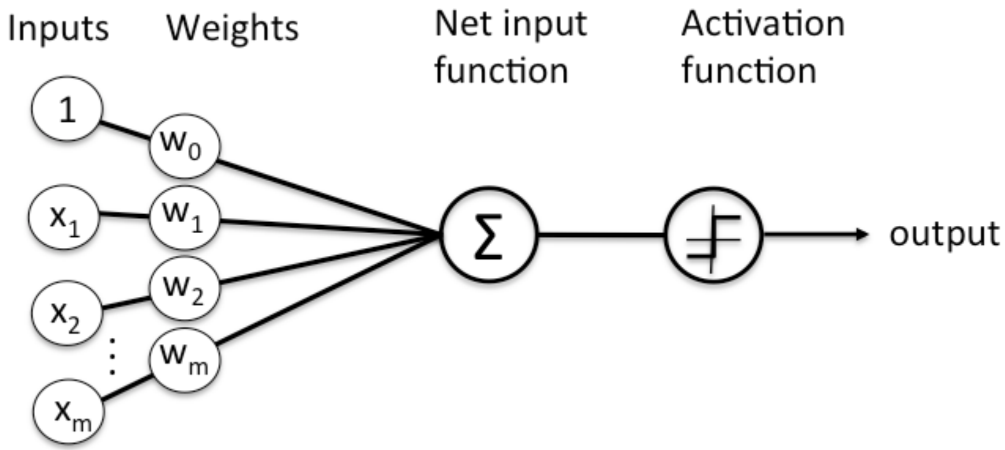
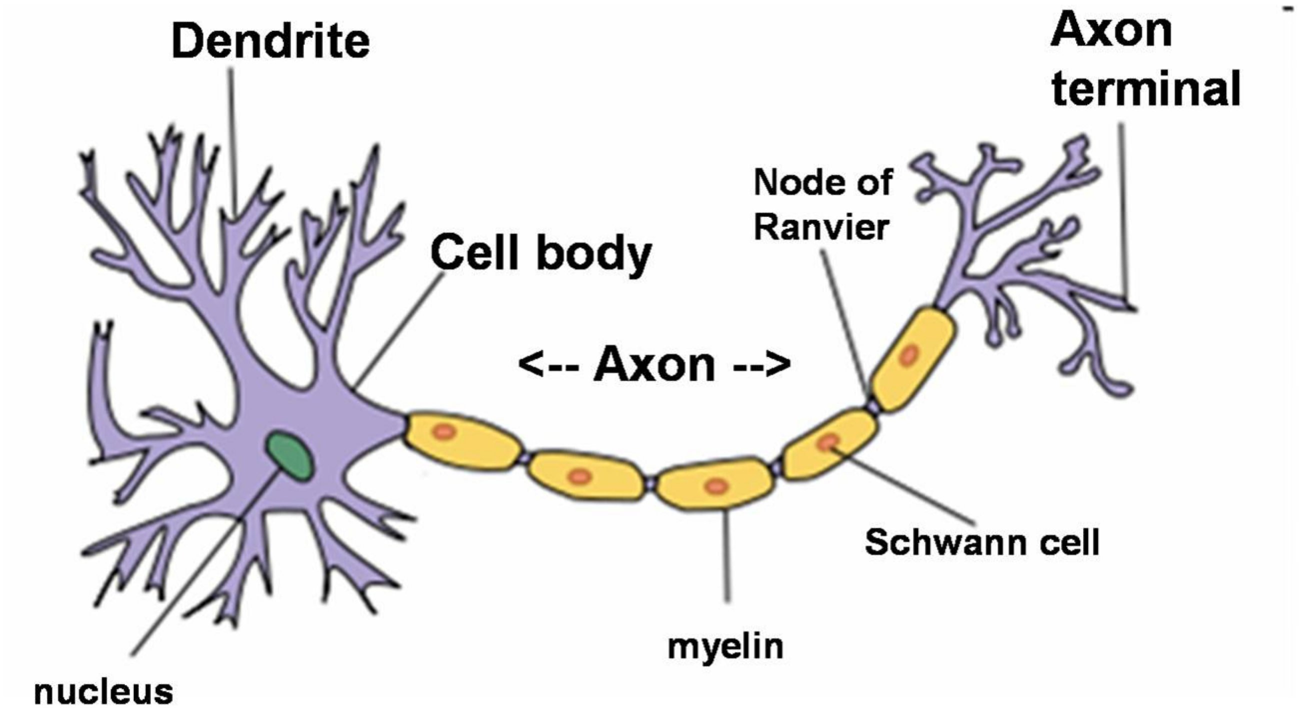
Unit 2
Neural Networks

TFIP-AI Artificial Neural Networks and Deep Learning

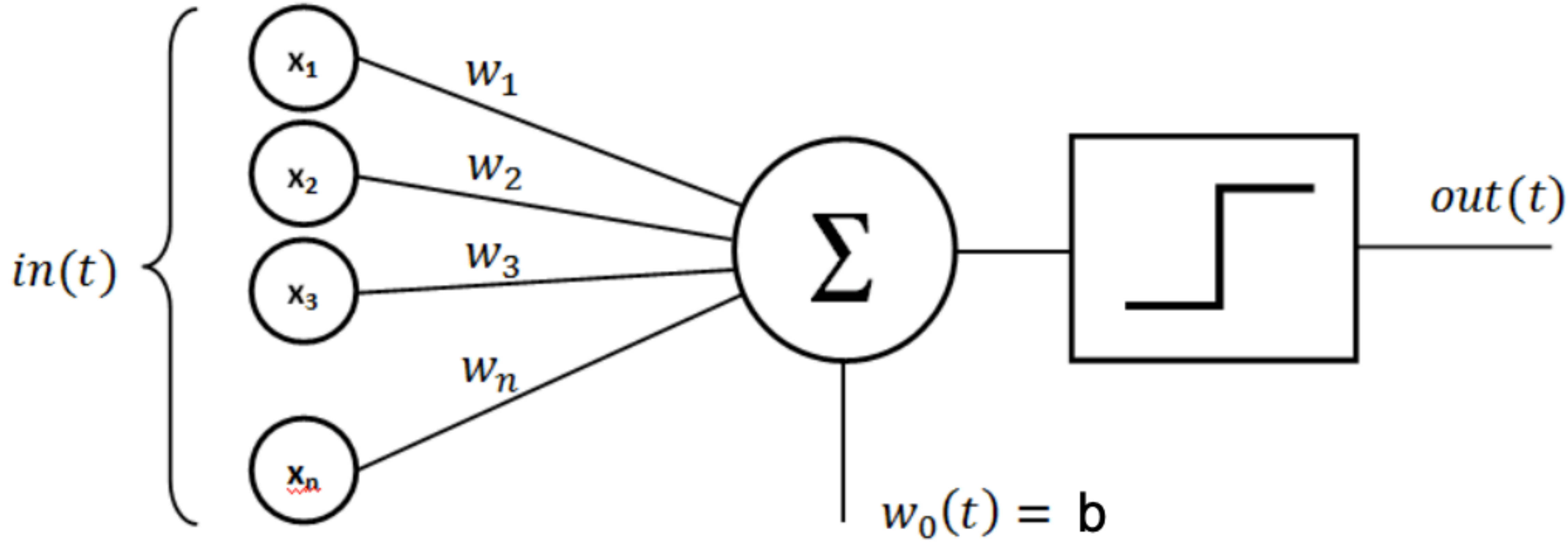
Feed-forward Neural Network

Neuron - perceptron





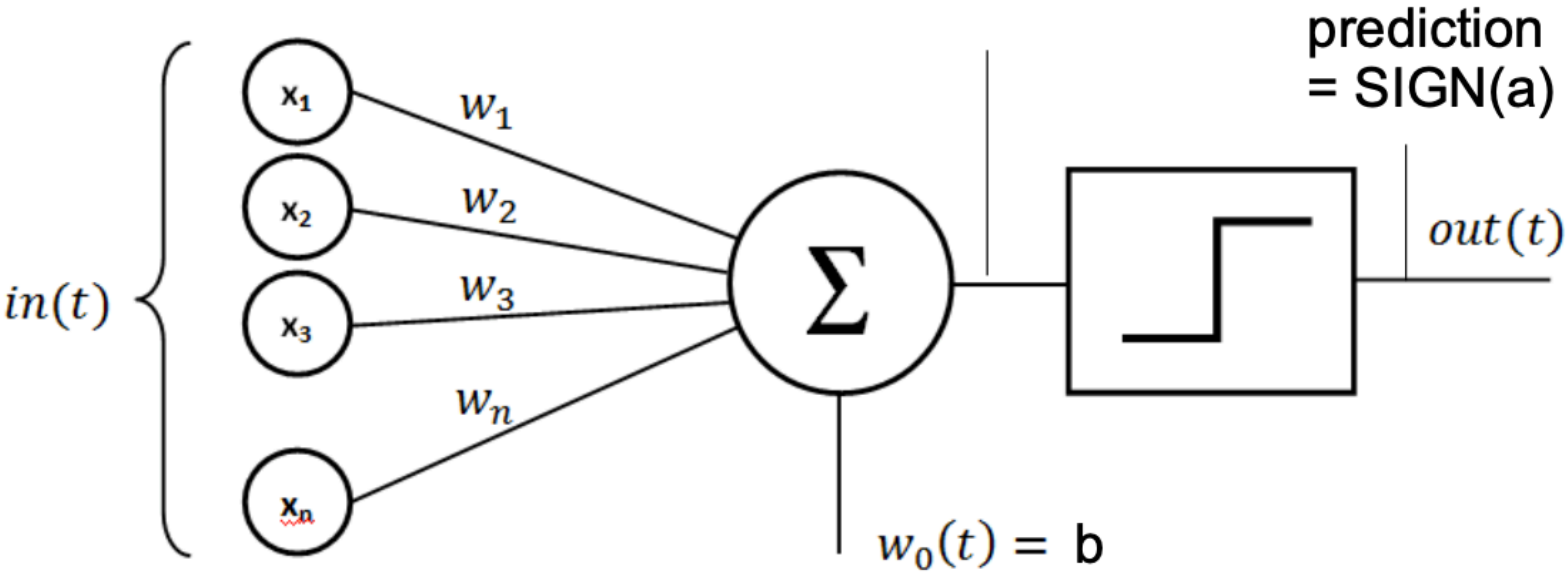
Perceptrons



Perceptrons cont...

Perceptron

$$a = b + \sum_{d=1}^{D} w_d x_d$$



Perceptrons cont...

Error driven learning

$$a = b + \sum_{d=1}^{D} w_d x_d$$

- At each step, return SIGN(a)
- if SIGN(a)≠ y update parameter
- otherwise don't change

Perceptrons cont…

Algorithm 5 PERCEPTRONTRAIN($\mathbf{D}$, $MaxIter$)

$w_d \leftarrow 0$, for all $d = 1 \dots D$ // initialize weights
$b \leftarrow 0$ // initialize bias
for $iter = 1 \dots MaxIter$ **do**
  for all $(x,y) \in \mathbf{D}$ **do**
    $a \leftarrow \sum_{d=1}^{D} w_d \, x_d + b$ // compute activation for this example
    if $ya \leq 0$ **then**
      $w_d \leftarrow w_d + yx_d$, for all $d = 1 \dots D$ // update weights
      $b \leftarrow b + y$ // update bias
    end if
  end for
end for
return $w_0, w_1, \dots, w_D, b$

Perceptrons cont...

Does this move a in the right direction?

- update $\mathbf{w}' = \mathbf{w} + y\mathbf{x} = \mathbf{w} + \mathbf{x}$
- $b' = b + y = b+1$

$$
\begin{aligned}
a' &= \sum_{d=1}^{D} w'_d x_d + b' \\
&= \sum_{d=1}^{D} (w_d + x_d) x_d + (b + 1) \\
&= \sum_{d=1}^{D} w_d x_d + b + \sum_{d=1}^{D} x_d x_d + 1 \\
&= a + \sum_{d=1}^{D} x_d^2 + 1 \quad > \quad a
\end{aligned}
$$

Perceptrons cont...

Does this move a in the right direction?

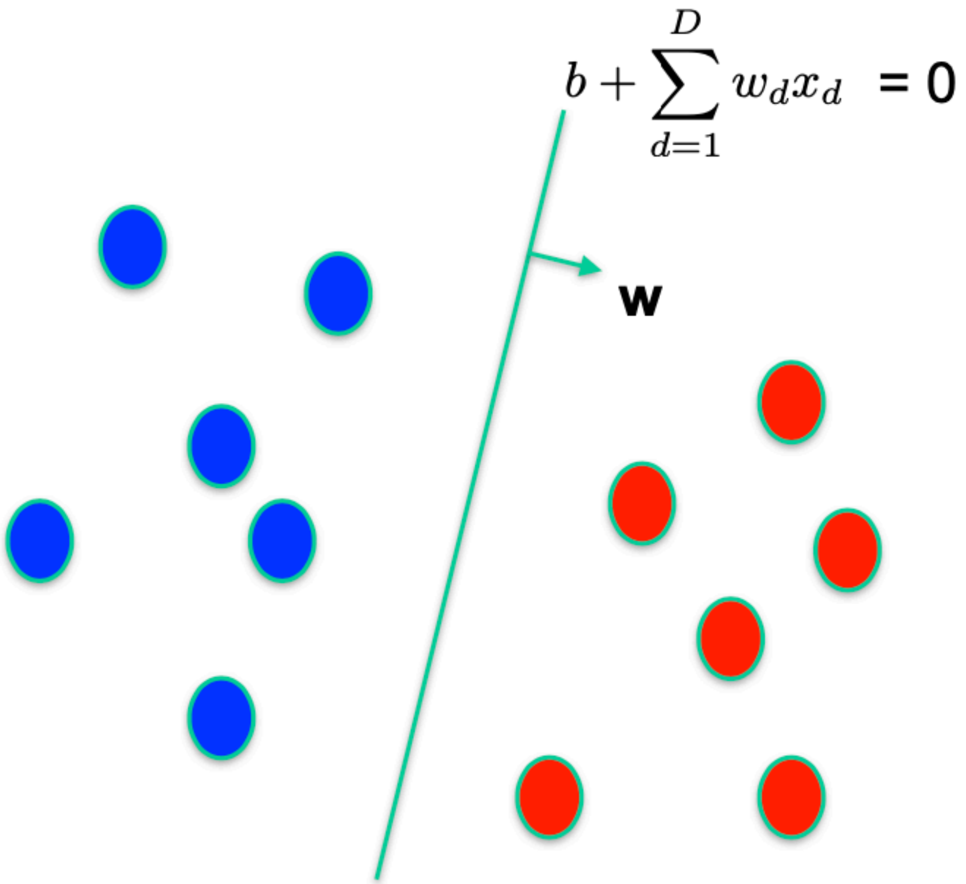
- update **w'** = **w** +y**x** = **w** + **x**
- b' = b + y = b+1

$$
\begin{aligned}
a' &= \sum_{d=1}^{D} w'_d x_d + b' \\
&= \sum_{d=1}^{D} (w_d + x_d) x_d + (b + 1) \\
&= \sum_{d=1}^{D} w_d x_d + b + \sum_{d=1}^{D} x_d x_d + 1 \\
&= a + \sum_{d=1}^{D} x_d^2 + 1 \quad > \quad a
\end{aligned}
$$

a becomes more positive
(not guaranteed that a>0)

Perceptrons cont...

What is the decision boundary?



Perceptrons cont…

How good is this algorithm?

- Convergence: an entire pass without changing the weights.

- If the data is linearly separable, the algorithm will converge. But not necessarily to the “best” boundary

Perceptrons cont...

Notion of margin

$$margin(\mathbf{D}, \boldsymbol{w}, b) = \begin{cases} \min_{(\boldsymbol{x},y)\in\mathbf{D}} y(\boldsymbol{w}\cdot\boldsymbol{x}+b) & \text{if } \boldsymbol{w} \text{ separates } \mathbf{D} \\ -\infty & \text{otherwise} \end{cases}$$

$$margin(\mathbf{D}) = \sup_{\boldsymbol{w},b} margin(\mathbf{D}, \boldsymbol{w}, b)$$

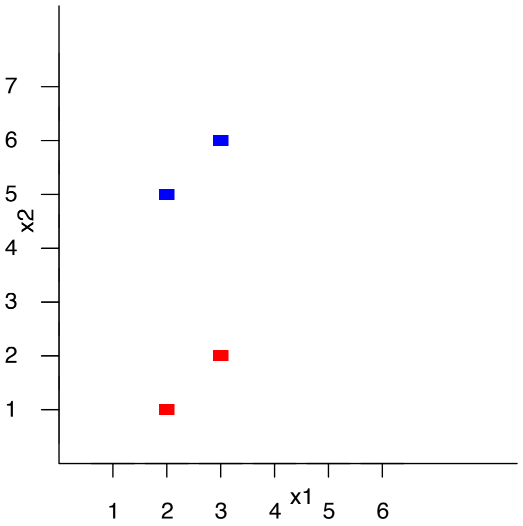
If data is linearly separable with margin γ and $||\mathbf{x}|| \leq 1$, then algorithm will converge in $\frac{1}{\gamma^2}$ updates

Perceptrons cont...

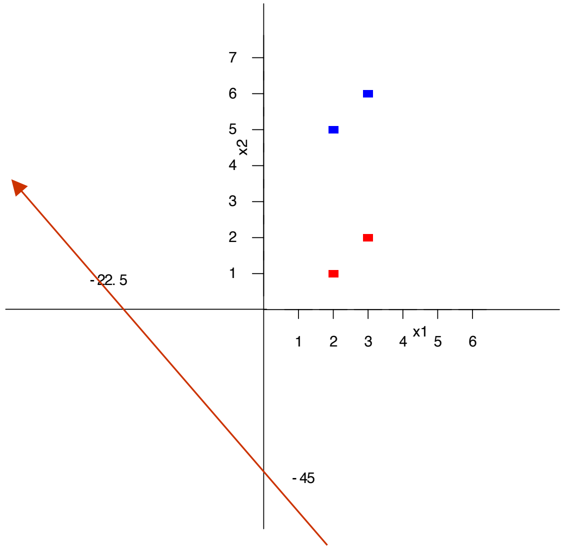
- The first generation of neural networks
- They were popularised by Frank Rosenblatt in the early 1960’s.
  - They appeared to have a very powerful learning algorithm.
  - Lots of grand claims were made for what they could learn to do.
- In 1969, Minsky and Papert published a book called “Perceptrons” that analysed what they could do and showed their limitations.
  - Many people thought these limitations applied to all neural network models.
- The perceptron learning procedure is still widely used today for tasks with enormous feature vectors that contain many millions of features.

Perceptrons: training

- Same as the learning method of logistic regression (with hard threshold or sigmoid function)
- Pick training cases using any policy that ensures that every training case will keep getting picked.
  - If the output unit is correct, leave its weights alone.
  - If the output unit incorrectly outputs a zero, add the input vector to the weight vector.
  - If the output unit incorrectly outputs a 1, subtract the input vector from the weight vector.
- This is guaranteed to find a set of weights that gets the right answer for all the training cases if any such set exists.

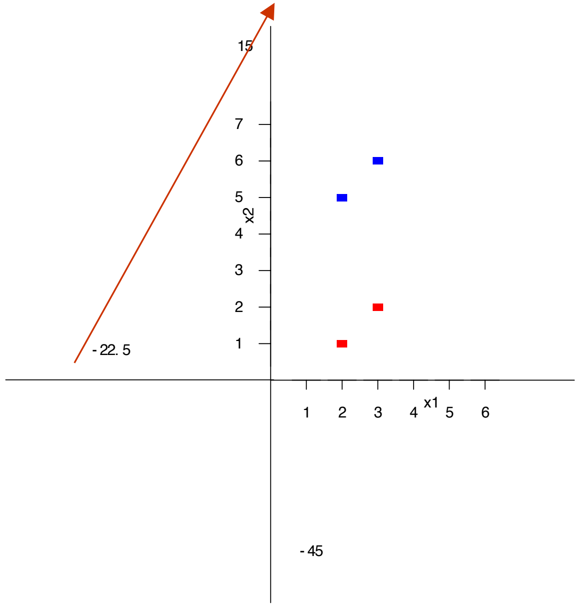


W= (0,0,0)



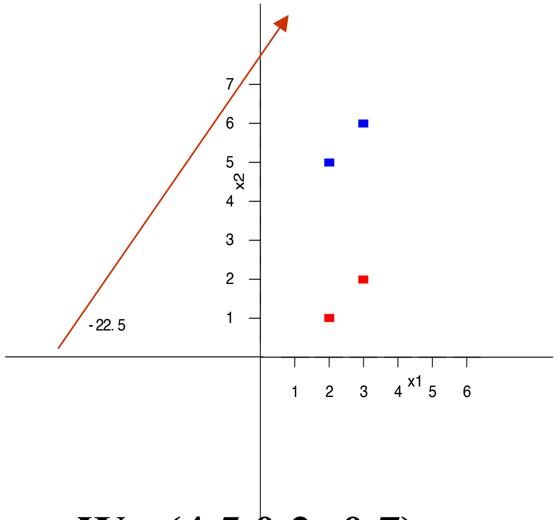
W= (4.5,0.2,0.1)

$0.2 \times x_1 + 0.1 \times x_2 + 4.5 = 0$



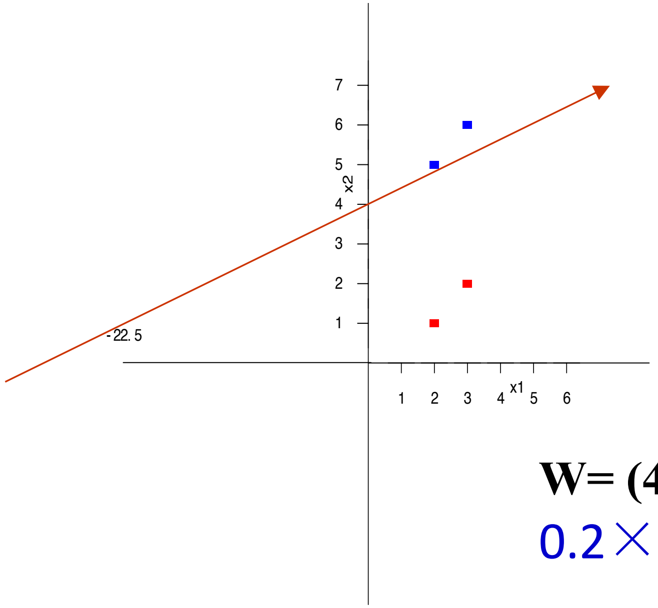
W= (4.5,0.2,-0.3)

$0.2 \times x_1 - 0.3 \times x_2 + 4.5 = 0$



W= (4.5,0.2,-0.7)

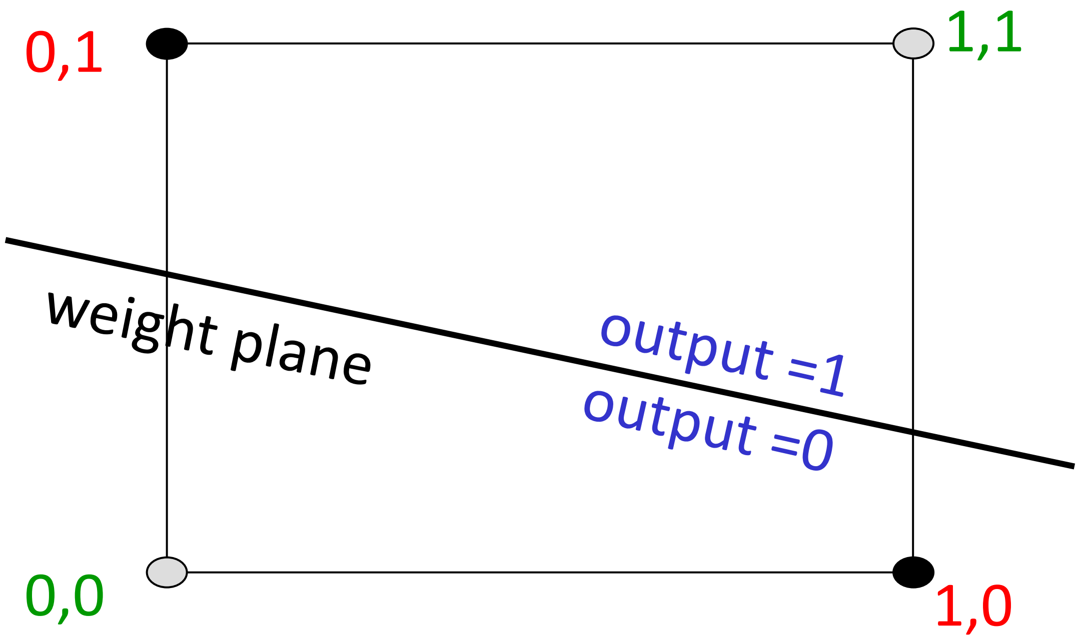
$0.2 \times x_1 - 0.7 \times x_2 + 4.5 = 0$



W= (4.5,0.2,-1.1)

$0.2 \times x_1 - 1.1 \times x_2 + 4.5 = 0$

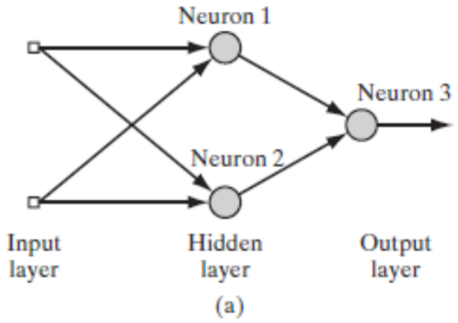
What perceptrons can't do



The positive and negative cases
cannot be separated by a plane

Hidden units

- Without hidden units, perceptrons are very limited
  - More layers of linear units do not help. It's still linear.
- We need multiple layers of adaptive, non-linear hidden units.

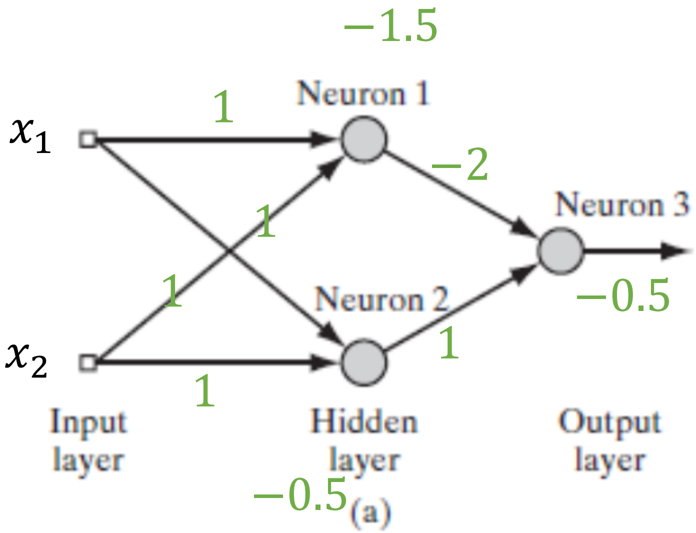


Neuron1 $w_{11} = w_{12} = +1 \quad b_1 = -\frac{3}{2}$

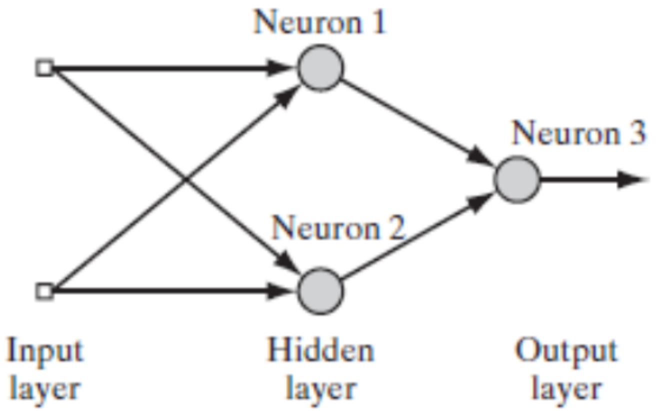
Neuron2 $w_{21} = w_{22} = +1 \quad b_2 = -\frac{1}{2}$

Neuron3 $w_{31} = -2 \quad w_{32} = +1 \quad b_3 = -\frac{1}{2}$

N.N. Inference

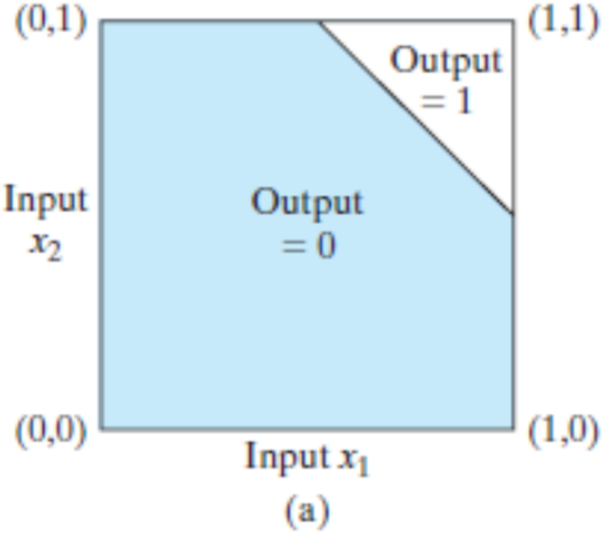


N.N. Inference

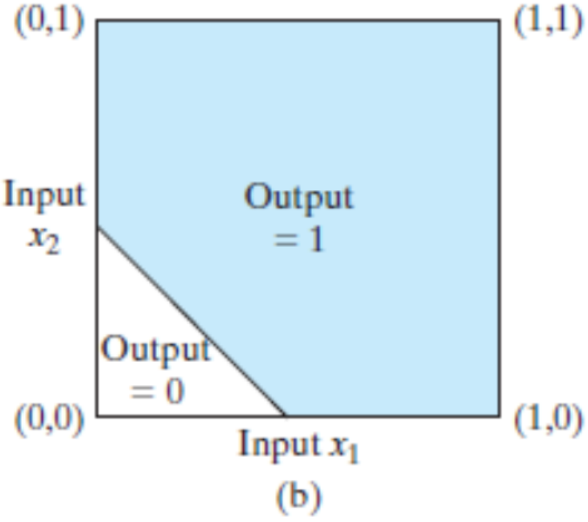


(a)

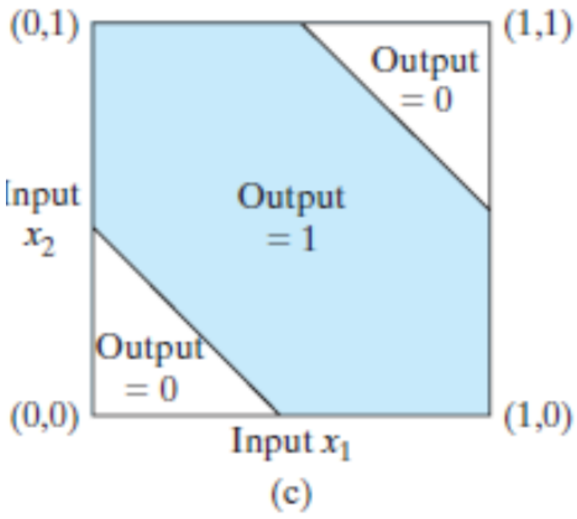
XOR Problem



Neuron1



Neuron2

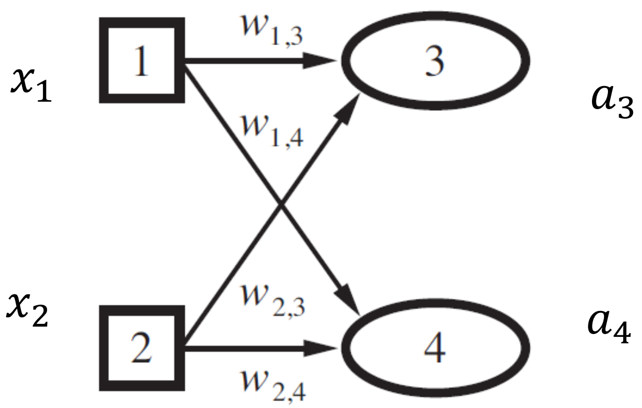


Neuron3

Loss function

- how can we train such nets?
  - We need an efficient way of adapting all the weights, not just the last layer. This is hard.
  - Learning the weights going into hidden units is equivalent to learning features.
  - This is difficult because nobody is telling us directly what the hidden units should do.

Learning - single layered



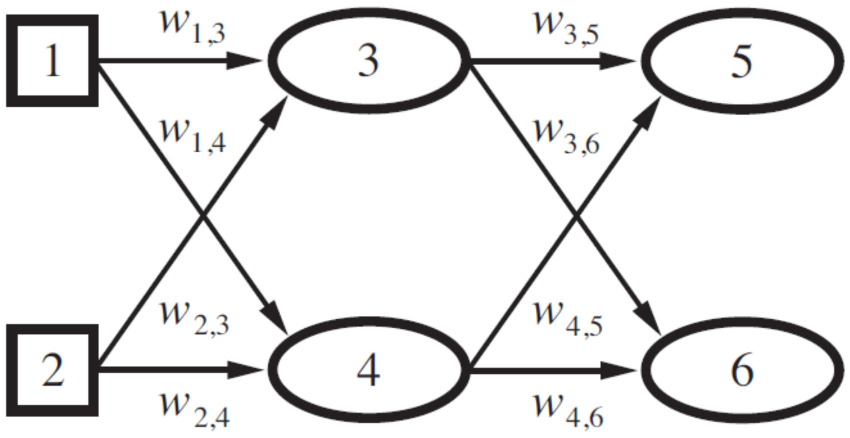
For a single sample (x1,x2)

$$a3 = g\left(w_{1,3} * x_1 + w_{2,3} * x_2 + w_{0,3}\right)$$
$$a4 = g\left(w_{1,4} * x_1 + w_{2,4} * x_2 + w_{0,4}\right)$$

$$E(\boldsymbol{w}) = \frac{1}{2}[(y_1 - a_3)^2 + (y_2 - a_4)^2]$$

$$\frac{\partial}{\partial \boldsymbol{w}} E(\boldsymbol{w}) = \frac{\partial}{\partial \boldsymbol{w1}} \frac{1}{2}(y_1 - a_3)^2 + \frac{\partial}{\partial \boldsymbol{w2}} \frac{1}{2}(y_2 - a_4)^2$$

Difficulty in learning with multilayer



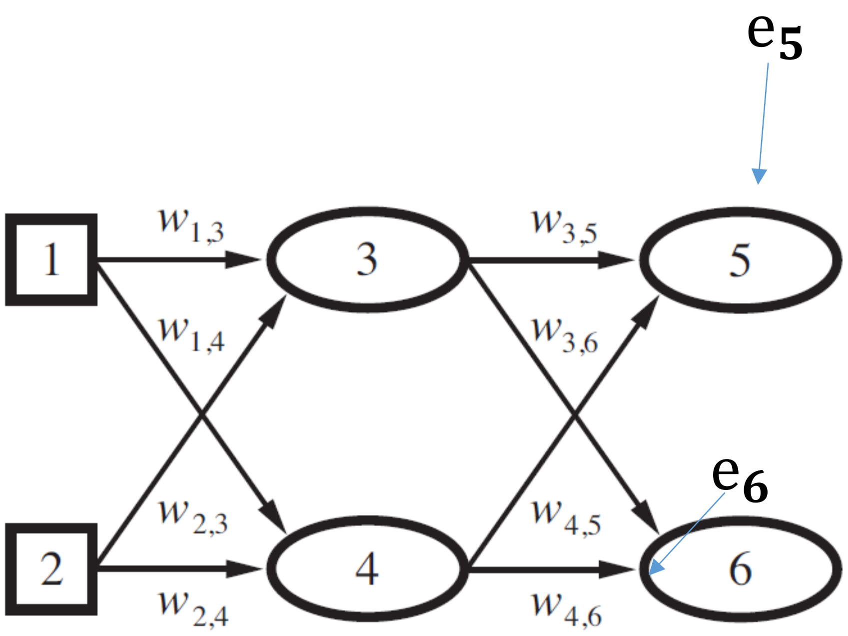
Error at the output layer is clear,

Error at the hidden layers seems mysterious

- the training data do not say what value the hidden nodes should have.

Error Back-propagate

Now Error at the hidden layers can be estimated



Error of Node3 is back propagated from e5 and e6

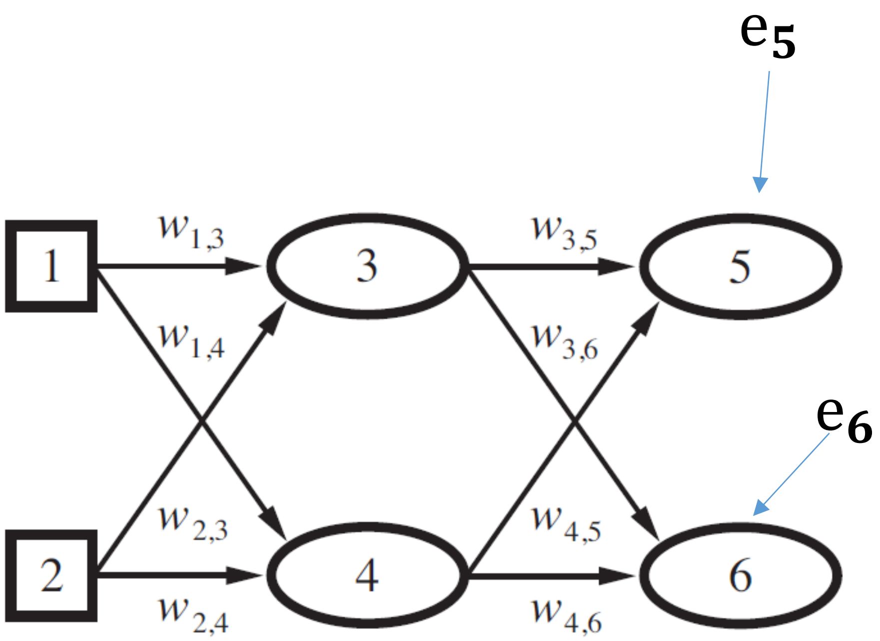
Output layer weight learning – chain rule

$$e_5 = \frac{1}{2}(y_1 - o_5)^{2}$$

$$o_5 = sigmoid(in_5)$$

Now Error at the hidden layers can be estimated

$$in_5 = w_{3,5} * o_3 + w_{4,5} * o_4 + w_{0,5}$$

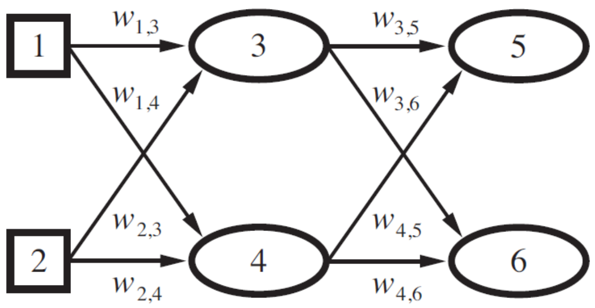


$$\frac{\partial e_5}{\partial w_{3,5}} = \frac{\partial e_5}{\partial o_5} * \frac{\partial o_5}{\partial in_5} * \frac{\partial in_5}{\partial w_{3,5}}$$

$$= -(y_1 - o_5) * sigmoid(in_5) * (1 - sigmoid(in_5)) * o_3$$

$$sigmoid(x)' = sigmoid(x) * (1 - sigmoid(x))$$

Output layer weight learning – delta rule



Gradient e5

$$\frac{\partial e_5}{\partial w_{3,5}} = -(y_1 - o_5) * sigmoid(in_5) * (1 - sigmoid(in_5)) * o_3$$

$$= \underbrace{-(y_1 - o_5) * o_5 * (1 - o_5)}_{\delta_5} * o_3$$

Gradient Descent

$$w_{3,5} = w_{3,5} - \alpha * \delta_5 * o_3$$

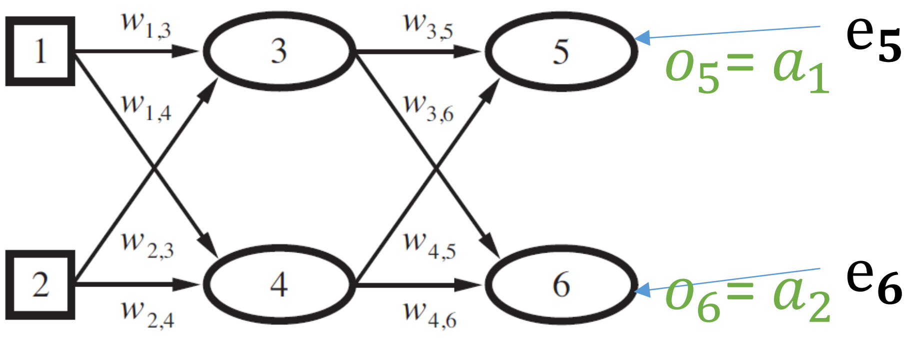
$$w_{4,5} = w_{4,5} - \alpha * \delta_5 * o_4$$

$$w_{0,5} = w_{0,5} - \alpha * \delta_5$$

$$\frac{\partial e_5}{\partial w_{4,5}} = \delta_5 * o_4$$

$$\frac{\partial e_5}{\partial w_{0,5}} = \delta_5$$

Hidden layer weight learning



$$e_3 = e_5 + e_6$$

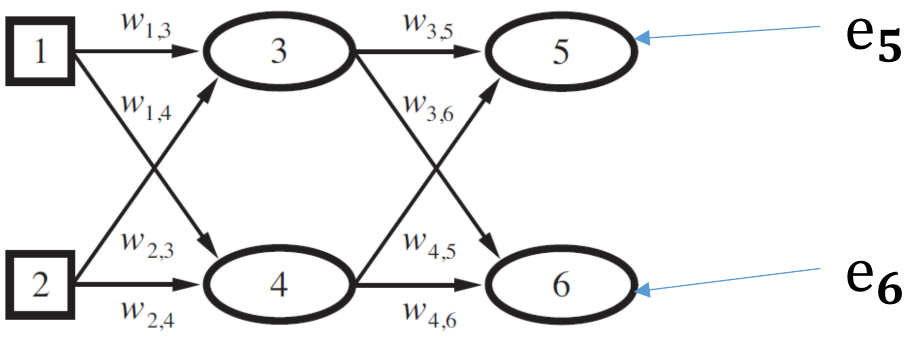
$$\frac{\partial e_3}{\partial w_{1,3}} = \frac{\partial e_5}{\partial w_{1,3}} + \frac{\partial e_6}{\partial w_{1,3}}$$

$$\frac{\partial e_5}{\partial w_{1,3}} = \delta_5 * w_{3,5} * o_3 * (1 - o_3) * x_1$$

$$\frac{\partial e_5}{\partial w_{2,3}} = \delta_5 * w_{3,5} * o_3 * (1 - o_3) * x_2$$

…

Hidden layer weight learning



$$e_3 = e_5 + e_6$$

$$\frac{\partial e_3}{\partial w_{1,3}} = \frac{\partial e_5}{\partial w_{1,3}} + \frac{\partial e_6}{\partial w_{1,3}}$$

$$\frac{\partial e_5}{\partial w_{1,3}} = \delta_5 * w_{3,5} * o_3 * (1 - o_3) * x_1$$

$$\frac{\partial e_5}{\partial w_{2,3}} = \delta_5 * w_{3,5} * o_3 * (1 - o_3) * x_2$$

…

$$\frac{\partial e_3}{\partial w_{1,3}} = \delta_5 * w_{3,5} * o_3 * (1 - o_3) * x_1 + \delta_6 * w_{3,6} * o_3 * (1 - o_3) * x_1$$

$$= \underbrace{(\delta_5 * w_{3,5} + \delta_6 * w_{3,6})}_{\delta_3} * \underbrace{o_3 * (1 - o_3)}_{sigmoid(x)'} * x_1$$

BP algorithm (stochastic + sigmoid)

- Step0 define # of layers, # of nodes
- Step1 initialize parameters: weights and bias
- Step2 feed forward
  - Calculate output for each non-input layer node
- Step3 back propagate
  - Compute sensitivity for each non-input layer node i
  - Update parameters
- Step4 Convergence
  - Compute cost function
  - Repeat step2 until convergence

sensitivity

$$\delta_i = error_i \cdot o_i \cdot (1 - o_i)$$

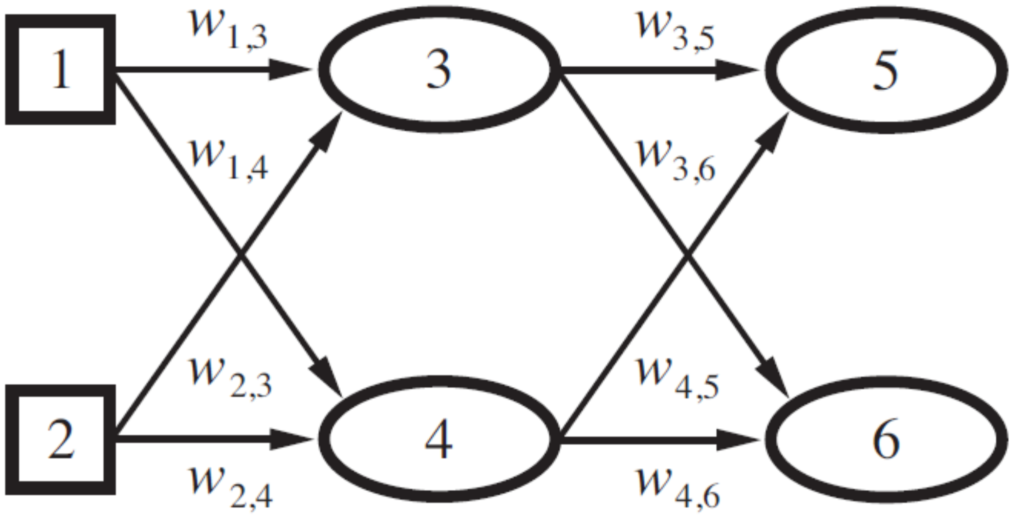
$$error_i = -(y - o_i) \qquad \text{output-layer}$$

$$error_i = \sum_j (\delta_j * w_{i,j}) \qquad \text{hidden-layer}$$

$$w_{k,i} = w_{k,i} - \alpha \cdot \delta_i \cdot o_k$$

Back-propagate - Example

B.P. Example – the Network



$$in_3 = x_1 \cdot w_{1,3} + x_2 \cdot w_{2,3} + w_{0,3}$$

$$o_3 = sigmoid(in_3)$$

B.P. Example – Feed Forward

$$o_3 = \text{sigmoid}(x_1 \cdot w_{1,3} + x_2 \cdot w_{2,3} + w_{0,3})$$

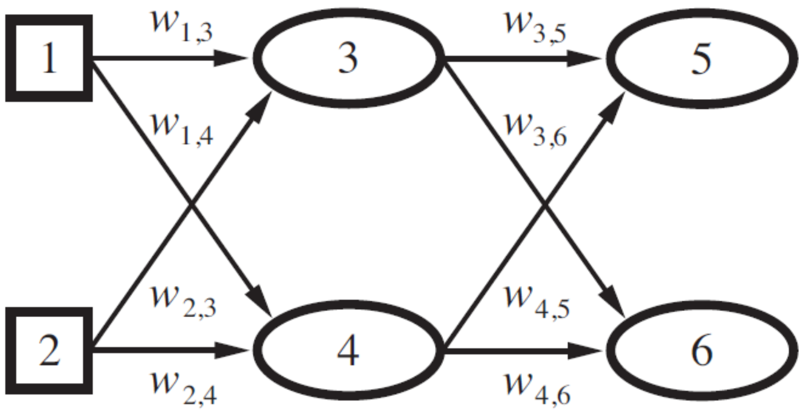
$$o_4 = \text{sigmoid}(x_1 \cdot w_{1,4} + x_2 \cdot w_{2,4} + w_{0,4})$$

$$o_5 = \text{sigmoid}(o_3 \cdot w_{3,5} + o_4 \cdot w_{4,5} + w_{0,5})$$

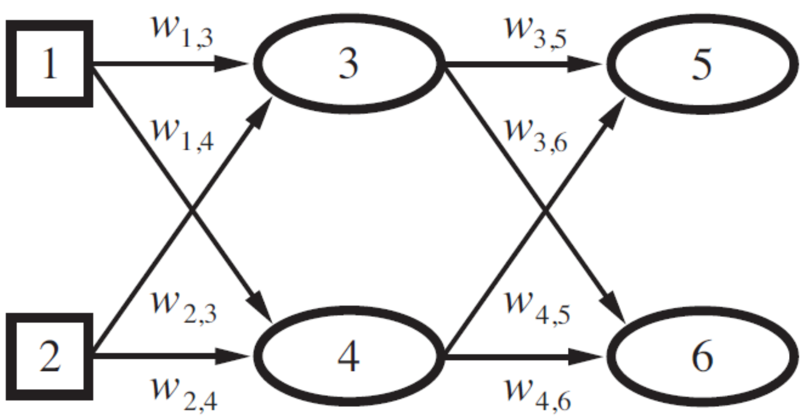
$$o_6 = \text{sigmoid}(o_3 \cdot w_{3,6} + o_4 \cdot w_{4,6} + w_{0,6})$$

$$\delta_5 = -(y_1 - o_5) \cdot o_5 \cdot (1 - o_5)$$

$$\delta_6 = -(y_2 - o_6) \cdot o_6 \cdot (1 - o_6)$$



B.P. Example – Learning (stochastic)



$$\delta_3 = o_3 \cdot (1 - o_3) \cdot (\delta_5 \cdot w_{3,5} + \delta_6 \cdot w_{3,6})$$

$$\delta_4 = o_4 \cdot (1 - o_4) \cdot (\delta_5 \cdot w_{4,5} + \delta_6 \cdot w_{4,6})$$

$$w_{3,5} = w_{3,5} - \alpha \cdot \delta_5 \cdot o_3$$

$$w_{4,5} = w_{4,5} - \alpha \cdot \delta_5 \cdot o_4$$

$$w_{0,5} = w_{0,5} - \alpha \cdot \delta_5 \cdot 1$$

$$w_{3,6} = w_{3,6} - \alpha \cdot \delta_6 \cdot o_3$$

$$w_{4,6} = w_{4,6} - \alpha \cdot \delta_6 \cdot o_4$$

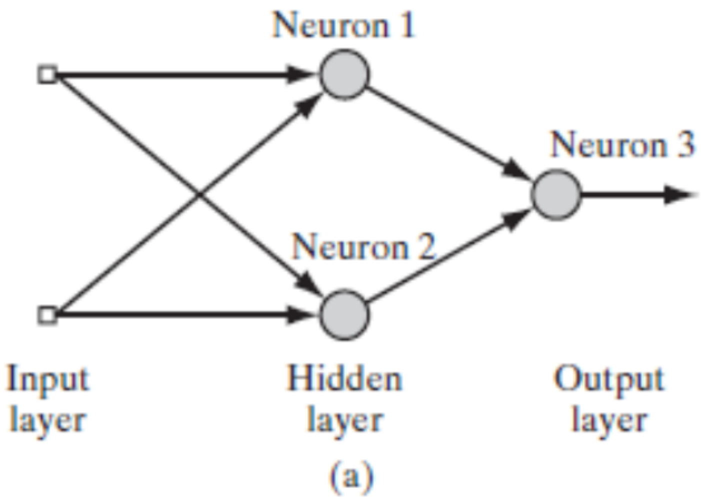
$$w_{0,6} = w_{0,6} - \alpha \cdot \delta_6 \cdot 1$$

$$w_{1,3} = w_{1,3} - \alpha \cdot \delta_3 \cdot x_1 \qquad w_{1,4} = w_{1,4} - \alpha \cdot \delta_4 \cdot x_1$$

$$w_{2,3} = w_{2,3} - \alpha \cdot \delta_3 \cdot x_2 \qquad w_{2,4} = w_{2,4} - \alpha \cdot \delta_4 \cdot x_2$$

$$w_{0,3} = w_{0,3} - \alpha \cdot \delta_3 \cdot 1 \qquad w_{0,4} = w_{0,4} - \alpha \cdot \delta_4 \cdot 1$$

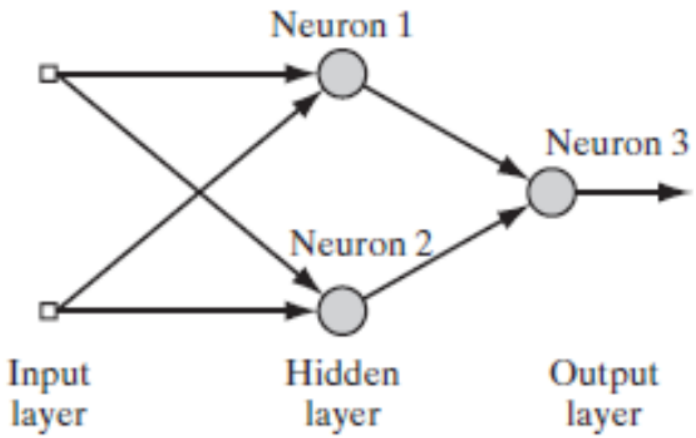
XOR with 2 nodes hidden layer



X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0

Parameter initialization

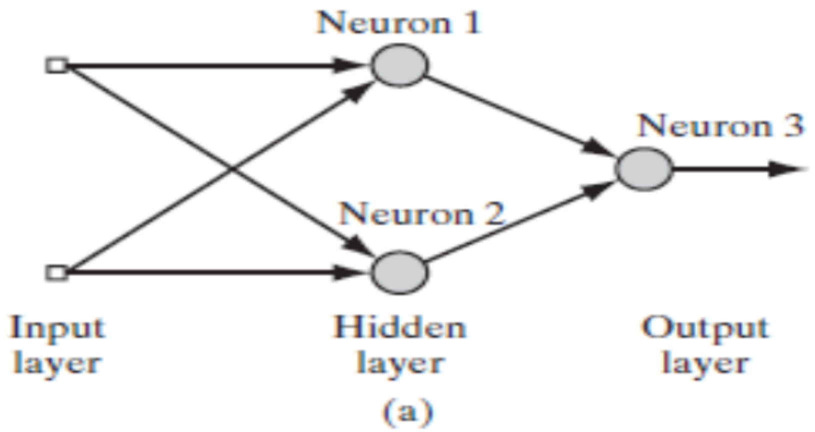
- To avoid symmetry problem, the initial value for the weights would not be zeros any longer,
- Instead, they would be a small randomized value
  - $w_{i1,1} = 0.02$ (randomly generated)
  - $w_{i2,1} = 0.03$
  - $w_{0,1} = 0$
  - $w_{i1,2} = 0.01$
  - $w_{i2,2} = 0.02$
  - $w_{0,2} = 0$
  - $w_{1,3} = 0.01$
  - $w_{2,3} = 0.03$
  - $w_{0,3} = 0$



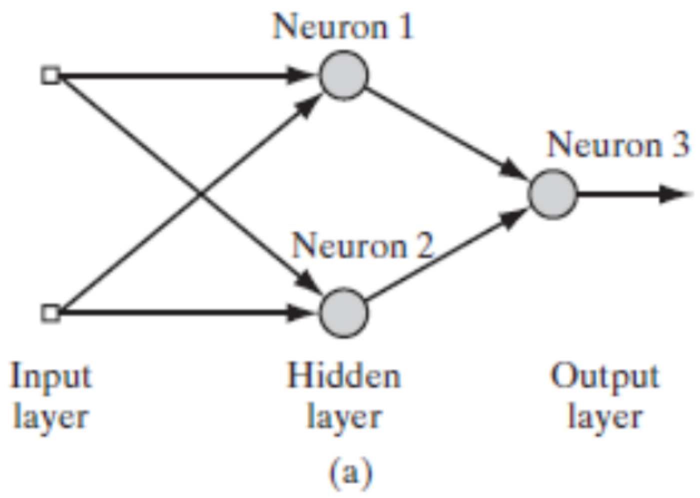
(a)

Feed forward – input(0,0)

- $o_1 = sigmoid(0) \ = 0.5$
- $o_2 = sigmoid(0) \ = 0.5$
- $o_3 = sigmoid(0.5 \cdot 0.01 + 0.5 \cdot 0.03 + 0) = 0.505$

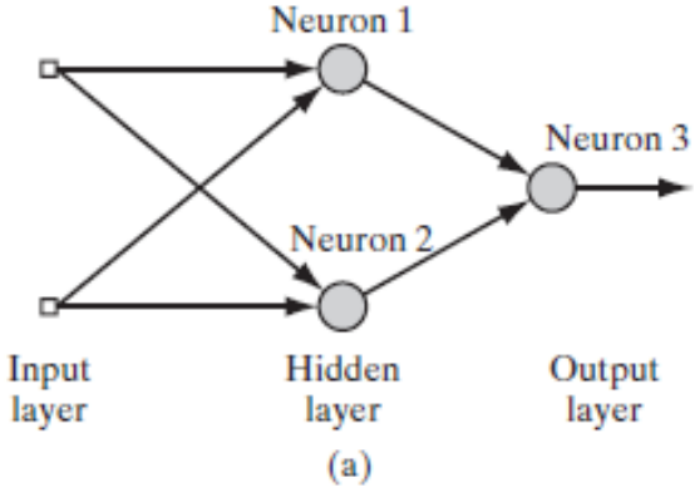


Back propagate – input(0,0)



- We have
  - o[3]={0.5, 0.5, 0.505}
  - $o_3 = 0.505$, and y = 0
- Sensitivities for all nodes
  - $\delta_3 = -(y - o_3) \cdot o_3 \cdot (1 - o_3) = -(0 - 0.505)0.505(1 - 0.505) = 0.126$
  - $\delta_1 = \left(\delta_3 w_{1,3}\right) o_1 (1 - o_1) = (0.126) \cdot 0.01 \cdot 0.5 \cdot (1 - 0.5) = 0.000315$
  - $\delta_2 = \left(\delta_3 w_{2,3}\right) o_2 (1 - o_2) = (0.126) \cdot 0.03 \cdot 0.5 \cdot (1 - 0.5) = 0.000945$
- Hidden-Output Weight
  - $w_{1,3} = w_{1,3} - \alpha \cdot \delta_3 \cdot o_1 = 0.01 - 0.1 \cdot 0.126 \cdot 0.5 = -0.0037$
  - $w_{2,3} = w_{2,3} - \alpha \cdot \delta_3 \cdot o_2 = 0.03 - 0.1 \cdot 0.126 \cdot 0.5 = -0.0237$
  - $w_{0,3} = w_{0,3} - \alpha \cdot \delta_3 = 0 - 0.1 \cdot 0.126 = -0.0126$

Back propagate – input(0,0)



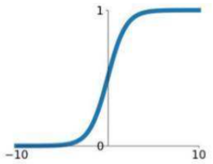
- We have
  - o[3]={0.5, 0.5, 0.505}
  - δ[3]={0.000315, 0.000945, 0.126}
- Input-Hidden
  - $w_{i1,1} = w_{i1,1} - \alpha \cdot \delta_1 \cdot x_1 = 0.02 - 0.1 \cdot 0.000315 \cdot 0 = 0.02$
  - $w_{i2,1} = w_{i2,1} - \alpha \cdot \delta_1 \cdot x_2 = 0.03 - 0.1 \cdot 0.000315 \cdot 0 = 0.03$
  - $w_{0,1} = w_{0,1} - \alpha \cdot \delta_1 = 0 - 0.1 \cdot 0.000315 = -0.0000315$
  - $w_{i1,2} = w_{i1,2} - \alpha \cdot \delta_2 \cdot x_1 = 0.01 - 0.1 \cdot 0.000945 \cdot 0 = 0.01$
  - $w_{i2,2} = w_{i2,2} - \alpha \cdot \delta_2 \cdot x_2 = 0.02 - 0.1 \cdot 0.000945 \cdot 0 = 0.02$
  - $w_{0,2} = w_{0,2} - \alpha \cdot \delta_2 = 0 - 0.1 \cdot 0.000945 = -0.0000945$

Activation Functions

- sigmoid
  - Gradient vanishing
  - Output is not zero-centered – slow down the learning
  - Power operation – time cost
- tanh (Hyperbolic Tangent)
  - Gradient vanishing
  - Output is not zero-centered
  - Power operation
- Relu
  - Gradient vanishing
  - Output is not zero-centered
  - Power operation
  - Dead Relu Problem

Sigmoid

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

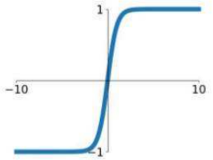


$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$\sigma(x) \cdot (1 - \sigma(x))$

tanh

$\tanh(x)$

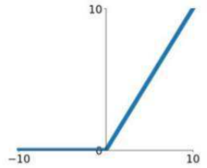


$$\tanh x = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

$1 - \tanh(x)^2$

ReLU

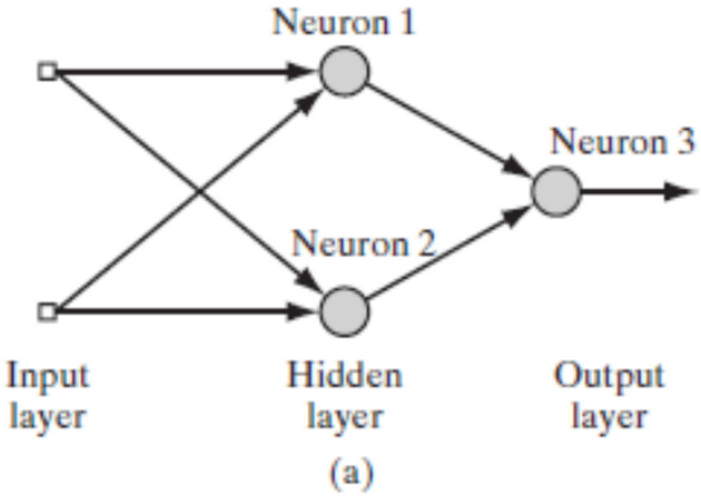
$\max(0, x)$



$$\mathrm{ReLU} = \max(0, x)$$

$$ReLU'(x) = \begin{cases} 0, x < 0 \\ 1, x \geq 0 \end{cases}$$

XOR with 2 nodes hidden layer - vectorization



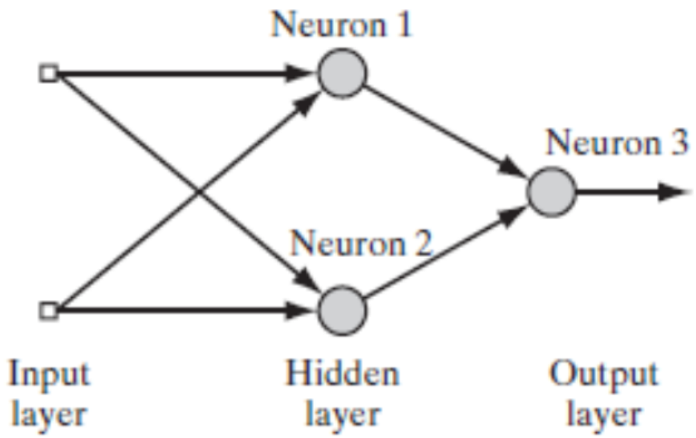
- Hidden layer
  - tanh
- Output layer
  - sigmoid

X0	X1	X2	Y
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

- X is a 4×3 matrix
- Y is a 4×1 vector

Parameter initialization

- Hidden layer
  - Weight_hidden = $\begin{bmatrix} 0.02 & 0.01 \\ 0.03 & 0.02 \end{bmatrix}$
  - Bias_hidden= $[0 \quad 0]$
- Output layer
  - Weight_output = $\begin{bmatrix} 0.01 \\ 0.03 \end{bmatrix}$
  - Bias_output= $[0]$



(a)

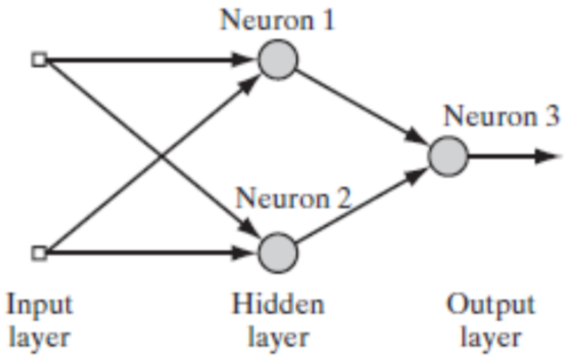
$w_{i1,1} = 0.02$
$w_{i2,1} = 0.03$
$w_{0,1} = 0$
$w_{i1,2} = 0.01$
$w_{i2,2} = 0.02$
$w_{0,2} = 0$
$w_{1,3} = 0.01$
$w_{2,3} = 0.03$
$w_{0,3} = 0$

Feed forward – Hidden layer output

- X = $\begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$ Y = $\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$
- Weight_hidden = $\begin{bmatrix} 0.02 & 0.01 \\ 0.03 & 0.02 \end{bmatrix}$
- Bias_hidden= $[0 \quad 0]$

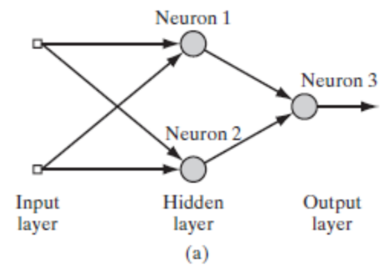
- hidden_output = tanh(np.dot(X,weight_hidden)+ bias_hidden))

hidden_output = $\begin{bmatrix} o_1^{(1)} & o_2^{(1)} \\ o_1^{(2)} & o_2^{(2)} \\ o_1^{(3)} & o_2^{(3)} \\ o_1^{(4)} & o_2^{(4)} \end{bmatrix}$



(a)

Feed forward – output layer output

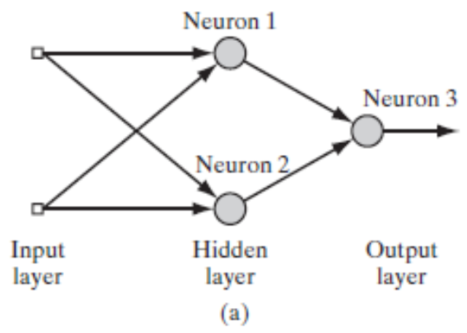


- yhat = expit(np.dot(hidden_output,weight_output) + bias_output)

- hidden_output = $\begin{bmatrix} o_1^{(1)} & o_2^{(1)} \\ o_1^{(2)} & o_2^{(2)} \\ o_1^{(3)} & o_2^{(3)} \\ o_1^{(4)} & o_2^{(4)} \end{bmatrix}$, yhat = $\begin{bmatrix} o_3^{(1)} \\ o_3^{(2)} \\ o_3^{(3)} \\ o_3^{(4)} \end{bmatrix}$

- Weight_output = $\begin{bmatrix} 0.01 \\ 0.03 \end{bmatrix}$
- Bias_output=$[0]$

Sensitivity computing



- delta_output = -(Y-yhat)*(yhat)*(1-yhat)
- delta_hidden = np.dot(delta_output,weight_output.T)*(1-np.square(hidden_output))
- delta_output = $\begin{bmatrix} \delta_3^{(1)} \\ \delta_3^{(2)} \\ \delta_3^{(3)} \\ \delta_3^{(4)} \end{bmatrix}$, delta_hidden = = $\begin{bmatrix} \delta_1^{(1)} & \delta_2^{(1)} \\ \delta_1^{(2)} & \delta_2^{(2)} \\ \delta_1^{(3)} & \delta_2^{(3)} \\ \delta_1^{(4)} & \delta_2^{(4)} \end{bmatrix}$

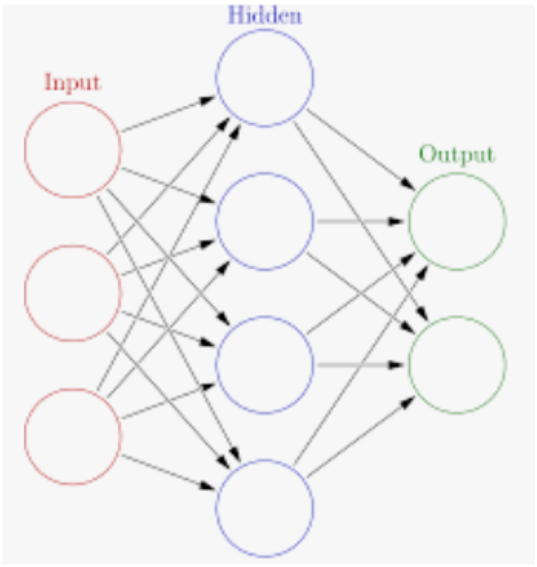
Parameters Update

- weight_output = weight_output - α*np.dot(hidden_output.T,delta_output)/4
- bias_output = bias_output -α*np.sum(delta_output)/4

- weight_hidden = weight_hidden -α*np.dot(X.T, delta_hidden)/4
- bias_hidden = bias_hidden -α*np.sum(delta_hidden,axis=0)/4

Weight Initialization

All Zero Initialization(pitfall)



- We assume: with proper data normalization
  - Half of the weights will be positive
  - Half of them will be negative
  - 0 would be the best guess
- It is a mistake
  - Same output → same gradient → same parameter updates
- Logistic Regression
  - All 0 initialization is ok

Small random numbers

- Close to zero, but not identically zero
- Symmetry breaking
- $W \sim N(0, \sigma)$
  - W is initialized as a random vector sampled from a multi-dimensional gaussian
- Smaller numbers $\neq$ work better
  - very small weights → small gradient → gradient diminish

Uniform distribution initialization

$$W \sim U[-\frac{1}{\sqrt{n_{in}}}, \frac{1}{\sqrt{n_{in}}}]$$

where $U[-a, a]$ is the uniform distribution in the interval (−a,a), and n_{in} is the size of the previous layer

Xavier initialization

- Problems in weights initialization
  - If the weights in a network start too small, then the signal shrinks as it passes through each layer until it’s too tiny to be useful. (Gradient Vanishing)
  - If the weights in a network start too large, then the signal grows as it passes through each layer until it’s too massive to be useful. (Gradient Exploding)

- Gaussian distribution with zero mean and a suitable variance

Xavier initialization

$$n_{in} = n, n_{out} = m$$

- For simplicity, let's assume $m = 1$

$$y = W^T X = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$$

  - to help keep the signal from exploding to a high value or vanishing to zero
  - we want the variance to remain the same after passing the layer

$$\mathrm{var}(y) = \mathrm{var}(w_1 x_1 + w_2 x_2 + \cdots + w_n x_n) = \mathrm{var}(w_1 x_1) + \cdots \mathrm{var}(w_n x_n)$$

  - Assume W and X are independent, both of them have zero mean

$$\mathrm{var}(w_i x_i) = E[x_i]^2 \mathrm{var}(w_i) + E[w_i]^2 \mathrm{var}(x_i) + \mathrm{var}(w_i)\,\mathrm{var}(x_i)$$

Xavier initialization

$$\text{var}(w_i x_i) = E[x_i]^2 \text{var}(w_i) + E[w_i]^2 \text{var}(x_i) + \text{var}(w_i)\,\text{var}(x_i)$$

- I.ID assumption on each w_i and x_i

$$\text{var}(y) = n \cdot \text{var}(w_i)\,\text{var}(x_i)$$

- if we want to make sure the variance of y to be the same as X , then we need $n \cdot \text{var}(w_i) = 1$. Hence,

$$\text{var}(w_i) = \frac{1}{n} = \frac{1}{n_{in}}$$

- Similarly, if you go through the same steps for the backpropagated signal, you find that you need

$$\text{var}(w_i) = \frac{1}{n_{out}}$$

Xavier initialization

- These two constraints can only be satisfied simultaneously if $n_{in} = n_{out}$
- As a compromise,

$$\mathrm{var}(w_i) = \frac{2}{n_{in} + n_{out}}$$

- Moreover, the author also introduced a normalized initialization version follows uniform distribution

$$W \sim U[-\frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}]$$

- What's more, Xavier with uniform distribution + sigmoid activation

$$W \sim U[-\frac{4 \cdot \sqrt{6}}{\sqrt{n_{in} + n_{out}}}, \frac{4 \cdot \sqrt{6}}{\sqrt{n_{in} + n_{out}}}]$$

Xavier initialization

- Xavier is aimed to deal with gradient vanishing or exploding problems

- **Xavier initialization would not be use with ReLU activation,** which would not lead to vanishing or exploding gradients

ReLU initialization

- Proposed to handle the issue that some very deep CNNs have difficulties to converge
- Weights in those CNN are initialized by random weights drawn from Gaussian distributions with fixed standard deviations
- n_l: size of layer l

$$W \sim N(0, \sqrt{\frac{2}{n_l}}) \qquad W \sim U[-\sqrt{\frac{6}{n_l}}, \sqrt{\frac{6}{n_l}}]$$