Lecture 6

Artificial Neural Networks

Physical Address for Prof. Ilias Tagkopoulos
Computer Science:
Office: 3063 Kemper Hall
Phone: (530) 752-4821
Fax: (530) 752-4767

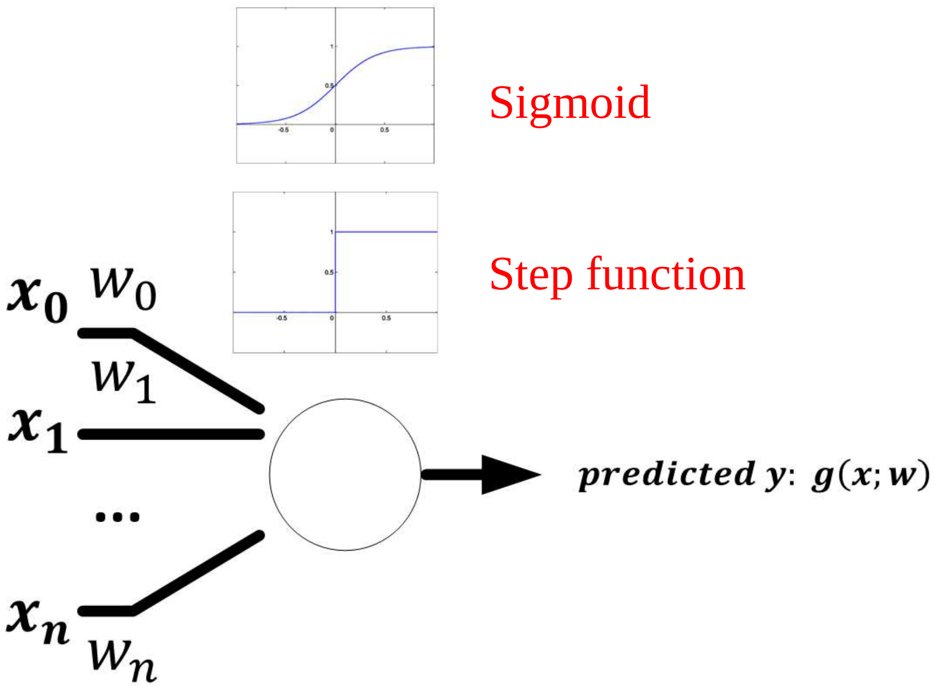
Genome and Biomedical Sciences Facility:
Office: 5313 GBSF
Phone: (530) 752-7707
Fax: (530) 754-9658

Instructor: Ilias Tagkopoulos
iliast@ucdavis.edu

Building Boolean Functions

Lecture	Date	Topic	Comments
1	9/24/2015	**Introduction**	HW1 posted
2	9/29/2015	Linear Regression	
3	10/1/2015	Other Regression methods	
4	10/6/2015	**Classification**	HW1 due - HW2 posted
5	10/8/2015	Artificial Neural Networks	**Project Topics**
6	10/13/2015	Artificial Neural Networks	
7	10/15/2015	Support Vector Machines	**Projects Assigned**
8	10/20/2015	Support Vector Machines	
9	10/22/2015	Support Vector Machines	HW2 due - HW3 posted
10	10/27/2015	**Midterm**	
11	10/29/2015	Classification issues: Kernels, Overfitting, Regularization	
12	11/3/2015	Dimensionality Reduction	
13	11/5/2015	Reinforcement Learning	
14	11/10/2015	Decision support: Markov Decision Processes	
15	11/12/2015	Graphical Models - Naïve Bayes	
16	11/17/2015	**Clustering**: K-means - Hierarchical	HW3 due - HW4 posted
17	11/19/2015	Special topics: Deep Learning	
18	11/24/2015	Project Presentation I	**Project Reports Due**
19	11/26/2015	**NO CLASS** (Thanksgiving)	HW4 due
20	12/1/2015	Project Presentation II	
21	12/3/2015	Project Presentation III - Overview	

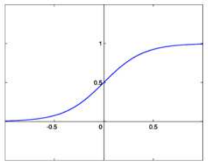
Building Boolean Functions



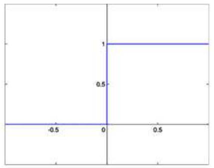
Picking the right weights, function and thresholds (whenever needed), we can represent any Boolean function.

For simplicity assume here that we use the step function.

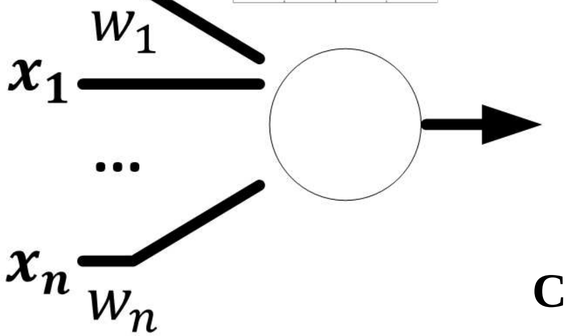
Building Boolean Functions



Sigmoid



Step function



predicted y: $\boldsymbol{g(x;w)}$

Picking the right weights, function and thresholds (whenever needed), we can represent any Boolean function.

For simplicity assume here that we use the step function.

$$g(x;w) = \begin{cases} 0 & if w^T x < 0 \\ 1 & if w^T x \geq 0 \end{cases}$$

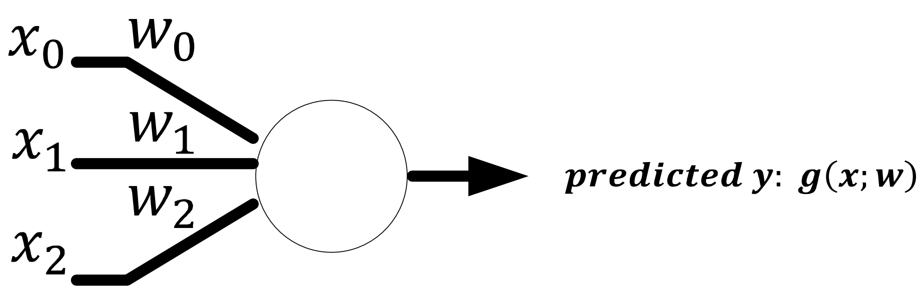
Case 1: Construct $g(x;w) = x_1\overline{x_2}$

$$w = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \dots \\ w_n \end{bmatrix} = \begin{bmatrix} 0.5 \\ 1 \\ -1 \\ \dots \\ 0 \end{bmatrix}$$

Note: this solution is not unique, e.g.

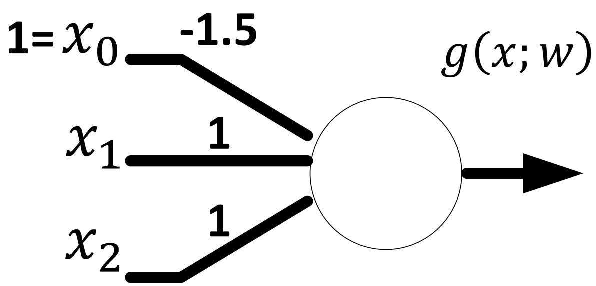
$$w = \begin{bmatrix} 105 \\ 200 \\ -100 \\ \dots \\ 0 \end{bmatrix}$$

Building Boolean Functions



Case 2: Construct $g(x;w) = x_1\ AND\ x_2$

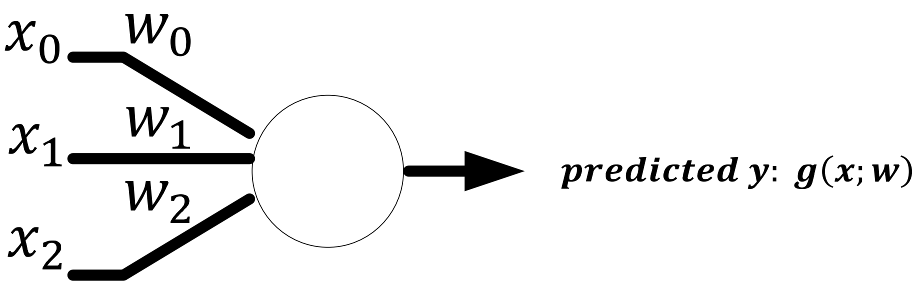
x_1	x_2	$y = g(x;w)$
0	0	0
0	1	0
1	0	0
1	1	1



$g(x;w) = -1.5 + x_1 + x_2$

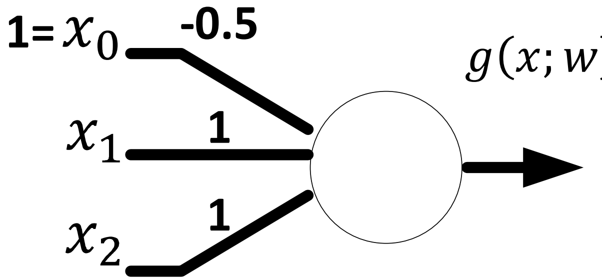
$$w = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} -1.5 \\ 1 \\ 1 \end{bmatrix}$$

Building Boolean Functions



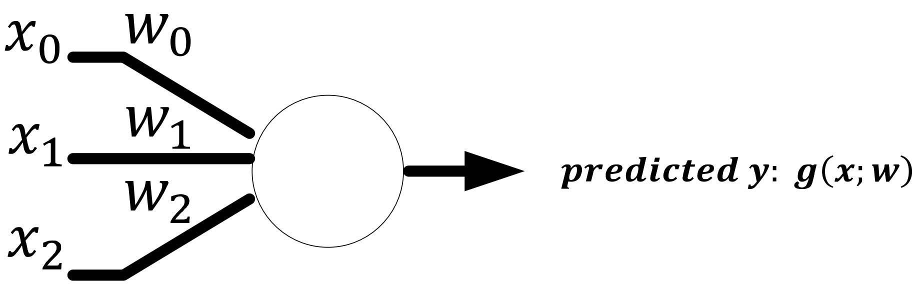
Case 3: Construct $g(x; w) = x_1\ OR\ x_2$

x_1	x_2	$y = g(x; w)$
0	0	0
0	1	1
1	0	1
1	1	1



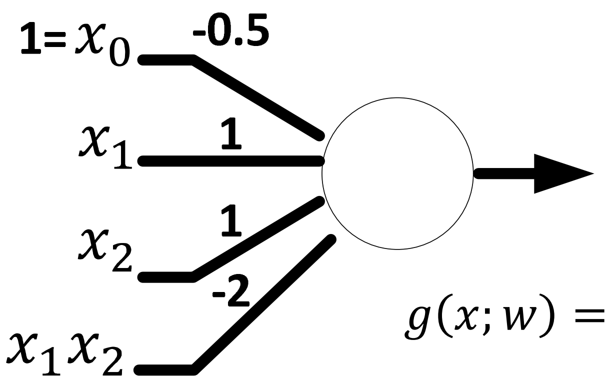
$$g(x; w) = -0.5 + x_1 + x_2$$

$$w = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} -0.5 \\ 1 \\ 1 \end{bmatrix}$$



Case 3: Construct $g(x; w) = x_1\ XOR\ x_2$

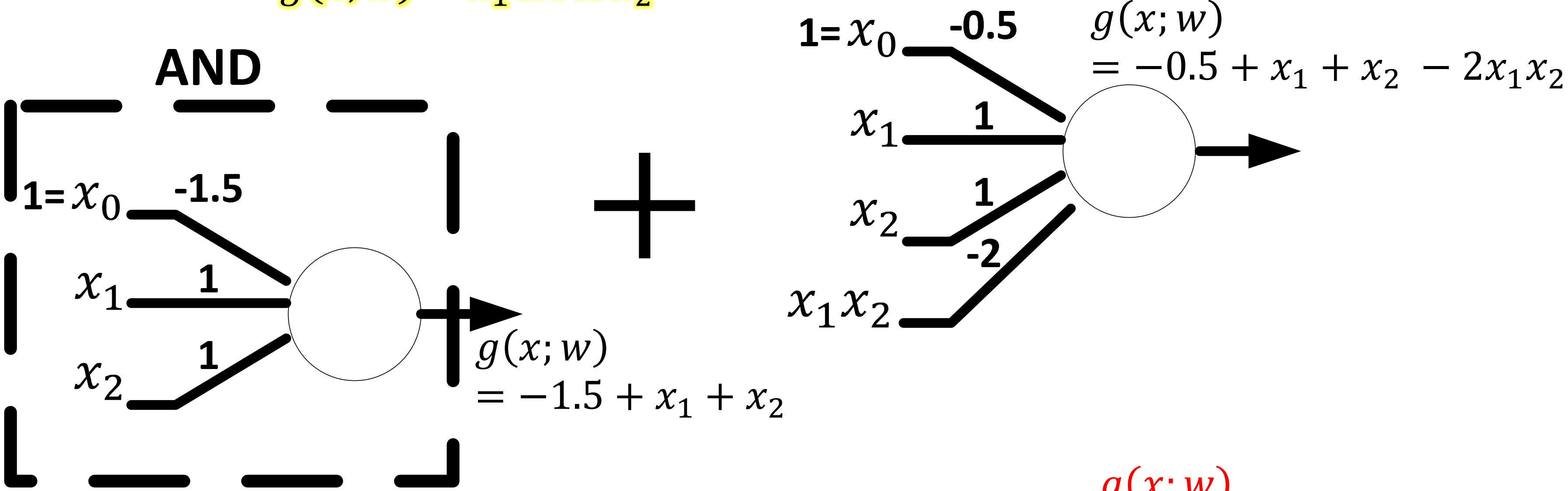
x_1	x_2	$y = g(x; w)$
0	0	0
0	1	1
1	0	1
1	1	0



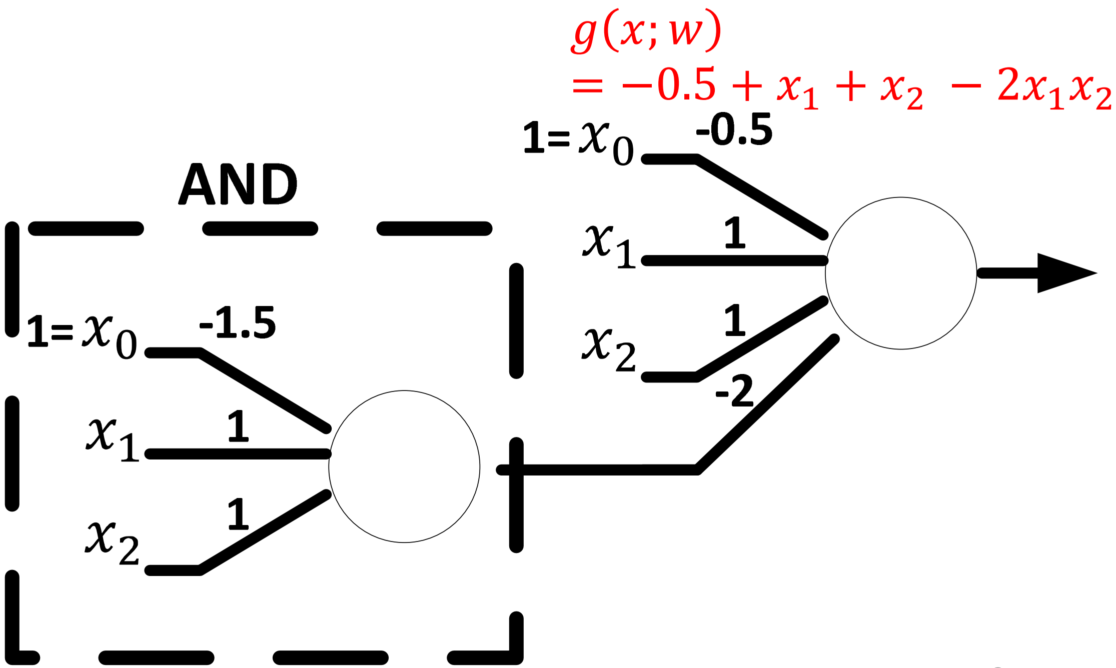
$$g(x; w) = -0.5 + x_1 + x_2 - 2x_1x_2$$

3-layer ANN encoding for an XOR

Case 3: Construct $g(x;w) = x_1\ XOR\ x_2$

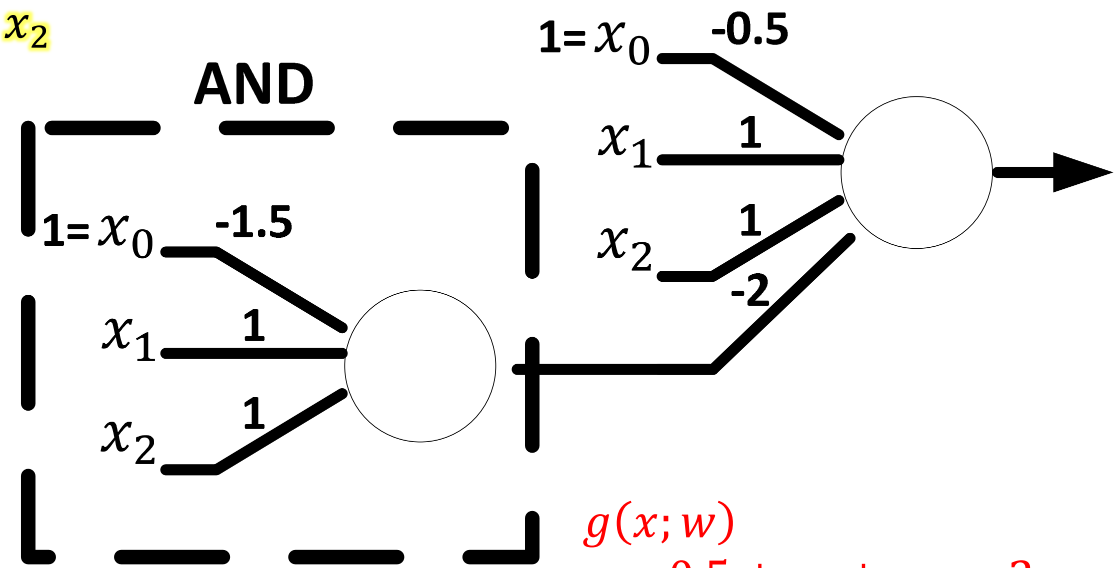


x_1	x_2	$y = g(x;w)$
0	0	0
0	1	1
1	0	1
1	1	0



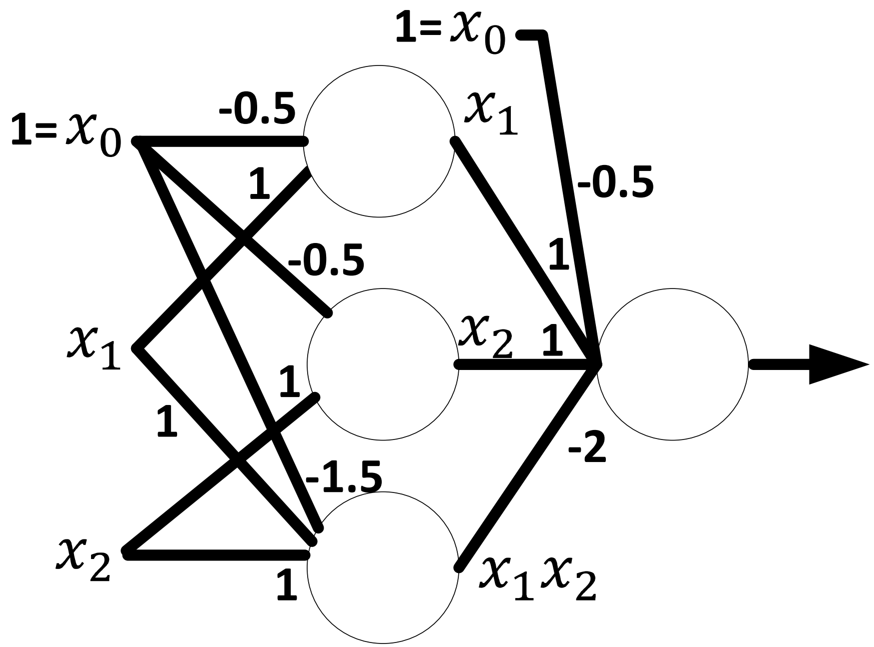
3-layer ANN encoding for an XOR

Case 3: Construct $g(x; w) = x_1\ XOR\ x_2$



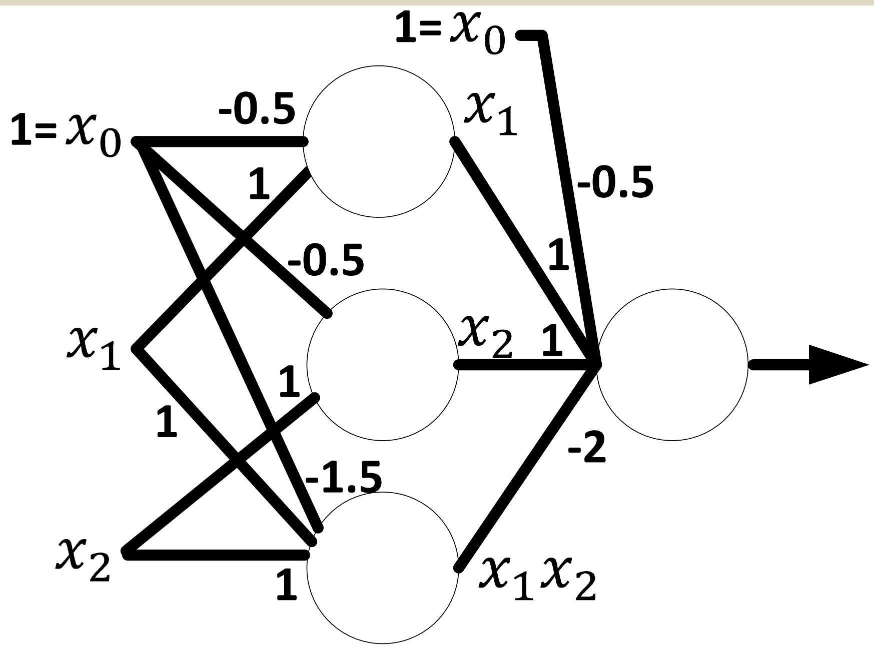
$g(x; w)$
$= -0.5 + x_1 + x_2 - 2x_1x_2$

x_1	x_2	$y = g(x; w)$
0	0	0
0	1	1
1	0	1
1	1	0



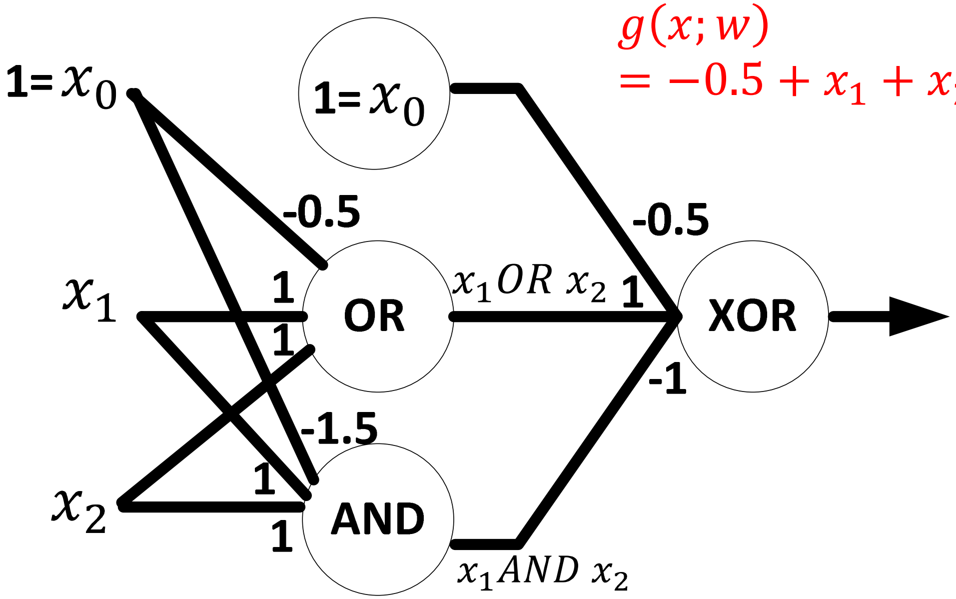
3-layer ANN encoding for an XOR

Case 3: Construct $g(x;w) = x_1\ XOR\ x_2$



x_1	x_2	$y = g(x;w)$
0	0	0
0	1	1
1	0	1
1	1	0

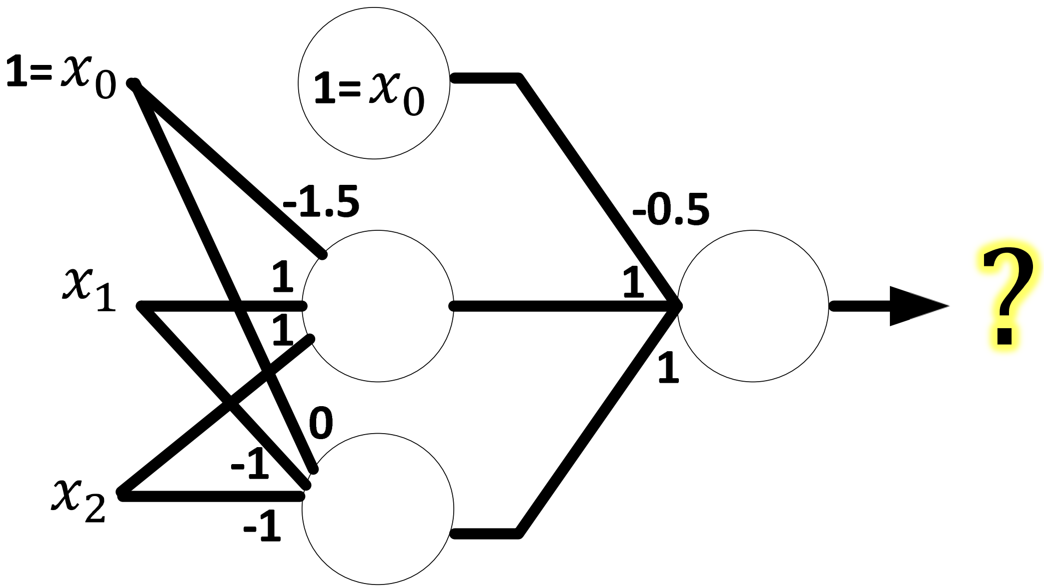
$g(x;w)$
$= -0.5 + x_1 + x_2\ - 2x_1x_2$



3-layer ANN encoding for an ?

Case 4: What is this gate? $g(x;w) = x_1 \,?\, x_2$

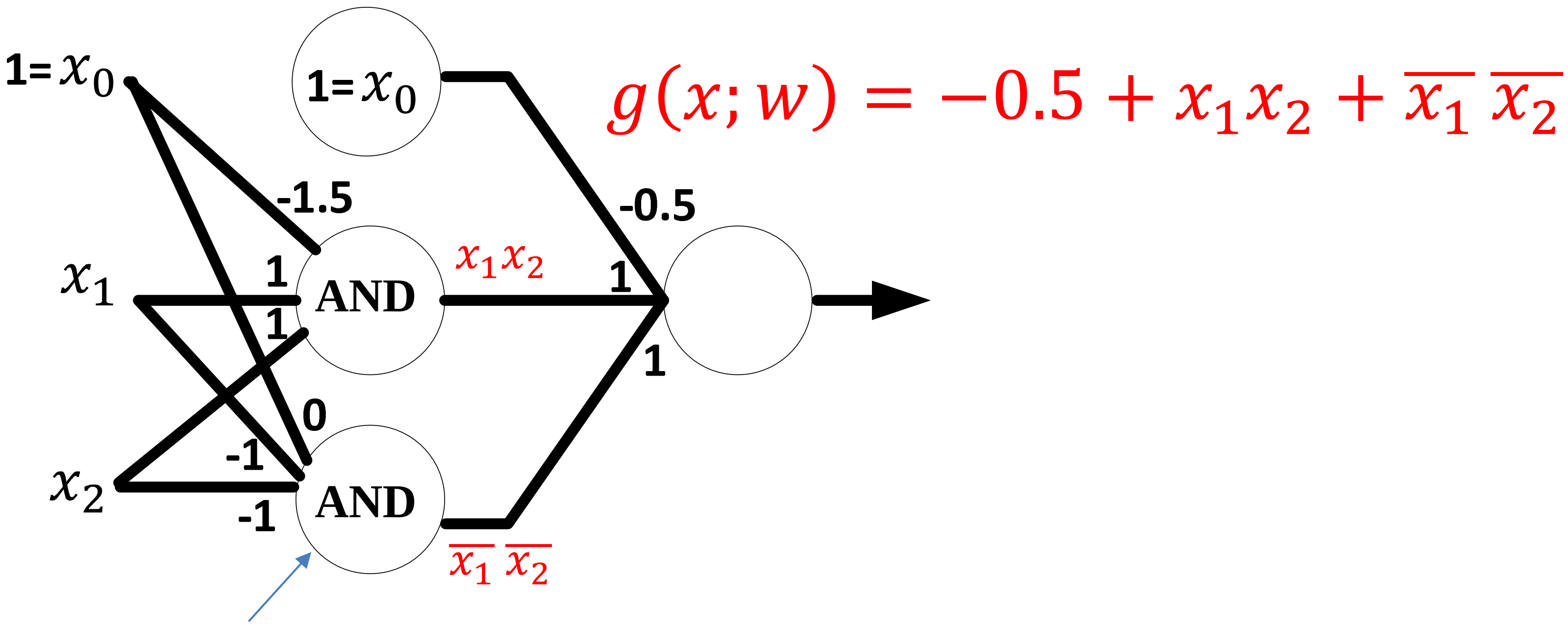
x_1	x_2	$y = g(x;w)$
0	0	?
0	1	?
1	0	?
1	1	?



3-layer ANN encoding for an XOR

Case 4: What is this gate? $g(x; w) = x_1\, XNOR\, x_2 = NOT(x_1\, XOR\, x_2)$

x_1	x_2	$y = g(x; w)$
0	0	1
0	1	0
1	0	0
1	1	1

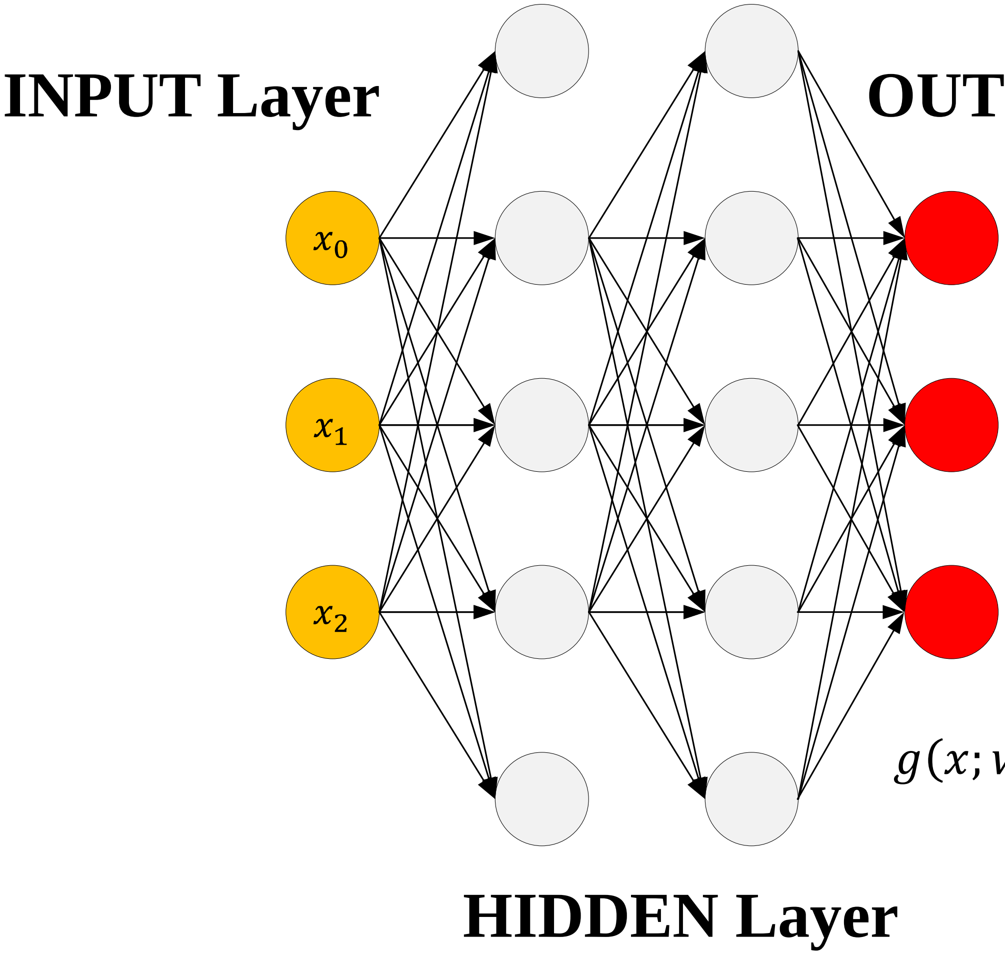


Actually a NOR, not an AND, on the original inputs

Multi-class ANN

Multi-class classification

- We can train the ANN to classify samples into multiple classes (mutual exclusive or not)
- For example: classify images to{bird, chimpanzee, human}



$g_1 = Pr(Bird)$

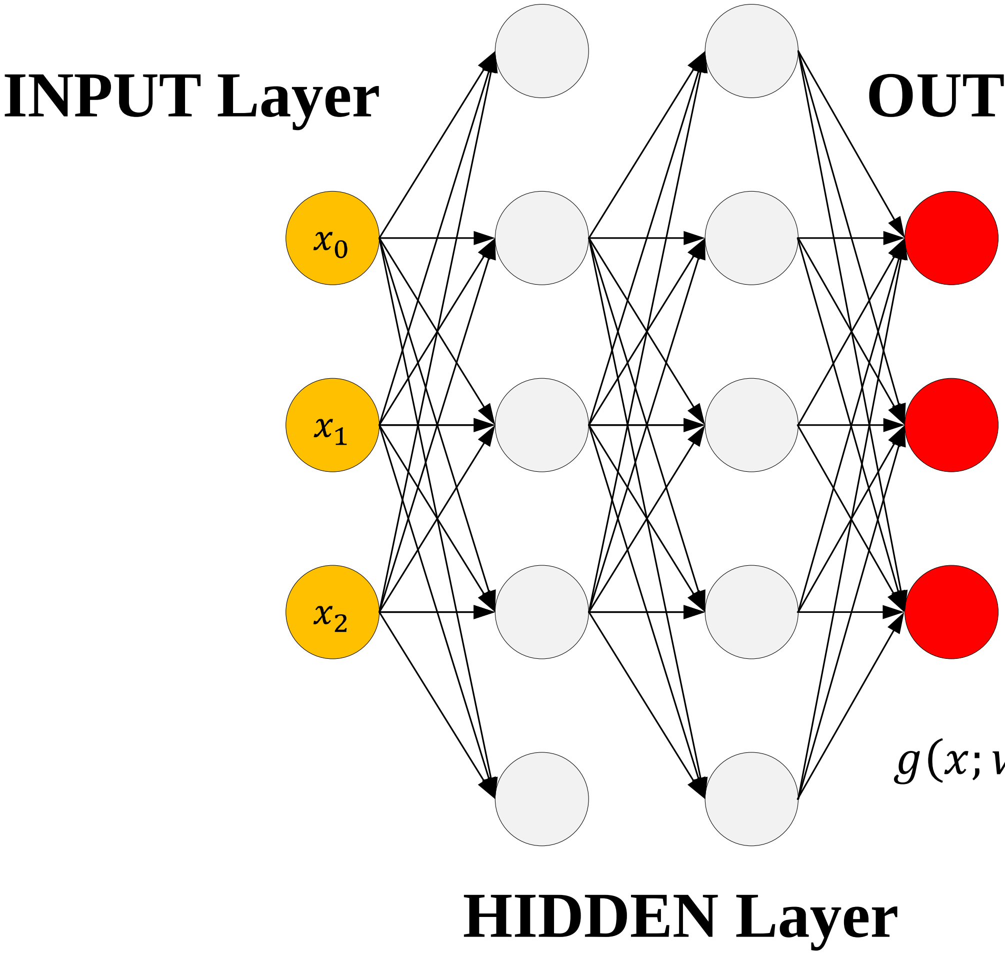
$g_2 = Pr(Chimpanzee)$

$g_3 = Pr(Human)$

$$g(x; w) = \begin{bmatrix} Pr(Bird) \\ Pr(Chimpanzee) \\ Pr(human) \end{bmatrix}$$

Multi-class classification

- We can train the ANN to classify samples into multiple classes (mutual exclusive or not)
- For example: classify images to{bird, chimpanzee, human}



$g_1 = Pr(Bird)$

$g_2 = Pr(Chimpanzee)$

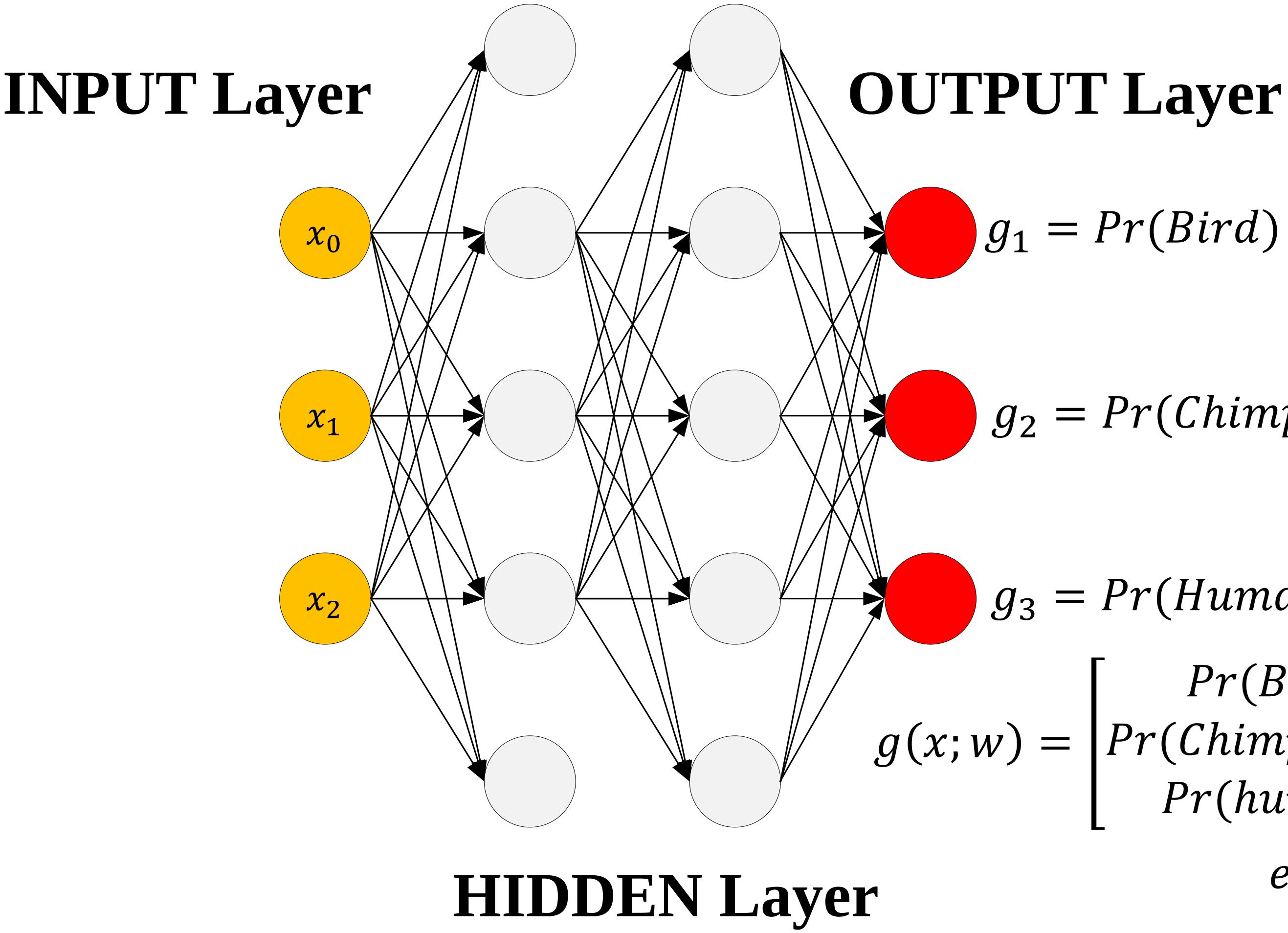
$g_3 = Pr(Human)$

$$g(x; w) = \begin{bmatrix} Pr(Bird) \\ Pr(Chimpanzee) \\ Pr(human) \end{bmatrix}$$

$$example\text{: } y = g(x; w) = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

Multi-class classification

- We can train the ANN to classify samples into multiple classes (mutual exclusive or not)
- For example: classify images to{bird, chimpanzee, human}



$g_1 = Pr(Bird)$

$g_2 = Pr(Chimpanzee)$

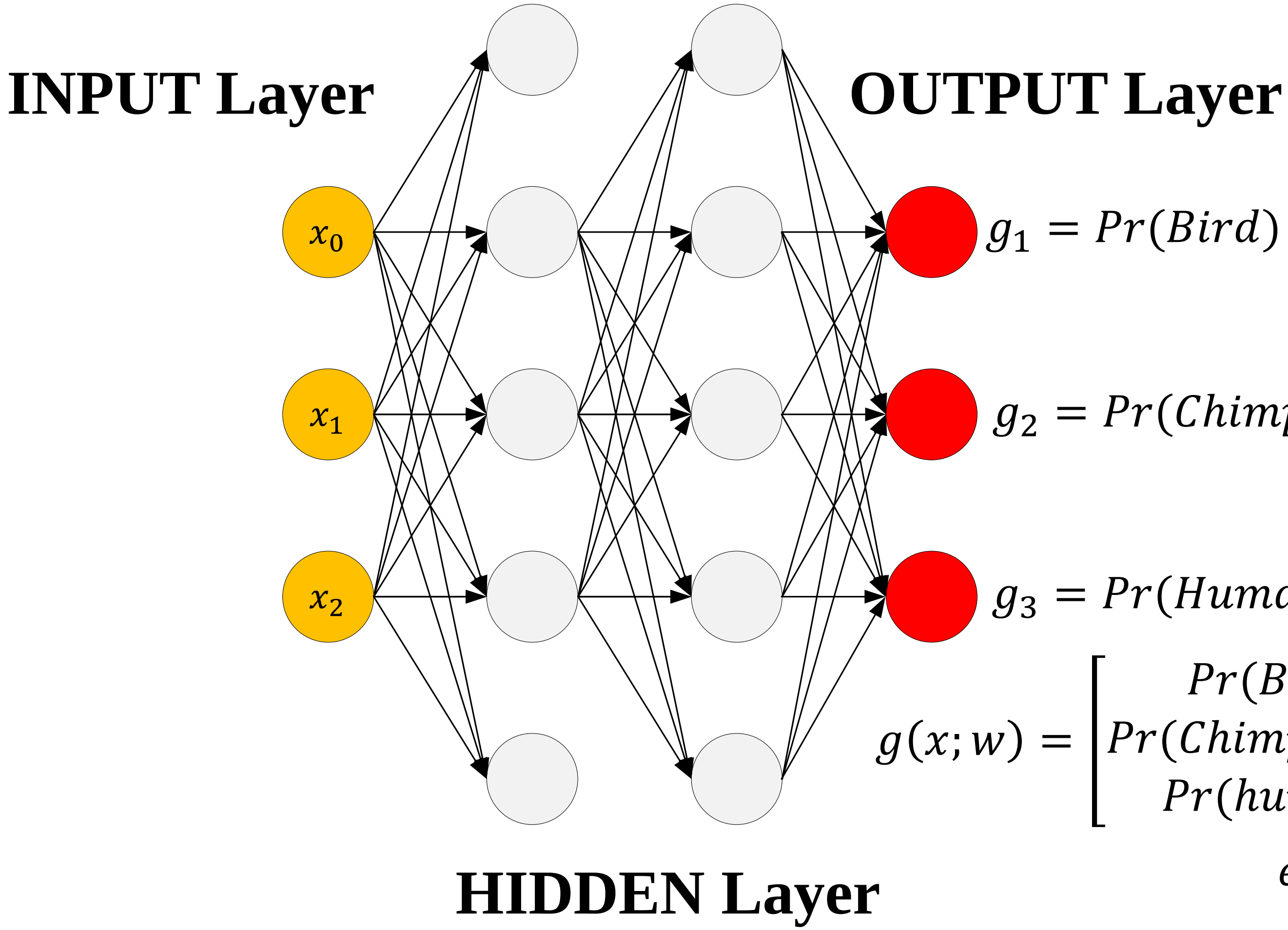
$g_3 = Pr(Human)$

$$g(x; w) = \begin{bmatrix} Pr(Bird) \\ Pr(Chimpanzee) \\ Pr(human) \end{bmatrix}$$

$$example: \ y = \ g(x; w) = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

Multi-class classification

- We can train the ANN to classify samples into multiple classes (mutual exclusive or not)
- For example: classify images to{bird, chimpanzee, human}



$g_1 = Pr(Bird)$

$g_2 = Pr(Chimpanzee)$

$g_3 = Pr(Human)$

$$g(x;w) = \begin{bmatrix} Pr(Bird) \\ Pr(Chimpanzee) \\ Pr(human) \end{bmatrix}$$

$$example\colon\ y = g(x;w) = \begin{bmatrix} 0 \\ ? \\ 1 \end{bmatrix}$$

ANN training and backpropagation

Training the parameters

- If we use logistic regression as the activation function, then that the log likelihood for one NN node is (lecture 4, slide 11):

$$l(\mathrm{w}) \triangleq log p(D|w) = \sum_{i=1}^{M} \left(y^{(i)} \log g\left(x^{(i)}; w\right) + (1 - y^{(i)}) \log(1 - g(x^{(i)}; w))\right)$$

- So if we have K nodes then the log likelihood becomes:

$$l(\mathrm{w}) = \sum_{k=1}^{K} \sum_{i=1}^{M} \left(y^{(i)}{}_{k} \log g\left(x^{(i)}; w\right)_{k} + (1 - y^{(i)}{}_{k}) \log\left(1 - g\left(x^{(i)}; w\right)_{k}\right)\right)$$

- Similarly as before (logistic regression) we want to maximize the log likelihood
  - or conversely minimize the negative log likelihood, that can be thought as representing the cost function.

Back propagation

- To train the parameters, we can use any optimization method, for example **Gradient Descent** (or "ascent" in the case of the log-likelihood):

$$w_{ij} := w_{ij} + a\frac{\partial l(w)}{\partial w_{ij}}$$

with

$$l(\mathrm{w}) = \sum_{k=1}^{K}\sum_{i=1}^{M}\left(y^{(i)}{}_k \log g\left(x^{(i)}; w\right)_k + (1 - y^{(i)}{}_k)\log\left(1 - g\left(x^{(i)}; w\right)_k\right)\right)$$

If we take the derivatives with respect to w, we see that:

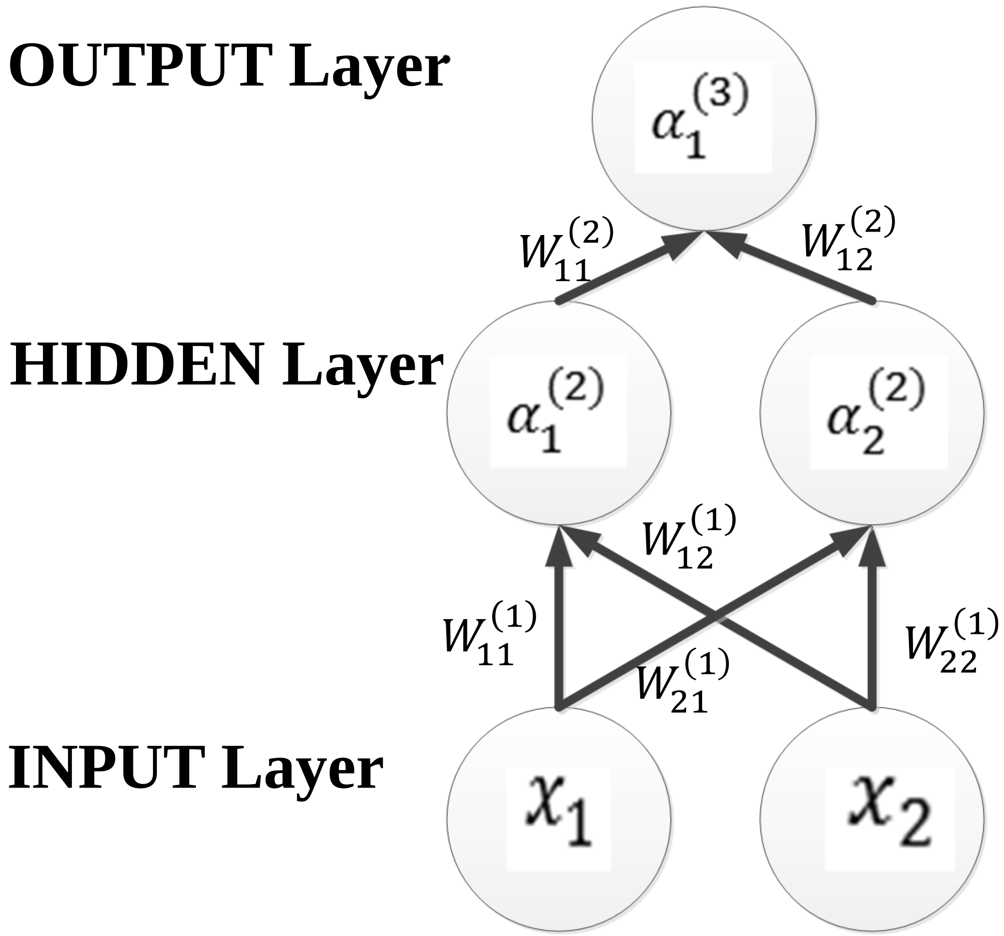
$$\frac{\partial l(W)}{\partial w_{ij}{}^{(l-1)}} = -\boldsymbol{a}_j^{(\boldsymbol{l}-1)}\boldsymbol{\delta}_{\boldsymbol{i}}^{(\boldsymbol{l})}$$

Where $\delta_i^{(l)}$ is the error of the i^{th} neuron (node) in the l^{th} layer.

Some definitions

- To calculate the error, lets go through some definitions first:

$$\alpha_i^{(l)} = \text{Activation of unit i in layer l}$$
$$W^{(l)} = \text{weight matrix from layer l to l} + 1$$



$$\alpha_1^{(3)} = g(W_{10}^{(2)}x_0 + W_{11}^{(2)}\alpha_1^{(2)} + W_{12}^{(2)}\alpha_2^{(2)})$$

$$\alpha_1^{(2)} = g(W_{10}^{(1)}x_0 + W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2)$$
$$\alpha_2^{(2)} = g(W_{20}^{(1)}x_0 + W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2)$$

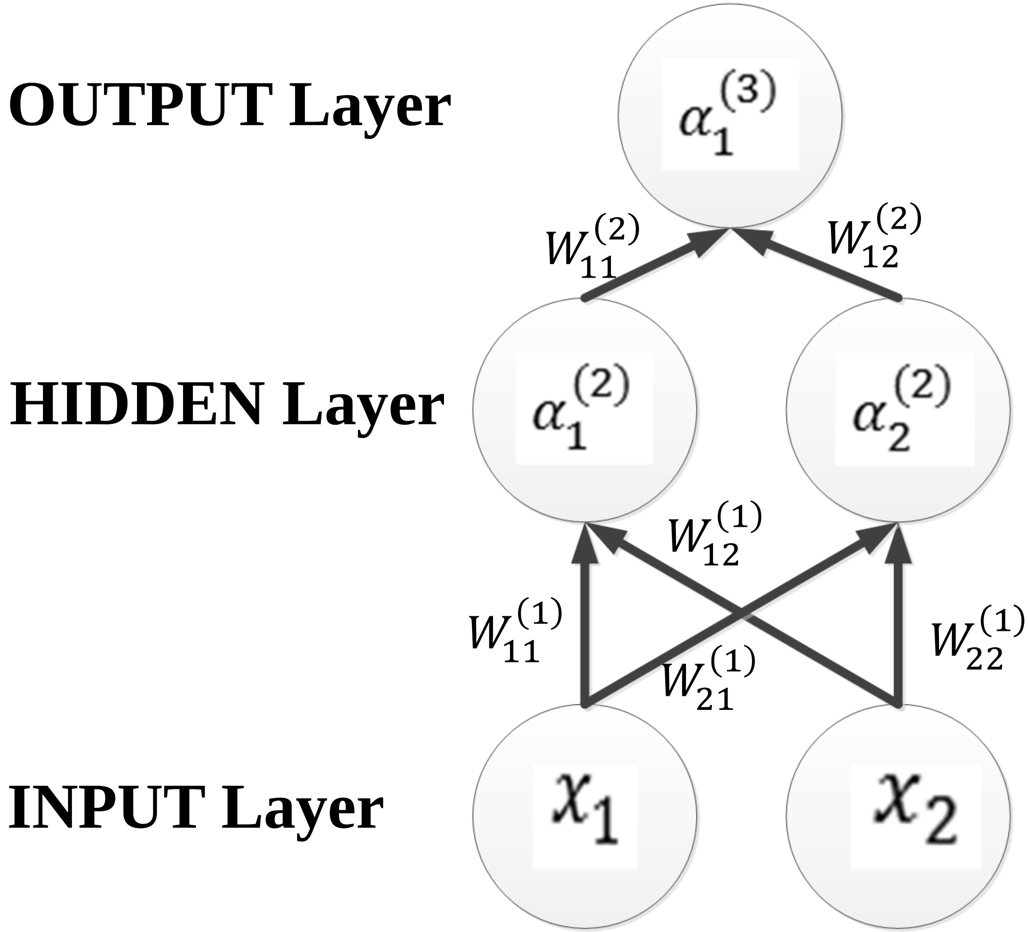
You can represent the sum in the () as ***z*** (since you didn't have enough variable definitions already…):

$$z_1^{(2)} = W_{10}^{(1)}x_0 + W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2$$

so that activation of unit 1 becomes:

$$\alpha_1^{(2)} = g(z_1^{(2)})$$

Artificial Neural Network



So output can be represented as

$$\boldsymbol{a}^{(l)} = \boldsymbol{g}(\boldsymbol{z}^{(l)})$$

With

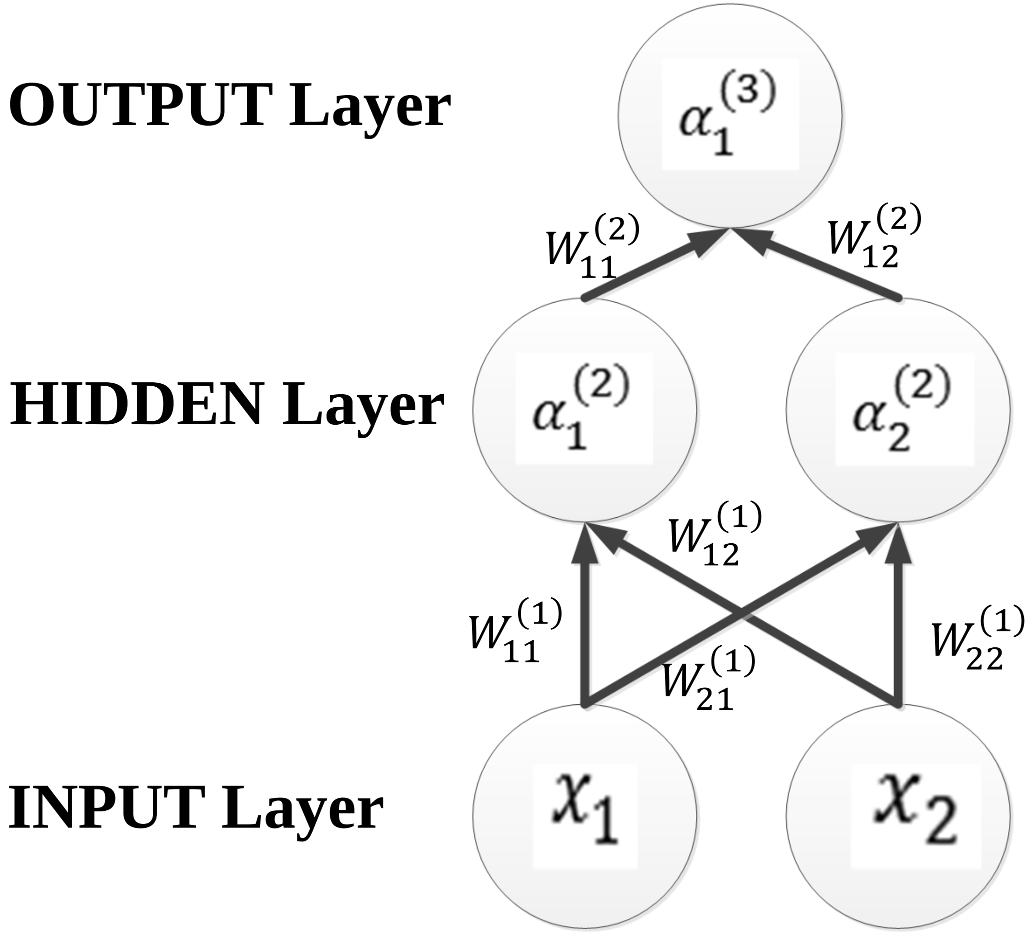
$$\boldsymbol{a}^{(l)} = \begin{bmatrix} a_0^{(l)} \\ a_1^{(l)} \\ a_2^{(l)} \end{bmatrix} \qquad \boldsymbol{z}^{(l)} = \begin{bmatrix} z_1^{(l)} \\ z_2^{(l)} \end{bmatrix}$$

Each $\boldsymbol{a}^{(l)}$ can be computed iteratively, starting from $\boldsymbol{l} = \mathbf{1}$ and ending at the output layer (here $\boldsymbol{l} = \mathbf{3}$). This is called forward-propagation.

The final output is given by:

$$\boldsymbol{a}^{(3)} = \boldsymbol{g}(\boldsymbol{z}^{(3)})$$

Back-propagation



Let the error of **node *i* in layer *l*** be

$$\delta_i^{(l)} = \alpha_i^{(l)} - y_i$$

Then we can **calculate iteratively the errors for each layer** as:

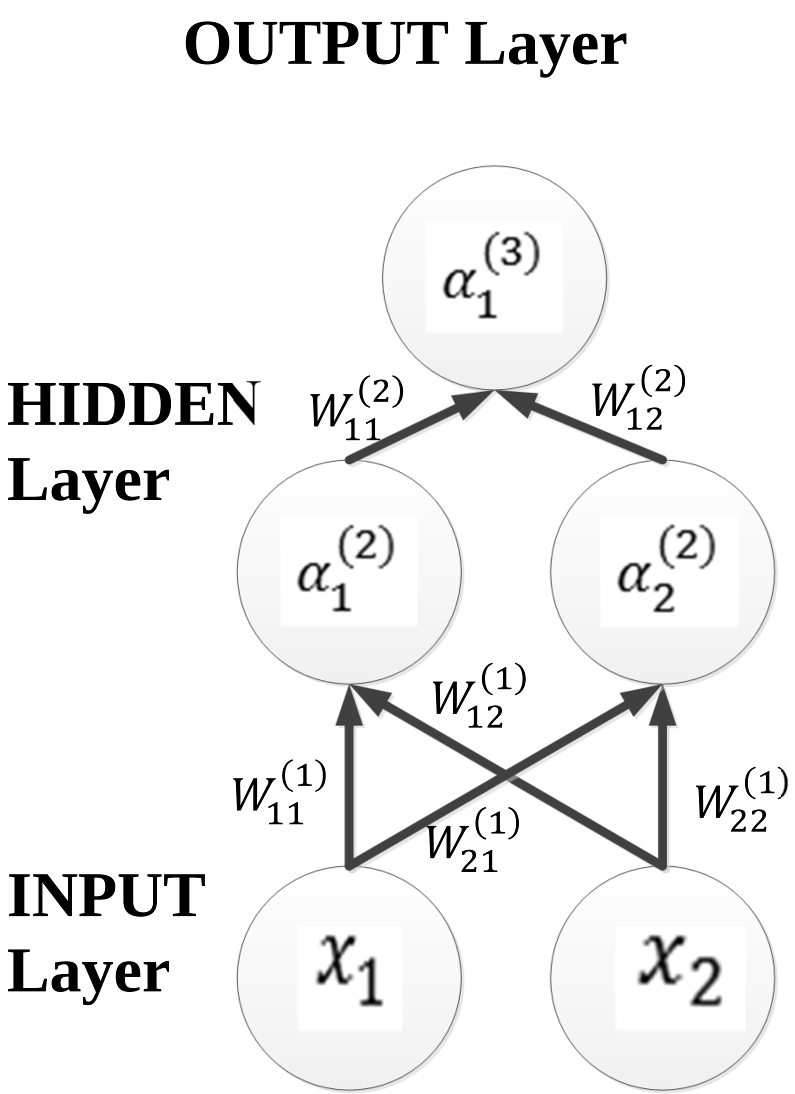
$$\delta_i^{(l-1)} = (W^{(l-1)})^T \delta^{(l)} \cdot a^{(l-1)} \cdot (1 - a^{(l-1)})$$

Where

$\delta^{(l)} = \begin{bmatrix} \delta_1^{(l)} \\ \dots \\ \delta_n^{(l)} \end{bmatrix}$ is the vector of errors

for the n nodes of the l^{th} layer.

Back-propagation



It turns out that the partial derivatives of the log-likelihood with respect to the weights $\boldsymbol{w}$ are given by:

$$\frac{\partial l(W)}{\partial w_{ij}{}^{(l-1)}} = -\boldsymbol{a}_j^{(l-1)} \boldsymbol{\delta}_i^{(l)}$$

Which you now can calculate recursively:

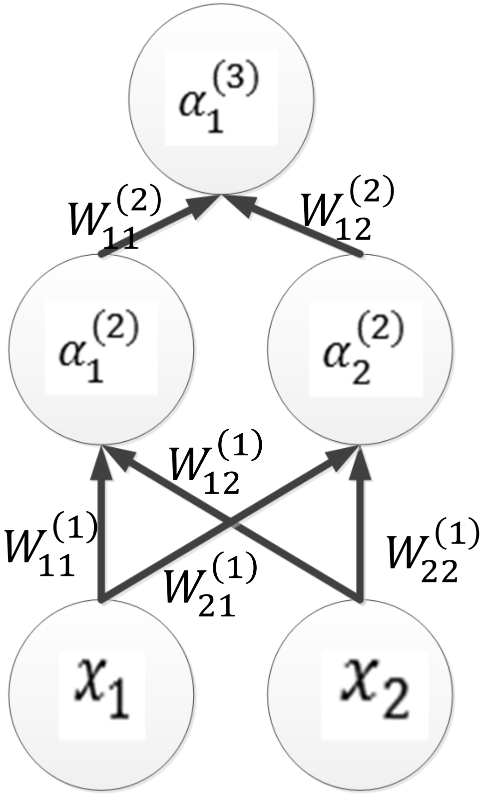
Element-by-element multiplication

$$\boldsymbol{\delta}^{(l-1)} = \left(\boldsymbol{W}^{(l-1)}\right)^T \boldsymbol{\delta}^{(l)} \cdot \boldsymbol{a}^{(l-1)} \cdot \left(\mathbf{1} - \boldsymbol{a}^{(l-1)}\right)$$

Why you need the partial derivatives?

- Gradient Descent or other optimization algorithms

Back-propagation



For example, for our 3-layer network on the left we have:

$$\boldsymbol{\delta}^{(3)} = \boldsymbol{\delta}_1^{(3)} = \boldsymbol{\alpha}_1^{(l)} - \boldsymbol{y}_1$$

$$\boldsymbol{W}^{(2)} = \begin{bmatrix} \boldsymbol{w}_{10}^{(2)} \\ \boldsymbol{w}_{11}^{(2)} \\ \boldsymbol{w}_{12}^{(2)} \end{bmatrix} \quad \boldsymbol{W}^{(1)} = \begin{bmatrix} \boldsymbol{w}_{10}^{(1)} \\ \boldsymbol{w}_{11}^{(1)} \\ \boldsymbol{w}_{12}^{(1)} \end{bmatrix} \quad \boldsymbol{a}^{(2)} = \begin{bmatrix} \mathbf{1} \\ \boldsymbol{a}_1^{(2)} \\ \boldsymbol{a}_2^{(2)} \end{bmatrix}$$

So that for $\boldsymbol{\delta}^{(2)}$ we have (following the equation from the previous slide:

$$\boldsymbol{\delta}^{(2)} = \begin{bmatrix} \boldsymbol{\delta}_0^{(2)} \\ \boldsymbol{\delta}_1^{(2)} \\ \boldsymbol{\delta}_2^{(l)} \end{bmatrix} = \begin{bmatrix} \boldsymbol{w}_{10}^{(2)} \\ \boldsymbol{w}_{11}^{(2)} \\ \boldsymbol{w}_{12}^{(2)} \end{bmatrix}^{\boldsymbol{T}} (\boldsymbol{\alpha}_i^{(l)} - \boldsymbol{y}_i)$$

Note: this is a vector/matrix representation. Follow the derivation that we did in class instead.

Slide 25

i1 iliast, 10/28/2014

Backpropagation of a FF-ANN

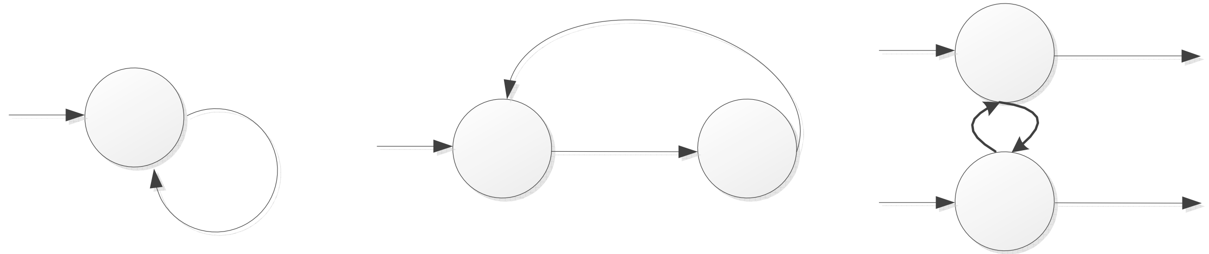
For each training sample $\left(x^{(k)}, y^{(k)}\right)$:

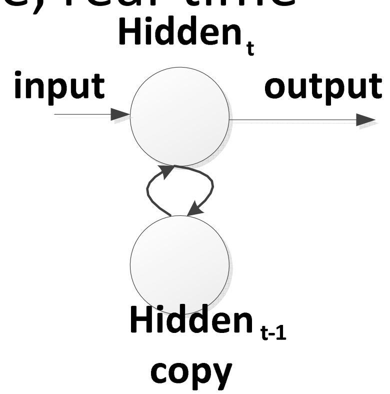
- **Set $a^{(1)} = x^{(k)}$**
- **Compute $a^{(l)}$ for all layers l (forward propagation)**
- **Compute the error in final layer $\delta_j^{(L)} = a_i^{(L)} - y^{(k)}$ and all hidden layers $\delta^{(l-1)} = (W^{(l-1)})^T \delta^{(l)} \cdot a^{(l-1)} \cdot \left(1 - a^{(l-1)}\right)$**
- **Compute partial derivatives $\frac{\partial l(W)}{\partial w_{ij}{}^{(l-1)}} = -a_j^{(l-1)} \delta_i^{(l)}$**
- **Use the derivatives to update with a heuristic optimization method:**

$$w_{ij}{}^{(l-1)} := w_{ij}{}^{(l-1)} - \alpha\left(a_j^{(l-1)} \delta_i^{(l)}\right)$$

Recurrent Neural Networks

- Neural Networks can be split in two main categories:
  - Feed-forward networks (what we have done so far), when the ANN graph is acyclic
  - Recurrent networks, when the ANN graph contains cycles (feedback)
- The are many other types of networks that can fall into these categories (e.g. Radial Basis Function Networks, Hopfield Networks, long-short memory etc.)
- How can we train Recurrent networks?
  - Elman network, Backpropagation through time, real-time recurrent learning, etc.





Recurrent Neural Networks

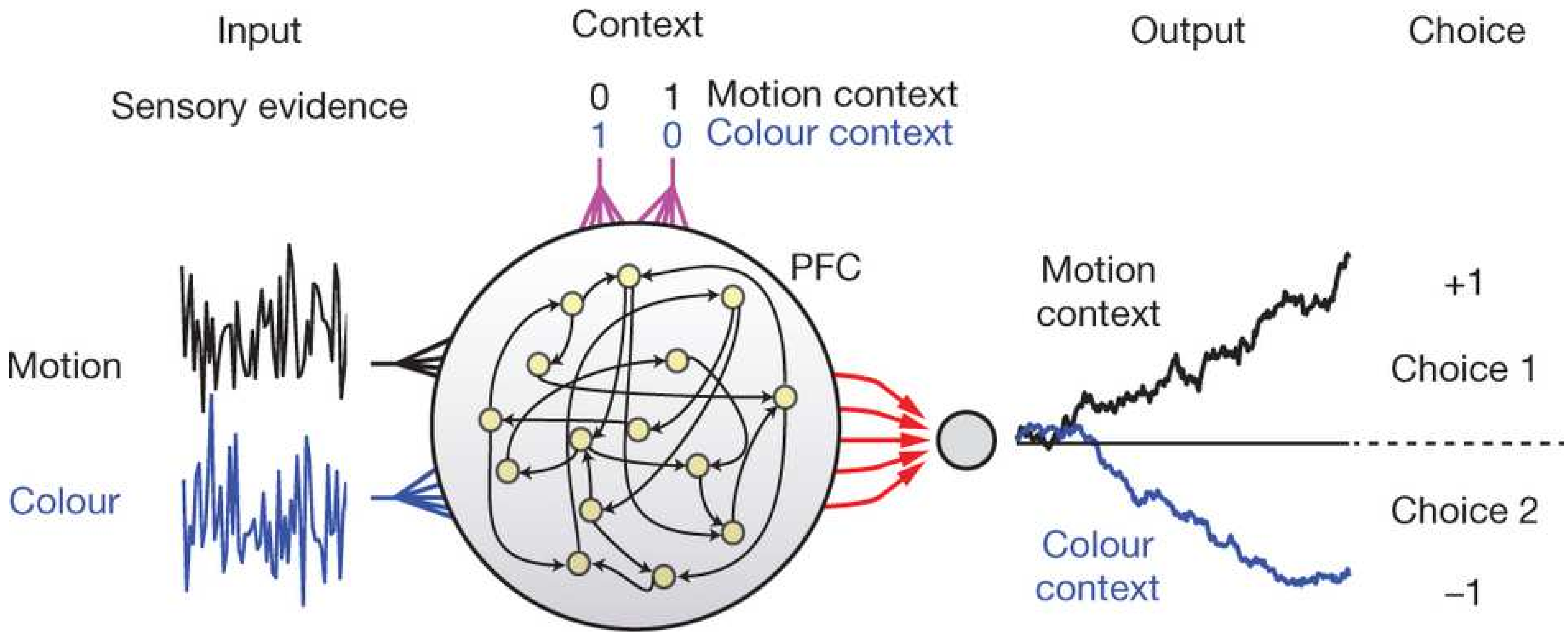


Figure 4 | A neural network model of input selection and integration. PFC is modelled as a network of recurrently connected, nonlinear, rate neurons that receive independent motion, colour and contextual inputs. The network is fully recurrently connected, and each unit receives both motion and colour inputs as well as two inputs that indicate context. At each time step, the sensory inputs are drawn from two normal distributions, the means of which correspond to the average strengths of the motion and colour evidence on a given trial. The contextual inputs take one of two values (0 or 1), which instruct the network to discriminate either the motion or the colour input. The network is read out by a single linear read-out, corresponding to a weighted sum over the responses of all neurons (red arrows). We trained the network (with back-propagation[35]) to make a binary choice, that is, to generate an output of +1 at the end of the stimulus presentation if the relevant evidence pointed towards choice 1, or a −1 if it pointed towards choice 2. Before training, all synaptic strengths were randomly initialized.

Context-dependent computation by recurrent dynamics in prefrontal cortex

Valerio Mante[1]†*, David Sussillo[2]*, Krishna V. Shenoy[2,3] & William T. Newsome[1]

78 | NATURE | VOL 503 | 7 NOVEMBER 2013

End of Lecture 6