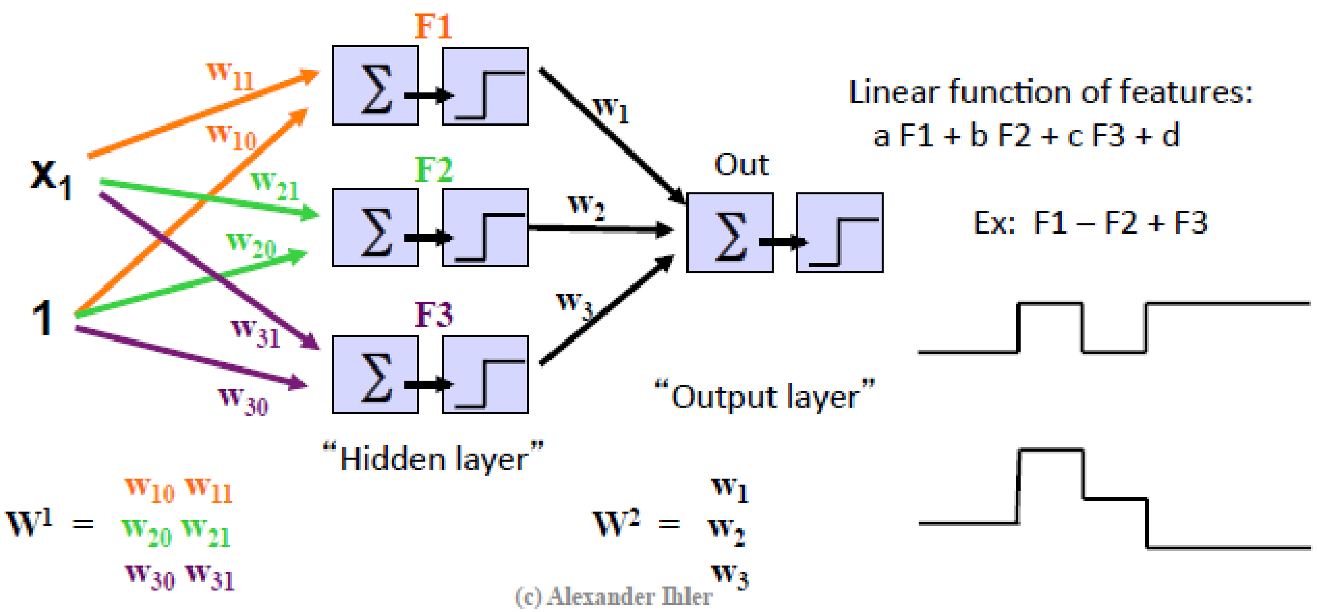
Two Layers Neural Network



Two Layers Neural Network

Multi-layer perceptron model

- Step functions are just perceptrons!
  - "Features" are outputs of a perceptron
  - Combination of features output of another



Recall Calculus

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

$$Find\ \ x\ that\,satisfy\ \ to\ \ \min value\ f(x)$$

$$y = 2x^2 - x$$

$$E = y' - y = y' - \left(2x^2 - x\right)$$

$$E = \frac{1}{2}\left(y' - \left(2x^2 - x\right)\right)^2$$

$$\frac{\partial E}{\partial x} = \frac{1}{2}2(-4x+1)\frac{\partial E}{\partial x}\left(y' - \left(2x^2 - x\right)\right)$$

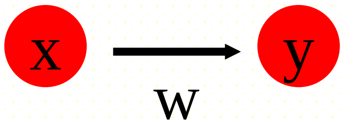
$$0 = -x\frac{\partial E}{\partial x}\left(y' - \left(2x^2 - x\right)\right)$$

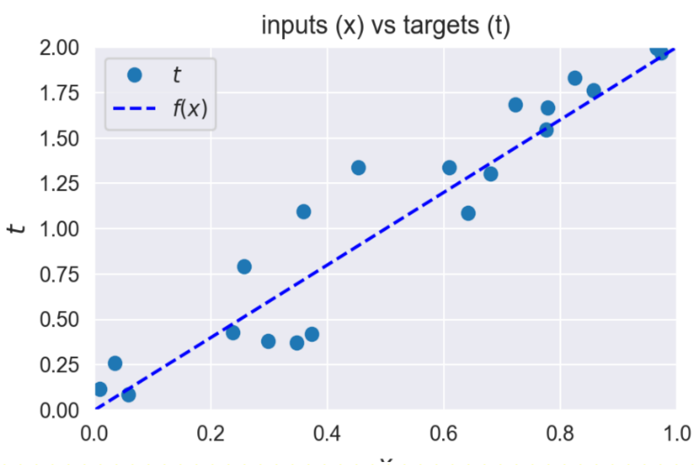
$$0 = -x(-4x+1)$$

$$It\ \ means\ \ x = 0\ or\ \ x = \frac{1}{4}$$

$$satisfies\ \ the\ \ restriction\ \ of\ \ \min of\ f(x)$$

Define Loss Function





We will optimize the model $y = x * w$ *by tuning parameter of* w

To do this, we will use Mean Squared Error (MSE) as a loss function

$$E = \sum_{i=1}^{n} \left\| t_i - y_i \right\|^2$$

For minimizing the loss function, we will use some math

Gradient Descent (derivative)

Gradient Descent (derivative)

The gradient descent algorithm works by taking the gradient (derivative) of the loss function, E,

with respect to parameters, at a specific position on this loss function,

and updates the parameters in the direction of the negative gradient (down along the loss function).

The parameter w *is iteratively updated by taking steps proportional to the negative of the gradient:*

$$\mathrm{w}(\mathrm{k}+1) = \mathrm{w}(\mathrm{k}) - \Delta\,\mathrm{w}(\mathrm{k})$$

Loss Function is MSE as

$$E=\sum_{i=1}^{n}\left\|t_i-y_i\right\|^2$$

Gradient Descent (derivative) :

$$w(k+1)=w(k)-\Delta w(k)$$

$$\Delta w=\eta\frac{\partial E}{\partial w},\ \textit{where}\ \eta\ \textit{is learning rate}$$

For each sample i this gradient can be splitted according to the chain rule into :

$$\frac{\partial E_i}{\partial w_i}=\frac{\partial E_i}{\partial y_i}\frac{\partial y_i}{\partial w_i}$$

$$\frac{\partial E_i}{\partial y_i}=\frac{\partial\left(t_i-y_i\right)^2}{\partial y_i}=-2\left(t_i-y_i\right)=2\left(y_i-t_i\right)$$

Since $y_i=x_i*w_i$, then we can write $\partial y_i/\partial w_i$ as :

$$\frac{\partial y_i}{\partial w_i}=\frac{\partial\left(x_i*w_i\right)}{\partial y_i}=x_i$$

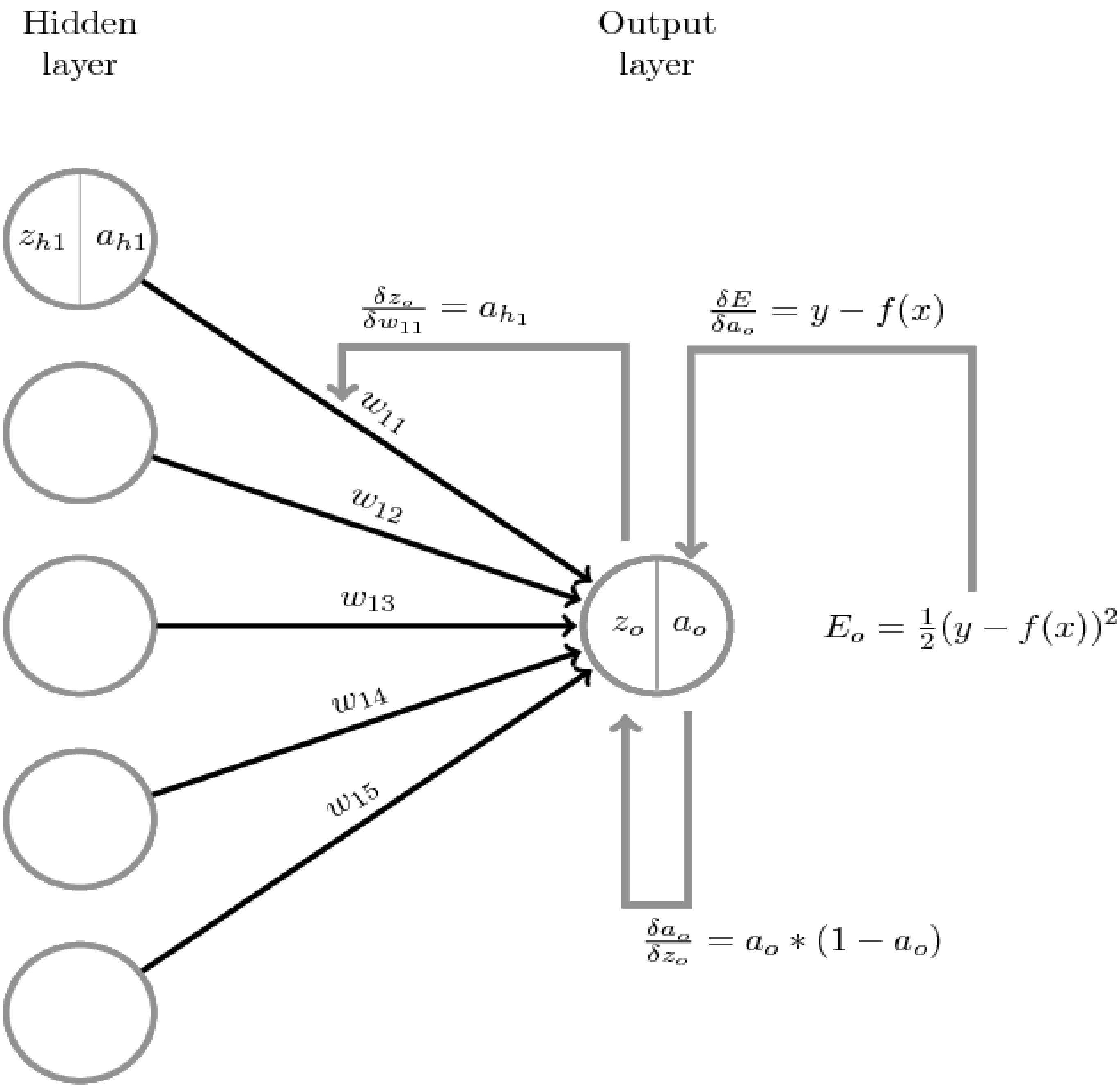
So the full update will be :

$$\Delta w_i=\eta\frac{\partial E}{\partial w}=\eta\frac{\partial E_i}{\partial y_i}\frac{\partial y_i}{\partial w_i}=\eta*2\left(y_i-t_i\right)*x_i\quad \textit{recalll perceptron}$$

In batch process case :

$$\Delta w_i=\eta*2*\frac{1}{N}\sum_{i=1}^{N}\left(x_i*\left(y_i-t_i\right)\right)$$

Update for Output Unit



$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial a_0} * \frac{\partial a_0}{\partial z_0} * \frac{\partial z_0}{\partial w}$$

$$E = \frac{1}{2}(t-y)^2$$

$$\frac{\partial E}{\partial a_0} = 2 * \frac{1}{2}(t-y)^{2-1}$$

$$\frac{\partial E}{\partial a_0} = (t-y)$$

Since the activation function is sigmoid

$$\frac{\partial a_0}{\partial z_0} = a_0 * (1 - a_0)$$

$$z_0 = \sum_{i=0}^{n-1} w_i * a_{hi}$$

$$z_0 = w_0 * a_{h0} + w_1 * a_{h1} + w_2 * a_{h2} + w_3 * a_{h3} + w_4 * a_{h4}$$

$$\frac{\partial z_0}{\partial w_0} = a_{h0}, \quad \frac{\partial z_0}{\partial w_1} = a_{h1}, \quad \frac{\partial z_0}{\partial w_2} = a_{h2}, \quad \frac{\partial z_0}{\partial w_3} = a_{h3}, \quad \frac{\partial z_0}{\partial w_3} = a_{h4}$$

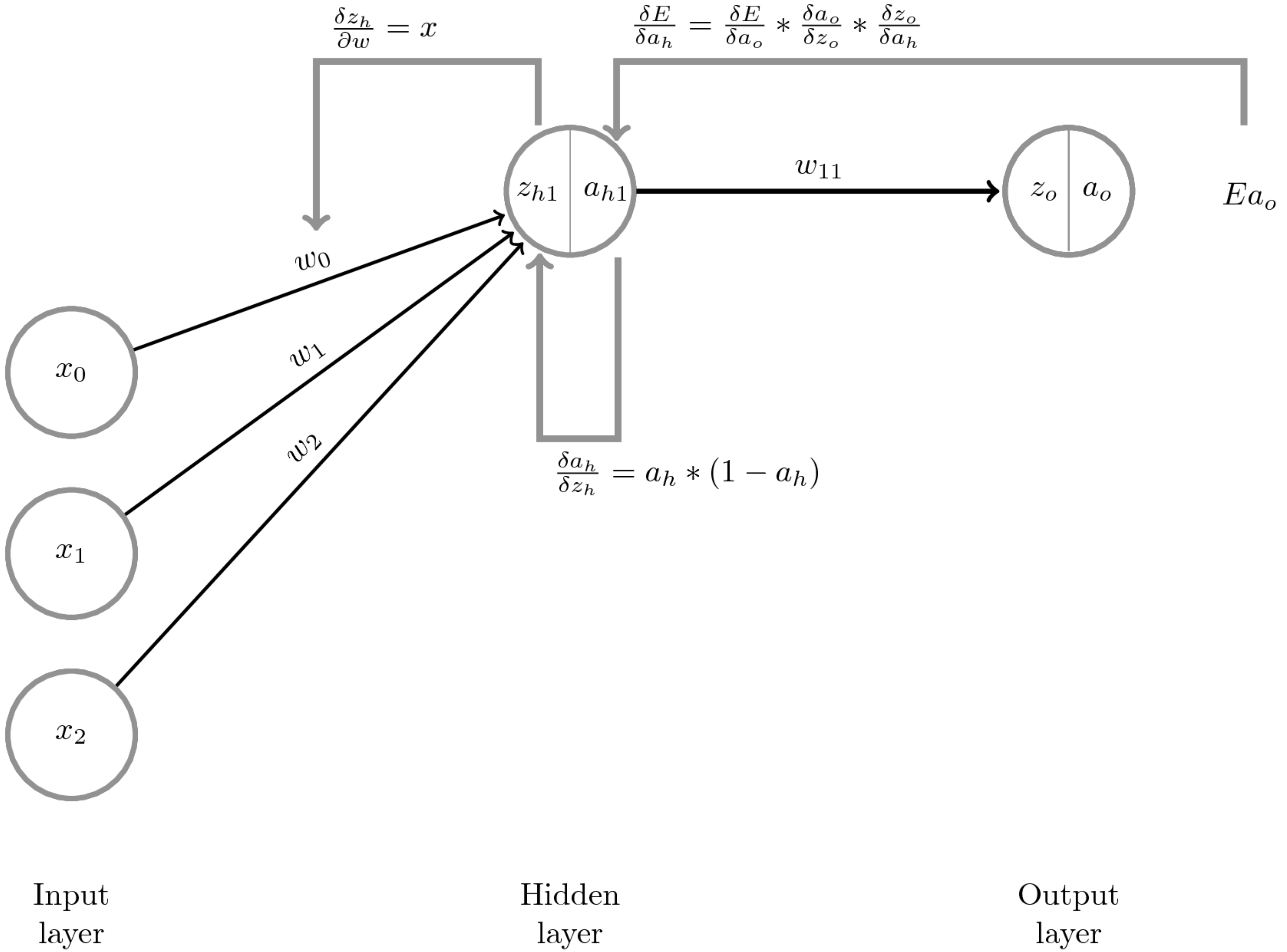
$$\frac{\partial z_0}{\partial w} = \left[a_{h0}, a_{h1}, a_{h2}, a_{h3}, a_{h4}\right]^T$$

So the full update will be:

$$\Delta w_i = \eta \frac{\partial E}{\partial a_0} * \frac{\partial a_0}{\partial z_0} * \frac{\partial z_0}{\partial w} = (t-y)\left(a_0 * (1-a_0)\right)\left[a_{h0}, a_{h1}, a_{h2}, a_{h3}, a_{h4}\right]^T$$

$$\delta_o = (t-y)\left(a_0 * (1-a_0)\right), \; input = \left[a_{h0}, a_{h1}, a_{h2}, a_{h3}, a_{h4}\right]^T$$

Update for Hidden Unit



For hidden unit

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial a_h} * \frac{\partial a_h}{\partial z_h} * \frac{\partial z_h}{\partial w}$$

First step

$$\frac{\partial E}{\partial a_h} = \frac{\partial E}{\partial z_0} * \frac{\partial z_0}{\partial a_h}$$

$$\frac{\partial E}{\partial z_0} = \frac{\partial E}{\partial a_0} * \frac{\partial a_0}{\partial z_0}$$

$$\frac{\partial E}{\partial a_0} = (t - y) \text{ previous error}$$

$$\frac{\partial a_0}{\partial z_0} = a_0 * (1 - a_0)$$

$$\frac{\partial z_0}{\partial a_h} = w \text{ since } z_0 = w_{11} * a_{h1}$$

Note that w is from output layer

$$\frac{\partial a_h}{\partial z_h} = a_h * (1 - a_h)$$

$$\frac{\partial z_h}{\partial w} = x$$

So the full update will be:

$$\Delta w_i = \frac{\partial E}{\partial a_h} * \left(\frac{\partial a_h}{\partial z_h} * \frac{\partial z_h}{\partial w} \right)$$

$$\frac{\partial E}{\partial a_h} = \frac{\partial E}{\partial a_0} * \frac{\partial a_0}{\partial z_0} * \frac{\partial z_0}{\partial a_h} = \underbrace{(t - y) * (a_0 * (1 - a_0))}_{\delta_o} * \underset{w_o}{w}$$

Note that δ_o *is delta from output layer*

$$\delta_h = \sum_{i=1}^{n} w_o * \delta_o$$

$$\Delta w_i = \underbrace{(t - y) * (a_0 * (1 - a_0))}_{\delta_o} * \underset{w_o}{w} * (a_h * (1 - a_h)) * (x)$$

BackPropagation

For each input vector:

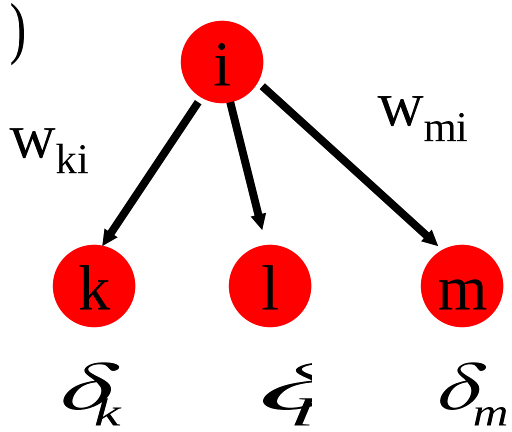
1. Propagate the inputs forward through the network.
2. Propagate the errors backward through the network.

<u>Error term for output units:</u>

$$\delta_i = -\frac{\partial E_d}{\partial sum_{id}} = -\frac{\partial E_{id}}{\partial sum_{id}} = (t_{id} - o_{id}) o_{id} (1 - o_{id})$$

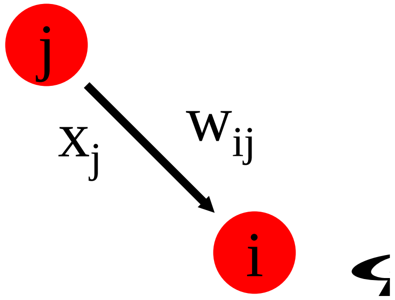
<u>Error term for hidden units:</u>

$$\delta_i = -\frac{\partial E_d}{\partial sum_{id}} = o_{id} (1 - o_{id}) \sum_{k \in Outputs} w_{ki} \delta_k$$



3. Compute all weights changes

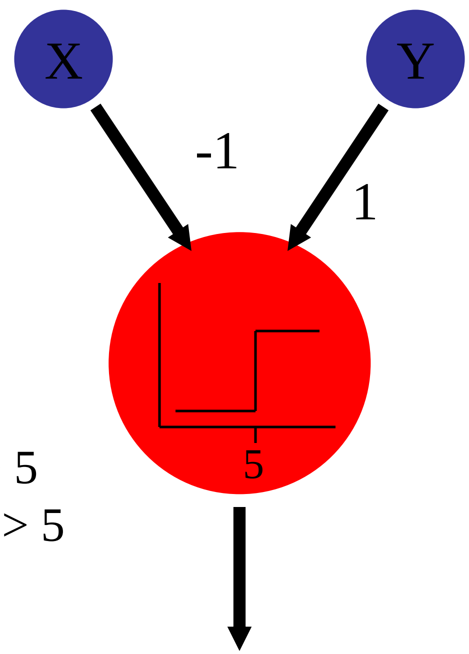
$$\Delta w_{ij} = \eta \delta_i x_j$$



Perceptrons

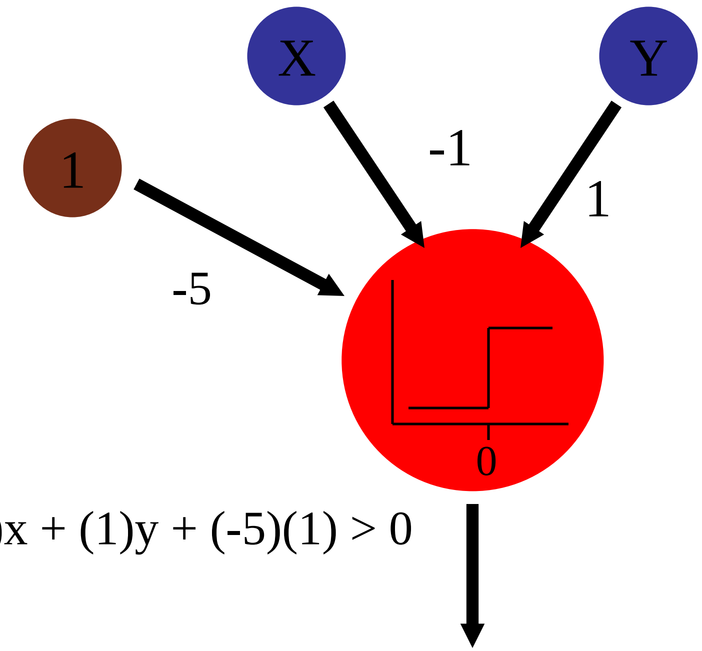
Perceptron: A machine that classifies input vectors by applying linear functions to them (Rosenblatt, 1958).

Perceptron Learning Algorithm: A stochastic gradient-descent method for finding a linear function that properly classifies a set of input vectors (Minsky & Papert, 1969).



$xw_x + yw_y > 5$

$(-1)x + (1)y > 5$

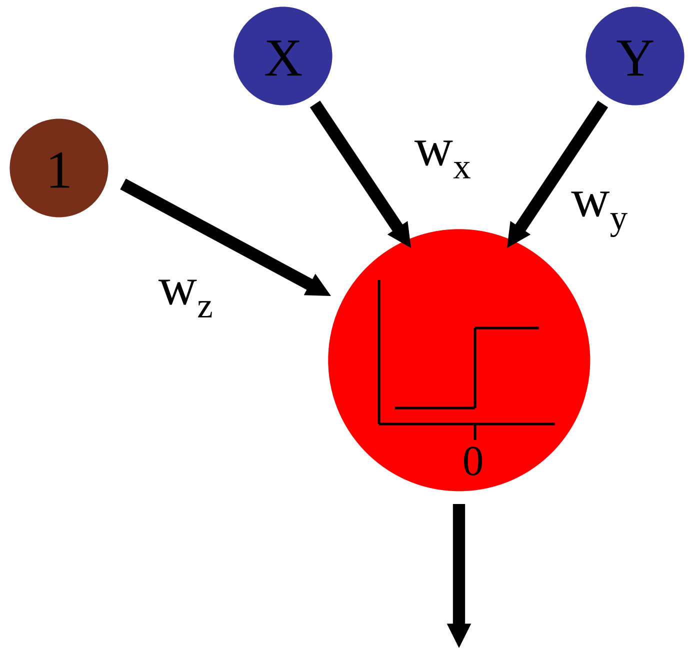


$(-1)x + (1)y + (-5)(1) > 0$

Classification & Learning

case	x	y	sum*	class
1	1	3	-3	-1
2	-5	2	2	+1
3	1	3	-3	-1
4	1	9	3	+1
5	-2	4	1	+1
6	-7	2	4	+1
7	5	5	-5	-1

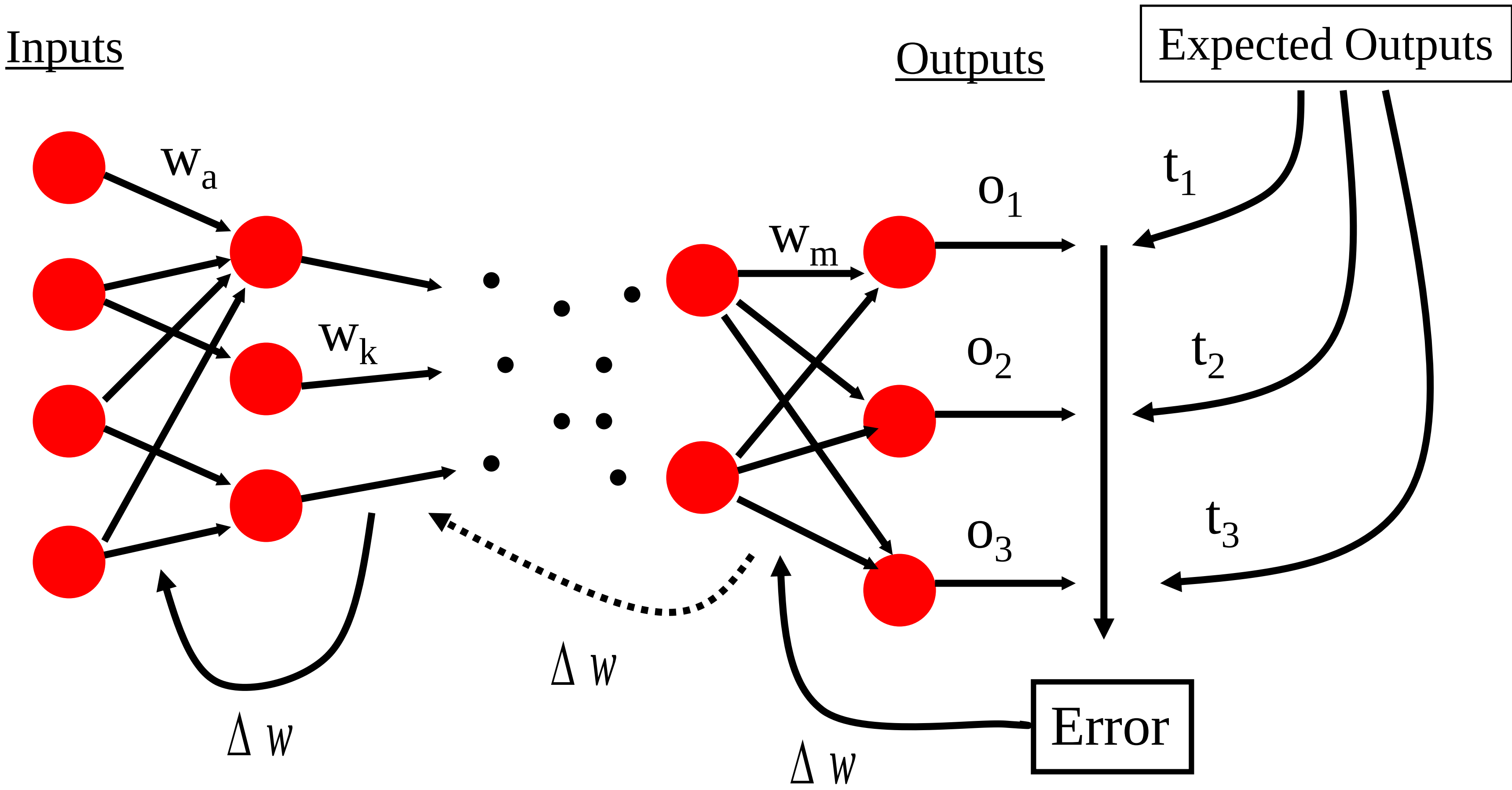
*Sum assumes values -1, 1, -5 for the weights.



Classification: The perceptron should compute the proper class for each input x-y pair. For a single perceptron, this is only possible when the input vectors are linearly separable

Learning: Find the proper values for weights w_x, w_y and w_z so that the perceptron properly classifies all input cases. This is a search problem in weight-vector space.

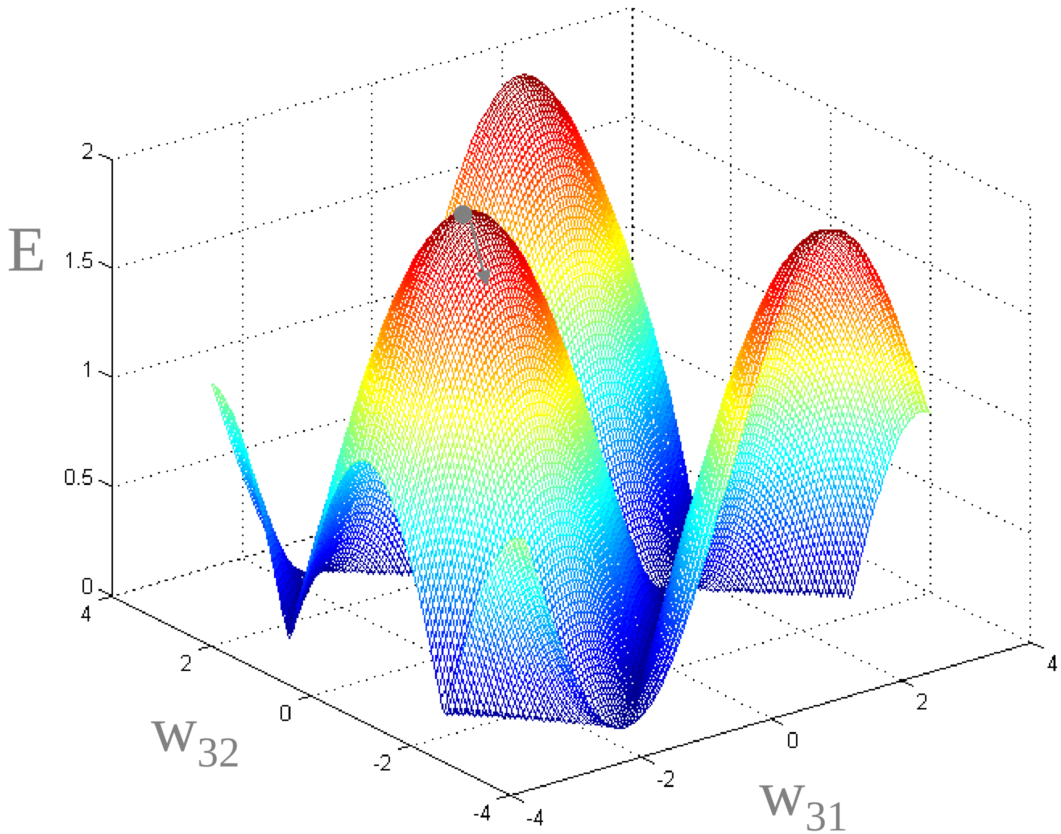
Gradient Descent Weight Learning



Base weight changes upon their contribution to the error such that the updated weights will create LESS error on the same training cases.

$$\text{Contribution} = \frac{\partial Error}{\partial w_{ij}}$$

Gradient Descent



$$\nabla E(\vec{w}) = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n} \right] \quad = \quad \text{Gradient of E w.r.t. the weight vector}$$

BackPropagation

For each input vector:

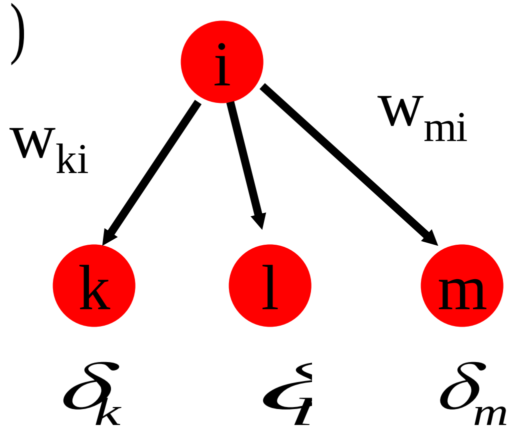
1. Propagate the inputs forward through the network.
2. Propagate the errors backward through the network.

<u>Error term for output units:</u>

$$\delta_i = -\frac{\partial E_d}{\partial sum_{id}} = -\frac{\partial E_{id}}{\partial sum_{id}} = (t_{id} - o_{id})\, o_{id}(1 - o_{id})$$

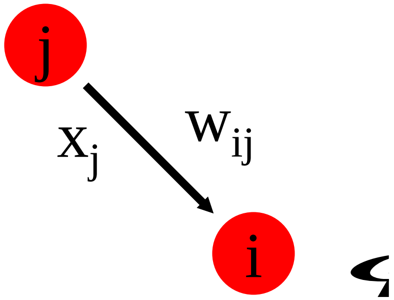
<u>Error term for hidden units:</u>

$$\delta_i = -\frac{\partial E_d}{\partial sum_{id}} = o_{id}(1 - o_{id}) \sum_{k \in Outputs} w_{ki}\delta_k$$

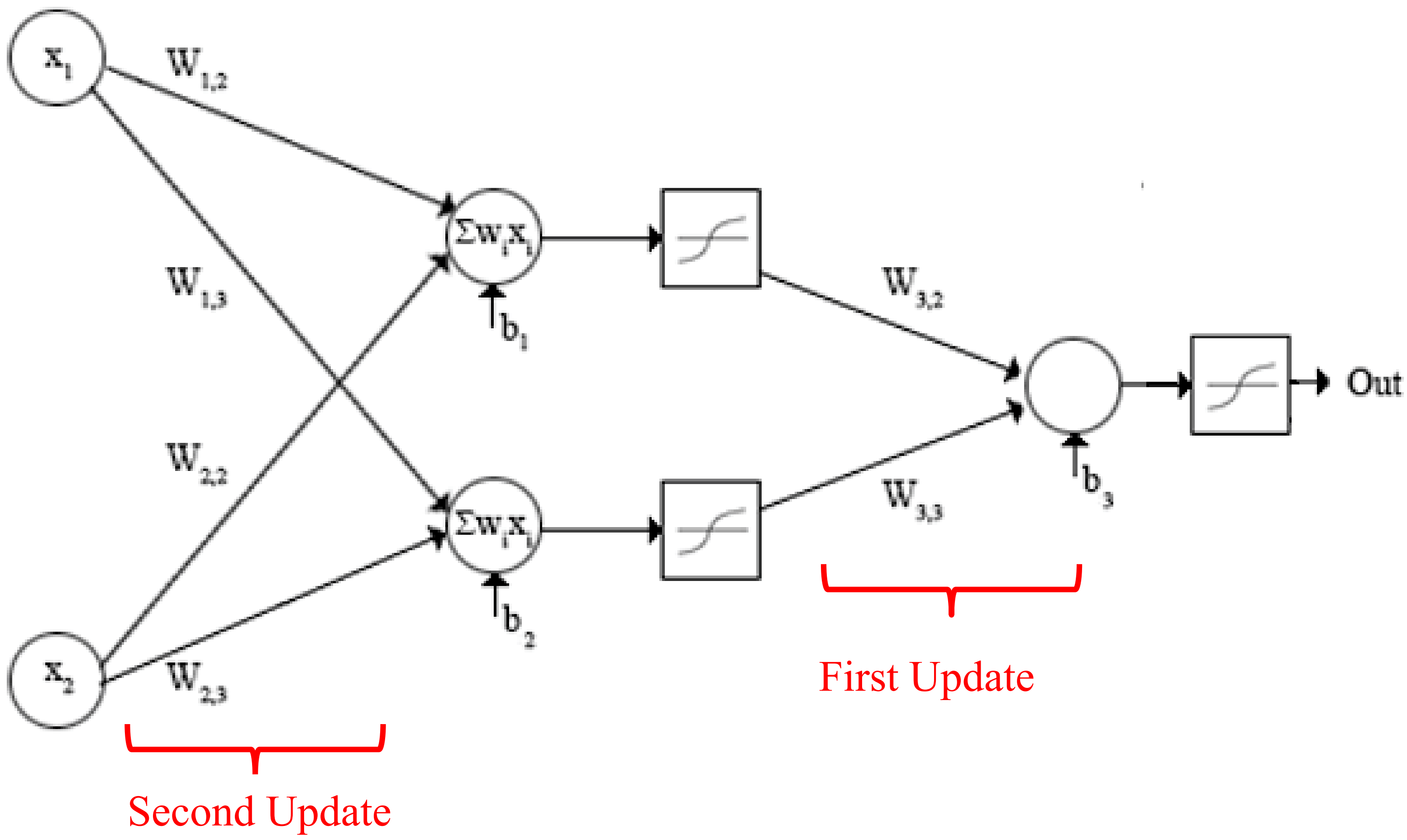


3. Compute all weights changes

$$\Delta w_{ij} = \eta\, \delta_i x_j$$



Two Layers Neural Network



MSE Loss Computing dE_i/dw_{ij}

$$\frac{\partial E_i}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \frac{1}{2} \sum_{d \in D} (t_{id} - o_{id})^2$$

$$= \frac{1}{2} \sum_{d \in D} 2(t_{id} - o_{id}) \frac{\partial}{\partial w_{ij}} (t_{id} - o_{id})$$

$$= \sum_{d \in D} (t_{id} - o_{id}) \frac{\partial}{\partial w_{ij}} (-o_{id})$$

$$= \sum_{d \in D} (t_{id} - o_{id}) \frac{\partial}{\partial w_{ij}} (-f_T(sum_{id})) \qquad \text{where} \quad sum_{id} = \sum_j w_{ij} x_{jd}$$

In general:

$$\frac{\partial}{\partial w_{ij}} f_T(sum_{id}) = \frac{\partial f_T}{\partial sum_{id}} \frac{\partial sum_{id}}{\partial w_{ij}} = \frac{\partial f_T}{\partial sum_{id}} x_{jd}$$

Computing $\frac{\partial}{\partial w_{ij}} f_T(sum_{id})$

Identity f_t : $f_T(sum_{id}) = sum_{id}$

$$\frac{\partial}{\partial sum_{id}} f_T = \frac{\partial}{\partial sum_{id}} sum_{id} = 1$$

$$\frac{\partial}{\partial w_{ij}} f_T(sum_{id}) = \frac{\partial f_T}{\partial sum_{id}} \frac{\partial sum_{id}}{\partial w_{ij}} = (1) x_{jd} = x_{jd}$$

If f_T is not continuous, and hence not differentiable everywhere, then we cannot use the Delta Rule.

Sigmoidal f_t : $f_T(sum_{id}) = \frac{1}{1+e^{-sum_{id}}}$

$$\frac{\partial}{\partial sum_{id}} \frac{1}{1+e^{-sum_{id}}} = \frac{e^{-sum_{id}}}{(1+e^{-sum_{id}})^2} = f_t(sum_{id})(1 - f_t(sum_{id}))$$

But since: $f_T(sum_{id}) = o_{id}$ $\quad \frac{\partial}{\partial w_{ij}} f_T(sum_{id}) = o_{id}(1-o_{id}) x_{jd}$

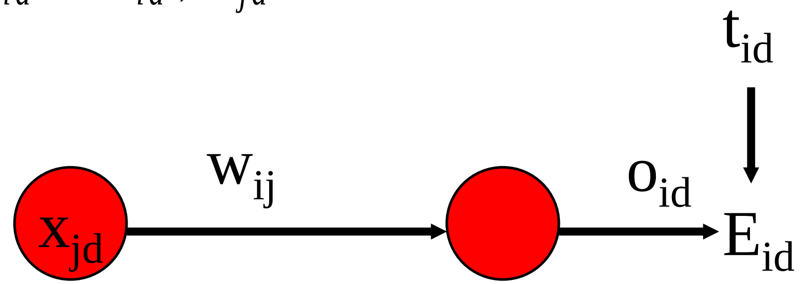
Weight Updates for Simple Units

f_T = identity function

$$\frac{\partial E_i}{\partial w_{ij}} = \sum_{d \in D} (t_{id} - o_{id}) \frac{\partial}{\partial w_{ij}} (-f_T(sum_{id}))$$

$$= \sum_{d \in D} (t_{id} - o_{id})(-x_{jd})$$

$$\Delta w_{ij} = -\eta \frac{\partial E_i}{\partial w_{ij}} = \eta \sum_{d \in D} (t_{id} - o_{id}) x_{jd}$$



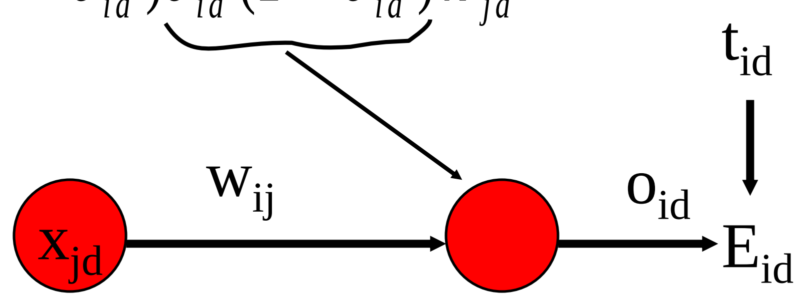
Weight Updates for Sigmoidal Units

f_T = sigmoidal function

$$\frac{\partial E_i}{\partial w_{ij}} = \sum_{d \in D} (t_{id} - o_{id}) \frac{\partial}{\partial w_{ij}} (-f_T(sum_{id}))$$

$$= \sum_{d \in D} (t_{id} - o_{id}) o_{id} (1 - o_{id})(-x_{jd})$$

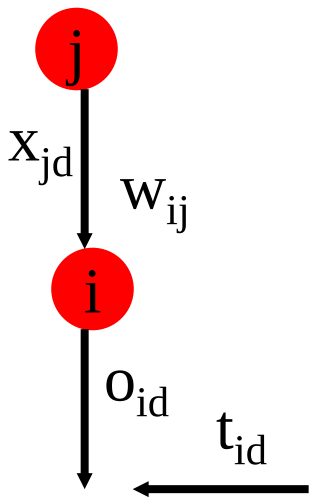
$$\Delta w_{ij} = -\eta \frac{\partial E_i}{\partial w_{ij}} = \eta \sum_{d \in D} (t_{id} - o_{id}) \underbrace{o_{id}(1 - o_{id})} x_{jd}$$



Incremental Gradient Descent

After each training instance, d, update the weights by:

$$\Delta w_{ij} = -\eta \frac{\partial E_{id}}{\partial w_{ij}} = \eta(t_{id} - o_{id}) \frac{\partial}{\partial w_{ij}} (f_T(sum_{id}))$$



Simple Unit:

$$\Delta w_{ij} = \eta (t_{id} - o_{id}) x_{jd}$$

*Same as perceptron rule

Sigmoidal Unit:

Error Term

$$\Delta w_{ij} = \eta (t_{id} - o_{id}) o_{id} (1 - o_{id}) x_{jd}$$

Backpropagation Learning in Multi-Layer ANNs

- Still use gradient-descent (delta) rule: $\Delta w_{ij} = -\eta \frac{\partial E_i}{\partial w_{ij}}$
  - But now the effects of an arc's weight change on the error need to be computed across all nodes along all arcs from the current arc to the output layer.

- Starting from the output layer and moving back through the hidden nodes, compute an error term δ_i for each node i.

- Then, for each arc going into node i, compute the contribution of that arc's weight w_{ij} to the total error as $x_{jd}\,\delta_i$, where x_{jd} is the output value of node j on training example d.

- When an error term has been calculated for every node to which node j sends outputs, then node j's error contribution can be computed as the product of :
  - a) the influence of j's input sum (net_j) upon j's output.
  - b) the sum of the contributions of j's output to each of its downstream neighbors' error terms.
    - Each such contribution is simply the weight along the arc times the error contribution of the node on the downstream end of that arc.

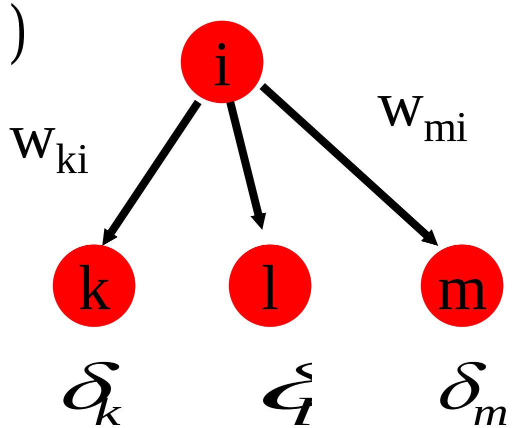
BackPropagation

For each input vector:

1. Propagate the inputs forward through the network.
2. Propagate the errors backward through the network.

Error term for output units:

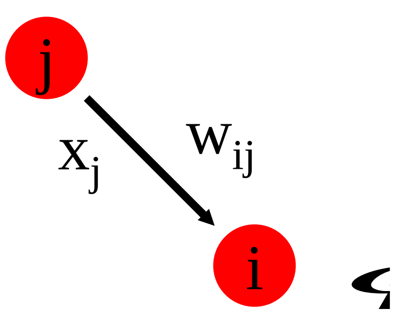
$$\delta_i = -\frac{\partial E_d}{\partial sum_{id}} = -\frac{\partial E_{id}}{\partial sum_{id}} = (t_{id} - o_{id})\, o_{id}(1 - o_{id})$$



Error term for hidden units:

$$\delta_i = -\frac{\partial E_d}{\partial sum_{id}} = o_{id}(1 - o_{id}) \sum_{k \in Outputs} w_{ki}\delta_k$$

3. Compute all weights changes



$$\Delta w_{ij} = \eta\, \delta_i\, x_j$$

Updating hidden-to-output

- We have teacher supplied desired values
- $\text{delta}_{wji} = \alpha * a_j * (T_i - Oi) * g'(in_i)$

  $= \alpha * a_j * (T_i - O_i) * O_i * (1 - O_i)$

  – for sigmoid the derivative is, $g'(x) = g(x) * (1 - g(x))$

derivative

alpha

Here we have general formula with derivative, next we use for sigmoid

miss

Updating interior weights

- Layer k units provide values to all layer k+1 units
  - "miss" is ***sum of misses*** from all units on k+1
  - $miss_j = \Sigma [a_i(1- a_i) (T_i - a_i) w_{ji}]$
  - weights coming into this unit are adjusted based on their contribution

For layer k+1

$delta_{kj} = \alpha * I_k * a_j * (1 - a_j) * miss_j$

Compute deltas

BackPropagation

```
epsilon = - (groundTruth - p);
for i = num_layers:-1:1

    gradStack{i} = struct; %initializing an empty cell

    gradStack{i}.W = epsilon * output{i}'/m;

    gradStack{i}.b = sum(epsilon,2)/m;

   epsilon = (stack{i}.W'*epsilon).*output{i}.*(1-output{i});

End
```

$$\Delta w_{ij} = \eta \delta_i x_j$$

$$\delta_i = -\frac{\partial E_d}{\partial sum_{id}} = -\frac{\partial E_{id}}{\partial sum_{id}} = (t_{id} - o_{id}) o_{id} (1 - o_{id})$$

```
Stack: keeps parameters of networks

Outputs: keeps output returned from each layer

Outputs{1}=1
```

$$\delta_i = -\frac{\partial E_d}{\partial sum_{id}} = o_{id}(1 - o_{id}) \sum_{k \in Outputs} w_{ki} \delta_k$$

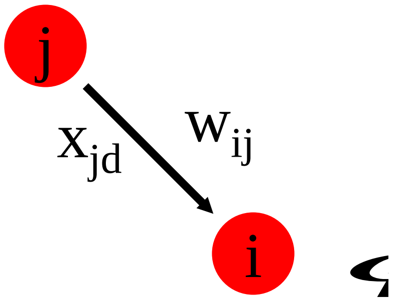
https://github.com/ericliu03/multilayer-neural-network

Explaining Error Terms

$$\delta_i = -\frac{\partial E_d}{\partial sum_{id}}$$

- What is node i's effect on the total error, E_d? It affects via its sum of inputs on case d, sum_{id}.
- Negative sign is merely for convenience when updating w_{ij}.

$$\Delta w_{ij} = -\eta \frac{\partial E_d}{\partial w_{ij}} = \eta \delta_i x_{jd}$$



$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial sum_{id}}{\partial w_{ij}} \frac{\partial E_d}{\partial sum_{id}}$$

- For any input weight, w_{ij}, to node i, its influence on E_d is simply its effect on sum_{id} times sum_{id}'s effect on E_d.

$$\frac{\partial sum_{id}}{dw_{ij}} = \frac{\partial}{dw_{ij}} \sum_k w_{ik} x_{kd} = x_{jd}$$

- A weight's influence on the sum is simply x_{jd}
- So once we compute a node's influence upon E_d, we can use that value to compute the influences of each weight, and thus to update each weight via the negative of its influence.

Output Node Error Term

- If node i is an output node, then the contribution of sum_{id} to E_d is its contribution to the error on the output of node i, E_{id}.

$$\delta_i = -\frac{\partial E_d}{\partial sum_{id}} = -\frac{\partial E_{id}}{\partial sum_{id}}$$

- The standard error function is a quadratic

$$= -\frac{\partial}{\partial sum_{id}}\left(\frac{1}{2}(t_{id} - o_{id})^2\right) = -(t_{id} - o_{id})\frac{\partial}{\partial sum_{id}}(-o_{id})$$

- The influence of the sum on the output is simply the derivative of the transfer function with respect to the sum. For a sigmoid unit, we've already shown that value to be $o_{id}(1-o_{id})$

$$= (t_{id} - o_{id})\frac{\partial f_T}{\partial sum_{id}} = (t_{id} - o_{id})o_{id}(1 - o_{id})$$

Hidden Node Error Term

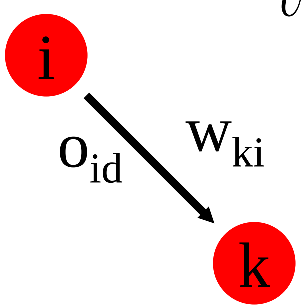
- If node i is a hidden node, then the contribution of sum_{id} to E_d is via its contributions to the error terms of each node that i outputs to.

$$\delta_i = -\frac{\partial E_d}{\partial sum_{id}} = -\frac{\partial o_{id}}{\partial sum_{id}} \sum_{k \in Outputs} \frac{\partial sum_{kd}}{\partial o_{id}} \frac{\partial E_d}{\partial sum_{kd}}$$

- The influence of output o_{id} on sum_k is simply the weight w_{ki}. And the other 2 derivatives were computed earlier.

$$\frac{\partial sum_{kd}}{\partial o_{id}} = \frac{\partial}{\partial o_{id}} \sum_{j \in Inputs} w_{kj} o_{jd} = w_{ki}$$

$$\frac{\partial E_d}{\partial sum_{kd}} = -\delta_k$$



$$\frac{\partial o_{id}}{\partial sum_{id}} = o_{id}(1 - o_{id})$$ For a sigmoid unit

- Putting it all together: $$\delta_i = o_{id}(1 - o_{id}) \sum_{k \in Outputs} w_{ki} \delta_k$$

BackPropagation

For each input vector:

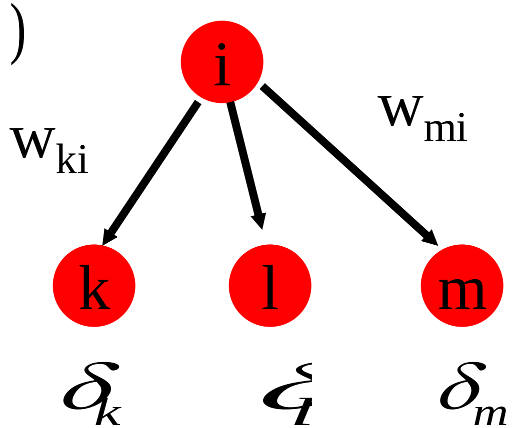
1. Propagate the inputs forward through the network.
2. Propagate the errors backward through the network.

<u>Error term for output units:</u>

$$\delta_i = -\frac{\partial E_d}{\partial sum_{id}} = -\frac{\partial E_{id}}{\partial sum_{id}} = (t_{id} - o_{id})\, o_{id}(1 - o_{id})$$

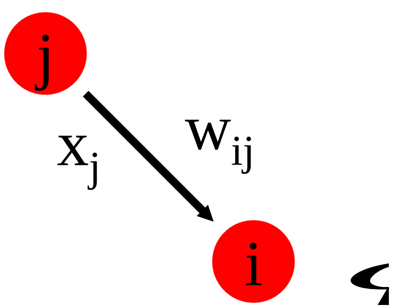
<u>Error term for hidden units:</u>

$$\delta_i = -\frac{\partial E_d}{\partial sum_{id}} = o_{id}(1 - o_{id}) \sum_{k \in Outputs} w_{ki}\delta_k$$



3. Compute all weights changes

$$\Delta w_{ij} = \eta\, \delta_i\, x_j$$



Backpropagation Algorithm

- Learning rule

$$\mathbf{w}(m+1) = \mathbf{w}(m) + \Delta\mathbf{w}(m),$$

- Hidden-to-output

$$\Delta w_{kj} = \eta \delta_k y_j = \eta (t_k - z_k) f'(net_k) y_j.$$

- Input-to-hidden

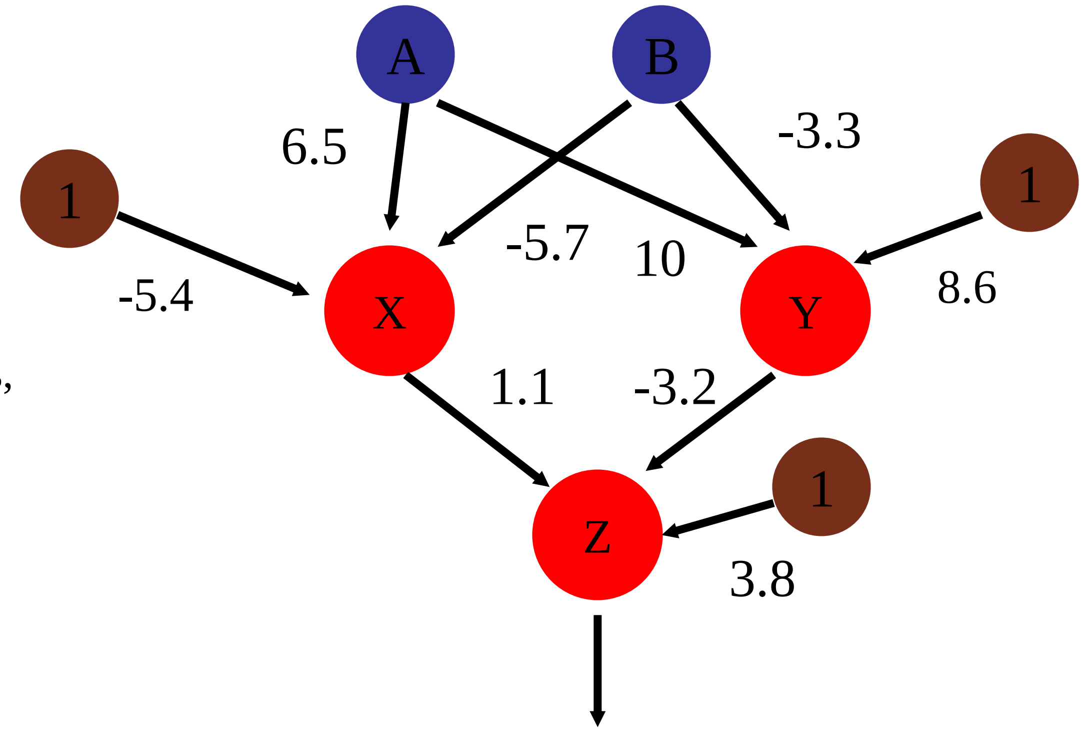
$$\Delta w_{ji} = \eta x_i \delta_j = \eta \underbrace{\left[\sum_{k=1}^{c} w_{kj} \delta_k \right] f'(net_j)}_{\delta_j} x_i.$$

- Note, that w_{ij} are initialized with random values

Learned XOR: Version I

Slightly sketchy: For this example, backpropagation was used with the perceptron training rule instead of the delta rule. This is necessary because f_T of the perceptron is not differentiable, and thus not amenable to the delta rule.

Cleaner Approach: If perceptron nets, use the GA to find a good weight set.



A	B	X
1	1	-1
1	-1	1
-1	1	-1
-1	-1	-1

$\Rightarrow A \wedge \neg B$

A	B	Y
1	1	1
1	-1	1
-1	1	-1
-1	-1	1

$\Rightarrow \neg(\neg A \wedge B)$

X	Y	Z
1	1	1
1	-1	1
-1	1	-1
-1	-1	1

$$\Rightarrow \neg(\neg X \wedge Y) \equiv X \vee \neg Y$$

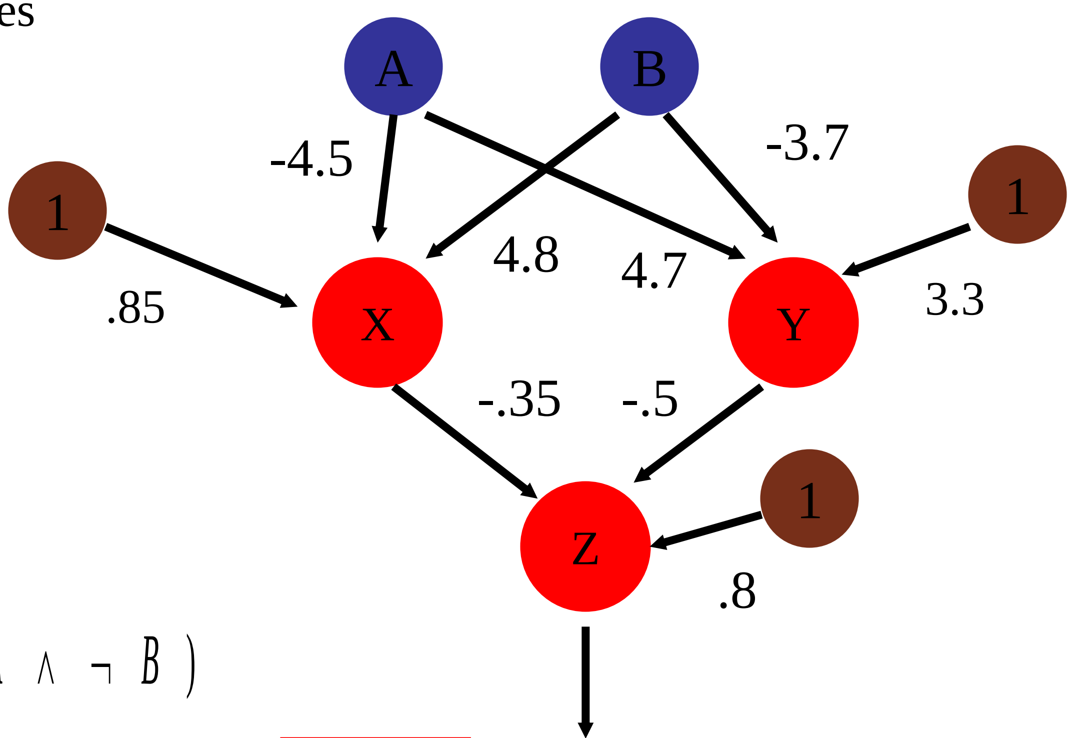
$$\equiv (A \wedge \neg B) \vee (\neg A \wedge B)$$

$$\equiv XOR(A, B)$$

see week5_ANN_learned_XOR_v1.m

Learned XOR: Version II

*The concepts that nodes X and Z represent are different in the two versions.



A	B	X
1	1	1
1	-1	-1
-1	1	1
-1	-1	1

$\rightarrow \neg(A \wedge \neg B)$

A	B	Y
1	1	1
1	-1	1
-1	1	-1
-1	-1	1

$\rightarrow \neg(\neg A \wedge B)$

X	Y	Z
1	1	-1
1	-1	1
-1	1	1
-1	-1	1

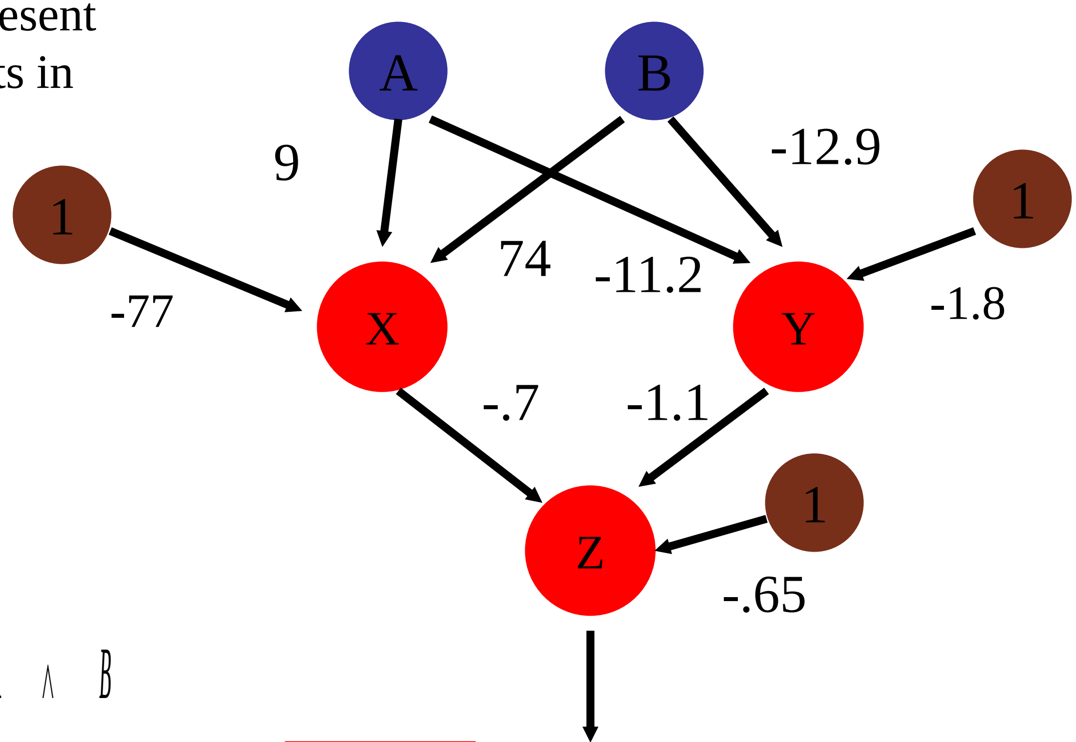
$$\rightarrow \neg(X \wedge Y) \equiv \neg X \vee \neg Y$$

$$\equiv (A \wedge \neg B) \vee (\neg A \wedge B)$$

$$\equiv XOR(A, B)$$

Learned XOR: Version III

*Nodes X, Y and Z represent different logical concepts in this version than in the previous 2 versions



A	B	X
1	1	1
1	-1	-1
-1	1	-1
-1	-1	-1

$\rightarrow A \wedge B$

A	B	Y
1	1	-1
1	-1	-1
-1	1	-1
-1	-1	1

$\rightarrow \neg A \wedge \neg B$

$\equiv \neg(A \vee B)$

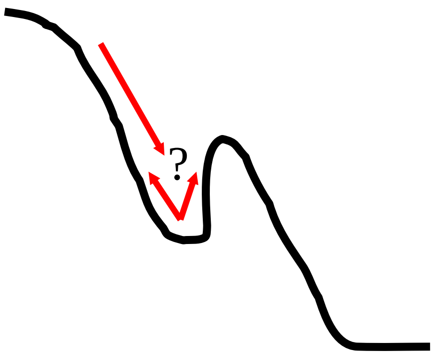
X	Y	Z
1	1	-1
1	-1	-1
-1	1	-1
-1	-1	1

$\rightarrow \neg X \wedge \neg Y$

$\equiv \neg(A \wedge B) \wedge (A \vee B)$

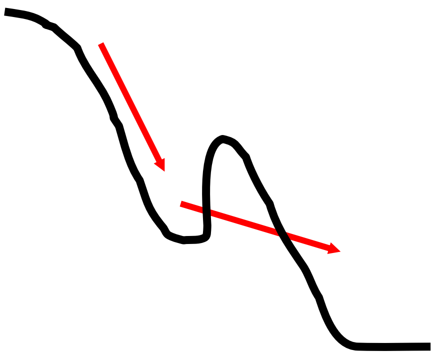
$\equiv XOR(A, B)$

Momentum



Gradient Descent can easily get stuck at a local minimum of the error landscape.

Momentum allows previous search direction to influence current direction.

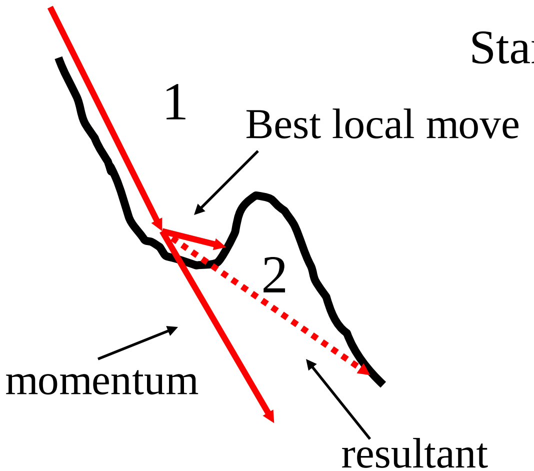


$$\Delta w_{kj}(t+1) = \eta \delta_k x_j + \alpha \Delta w_{kj}(t)$$

$$0 \le \eta , \alpha \le 1$$

Standard Delta-Rule update

Momentum term



Momentum smoothes the error landscape by guiding search in the <u>best average direction</u> (i.e., that which will, on average, decrease the error most) for a region (that may be quite jagged).

Design Issues for Learning ANNs

- Initial Weights
  - Random -vs- Biased
  - Width of init range
    - Typically: [-1 1] or [-0.5 0.5]
    - Too wide => large weights will drive many sigmoids to saturation => all output 1 => takes a lot of training to undo.

- Frequency of Weight Updates
  - Incremental - after each input.
    - In some cases, all training instances are not available at the same time, so the ANN must improve on-line.
    - Sensitive to presentation order.
  - Batch - after each epoch.
    - Uses less computation time
    - Insensitive to presentation order

Design Issues (2)

- Learning Rate
  - Low value => slow learning
  - High value => faster learning, but oscillation due to overshoot
  - Typical range: [0.1 0.9] - Very problem specific!
  - Dynamic learning rate (many heuristics):
    - Gradually decrease over the epochs
    - Increase (decrease) whenever performance improves (worsens)
    - Use 2nd deriv of error function
      - d^2E/dw^2 high => changing dE/dw => rough landscape => lower learning rate
      - d^2E/dw^2 low => dE/dw ~ constant => smooth landscape => raise learning rate
- Length of Training Period
  - Too short => Poor discrimination/classification of inputs
  - Too long => Overtraining => nearly perfect on training set, but not general enough to handle new (test) cases.
  - Many nodes & long training period = recipe for overtraining
  - Adding noise to training instances can help prevent overtraining.
    - (x1, x2…xn) => add noise => (x1+e1, x2 + e2….xn+en)

Design Issues (3)

- Size of Training Set
  - Heuristic: |Training Set | > k |Set of Weights| where k > 5 or 10.

- Stopping Criteria
  - Low error on training set
    - Can lead to overtraining if threshold is too low
  - Include extra validation set (preliminary test set) and test ANN on it after each epoch.
    - Stop when validation error is low enough.

Supervised Learning ANN Applications

- **Classification** *D: feature vectors => R: classes*
  - Medical Diagnosis: symptoms => disease
  - Letter Recognition: pixel array => letter

- **Control** *D: situation state vectors => R: responses/actions*
  - CMU's ALVINN: road picture => steering angle
  - Chemical plants:

    Temperature, Pressure, Chemical Concs in a container

    ⇒Valve settings for heat, chemical inputs/outputs

- **Prediction**

  D: Time series of previous states $s_1, s_2 \ldots s_n$ *=> R: next states* s_{n+1}

  - Finance: Price of a stock on days 1…n => price on day n+1
  - Meteorology: Cloud cover on days 1…n => cloud cover on day n+1