50.007
Machine Learning

Lu, Wei

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

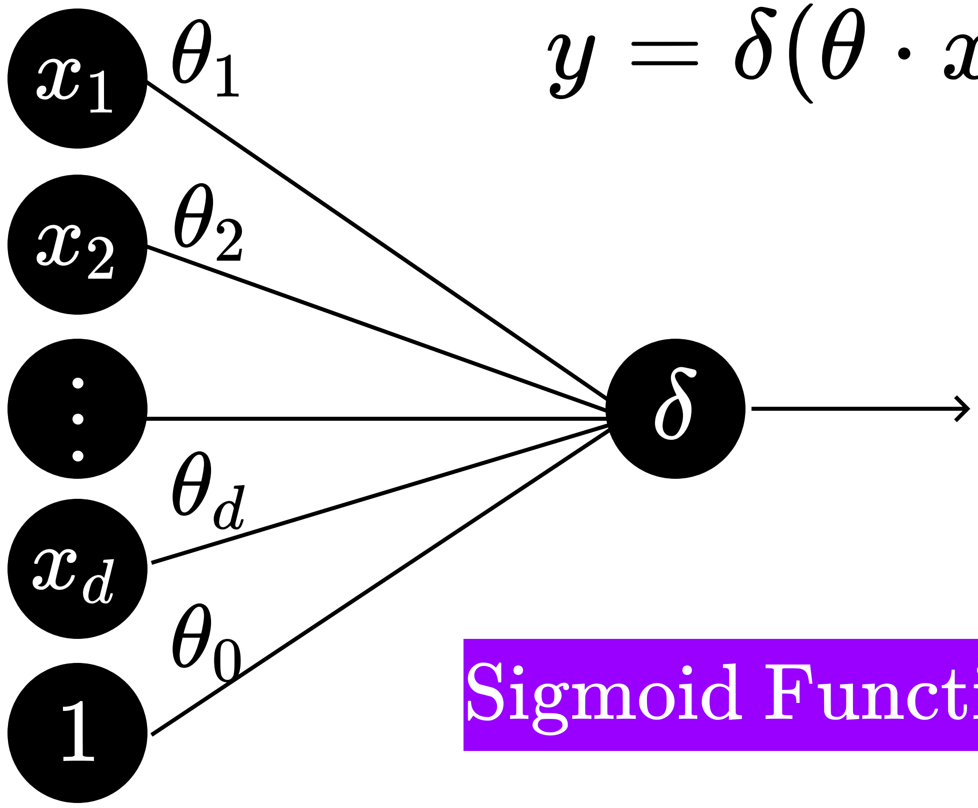
Neural Networks and Deep Learning

Logistic Regression

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}$$

$$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_d \end{bmatrix}$$

$$y = \delta(\theta \cdot x + \theta_0)$$



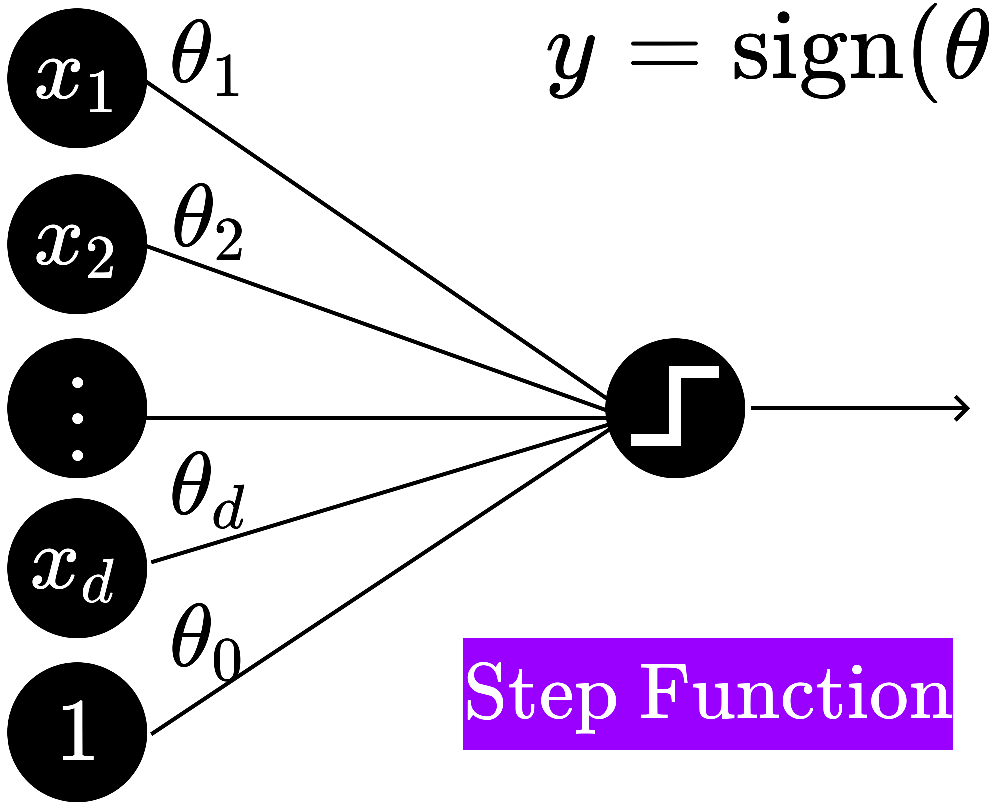
Sigmoid Function

Perceptron

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}$$

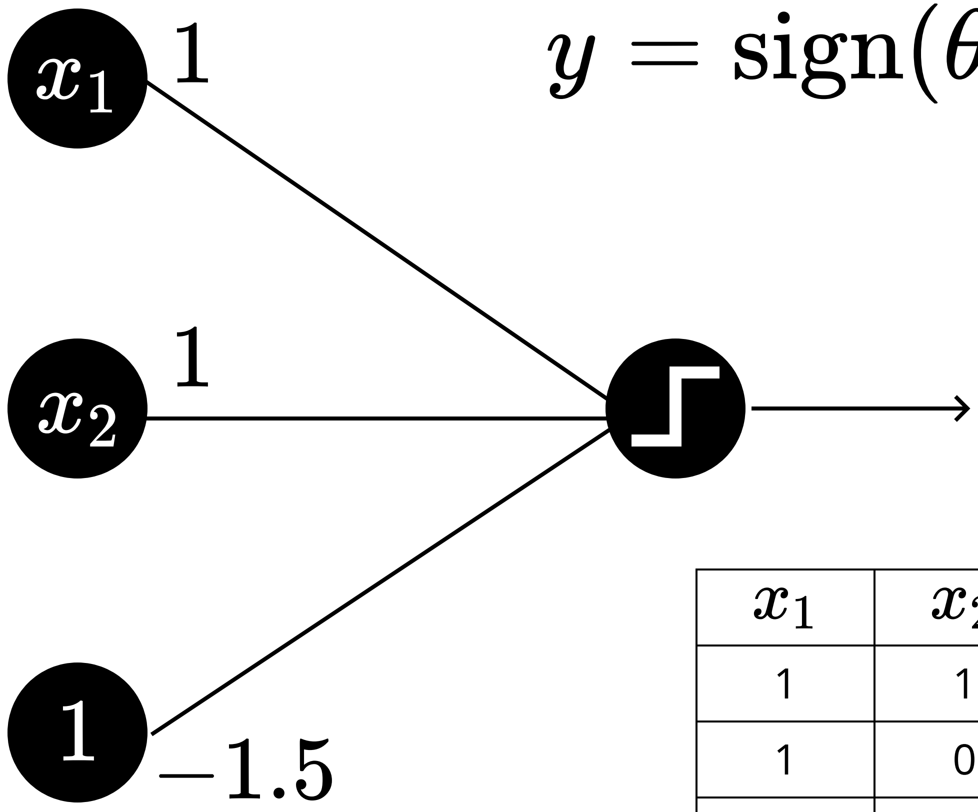
$$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_d \end{bmatrix}$$

$$y = \text{sign}(\theta \cdot x + \theta_0)$$



Step Function

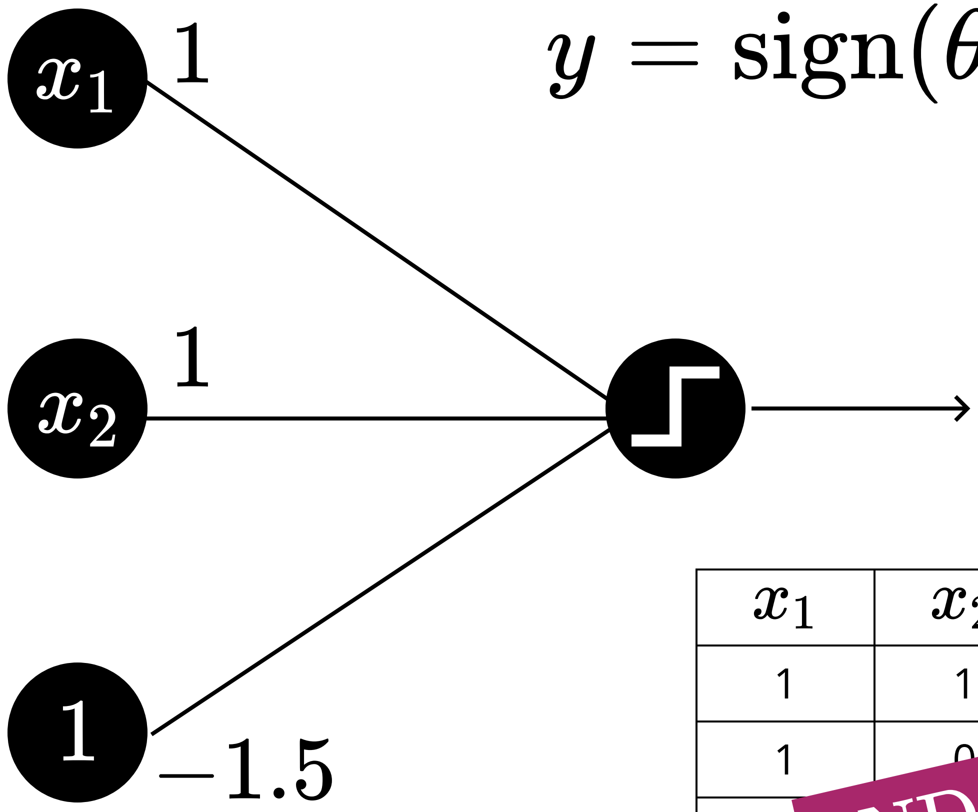
Perceptron



$$y = \text{sign}(\theta \cdot x + \theta_0)$$

x_1	x_2	$\theta \cdot x + \theta_0$	y
1	1	0.5	1
1	0	-0.5	-1
0	1	-0.5	-1
0	0	-1.5	-1

Perceptron

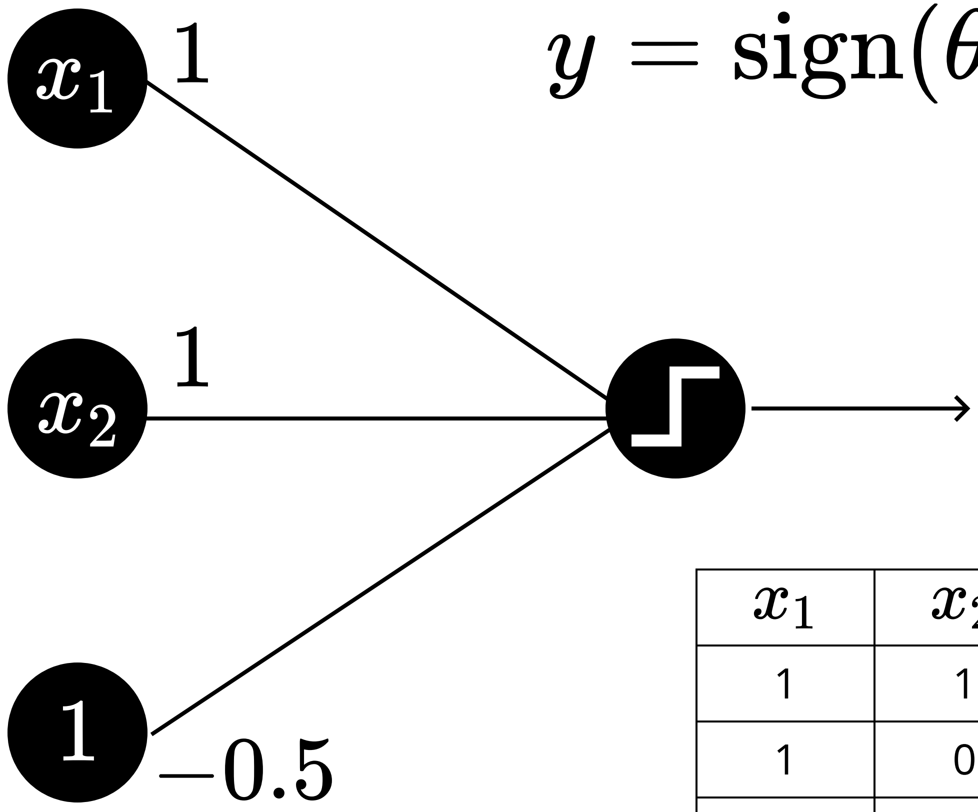


$$y = \operatorname{sign}(\theta \cdot x + \theta_0)$$

x_1	x_2	$\theta \cdot x + \theta_0$	y
1	1	0.5	1
1	0	[illegible]	-1
0	[illegible]	-0.5	-1
0	0	-1.5	-1

AND Function

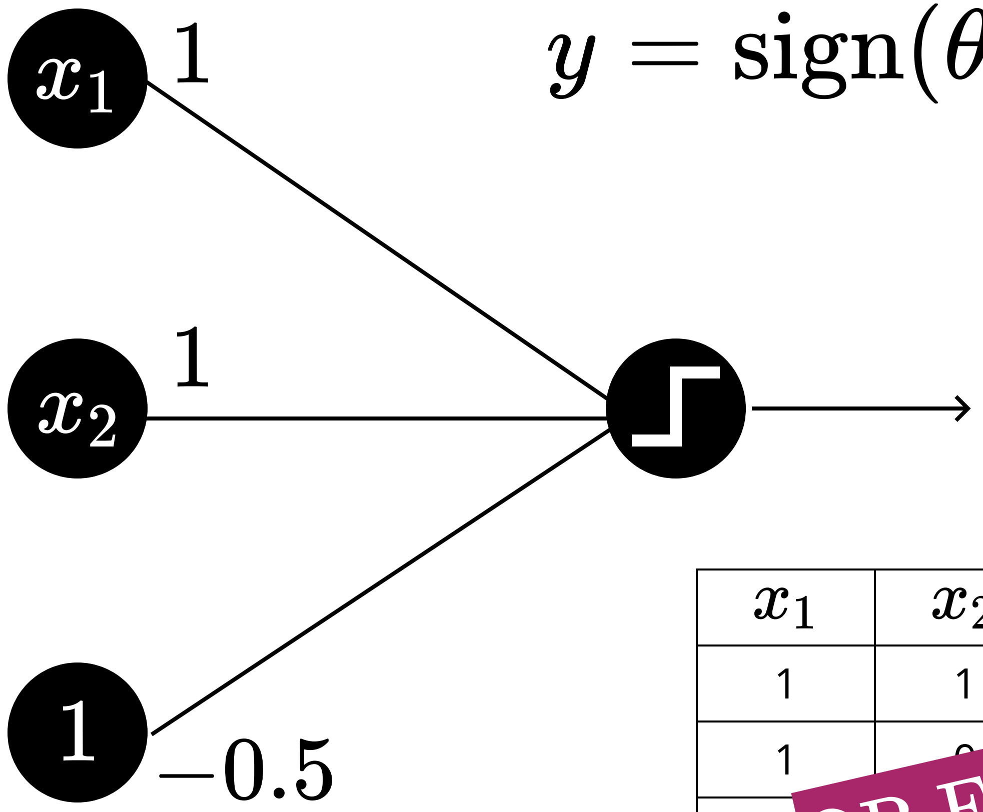
Perceptron



$$y = \text{sign}(\theta \cdot x + \theta_0)$$

x_1	x_2	$\theta \cdot x + \theta_0$	y
1	1	1.5	1
1	0	0.5	1
0	1	0.5	1
0	0	-0.5	-1

Perceptron

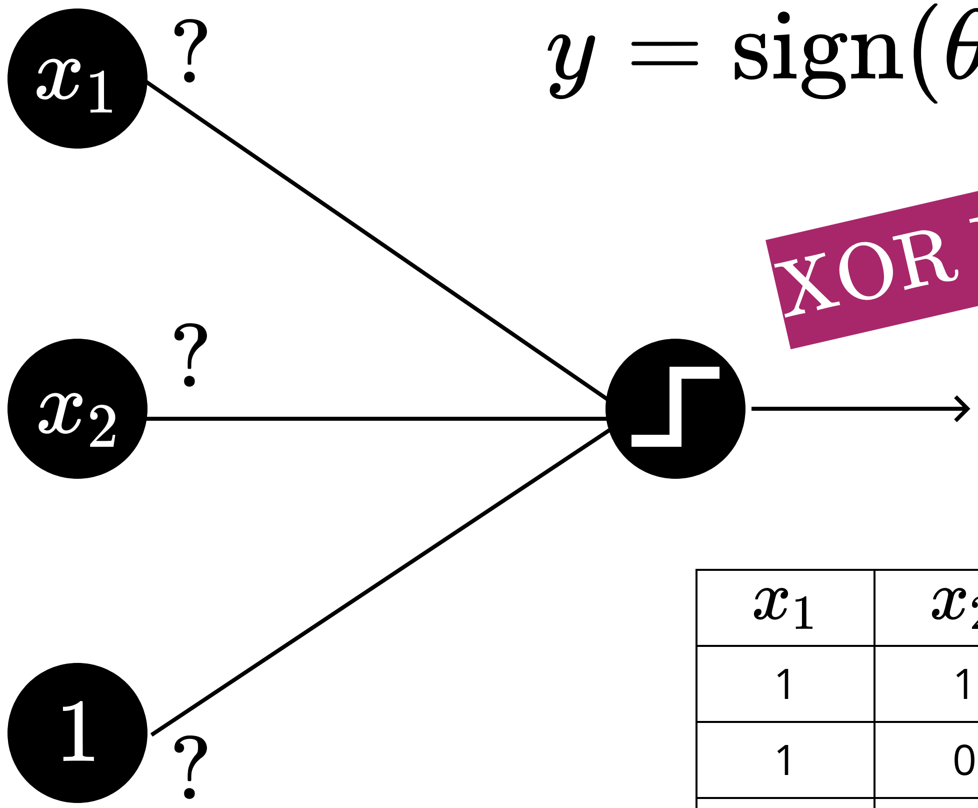


$$y = \text{sign}(\theta \cdot x + \theta_0)$$

x_1	x_2	$\theta \cdot x + \theta_0$	y
1	1	1.5	1
1	0	[illegible]	1
0	[illegible]	0.5	1
0	0	-0.5	-1

OR Function

Perceptron



$$y = \text{sign}(\theta \cdot x + \theta_0)$$

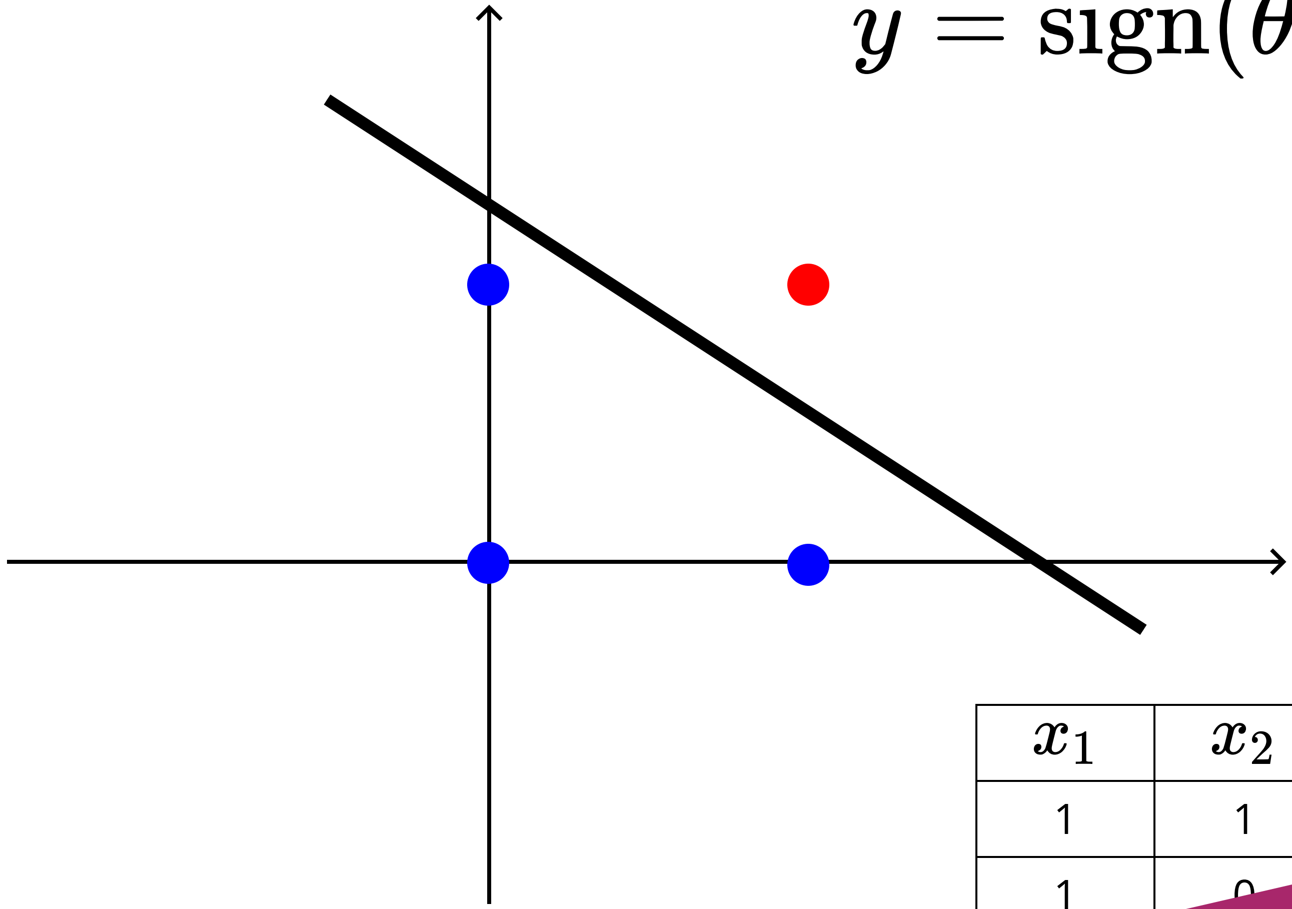
XOR Function

x_1	x_2	$\theta \cdot x + \theta_0$	y
1	1	?	-1
1	0	?	1
0	1	?	1
0	0	?	-1

Can you design the weights for the XOR function?

Perceptron

$$y = \text{sign}(\theta \cdot x + \theta_0)$$

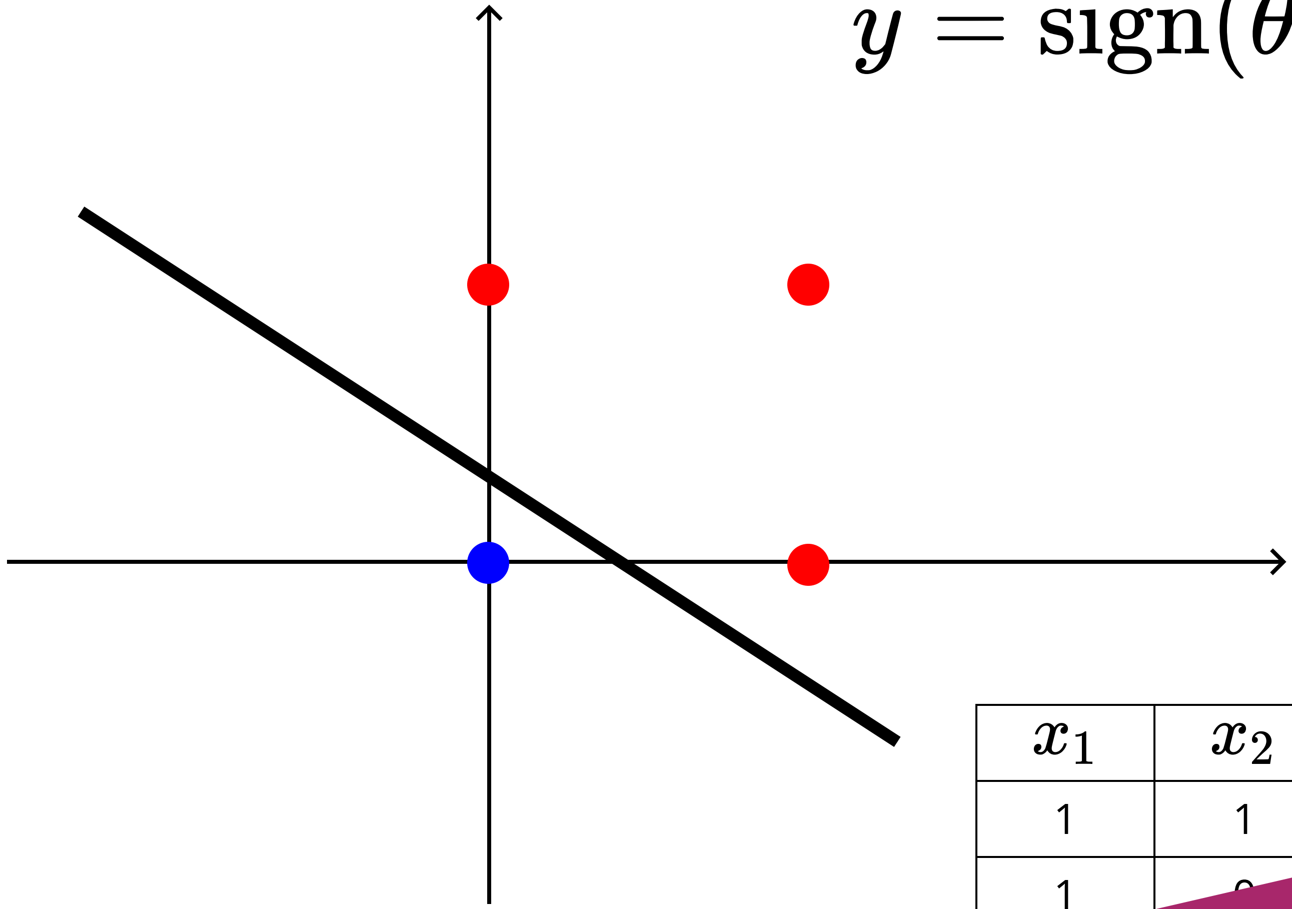


x_1	x_2	$\theta \cdot x + \theta_0$	y
1	1	0.5	1
1	0	[illegible]	-1
0	[illegible]	-0.5	-1
0	0	-1.5	-1

AND Function

Perceptron

$$y = \operatorname{sign}(\theta \cdot x + \theta_0)$$

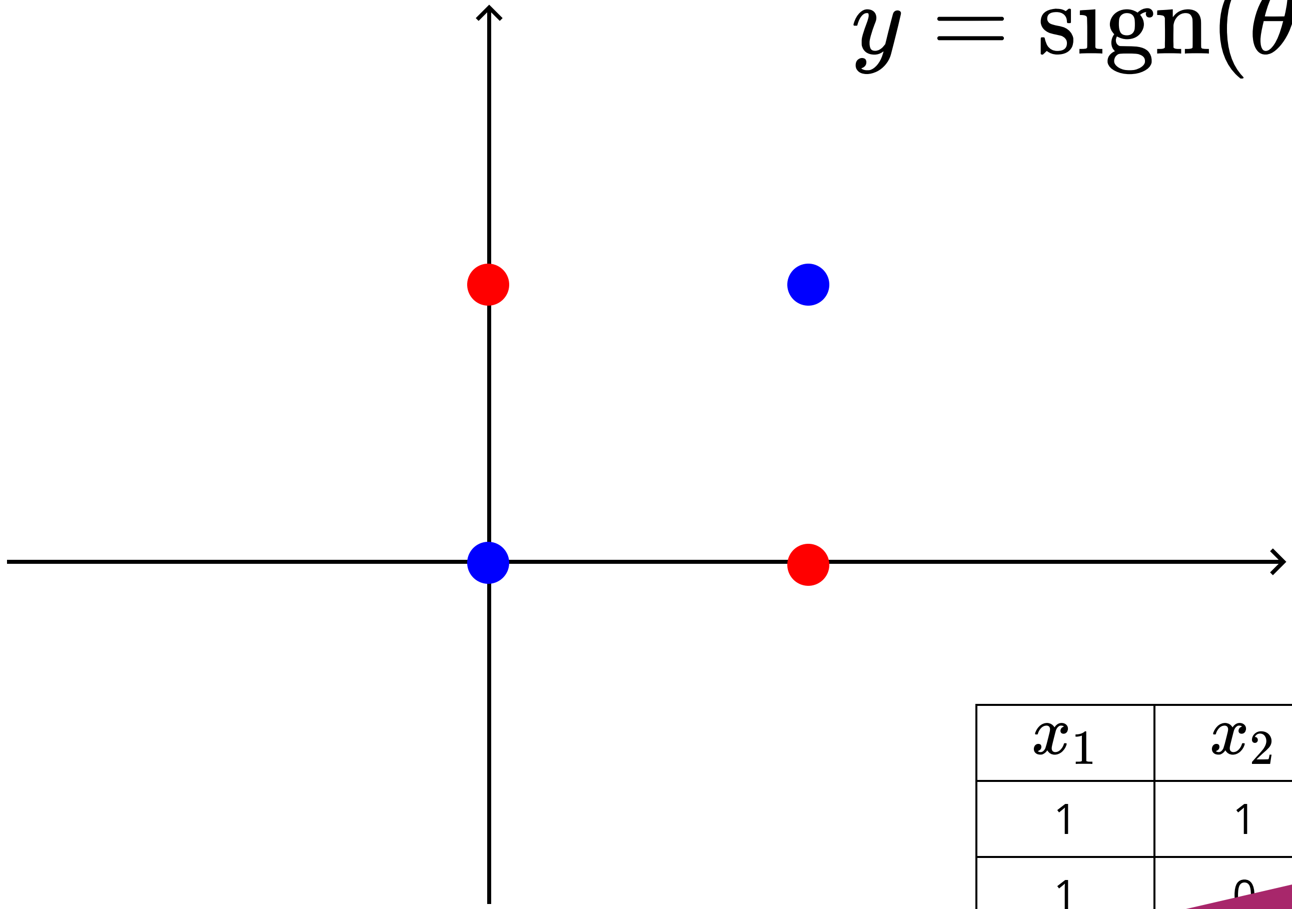


x_1	x_2	$\theta \cdot x + \theta_0$	y
1	1	1.5	1
1	0	[illegible]	1
0	[illegible]	0.5	1
0	0	-0.5	-1

OR Function

Perceptron

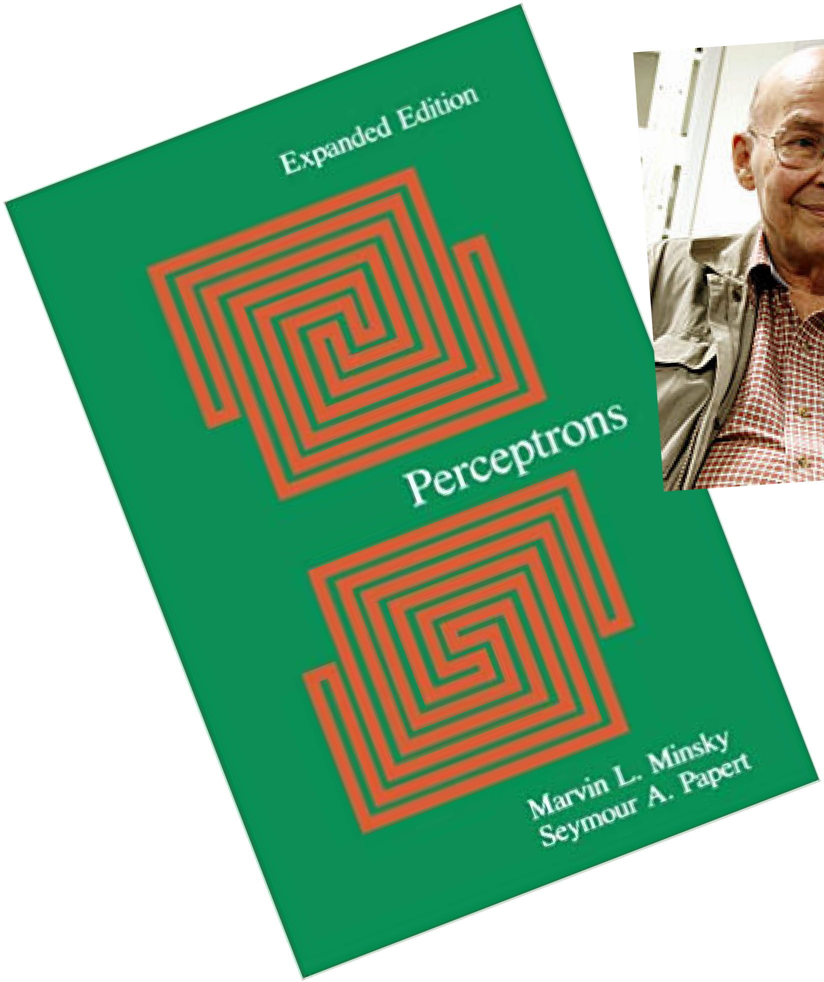
$$y = \text{sign}(\theta \cdot x + \theta_0)$$



x_1	x_2	$\theta \cdot x + \theta_0$	y
1	1	?	-1
1	0	[illegible]	1
0	[illegible]	?	1
0	0	?	-1

XOR Function

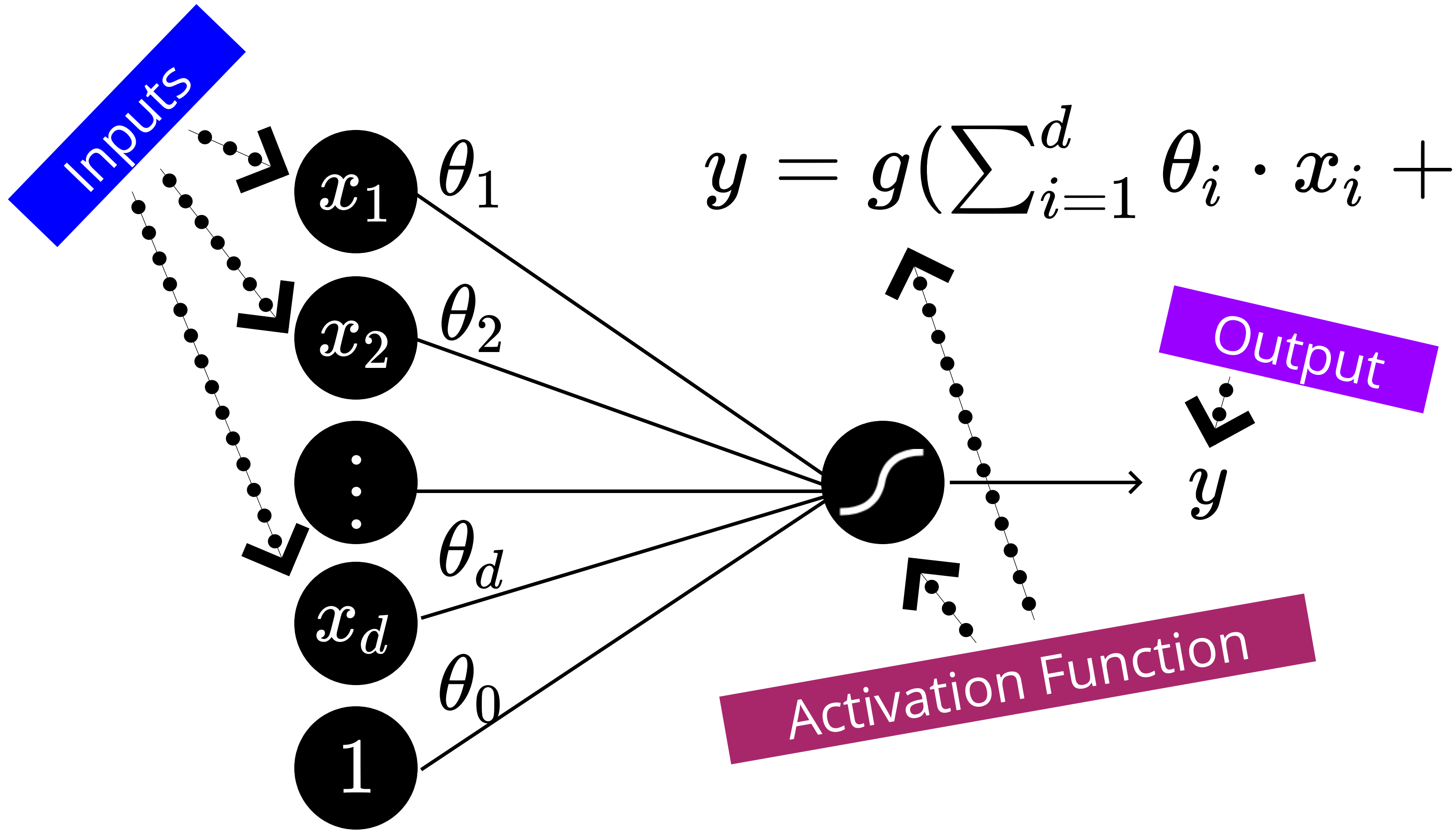
Perceptron



The book that attacks the Perceptron (and neural networks) -- it cannot even compute the XOR function

Neural Network

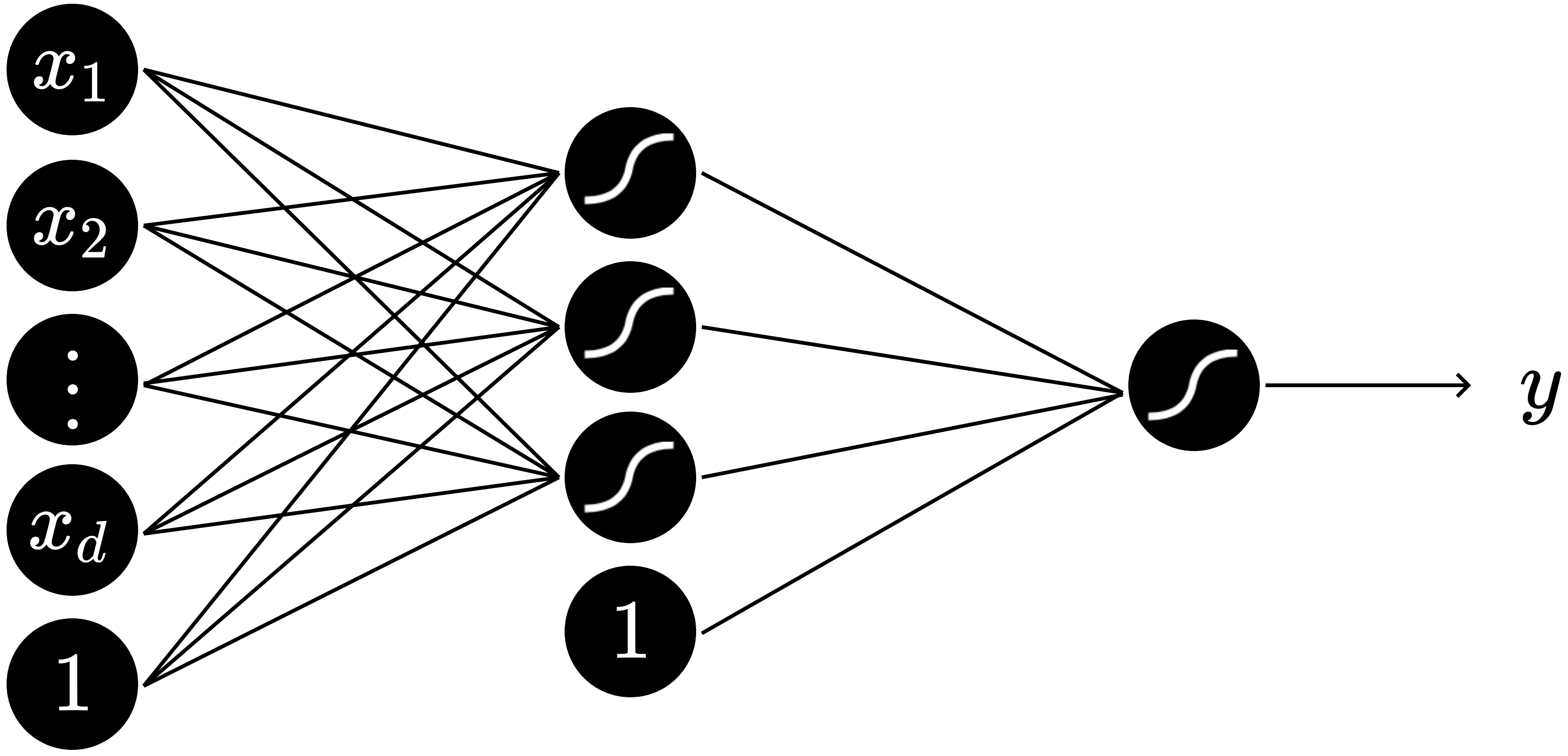
$$y = g\left(\sum_{i=1}^{d} \theta_i \cdot x_i + \theta_0\right)$$



Input Layer

Output Layer

Neural Network

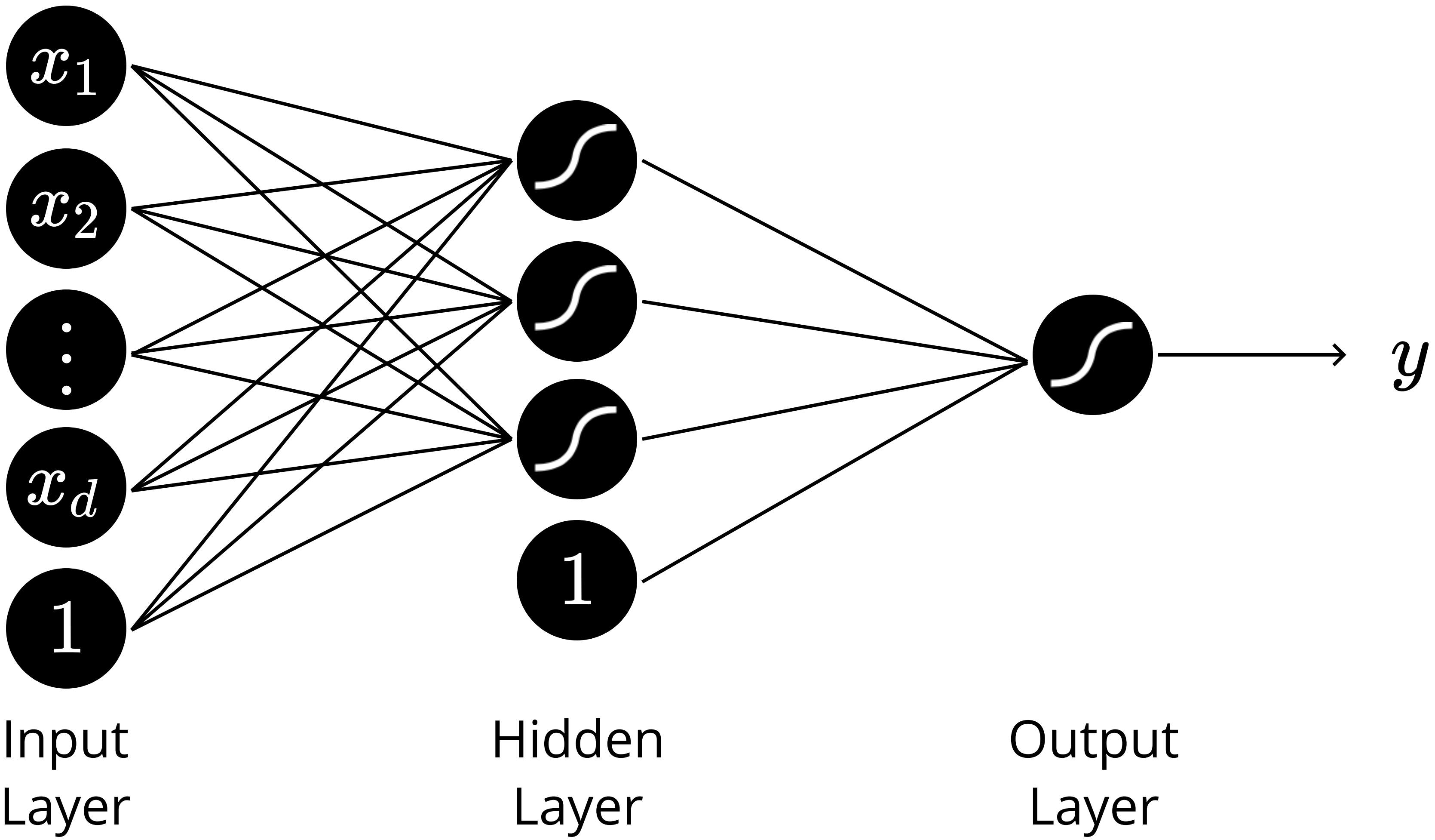


Input Layer

Hidden Layer

Output Layer

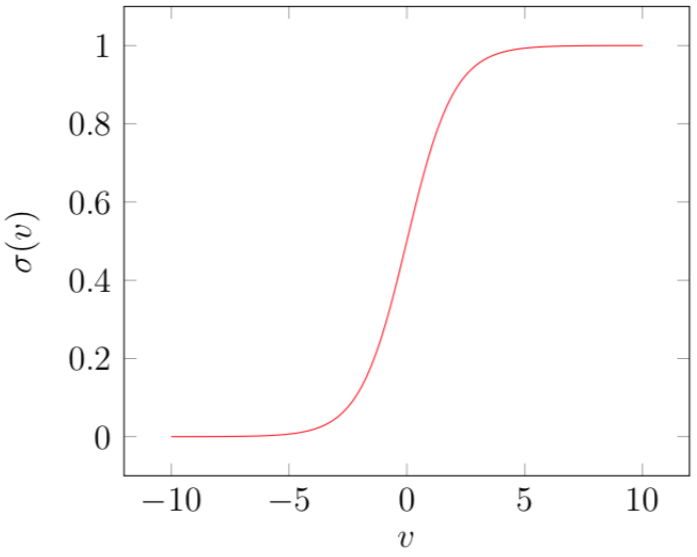
Neural Network



Can you design a neural network that can compute the XOR function?

Activation Function

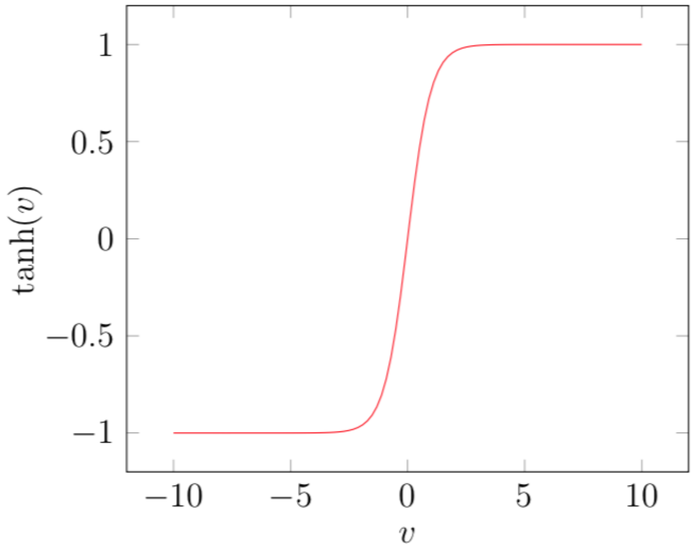
$$y = \sigma(\sum_{i=1}^{d} \theta_i \cdot x_i + \theta_0)$$



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

Activation Function

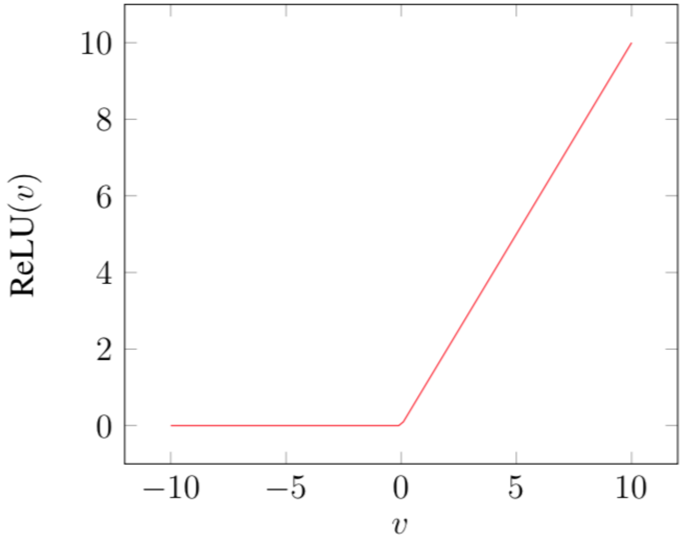
$$y = \tanh(\textstyle\sum_{i=1}^{d} \theta_i \cdot x_i + \theta_0)$$



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

Activation Function

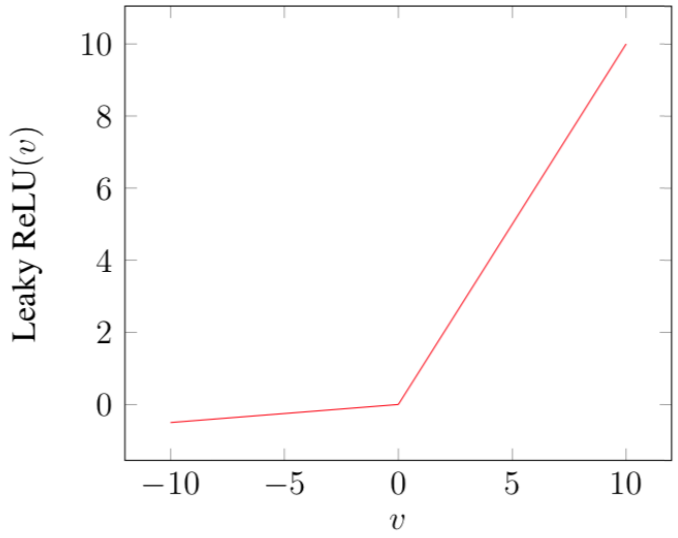
$$y = \mathrm{ReLU}(\sum_{i=1}^{d} \theta_i \cdot x_i + \theta_0)$$



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

Activation Function

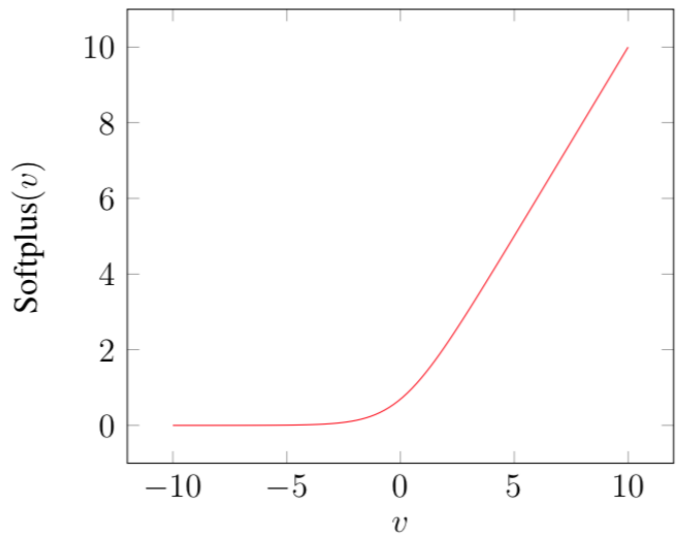
$$y = \text{Leaky ReLU}(\textstyle\sum_{i=1}^{d} \theta_i \cdot x_i + \theta_0)$$



$$\text{Leaky ReLU}(v) = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{otherwise} \end{cases}$$

Activation Function

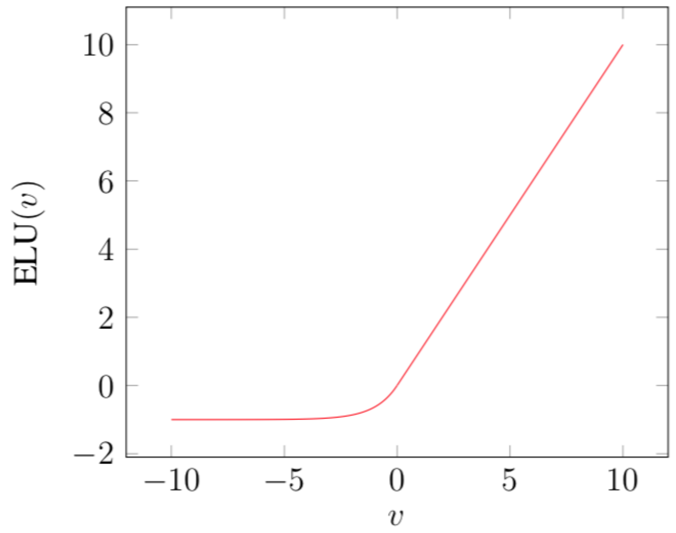
$$y = \text{Softplus}\left(\sum_{i=1}^{d} \theta_i \cdot x_i + \theta_0\right)$$



$$\text{Softplus}(x) = \log(1 + \exp(x))$$

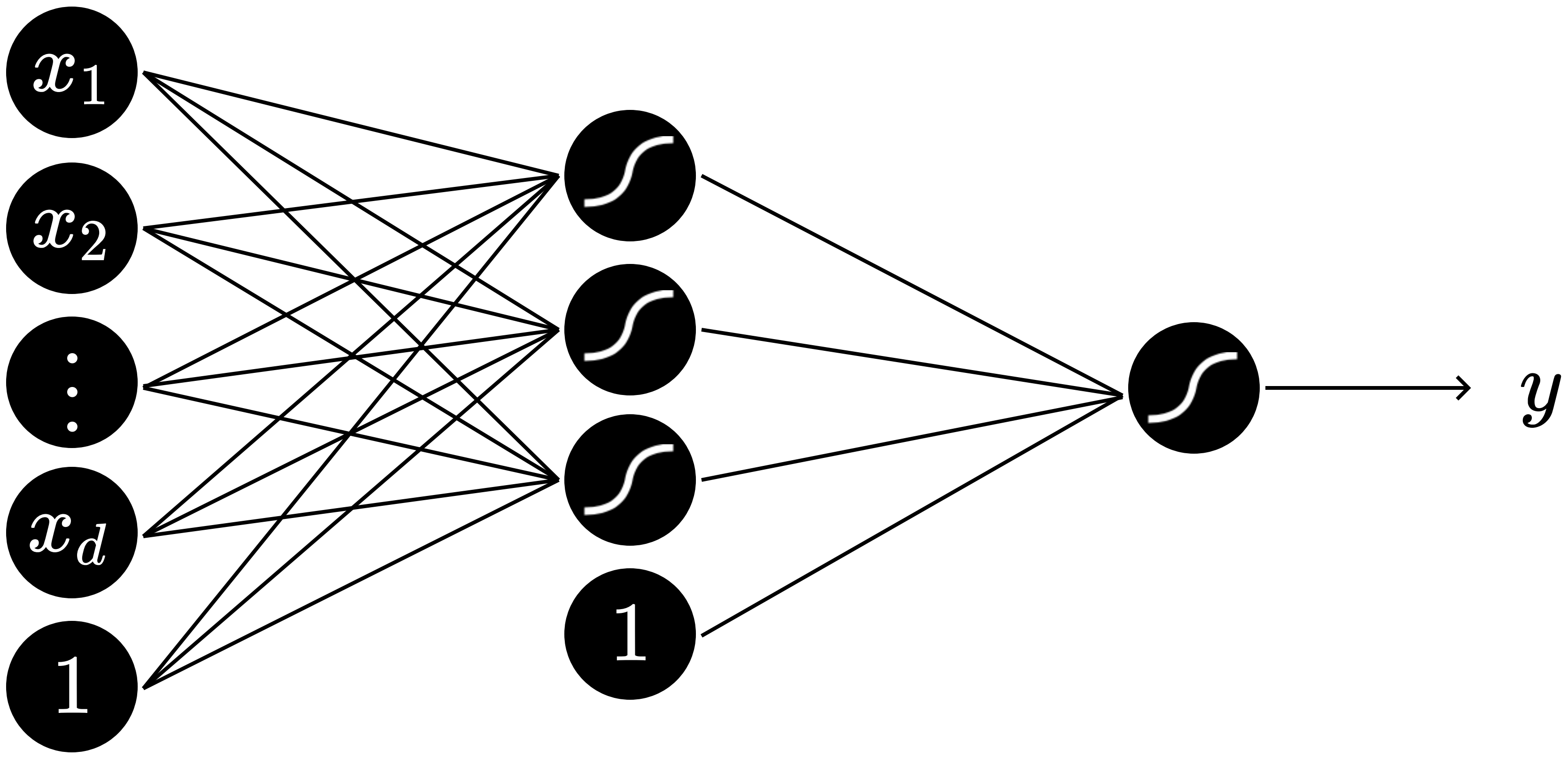
Activation Function

$$y = \text{ELU}(\sum_{i=1}^{d} \theta_i \cdot x_i + \theta_0)$$



$$\text{ELU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ a(\exp(x) - 1) & \text{otherwise} \end{cases}$$

Neural Network

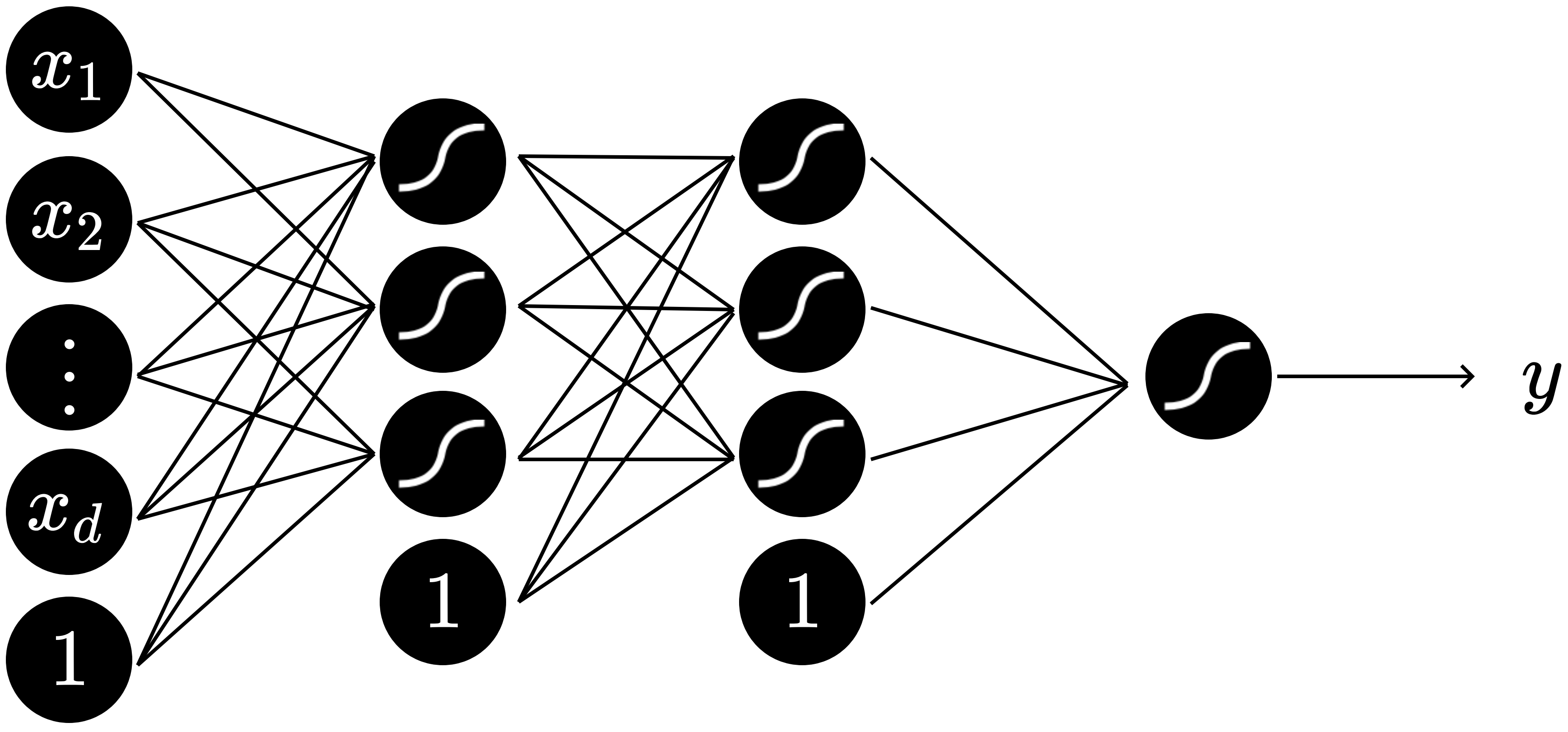


Input Layer

Hidden Layer

Output Layer

Deep Neural Network



Input Layer

Hidden Layer

Hidden Layer

Output Layer

Power of Neural Nets

Boolean Functions

Every boolean function can be represented exactly by some network with two layers (excluding input layer) of units, although the number of hidden units required grows exponentially in the worst case with the number of network inputs.

Can you think about why that is the case? Try to think about how to design such a two-layer network!

Power of Neural Nets

Continuous Functions

Every bounded continuous function can be approximated with arbitrarily small error by a neural network with two layers of units (Cybenko 1989; Hornik et al. 1989), under certain assumptions on the activation functions used.

Power of Neural Nets

Arbitrary Functions

Any function can be approximated to arbitrary accuracy by a neural network with three layers of units (Cybenko 1988), under certain assumptions on the activation functions.

Power of Neural Nets

Arbitrary Functions

Any function can be approximated to arbitrary accuracy by a n

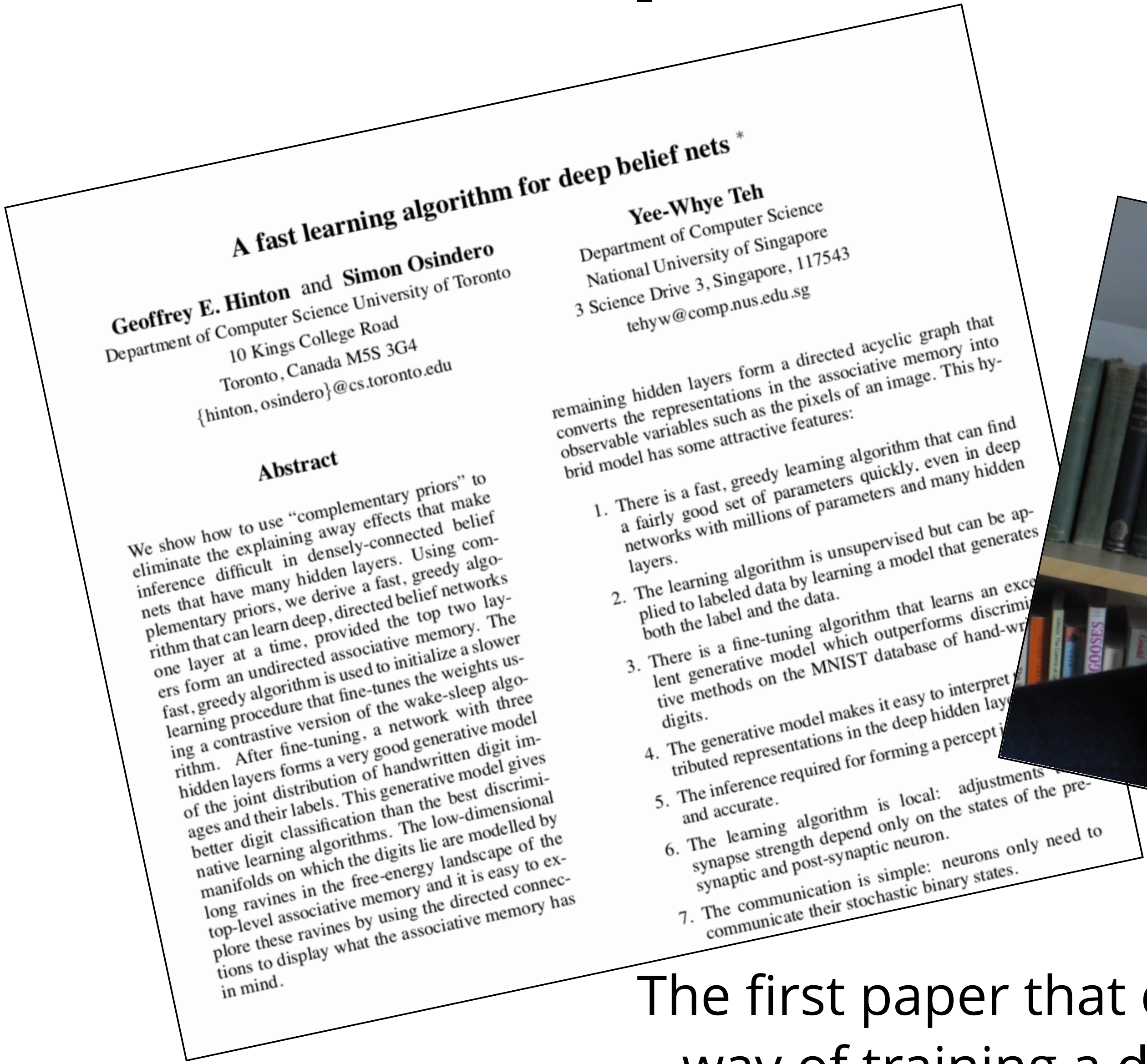
with thre

the

functions.

These Theorems are useful facts.
However, it does not mean it is easy to train
a neural network to approximate a function as desired.

Deep Learning



A fast learning algorithm for deep belief nets *

Geoffrey E. Hinton and **Simon Osindero**
Department of Computer Science University of Toronto
10 Kings College Road
Toronto, Canada M5S 3G4
{hinton, osindero}@cs.toronto.edu

Yee-Whye Teh
Department of Computer Science
National University of Singapore
3 Science Drive 3, Singapore, 117543
tehyw@comp.nus.edu.sg

Abstract

We show how to use "complementary priors" to eliminate the explaining away effects that make inference difficult in densely-connected belief nets that have many hidden layers. Using complementary priors, we derive a fast, greedy algorithm that can learn deep, directed belief networks one layer at a time, provided the top two layers form an undirected associative memory. The fast, greedy algorithm is used to initialize a slower learning procedure that fine-tunes the weights using a contrastive version of the wake-sleep algorithm. After fine-tuning, a network with three hidden layers forms a very good generative model of the joint distribution of handwritten digit images and their labels. This generative model gives better digit classification than the best discriminative learning algorithms. The low-dimensional manifolds on which the digits lie are modelled by long ravines in the free-energy landscape of the top-level associative memory and it is easy to explore these ravines by using the directed connections to display what the associative memory has in mind.

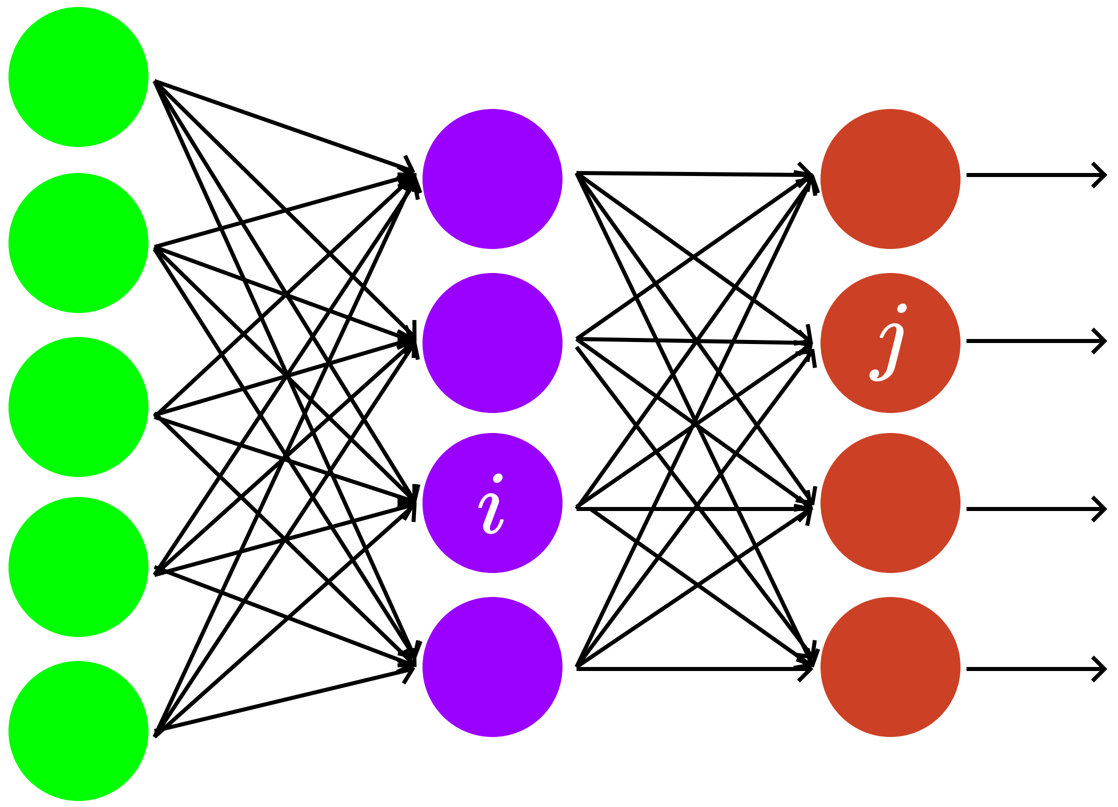
remaining hidden layers form a directed acyclic graph that converts the representations in the associative memory into observable variables such as the pixels of an image. This hybrid model has some attractive features:

1. There is a fast, greedy learning algorithm that can find a fairly good set of parameters quickly, even in deep networks with millions of parameters and many hidden layers.
2. The learning algorithm is unsupervised but can be applied to labeled data by learning a model that generates both the label and the data.
3. There is a fine-tuning algorithm that learns an exce lent generative model which outperforms discrimi tive methods on the MNIST database of hand-wr digits.
4. The generative model makes it easy to interpret tributed representations in the deep hidden lay
5. The inference required for forming a percept and accurate.
6. The learning algorithm is local: adjustments synapse strength depend only on the states of the pre-synaptic and post-synaptic neuron.
7. The communication is simple: neurons only need to communicate their stochastic binary states.

The first paper that describes an effective way of training a deep neural network

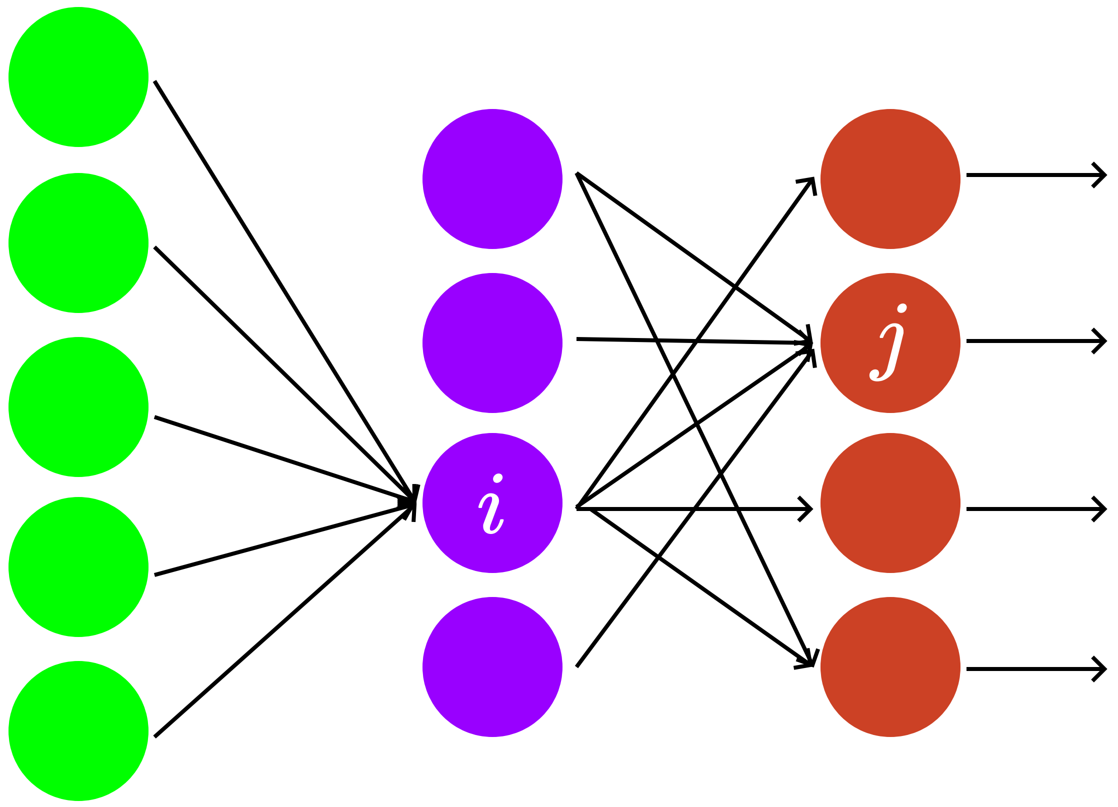
Deep Neural Networks

Learning



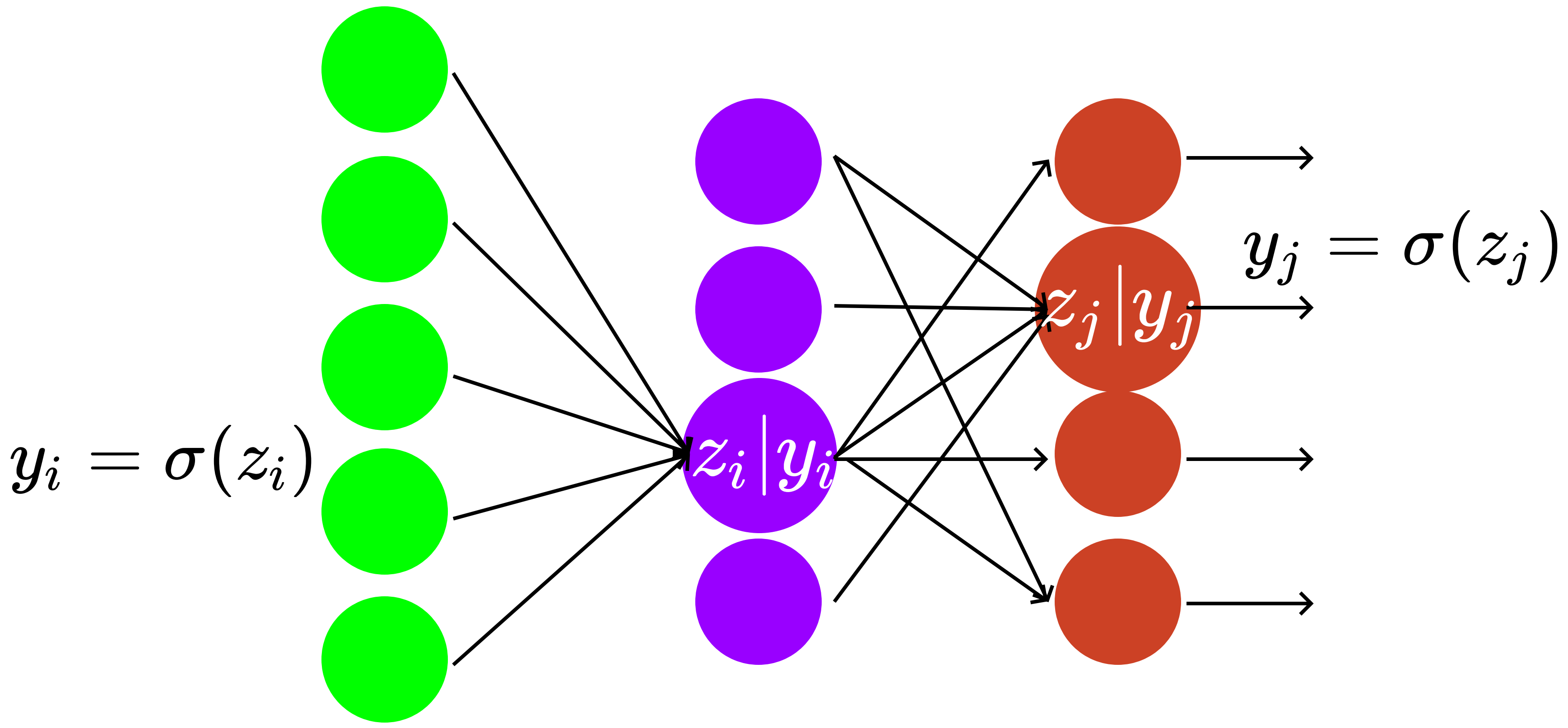
Deep Neural Networks

Learning



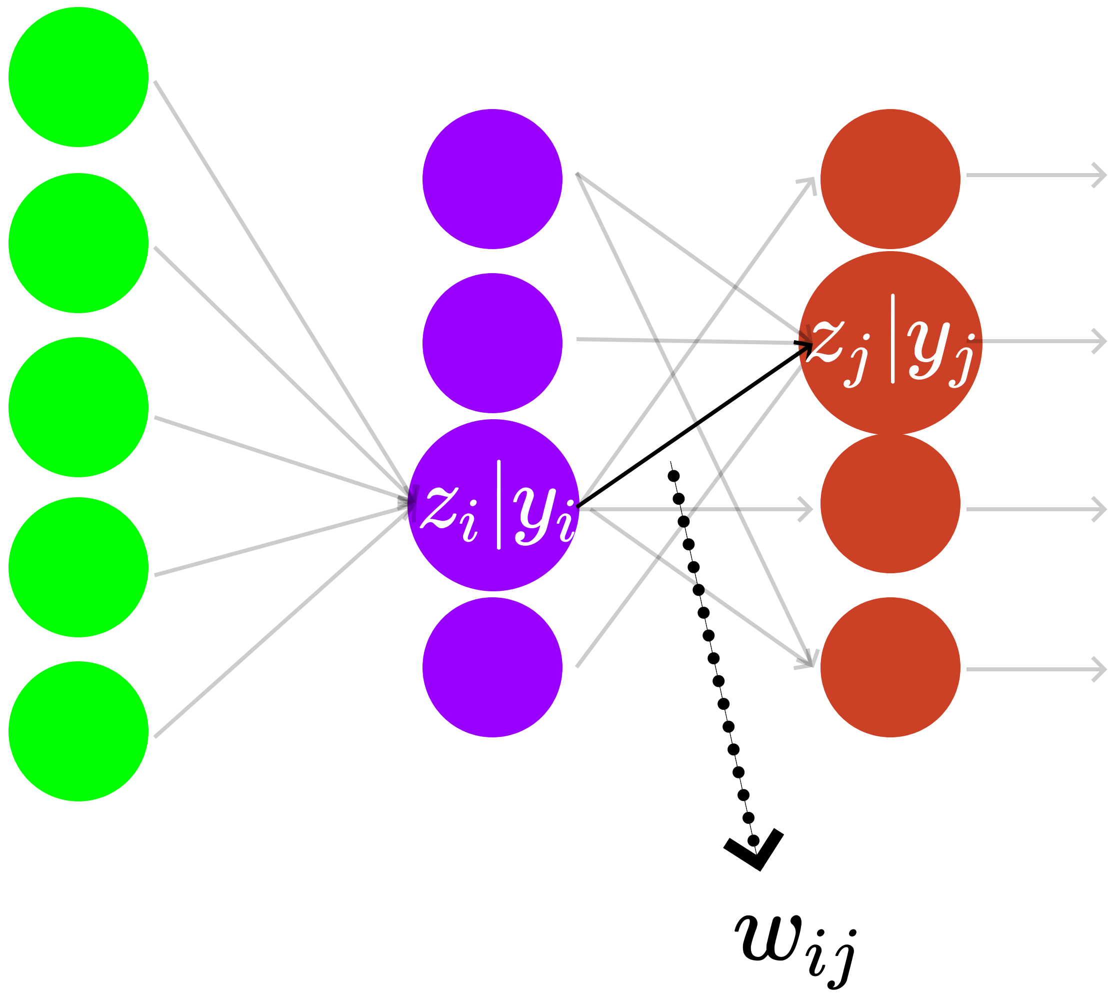
Deep Neural Networks

Learning



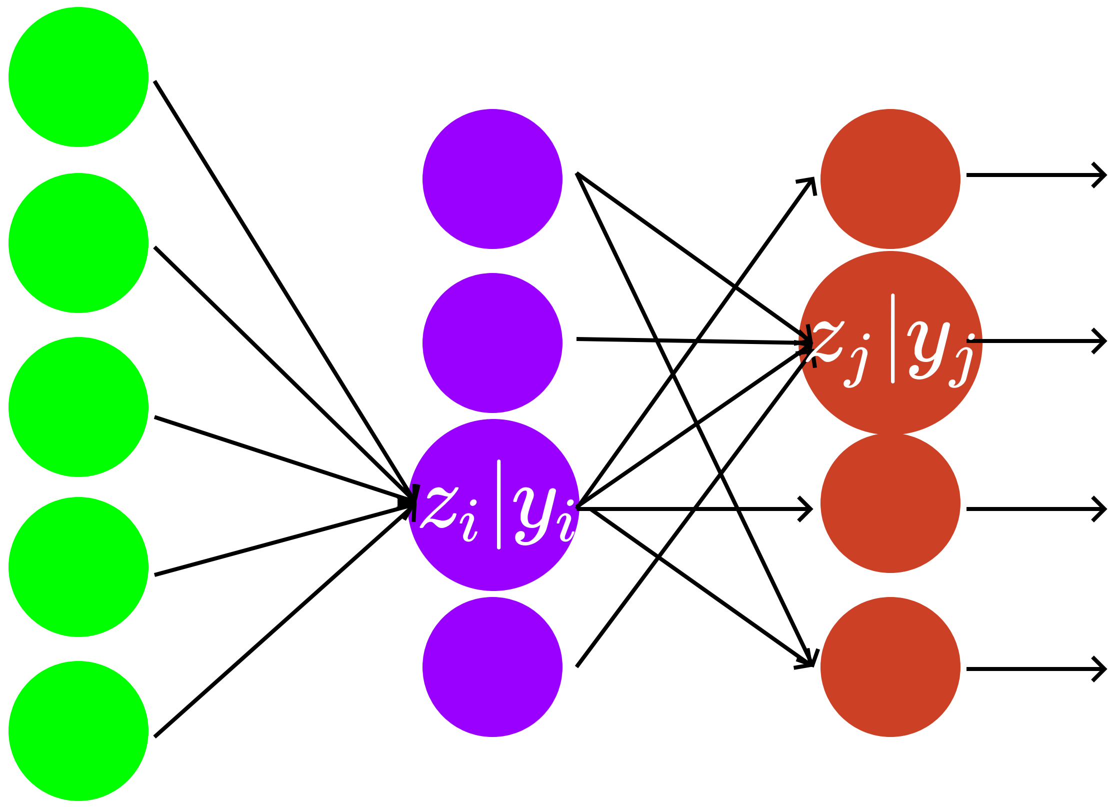
Deep Neural Networks

Learning



Deep Neural Networks

Learning

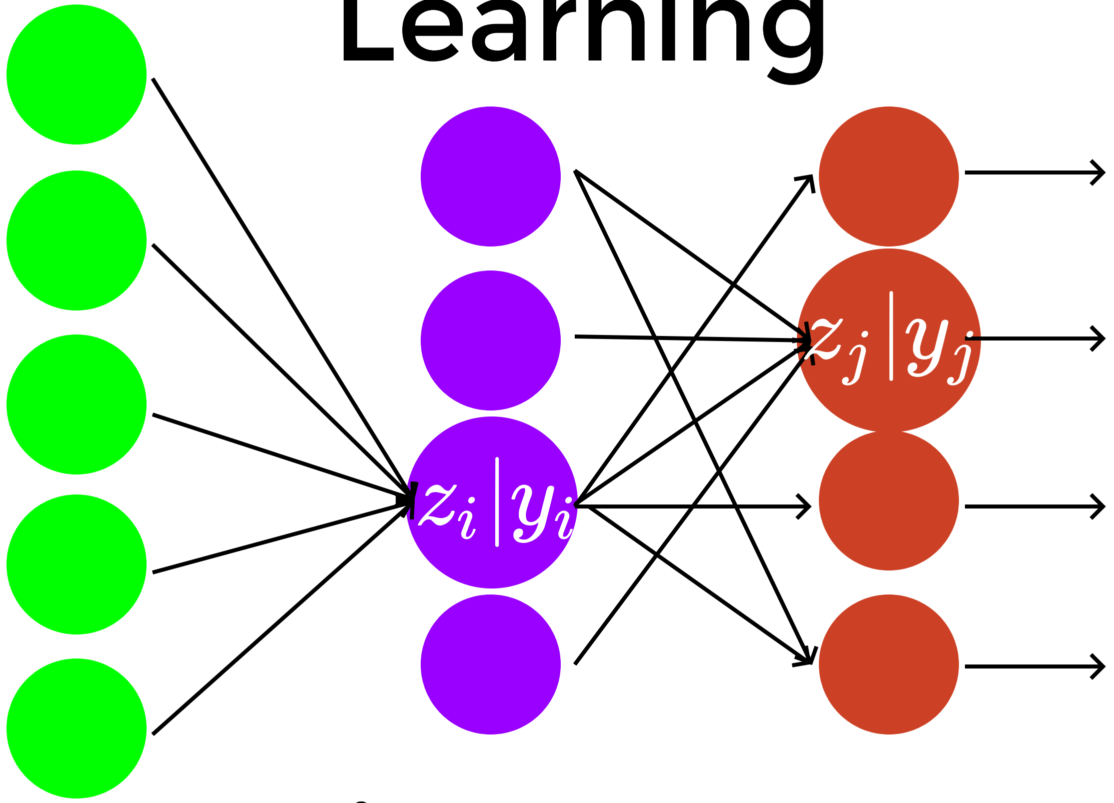


Assume the error function for the network is E.

What is $\frac{\partial E}{\partial w_{ij}}$?

Deep Neural Networks

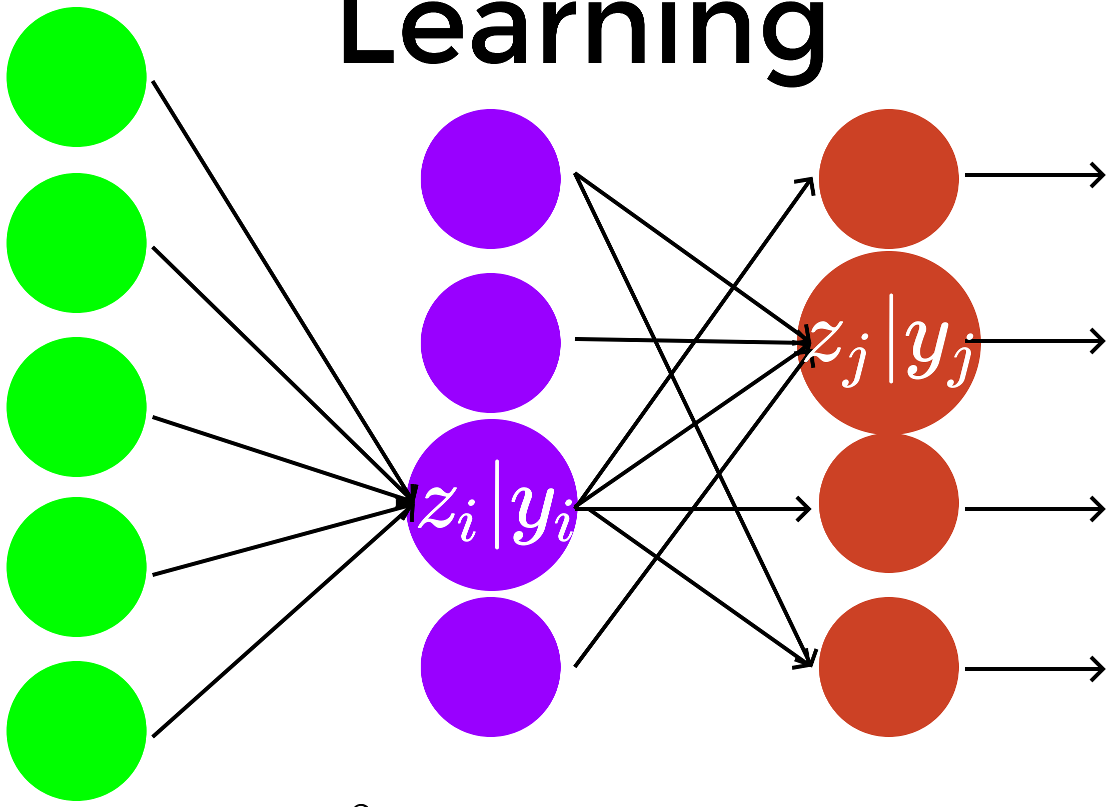
Learning



$$\frac{\partial E}{\partial z_i} = \frac{\partial y_i}{\partial z_i}\frac{\partial E}{\partial y_i} = y_i(1 - y_i)\frac{\partial E}{\partial y_i}$$

Deep Neural Networks

Learning

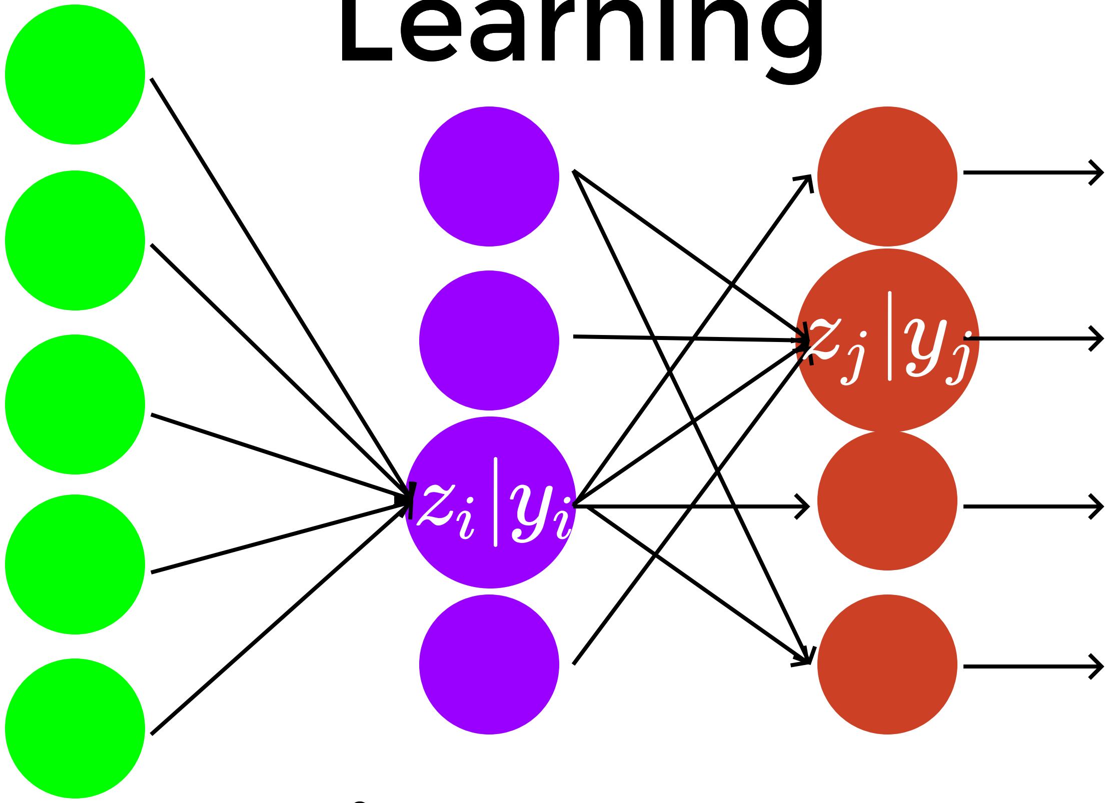


$$\frac{\partial E}{\partial z_i} = \frac{\partial y_i}{\partial z_i}\frac{\partial E}{\partial y_i} = y_i(1-y_i)\frac{\partial E}{\partial y_i}$$

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{\partial z_j}{\partial y_i}\frac{\partial E}{\partial z_j} = \sum_j w_{ij}\frac{\partial E}{\partial z_j}$$

Deep Neural Networks

Learning



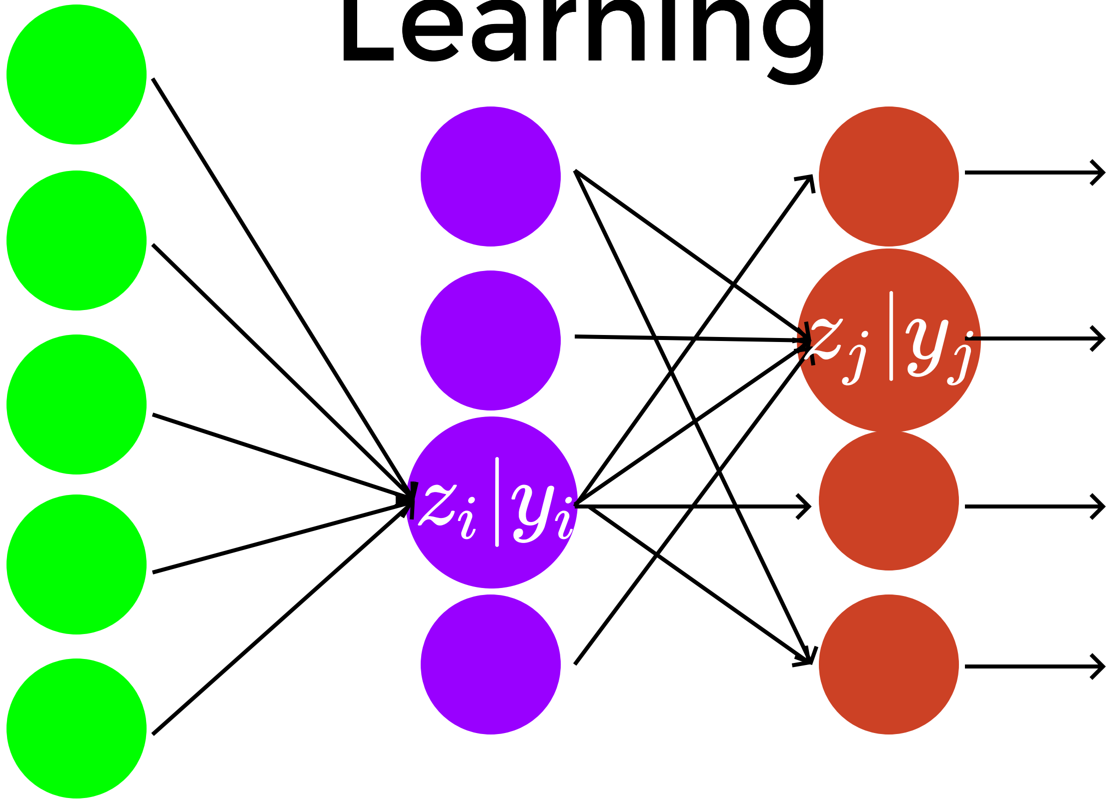
$$\frac{\partial E}{\partial z_i} = \frac{\partial y_i}{\partial z_i}\frac{\partial E}{\partial y_i} = y_i(1-y_i)\frac{\partial E}{\partial y_i}$$

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{\partial z_j}{\partial y_i}\frac{\partial E}{\partial z_j} = \sum_j w_{ij}\frac{\partial E}{\partial z_j}$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial z_j}{\partial w_{ij}}\frac{\partial E}{\partial z_j} = y_i\frac{\partial E}{\partial z_j}$$

Deep Neural Networks

Learning



$$\frac{\partial E}{\partial z_i} = \frac{\partial y_i}{\partial z_i}\frac{\partial E}{\partial y_i} = y_i(1-y_i)\frac{\partial E}{\partial y_i} = y_i(1-y_i)\sum_j w_{ij}\frac{\partial E}{\partial z_j}$$

$$\delta_i = y_i(1-y_i)\sum_j w_{ij}\ \ \delta_j$$

Backpropagation

1. Randomly initialize the weights (with small non-zero values).

2. Perform a forward pass to calculate the outputs (y values) from each neuron layer by layer.

3. For each output neuron t, calculate δ_t as follows:

   $\delta_t \leftarrow -y_t(1-y_t)(y_t^* - y_t)$ (Assume: $E = \frac{1}{2}\sum_t ||y_t - y_t^*||^2$)

4. For each hidden neuron j, calculate δ_j as follows:

   $\delta_j \leftarrow y_j(1-y_j)\sum_{k:j\rightarrow k} w_{jk}\delta_k$

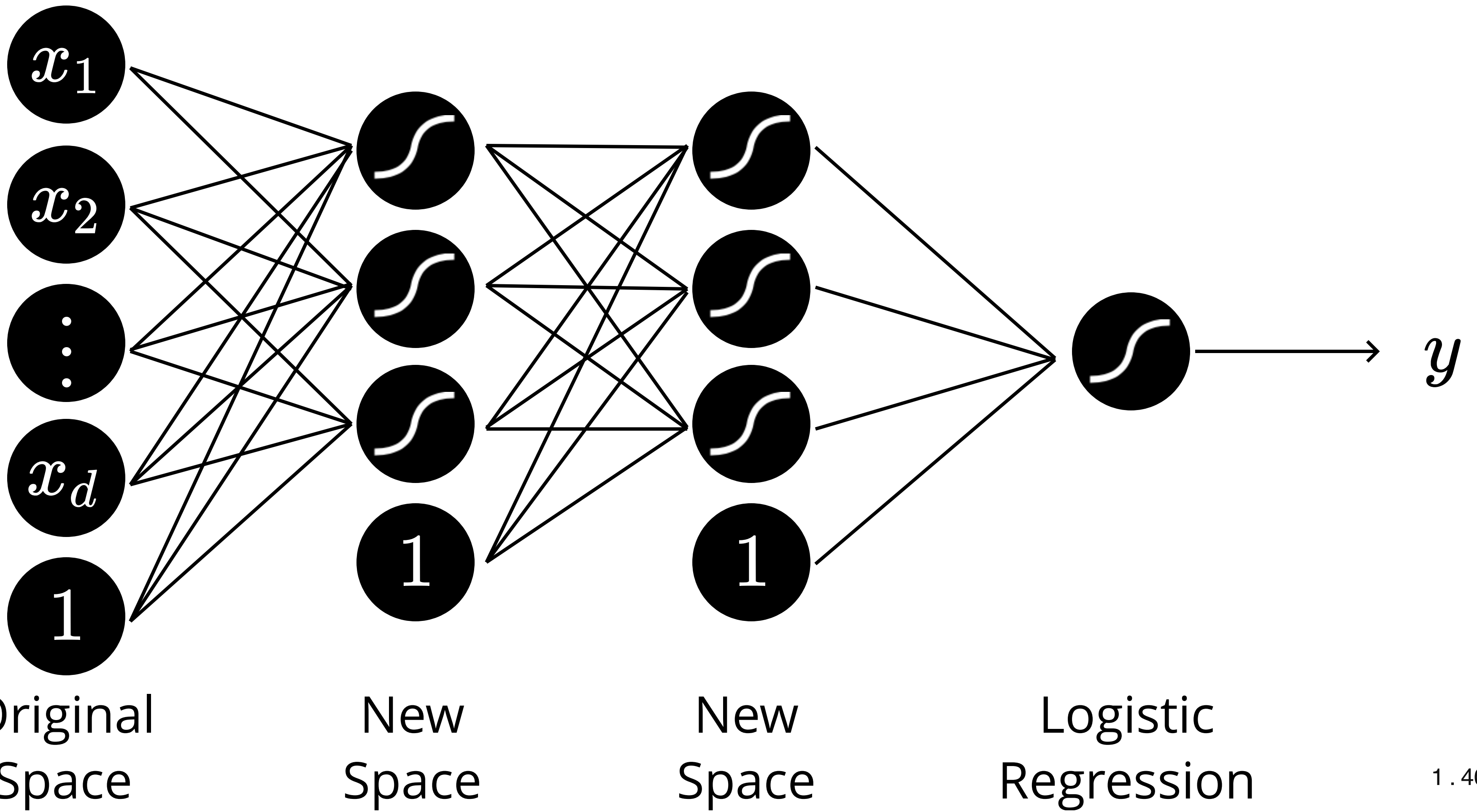
5. Update the weights:

   $w_{ij} \leftarrow w_{ij} - \eta\delta_j y_i$

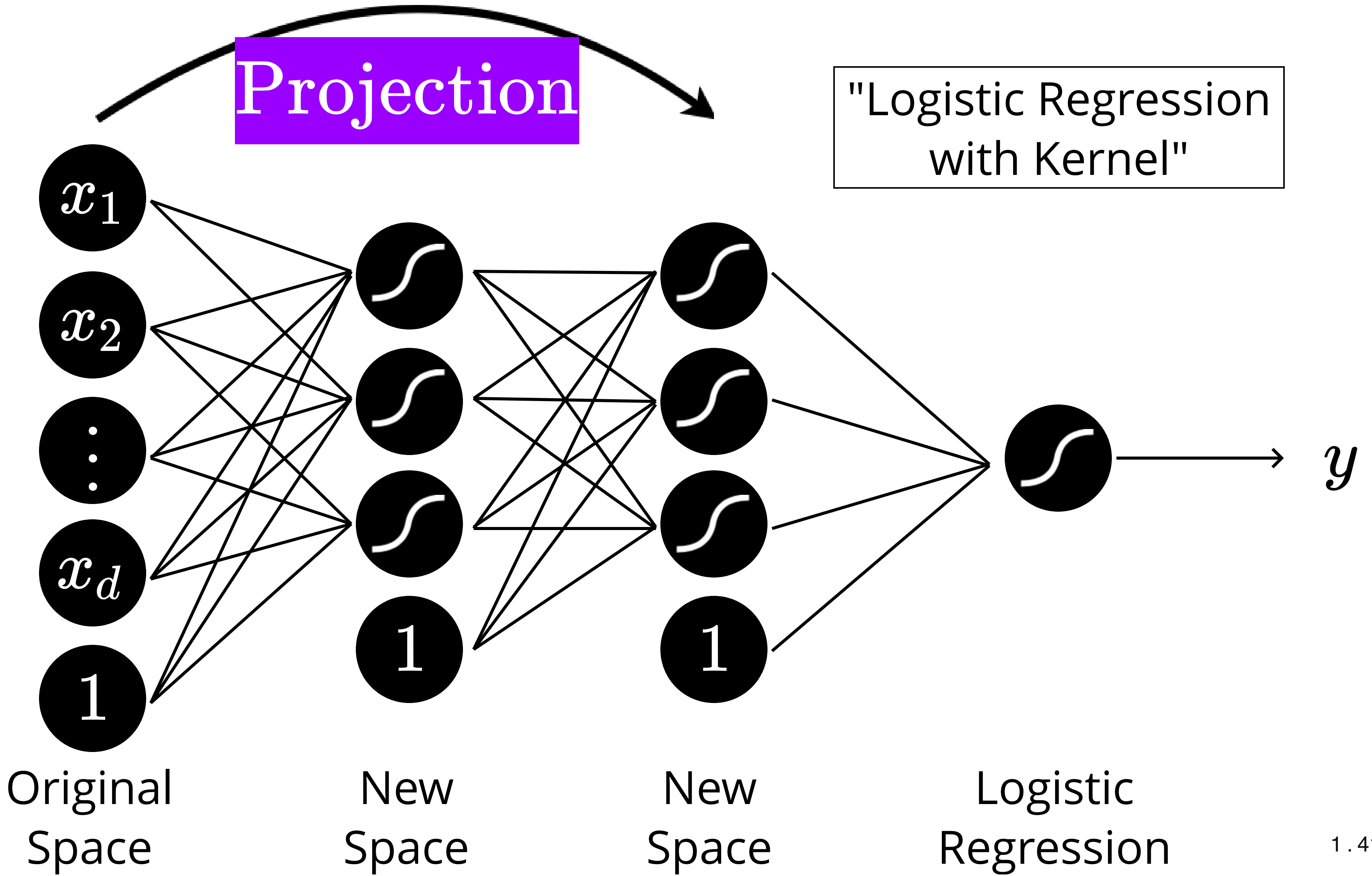
   where η is the learning rate.

6. Return to 2, unless you are satisfied.

Deep Neural Networks



Deep Neural Networks



Probabilistic Models

$$p(y|x)$$

Logistic Regression

$$p(x, y)$$

What about this?

Probabilistic Models

$p(y|x)$

Logistic Regression

$\text{Discriminative Model}$

$p(x, y)$

What about this?

☞ Generative Model