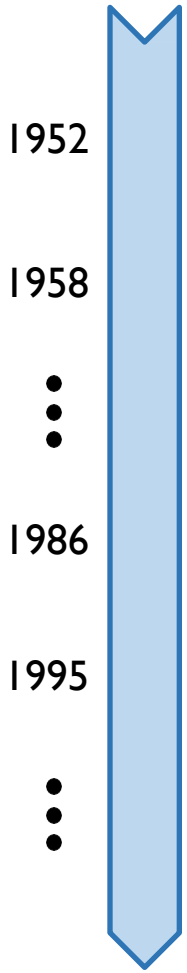
Deep Learning

Lecture 4: Neural Network

National University of Singapore

Instructor: Joey Tianyi Zhou

Neural Networks .. resurgence..



1952 Stochastic Gradient Descent

1958 Perceptron
- Learnable Weights

1986 Backpropagation
- Multi-Layer Perceptron

1995 Deep Convolutional NN
- Digit Recognition

1. Big Data
- Larger Datasets
- Easier Collection & Storage

2. Hardware
- Graphics Processing Units (GPUs)
- Massively Parallelizable

3. Software
- Improved Techniques
- New Models
- Toolboxes

Hebb's rule.. 1949

Hebb conjectured that:

a particular type of ***use-dependent modification*** **of the connection strength of synapses** might underlie learning in the nervous system.

Hebb introduced a neurophysiological postulate :

"…When an axon of cell A is near enough to excite a cell B and repeatedly and persistently takes part in firing it, *some growth process or metabolic change* takes place in one or both cells, such that A's efficiency as one of the cells firing B, is increased."

*Neurons that **FIRE** together **WIRE** together …*

Hebb's rule.. 1949

Simplest formalisation of Hebb's rule is ***to increase weight of connection at every next instant*** *as follows*:

$$w_{ji}^{k+1} = w_{ji}^{k} + \Delta w_{ji}^{k} \qquad (1)$$

where

$$\Delta w_{ji}^{k} = C a_{i}^{k} X_{j}^{k} \qquad (2)$$

w_{ji}^{k} is the ***weight*** of connection **at instant *k (or k-th input)***,

w_{ji}^{k+1} is the ***weight*** of connection at the following **instant *k+1 (or k+1-th input)***,

Δw_{ji}^{k} is ***increment*** by which the weight of connection is enlarged,

C is positive coefficient which determines ***learning rate***,

a_{i}^{k} is ***input*** value from the presynaptic neuron at instant *k,*

X_{j}^{k} is ***output*** of the postsynaptic neuron at the same instant *k*.

Hebb's rule.. 1949

Hebb Learning is fundamentally unstable as stronger connections will enforce themselves:

no notion of "competition"

no reduction in weights

learning is unbounded

Later modifications allowed for *weight normalization*, *forgetting* etc. e.g. Generalized Hebbian learning.

Perceptrons

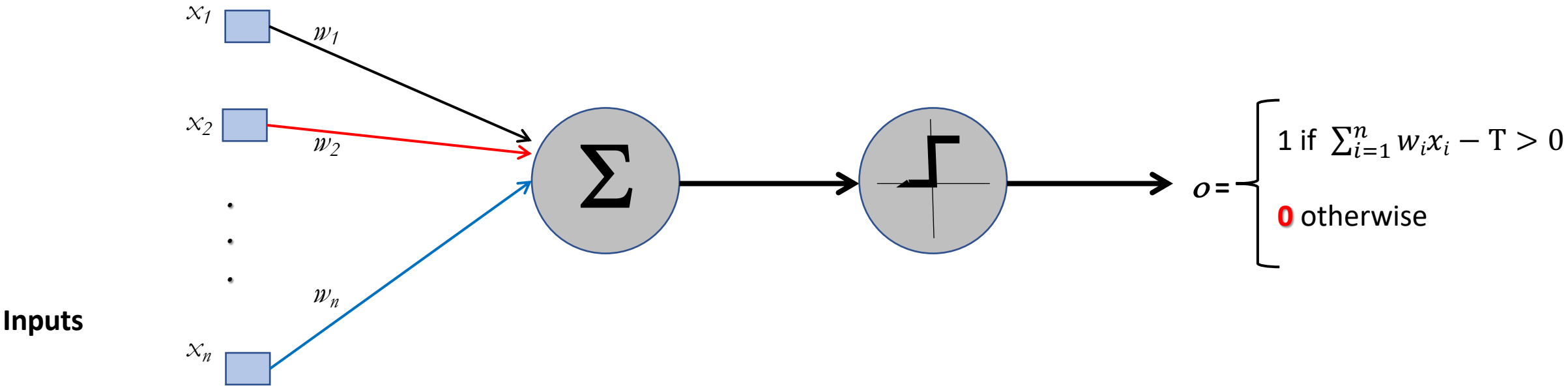
Frank Rosenblatt, Psychologist & Logician, introduced the **Perceptron** (1958);

- originally it was assumed that it could represent ***any*** Boolean circuit and perform ***any*** logic:
- "*the embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence*," New York Times (8 July) 1958
- "*Frankenstein Monster Designed by Navy That Thinks*," Tulsa, Oklahoma Times 1958.

Perceptrons

Rosenblatt's Perceptron for binary classifications:

- one weight w_i per input x_i
- multiply weights with respective inputs
- if result larger than threshold return 1, otherwise **0**



Perceptrons .. training.

Rosenblatt's innovative learning algorithm for Perceptrons:

initialize weights randomly

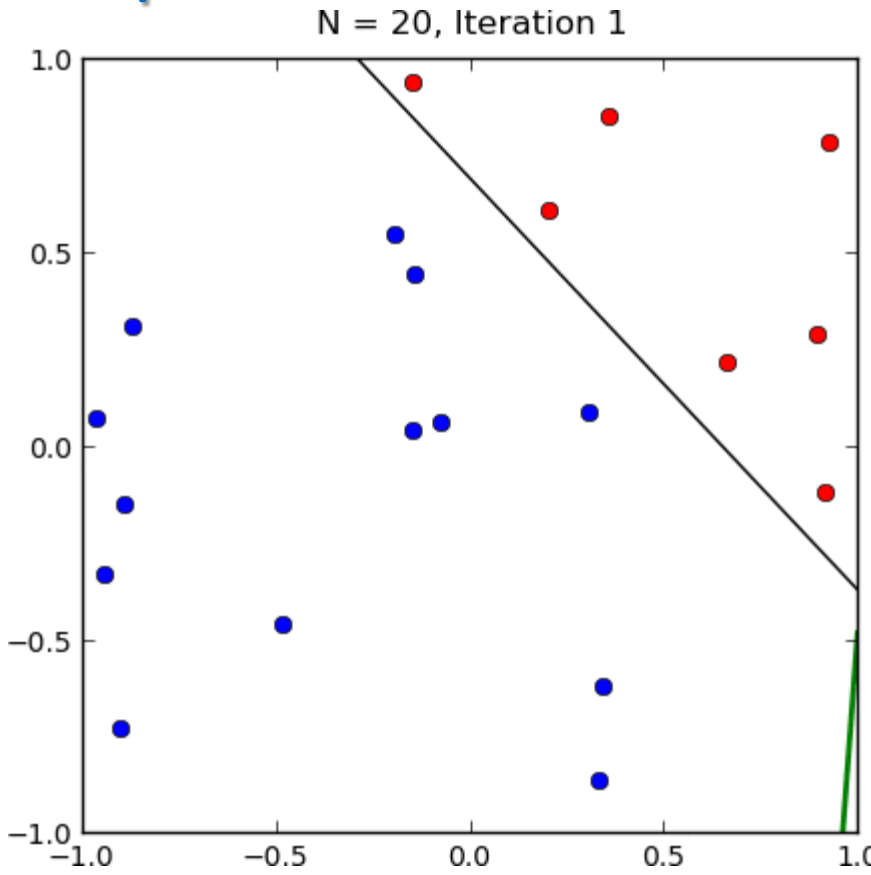
take one sample x_i and predict y_i

for erroneous predictions update weights:

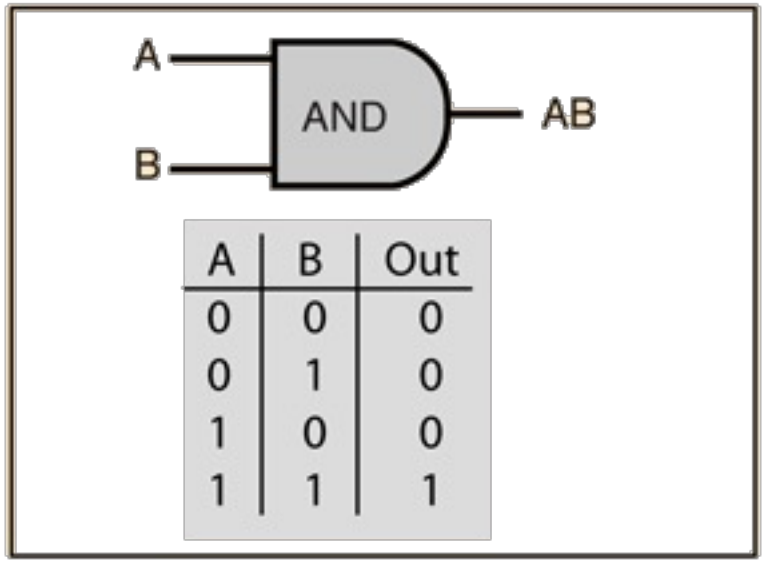
if output $\hat{y}_i = 0$ and $y_i = 1$, **increase weights**

if output $\hat{y}_i = 1$ and $y_i = 0$, **decrease weights**

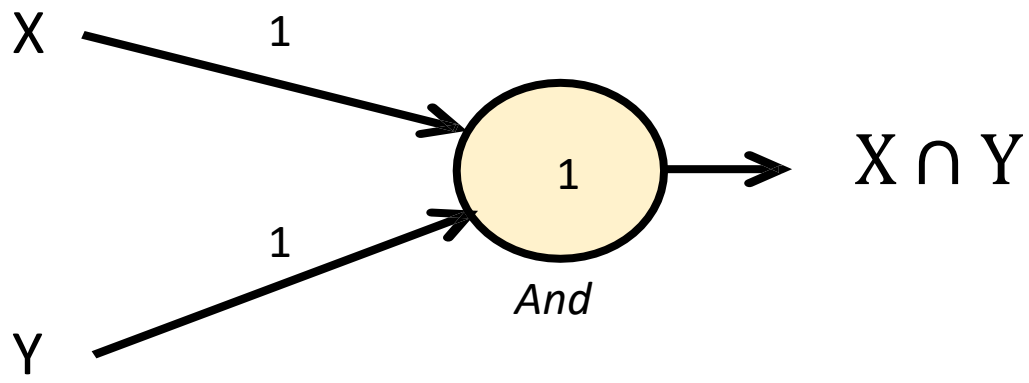
repeat until **no errors** are made…



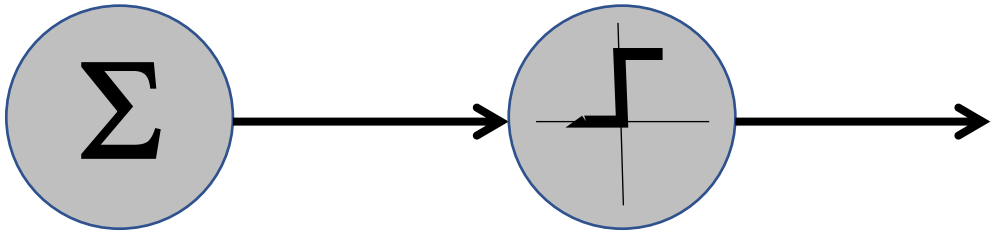
Perceptrons



A	B	Out
0	0	0
0	1	0
1	0	0
1	1	1



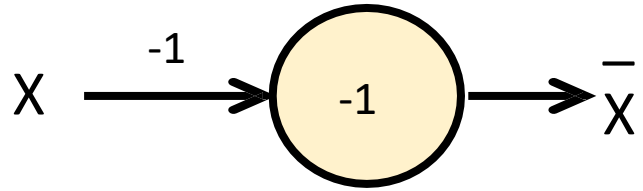
if result equal to or larger than threshold return 1, otherwise **0**



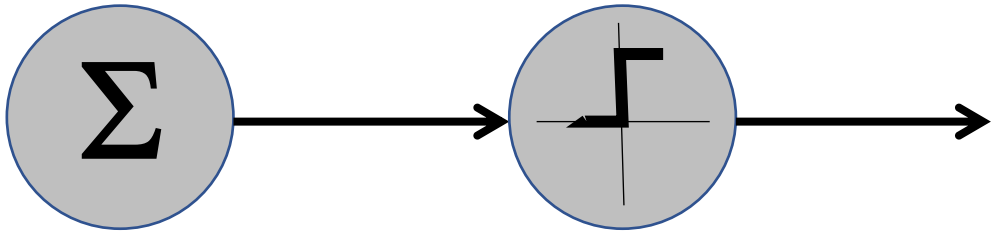
Perceptrons

Inverter

INPUT	OUTPUT
A	
0	1
1	0



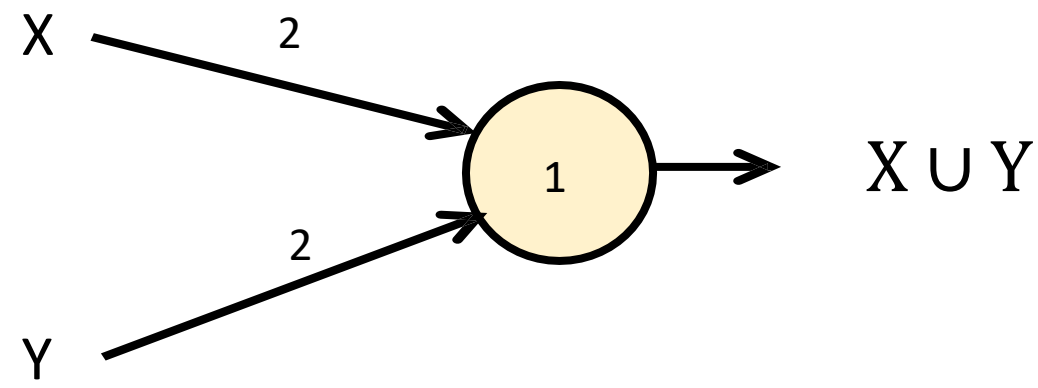
if result equal to or larger than threshold return 1, otherwise **0**



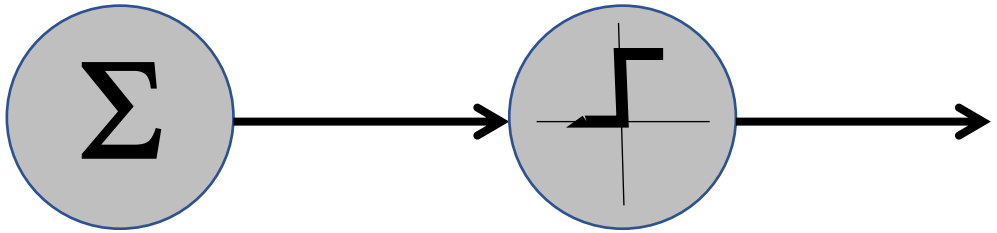
Perceptrons

Or

INPUT		OUTPUT
A	B	
0	0	0
1	0	1
0	1	1
1	1	1



if result equal to or larger than threshold return 1, otherwise **0**

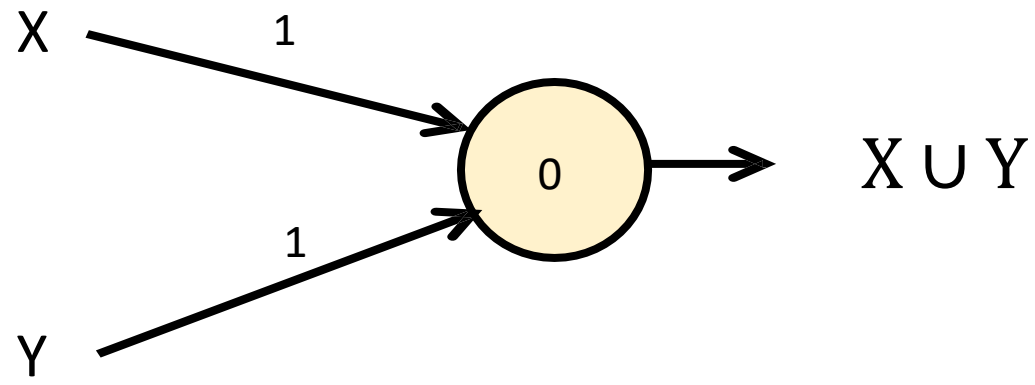


However, the form to implement the logic gate is exclusive.

Perceptrons

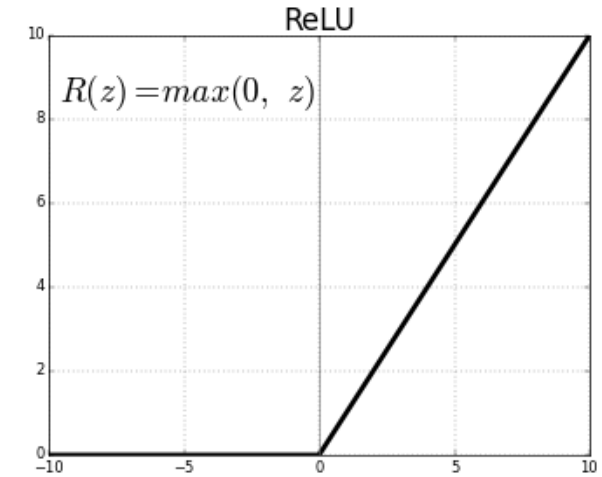
Or

INPUT		OUTPUT
A	B	
0	0	0
1	0	1
0	1	1
1	1	1

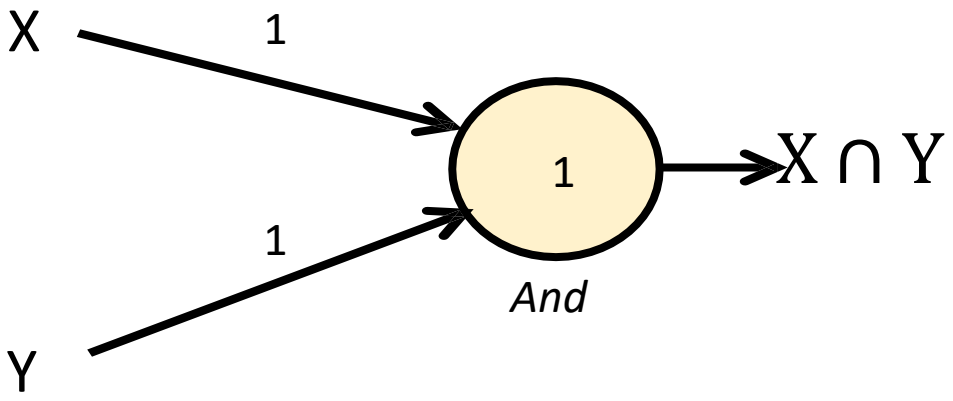


What is nonlinear activation here?

H=X+Y, $X \cup Y$= Max(0,H)



Perceptrons

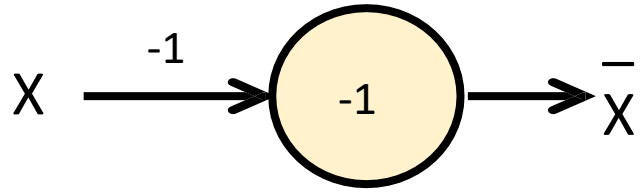


And

INPUT		OUTPUT
A	B	
0	0	0
1	0	0
0	1	0
1	1	1

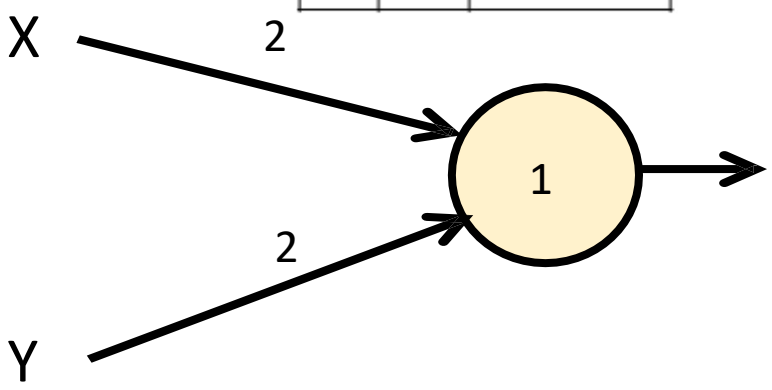
$X \cap Y$

Inverter



$\bar{X}$

INPUT	OUTPUT
A	
0	1
1	0



$X \cup Y$

Or

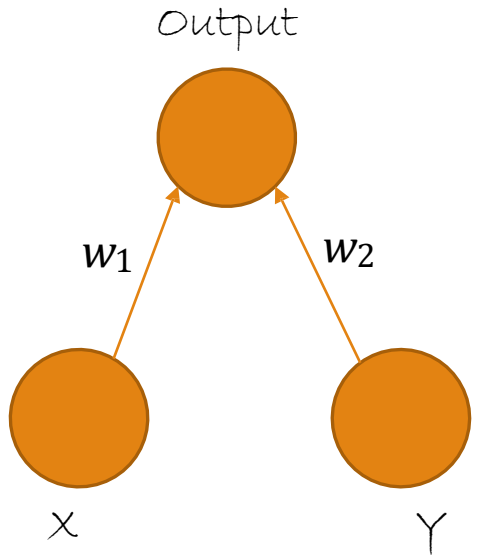
INPUT		OUTPUT
A	B	
0	0	0
1	0	1
0	1	1
1	1	1

able to mimic any Boolean gate … except for ?

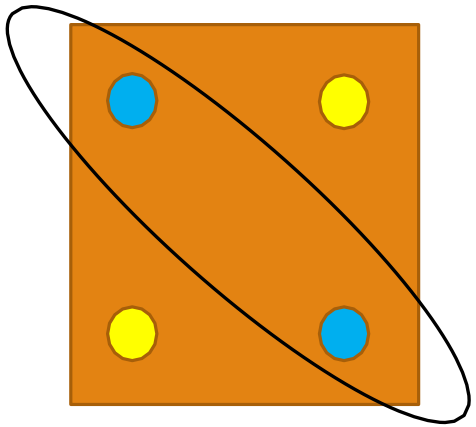
Perceptrons.. *XOR*

Exclusive or (XOR $\oplus$) cannot be solved by Perceptron which can only discriminate linearly separable classes

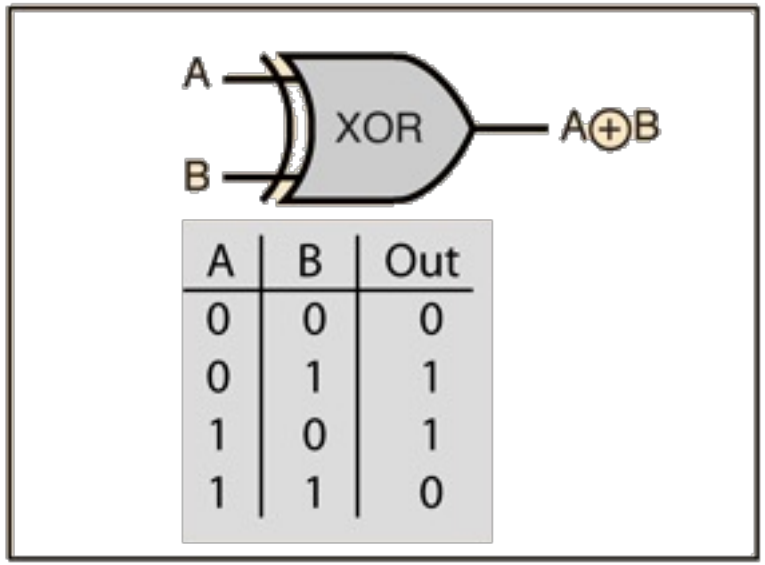
Minsky and Papert, "Perceptrons", 1969



- $0\ w_1 + 0w_2 < \theta \rightarrow 0 < \theta$
- $0\ w_1 + 1w_2 > \theta \rightarrow w_2 > \theta$
- $1\ w_1 + 0w_2 > \theta \rightarrow w_1 > \theta$
- $1\ w_1 + 1w_2 < \theta \rightarrow w_1 + w_2 < \theta$



Inconsistent!!



A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

X	Y	Desired Output
1	1	0
1	0	1
0	1	1
0	0	0

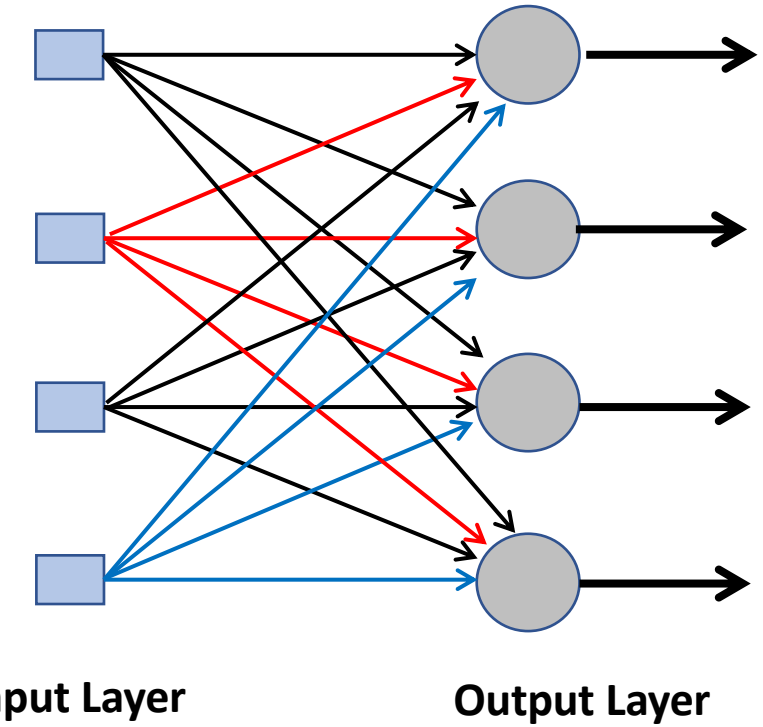
Building Neural Networks with Perceptrons

Perceptrons .. *to Neural Networks*

Each Perceptron represents a decision.

For multiple decisions, stack as many outputs as the possible outcomes into a layer ⇨ Neural Network.

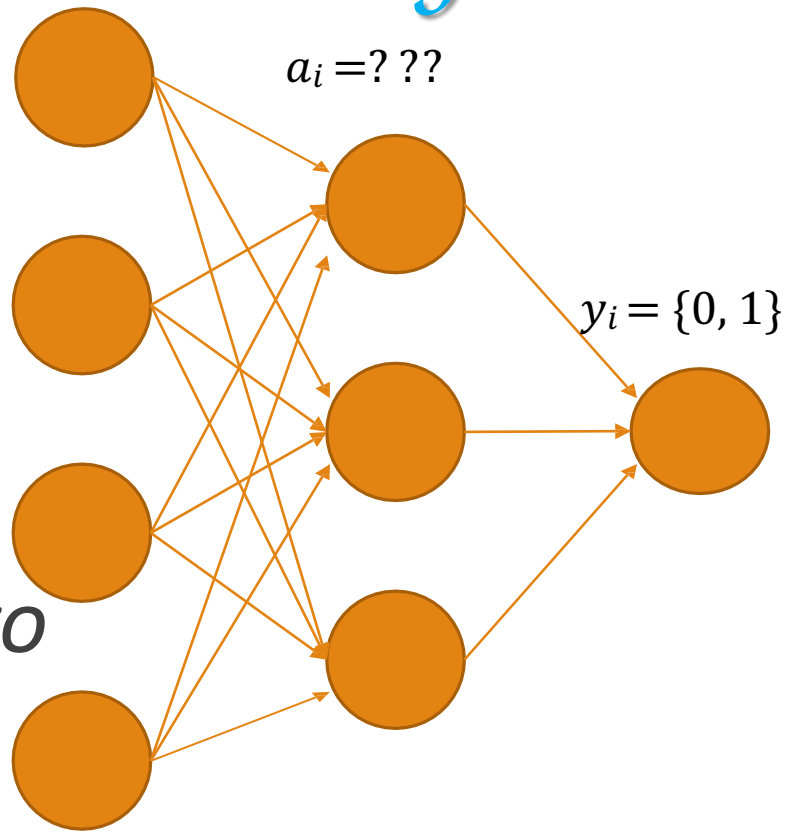
Use one layer as input to the next layer ⇨ Multi-Layer Perceptron (MLP)



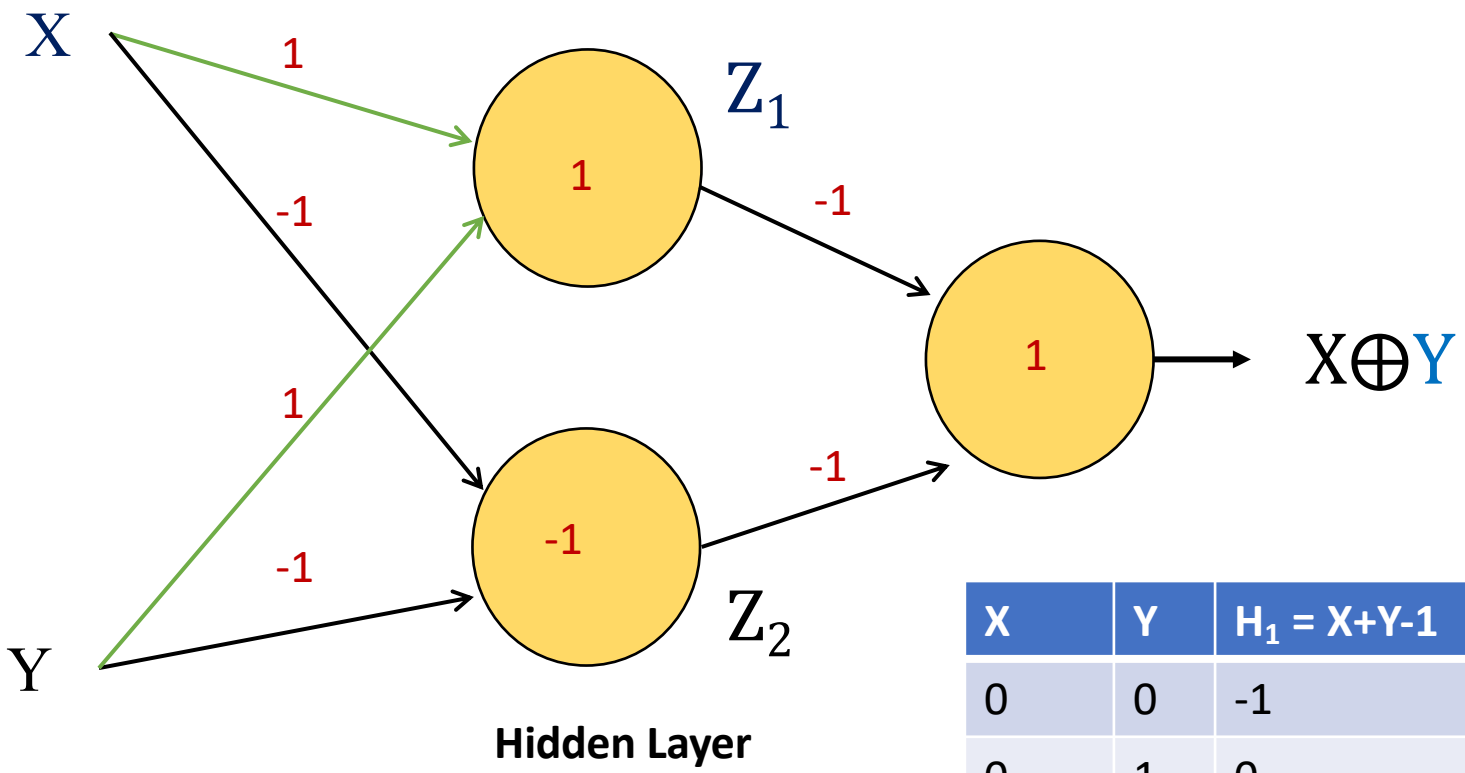
Multi-layer Perceptrons .. *Minsky*

XOR can't be realized with a single layer Perceptron

But a Multi-layer Perceptron can … *how to train such a Multi-layer Perceptron* ?



Multi-layer Perceptrons .. *Minsky*

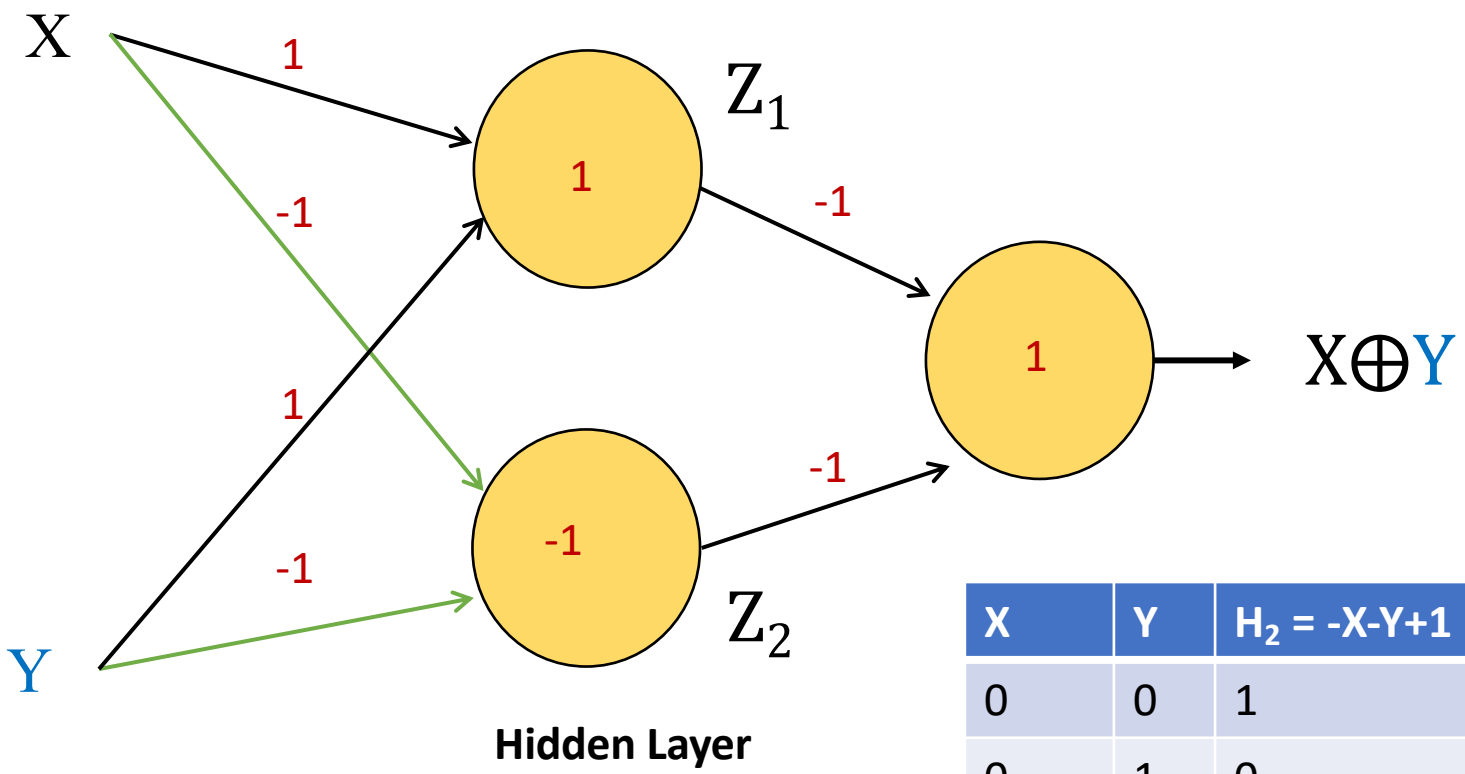


X	Y	$H_1 = X+Y-1$	$Z_1 = \max(0, H_1)$
0	0	-1	0
0	1	0	0
1	0	0	0
1	1	1	1

Has a “hidden” layer.

Originally suggested by *Minsky and Papert* 1968

Multi-layer Perceptrons .. *Minsky*

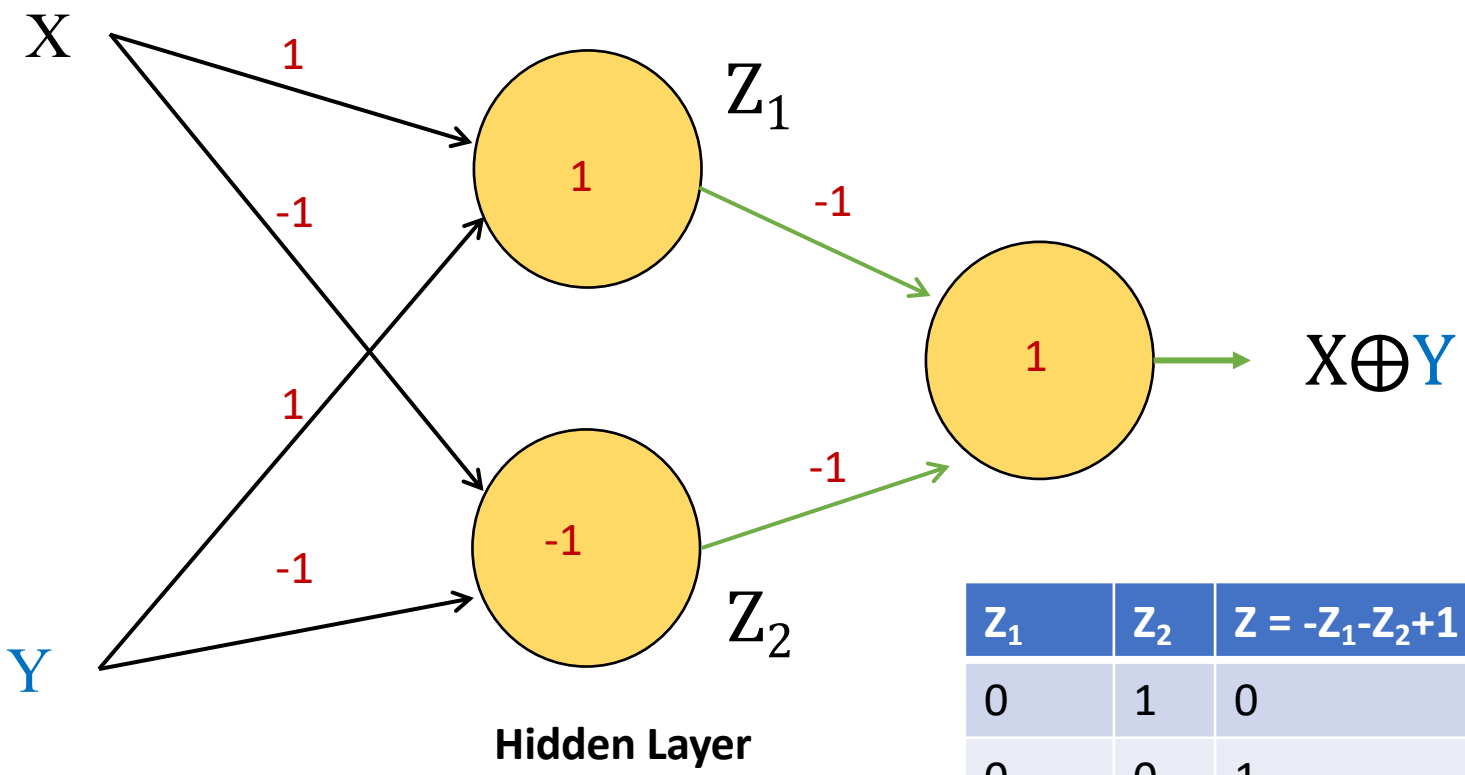


X	Y	$H_2 = -X-Y+1$	$Z_2 = \max(0, H_1)$
0	0	1	1
0	1	0	0
1	0	0	0
1	1	-1	0

Has a "hidden" layer.

Originally suggested by *Minsky and Papert* 1968

Multi-layer Perceptrons .. *Minsky*

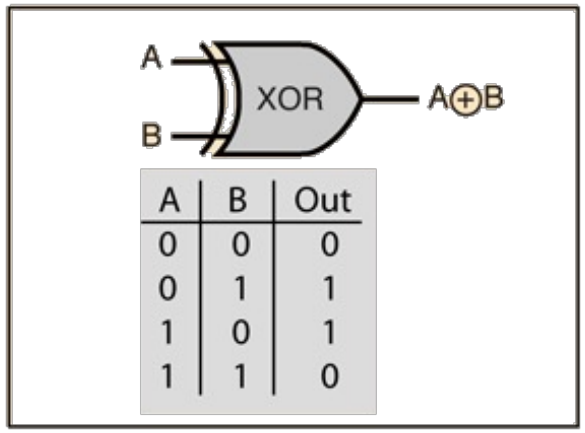


Z_1	Z_2	$Z = -Z_1 - Z_2 + 1$
0	1	0
0	0	1
1	1	0
1	0	0

Has a "hidden" layer.

Originally suggested by *Minsky and Papert* 1968

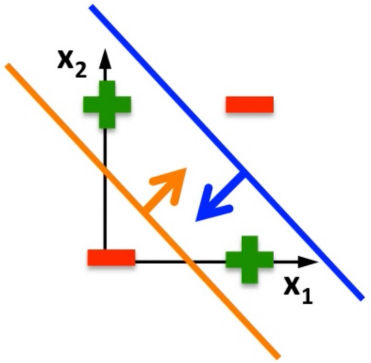
Multi-layer Perceptrons .. *Minsky*

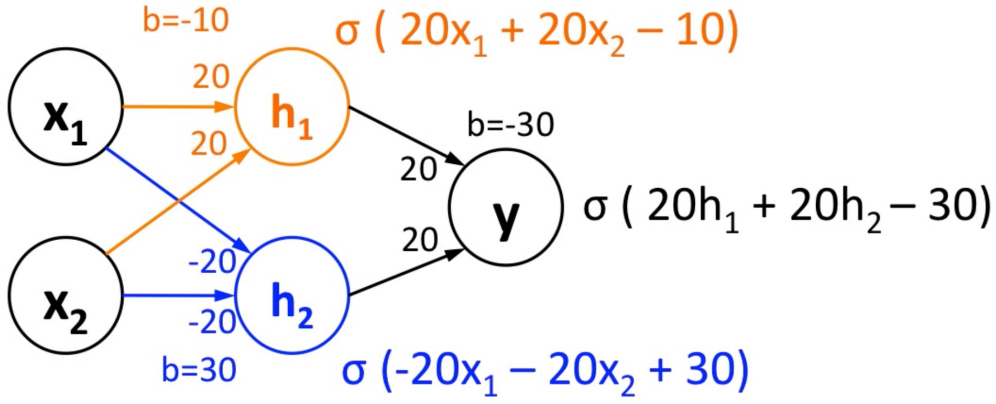


A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

Solving XOR with a Neural Net

Linear classifiers cannot solve this





σ(20*0 + 20*0 – 10) ≈ 0
σ(20*1 + 20*1 – 10) ≈ 1
σ(20*0 + 20*1 – 10) ≈ 1
σ(20*1 + 20*0 – 10) ≈ 1

σ (-20*0 – 20*0 + 30) ≈ 1
σ (-20*1 – 20*1 + 30) ≈ 0
σ (-20*0 – 20*1 + 30) ≈ 1
σ (-20*1 – 20*0 + 30) ≈ 1

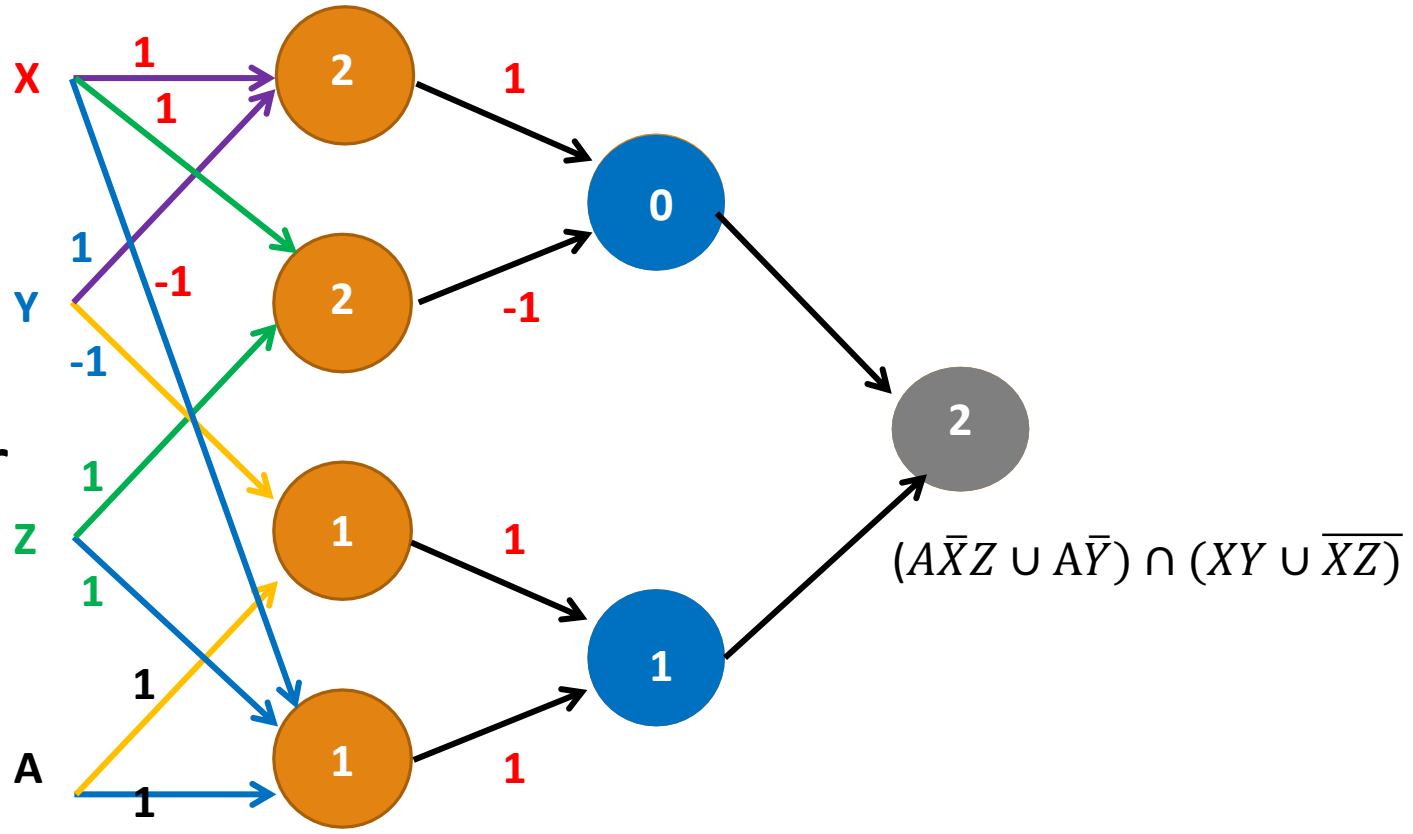
σ (20*0 + 20*1 – 30) ≈ 0
σ (20*1 + 20*0 – 30) ≈ 0
σ (20*1 + 20*1 – 30) ≈ 1
σ (20*1 + 20*1 – 30) ≈ 1

Has a “hidden” layer.

Originally suggested by *Minsky and Papert* 1968

Multi-layer Perceptrons .. *generic*

A Multi-Layer Perceptron can compose **arbitrarily complicated Boolean functions** with **any** number of inputs & **any** number of outputs!



But how many "layers" will it need?

Multi-layer Perceptrons.. number of layers

Disjunctive Normal Form (DNF), In boolean logic, a disjunctive normal form (DNF) is **a canonical normal form of a logical formula consisting of a disjunction of conjunctions**; it can also be described as an OR of ANDs, a sum of products.

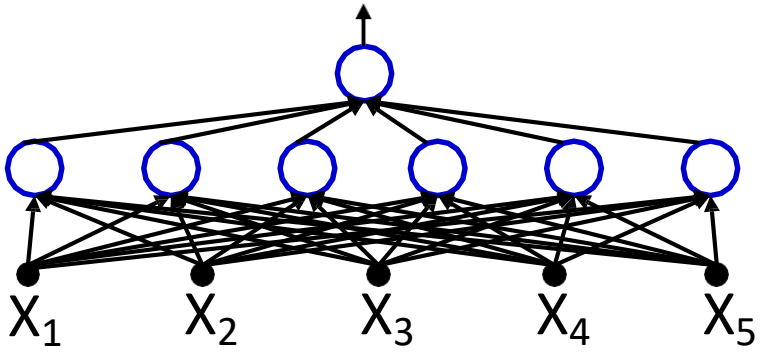
Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

Truth table shows all input combinations for which output is 1

$$Y = \overline{X_1}.\overline{X_2}.X_3.X_4.\overline{X_5} + \overline{X_1}.X_2.\overline{X_3}.X_4.X_5 + \overline{X_1}.X_2.X_3.\overline{X_4}.\overline{X_5} + X_1.\overline{X_2}.\overline{X_3}.\overline{X_4}.X_5 + X_1.\overline{X_2}.X_3.X_4.X_5 + X_1.X_2.\overline{X_3}.\overline{X_4}.X_5$$

ANY truth-table can be expressed using a **one-hidden-layer MLP** … **Universal Boolean Function**..



*How many **Perceptrons** (max) required in single hidden layer for an N-input function?*

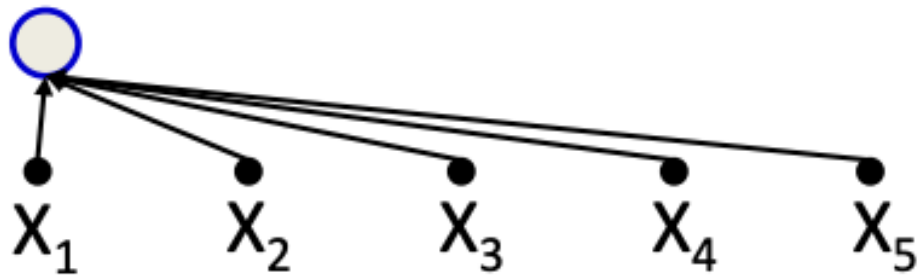
How many layers for a Boolean MLP?

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

Truth table shows *all* input combinations for which output is 1

$$Y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + X_1\bar{X}_2X_3X_4X_5 + X_1X_2\bar{X}_3\bar{X}_4X_5$$



- Expressed in disjunctive normal form

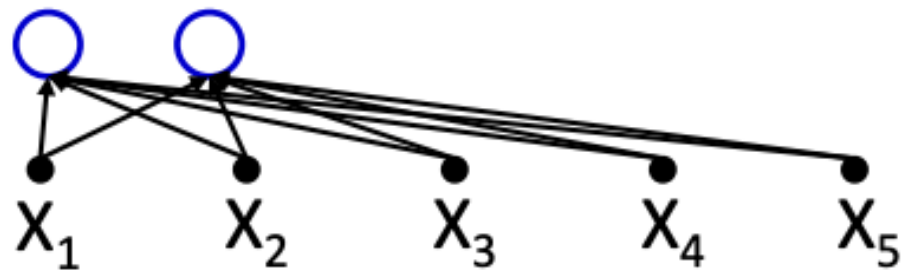
How many layers for a Boolean MLP?

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

Truth table shows *all* input combinations for which output is 1

$$Y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + X_1\bar{X}_2\bar{X}_3X_4X_5 + X_1X_2\bar{X}_3\bar{X}_4X_5$$



- Expressed in disjunctive normal form

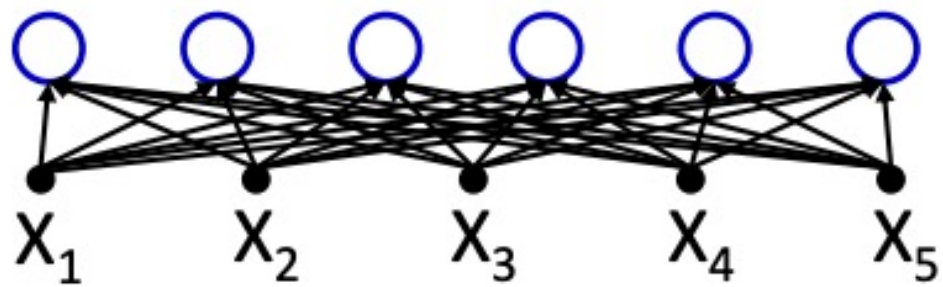
How many layers for a Boolean MLP?

Truth table shows *all* input combinations for which output is 1

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

$$Y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + X_1\bar{X}_2X_3X_4X_5 + X_1X_2\bar{X}_3\bar{X}_4X_5$$



- Expressed in disjunctive normal form

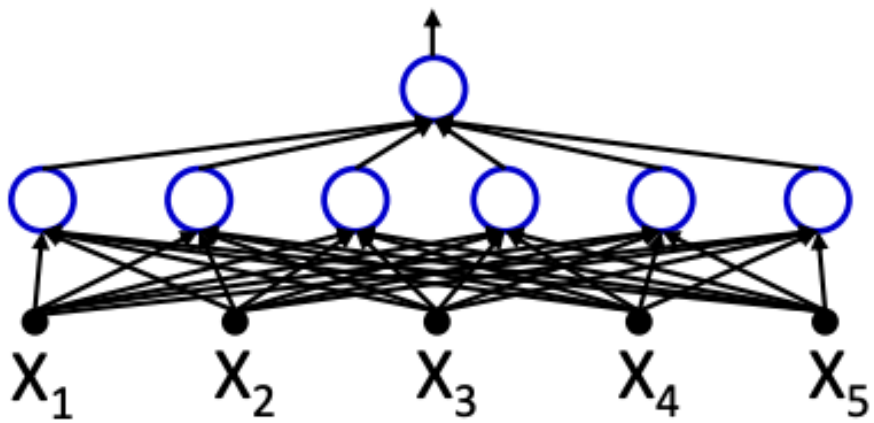
How many layers for a Boolean MLP?

Truth Table

X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

Truth table shows *all* input combinations for which output is 1

$$Y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + X_1\bar{X}_2X_3X_4X_5 + X_1X_2\bar{X}_3\bar{X}_4X_5$$



- Expressed in disjunctive normal form

Perceptrons.. reduction

$$Output = \overline{W}XYZ + \bar{W}X\overline{YZ} + WX\overline{YZ} + W\overline{XYZ} + \bar{W}X\bar{Y}Z + \overline{WX}Y\bar{Z} + W\bar{X}Y\bar{Z}$$

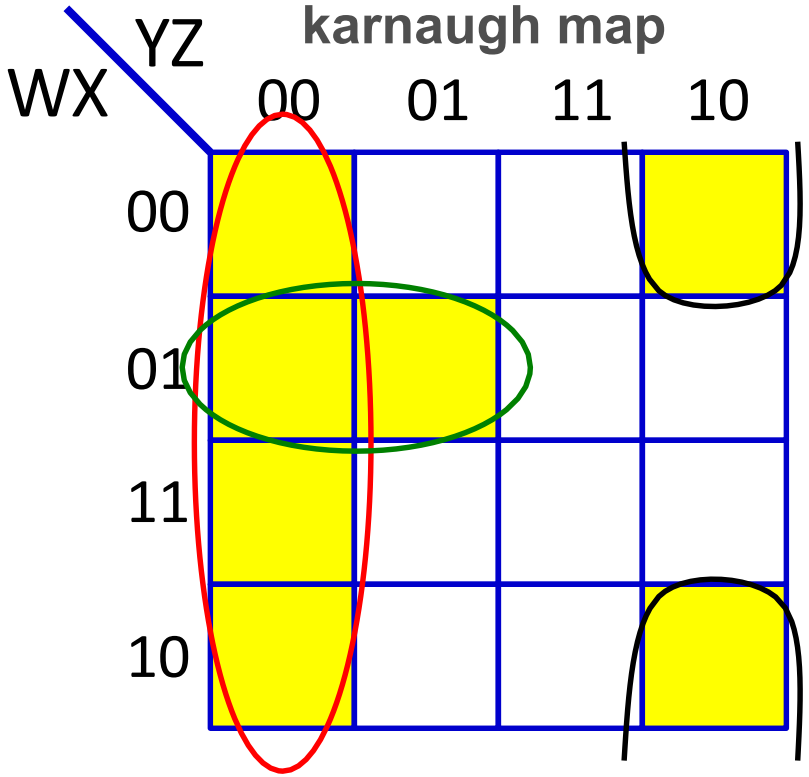
How many perceptrons do we need for the hidden layer needed for this function?

***7 Perceptrons** in **the** hidden layer needed for this function.*
How about with reduction??

$$Output = \overline{W}XYZ + \bar{W}X\overline{YZ} + WX\overline{YZ} + W\overline{XYZ} + \bar{W}X\bar{Y}Z + \overline{WX}Y\bar{Z} + W\bar{X}Y\bar{Z}$$

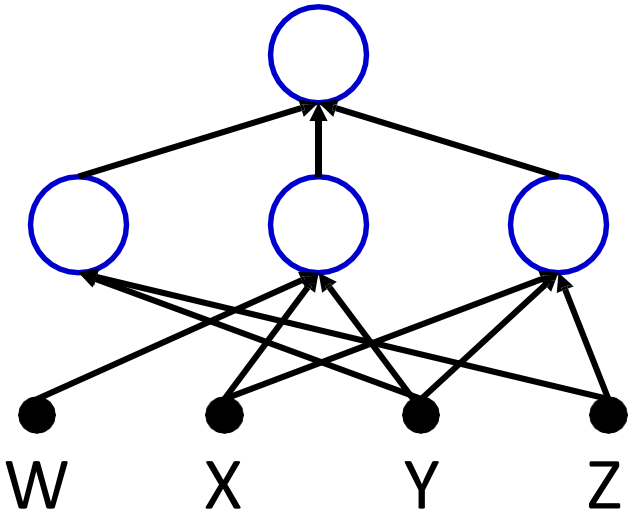
$$Output = \overline{YZ} + \bar{W}X\bar{Y} + \bar{X}Y\bar{Z}$$

Perceptrons.. reduction



Yellow --- 1
White---0

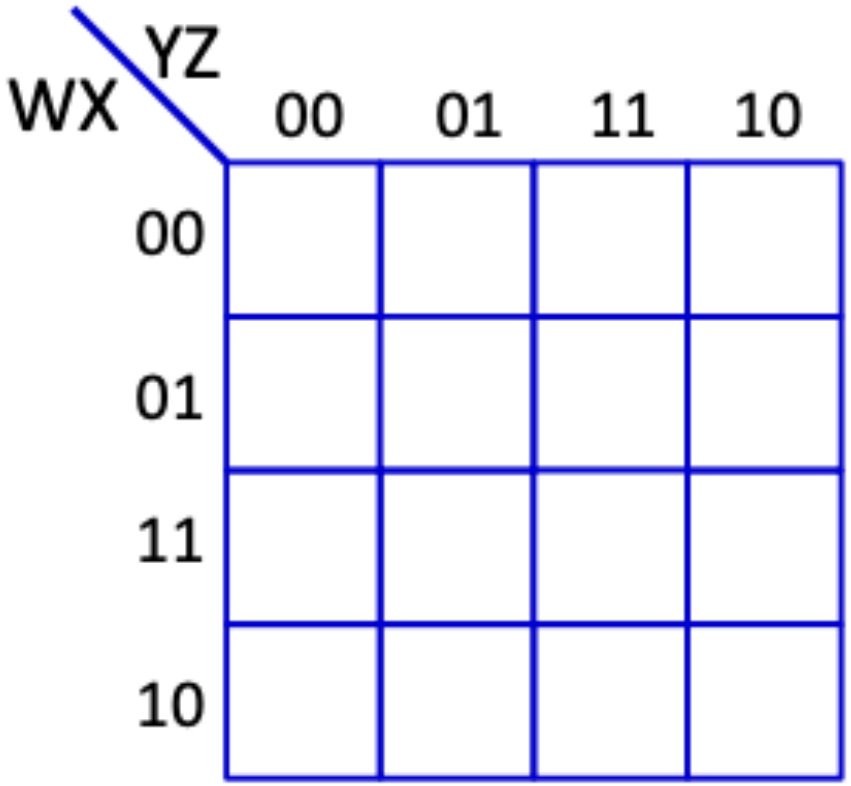
$$Output = \overline{YZ} + \bar{W}X\bar{Y} + \bar{X}Y\bar{Z}$$



***7 Perceptrons** in **the** hidden layer needed for this function (Why 7?).*
How about with reduction??

*ONLY **3 Perceptrons** !*

Largest irreducible DNF?

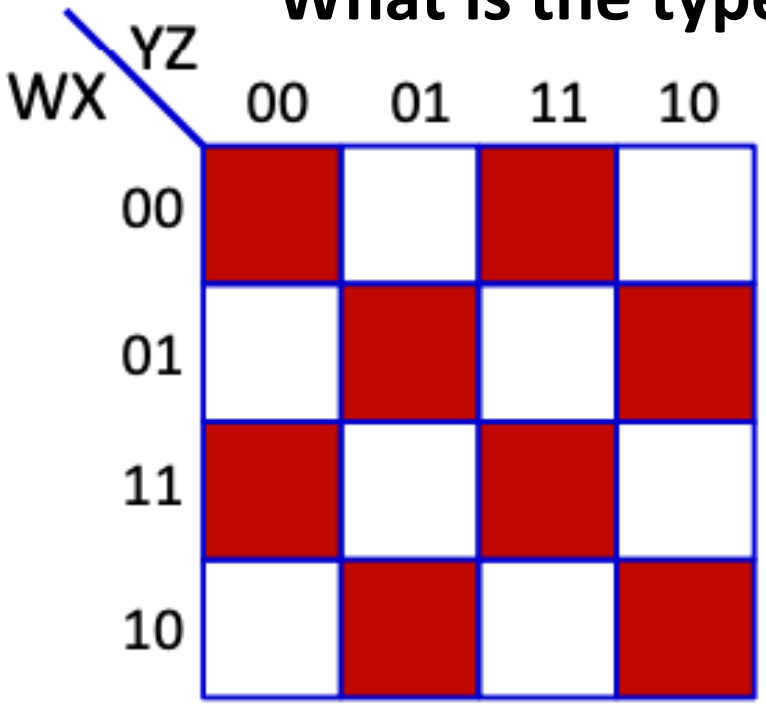


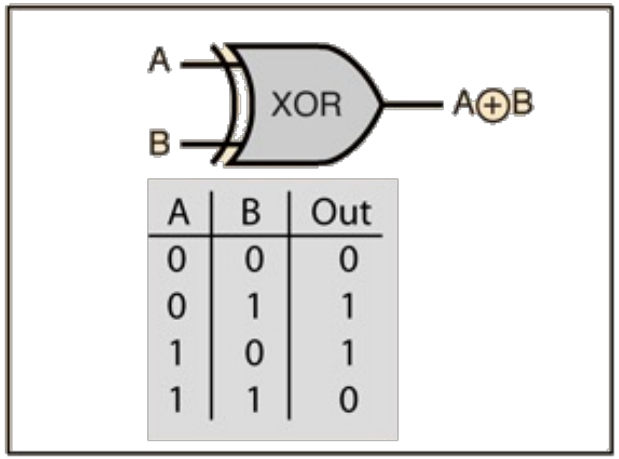
- What arrangement of ones and zeros simply cannot be reduced further?

Largest irreducible DNF?

What is the type of Boolean MLP here?

Red --- 0
White---1

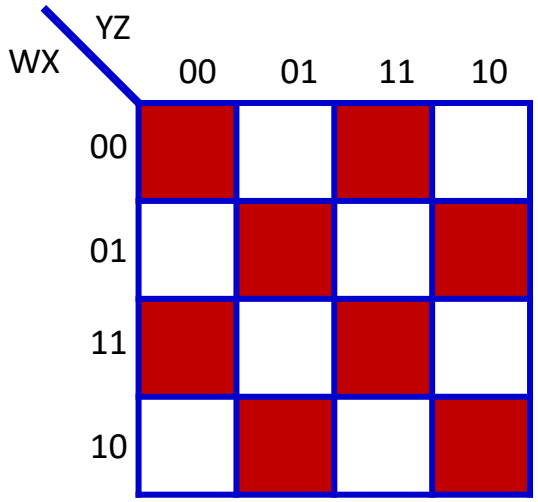




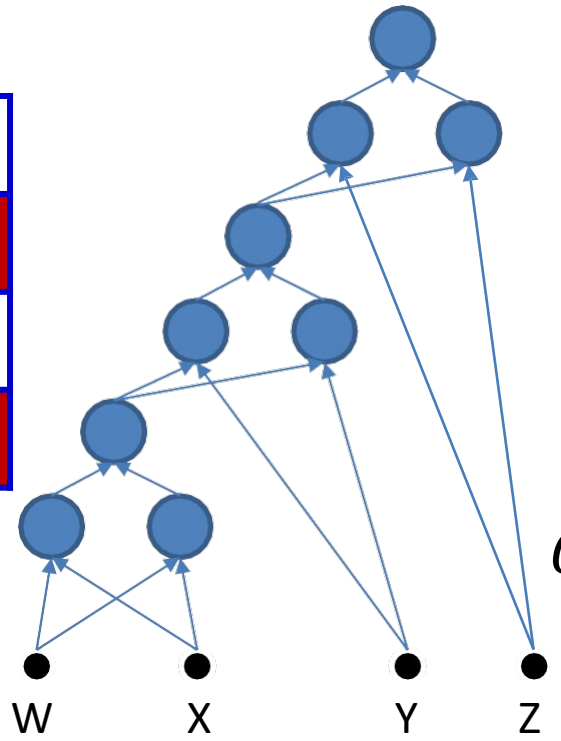
How many neurons in a DNF (one-hidden-layer) MLP for this Boolean function?

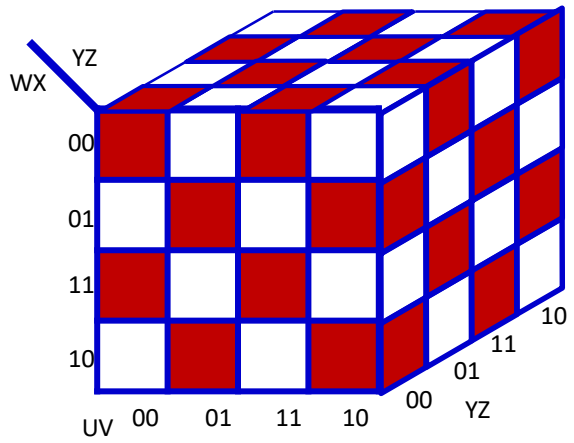
- What arrangement of ones and zeros simply cannot be reduced further?

MLP.. Depth

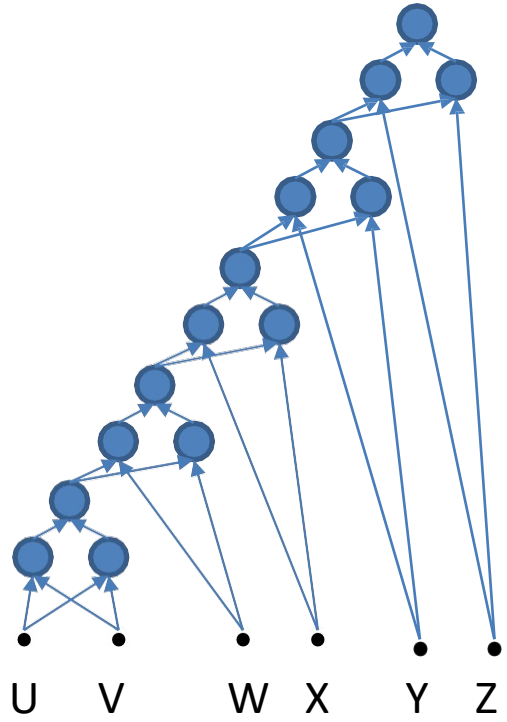


$$O = W \oplus X \oplus Y \oplus Z$$





$$O = U \oplus V \oplus W \oplus X \oplus Y \oplus Z$$



Exponential in ***N!***

These Boolean functions require 2^{N-1} **Perceptrons** in **SINGLE** hidden layer!

But this can be reduced with **multiple** hidden layers!

Generally, the XOR of ***N*** variables will require 3(***N*** -1) **Perceptrons** in a **DEEP Network** …. arranged in **"2 layers per XOR" (**$W \oplus X, Y \oplus Z$ in single layer**)**.. only $2\log_2(N)$ layers

Linear in ***N***!!!

MLP.. need for depth

DEEP Boolean MLPs scale **linearly** with number of inputs.

MLP, a universal Boolean function, represents a given function ONLY if its **sufficiently wide** & **sufficiently deep**.

Reducing number of layers below the “minimum” will result in an **exponentially** sized network to express the function.. *gets **exponentially large** if only one layer is used*.

A network with **fewer than the required** number of neurons cannot model the function.

Generally, **deeper networks** will require far fewer neurons for the same approximation error.

Time Break – 15mins

MLP.. Learning

Brain acquires/learns new knowledge through **experience**.

Parameters to adjust to resolve **learning** **without** changing "*pattern of connections*" are:

- the weights of connections w_{ji} and,
- the learning rule which defines how to adjust the weights of connections to get desirable outputs.

MLP Learning Algorithm

$$w = w + \eta\,(d(x) - y(x))\,x$$

Sequential Learning:

$d(x)$ is the desired output in response to input x

$y(x)$ is the actual output in response to x

Weights are updated when **Perceptron** output is wrong

proven convergence…

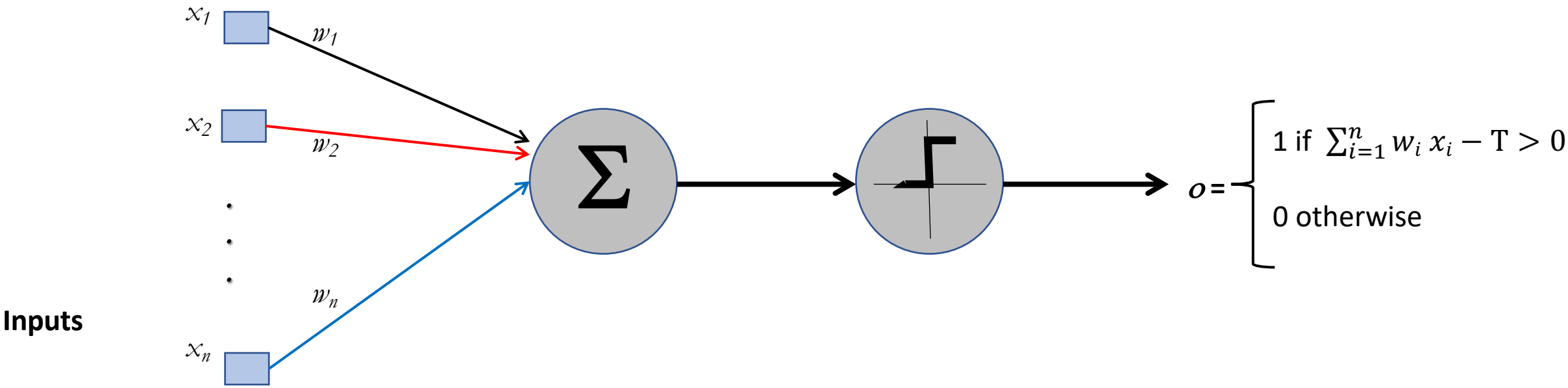
Our brain is not Boolean .. we have real inputs ..

we make non-Boolean inferences/predictions

Perceptrons with real inputs

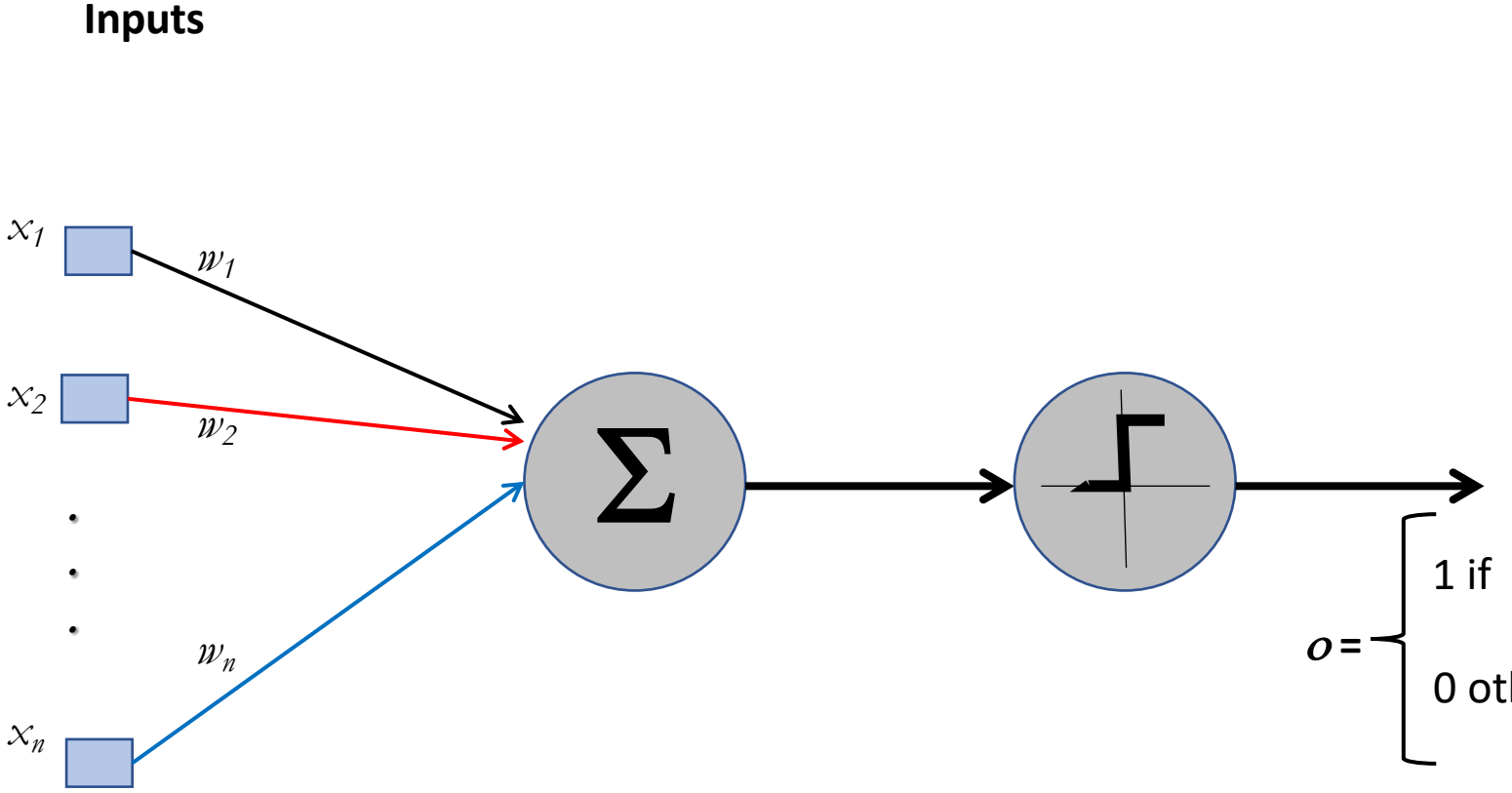
Inputs $x_1 \ldots x_n$ and weights $w_1 \ldots w_n$ are **real valued** :

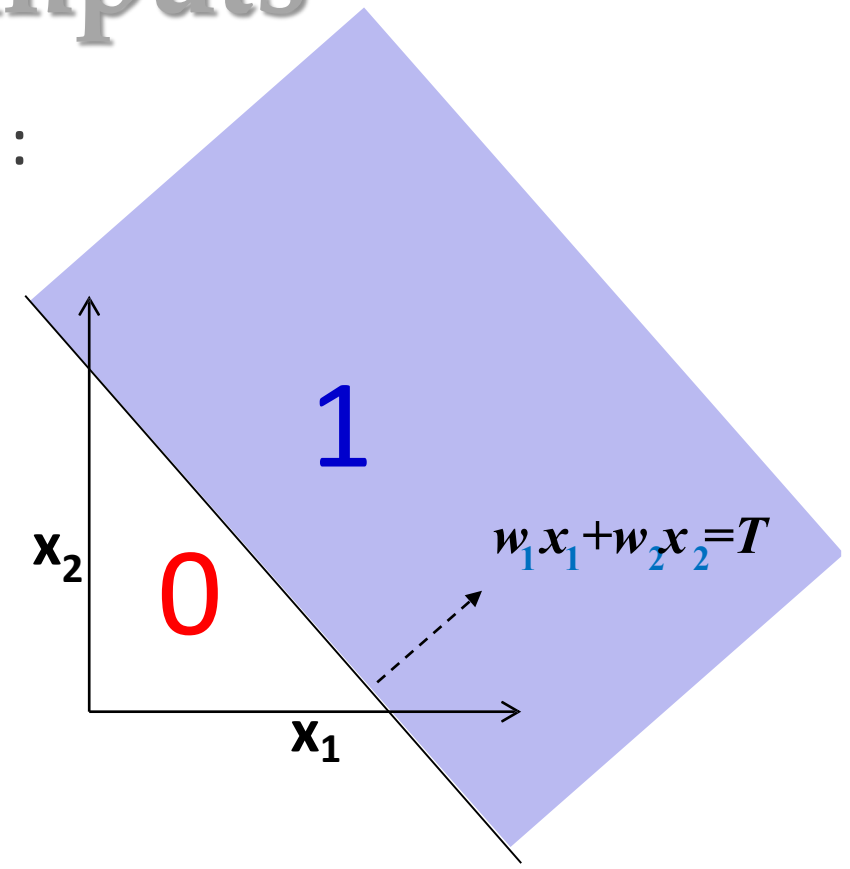
unit "fires" if weighted input exceeds a threshold



Perceptrons with real inputs

Inputs $x_1 \ldots x_n$ and weights $w_1 \ldots w_n$ are **real valued** :
unit "fires" if weighted input exceeds a threshold



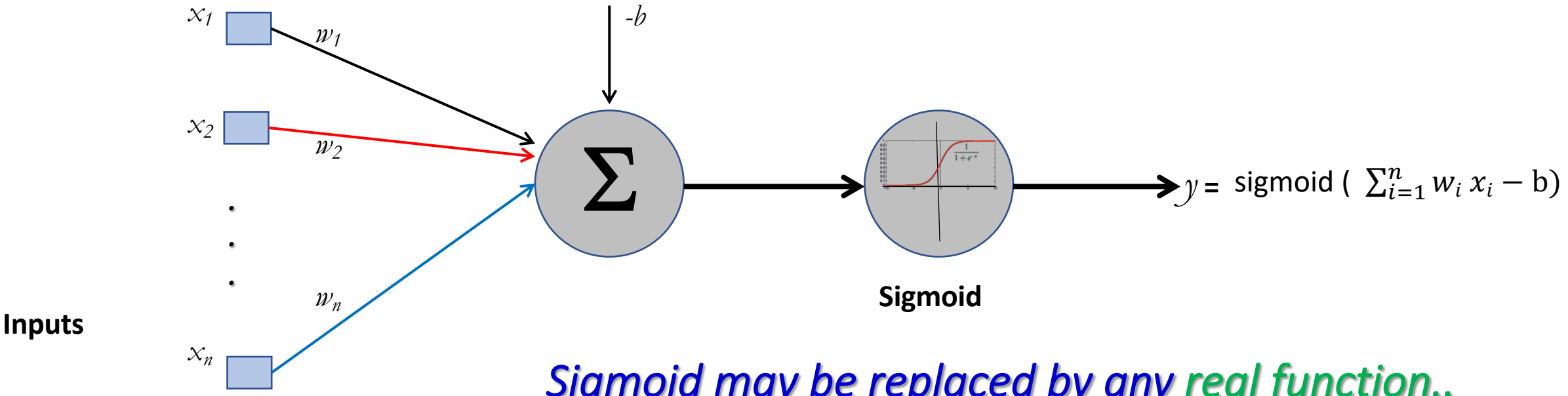


$$o = \begin{cases} 1 \text{ if } \sum_{i=1}^{n} w_i x_i - \mathrm{T} > 0 \\ 0 \text{ otherwise} \end{cases}$$

Perceptrons with real inputs & real output..

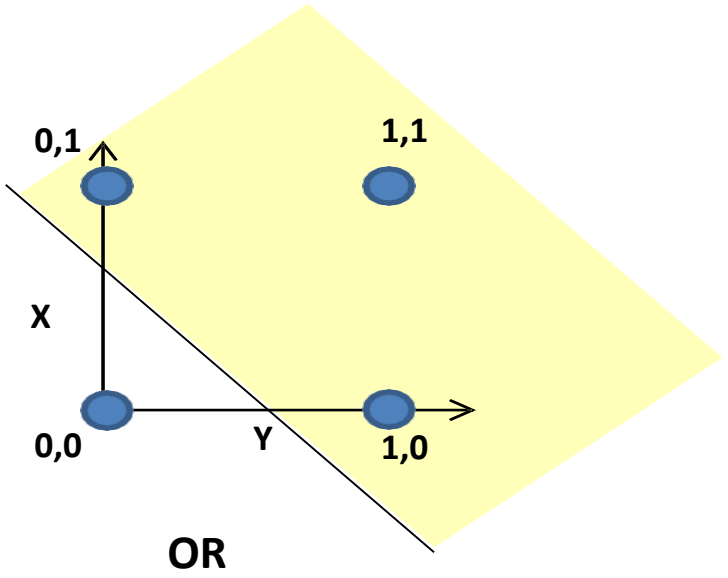
Inputs $x_1 \ldots x_n$ and weights $w_1 \ldots w_n$ are **real valued** and output y can also be **real valued**.

Sometimes viewed as the **"probability of firing"**.

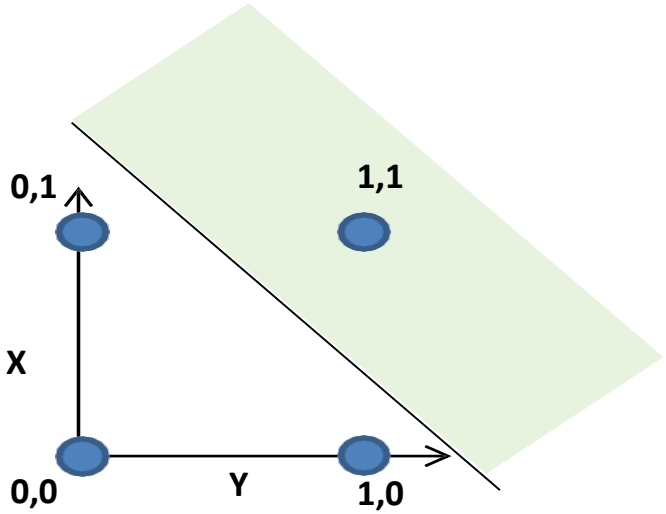


Sigmoid may be replaced by any real function..

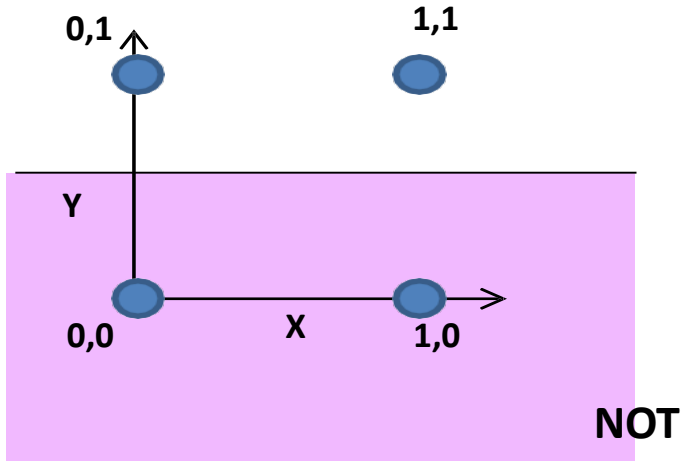
Perceptrons ..Boolean functions



OR



AND

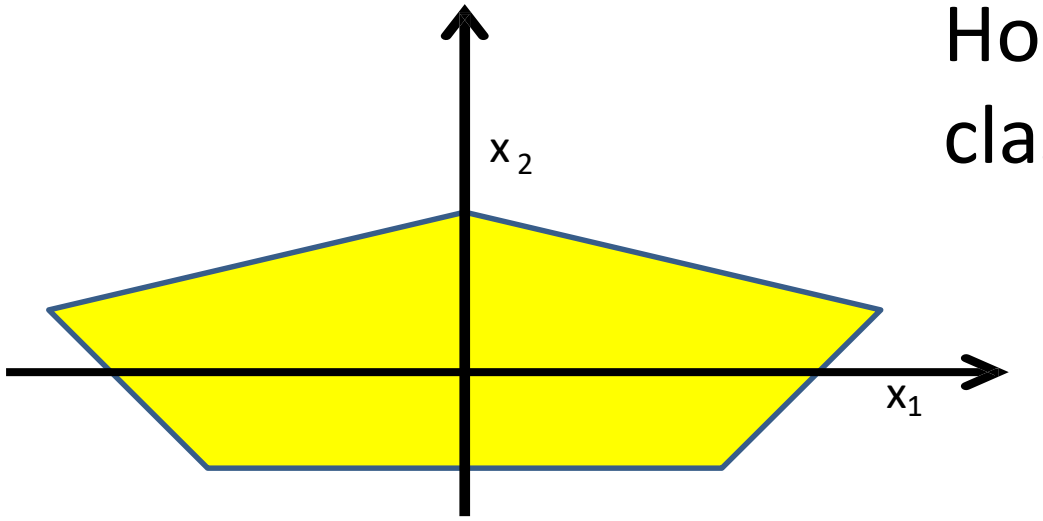


NOT

Such **Perceptrons** are also Linear Classifiers:

– implements Boolean functions.. except for the **XOR**

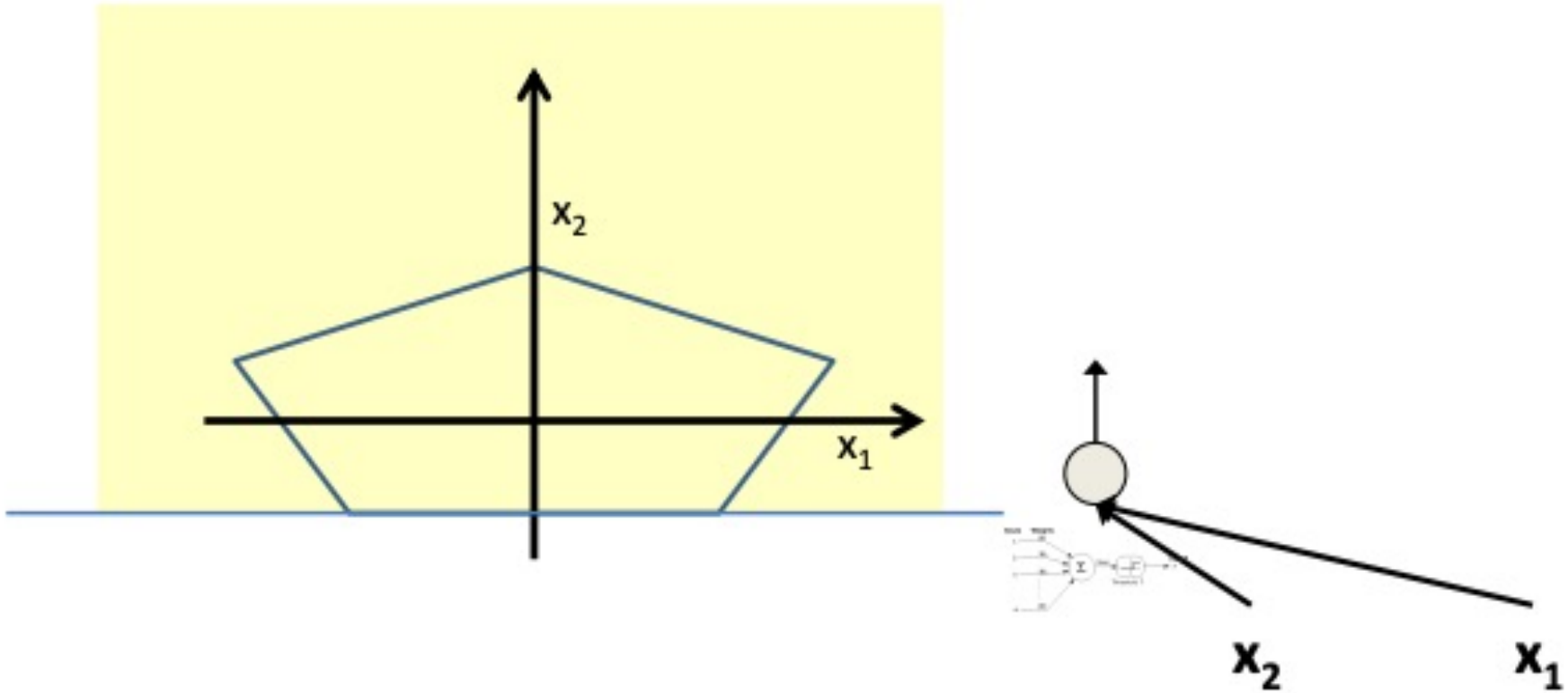
Perceptrons Composing Decision Boundaries



How to get such arbitrary classification “boundaries”?

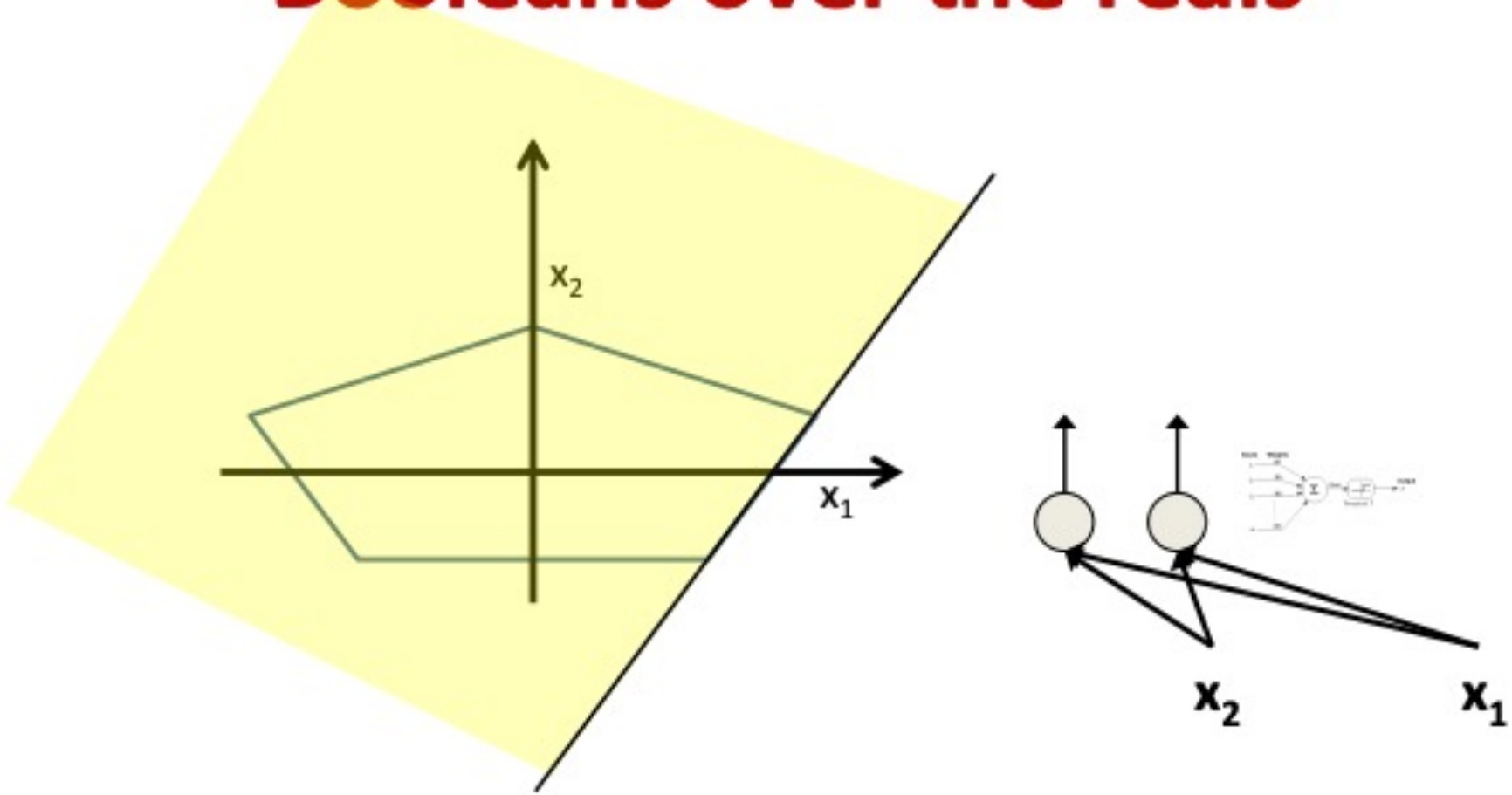
Build a network of single-output Perceptron units a that fires when input is in the shaded area.

Booleans over the reals



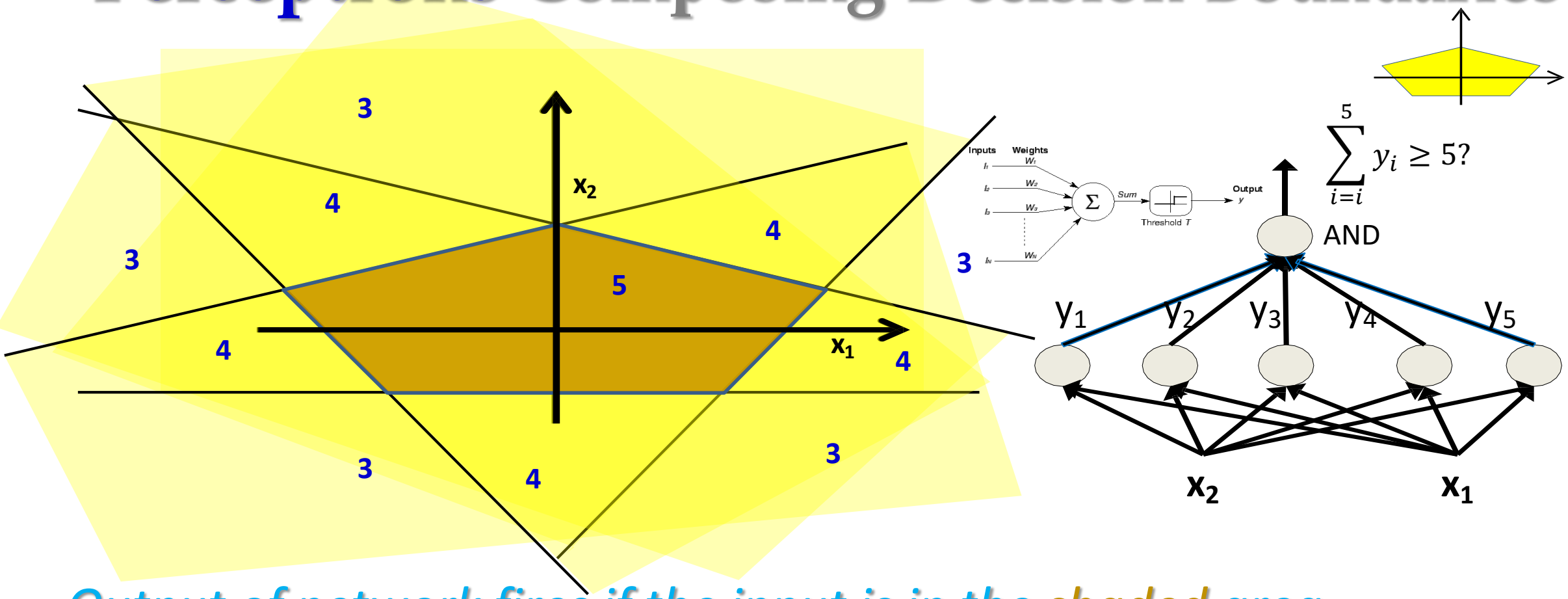
- The network must fire if the input is in the coloured area

Booleans over the reals



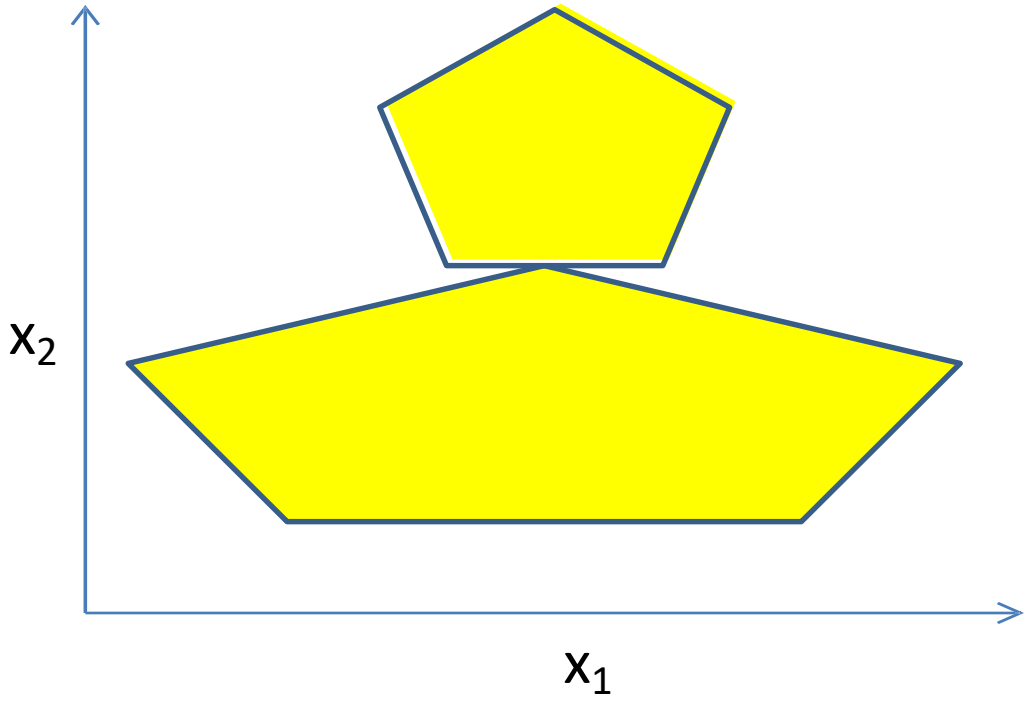
- The network must fire if the input is in the coloured area

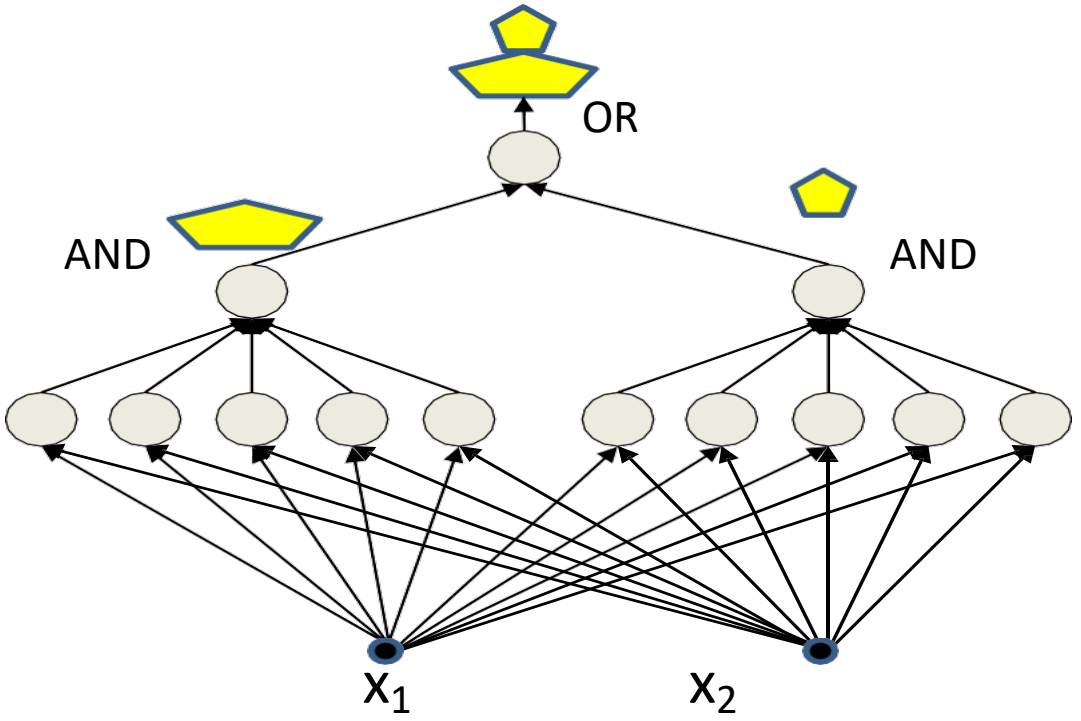
Perceptrons Composing Decision Boundaries



Output of network fires if the input is in the shaded area.

Perceptrons Composing Decision Boundaries

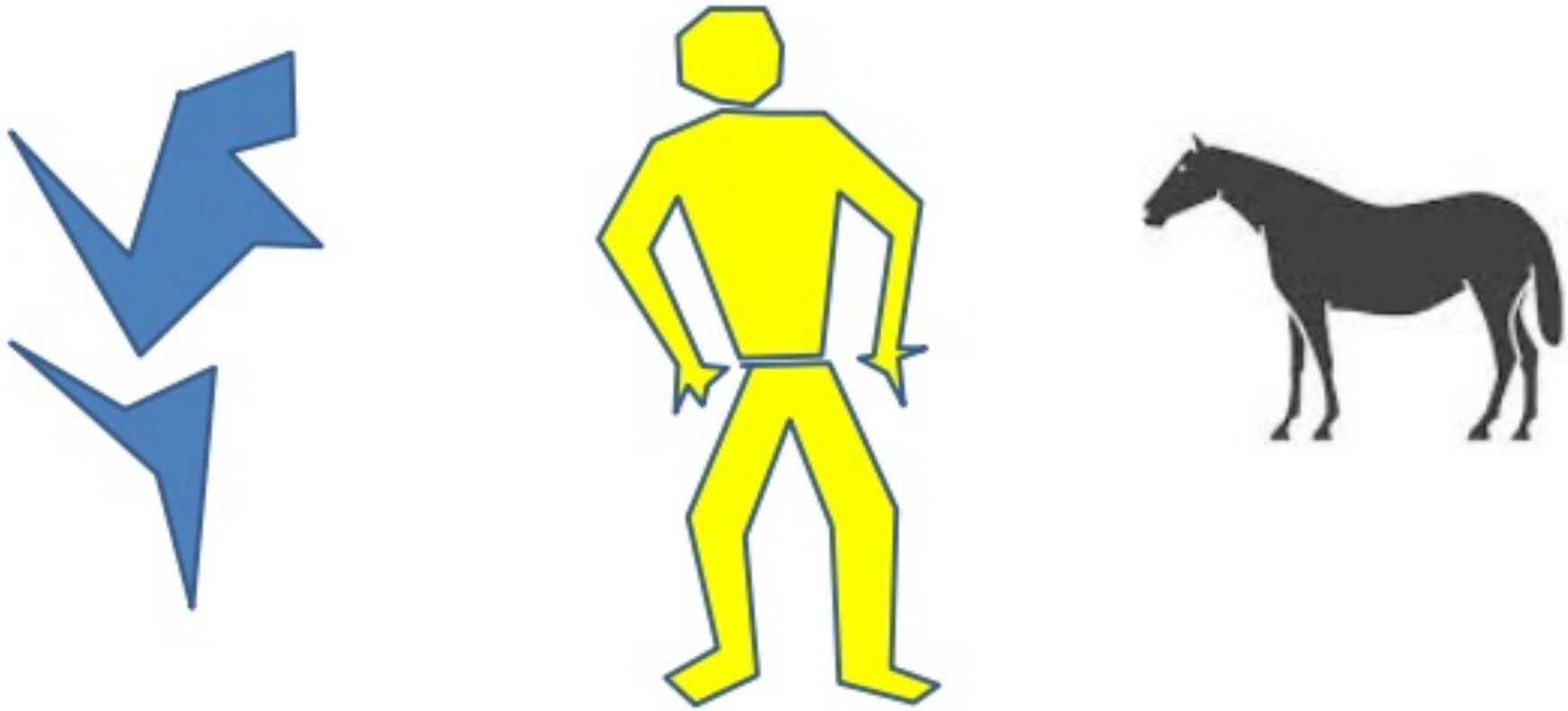




*Need a **THIRD LAYER** to "OR" the two polygon regions.*

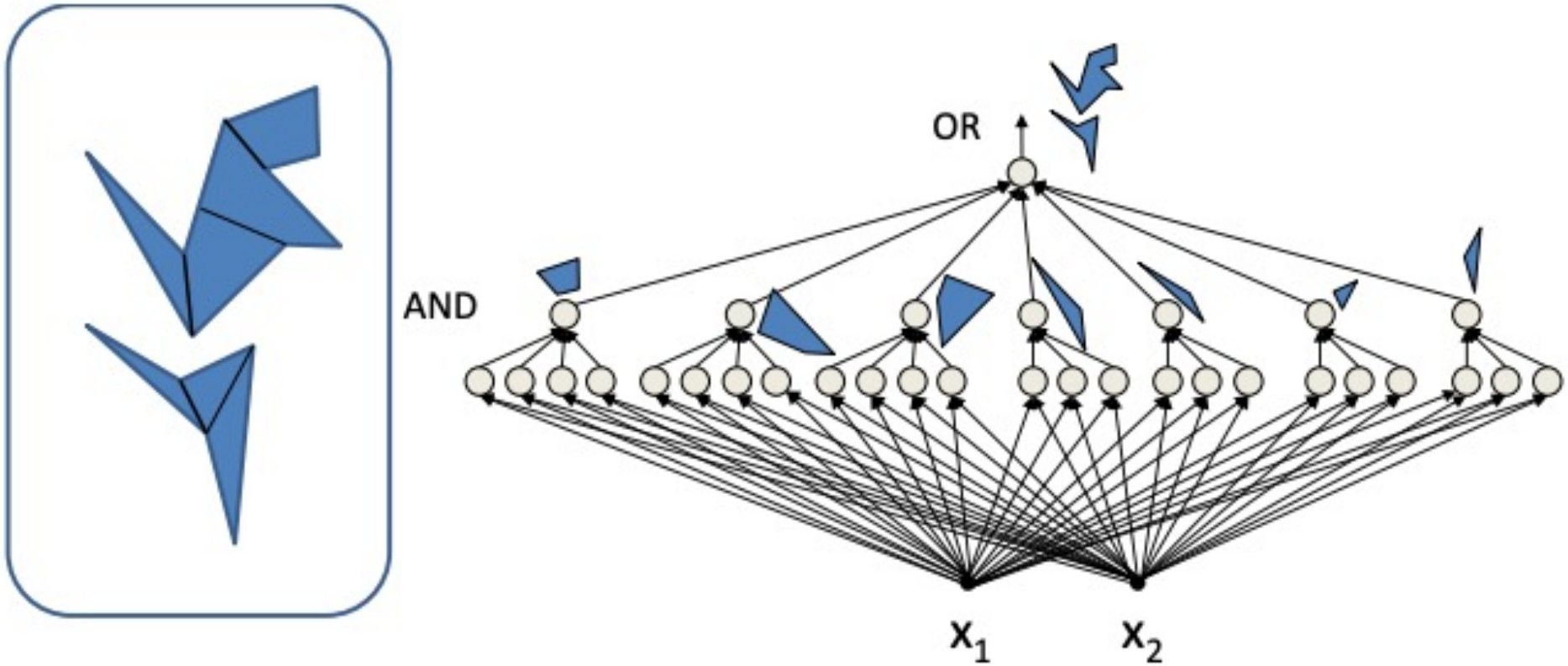
Similarly, even more complex regions can be categorized.

Complex decision boundaries



- Can compose *arbitrarily* complex decision boundaries

Complex decision boundaries



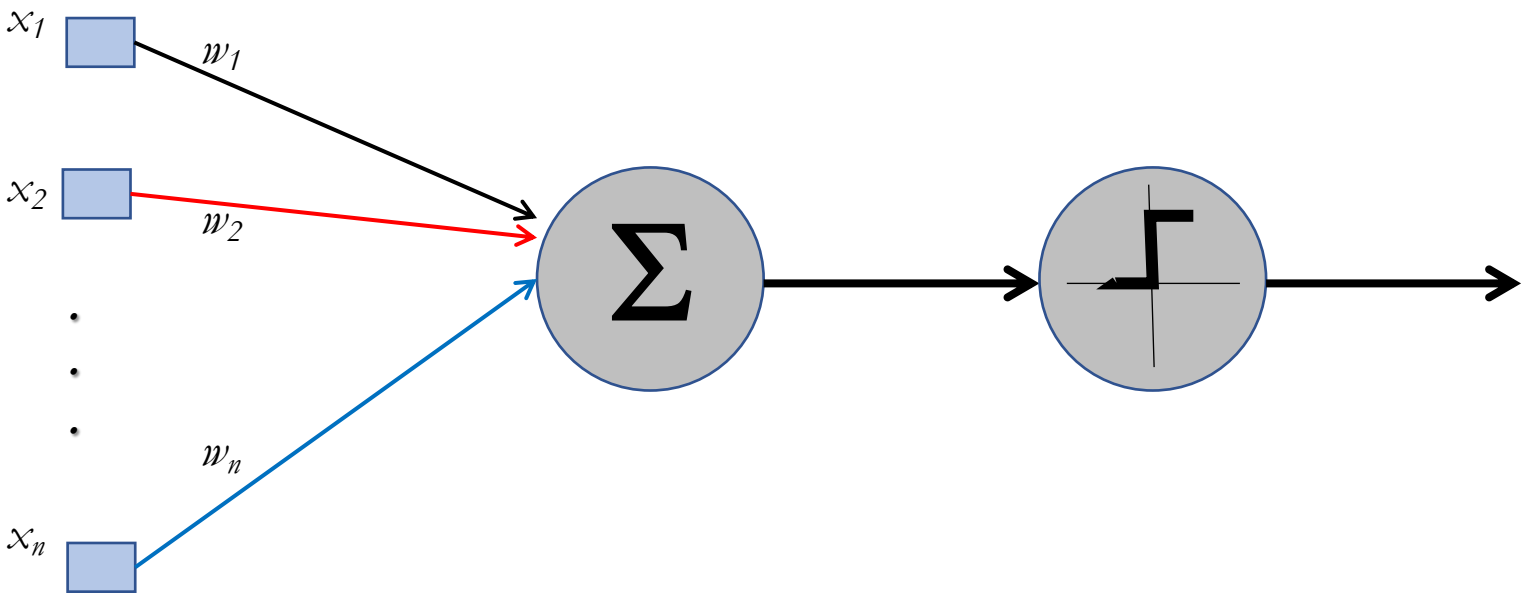
- Can compose *arbitrarily* complex decision boundaries

Complex Decision Boundaries .. Higher Dimensional Space ..

MLPs are **classification engines** … can identify classes in data..

Individual **Perceptrons** are **“feature detectors”** & network **fires** if combination of **detected basic features** matches an “acceptable” pattern for a desired class.

The Weights .. a closer look



$$y = \begin{cases} 1 \text{ if } \sum_{i=1}^{n} w_i\, x_i - \mathrm{T} > 0 \\ 0 \text{ otherwise} \end{cases}$$

$$y = \begin{cases} 1 \text{ if } \boldsymbol{x}^{\mathrm{T}}\boldsymbol{w} > \mathrm{T} \\ 0 \text{ otherwise} \end{cases}$$

Neuron **fires** if inner-product between weights & inputs exceeds a threshold.

The Weights … as a "template"

$$\boldsymbol{X^T W} > \boldsymbol{T}$$

$$\cos\theta > \frac{\boldsymbol{T}}{|\boldsymbol{X}|}$$

$$\theta < cos^{-1}\frac{\boldsymbol{T}}{|\boldsymbol{X}|}$$

Perceptron/Neuron fires if :

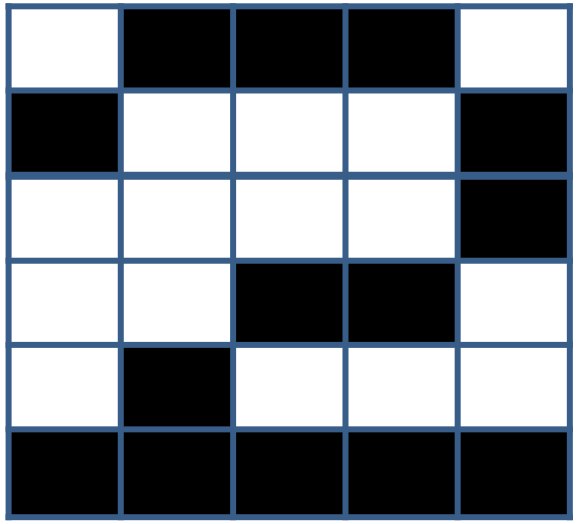
input is within a specified angle of the weight

input vector is close enough to weight vector

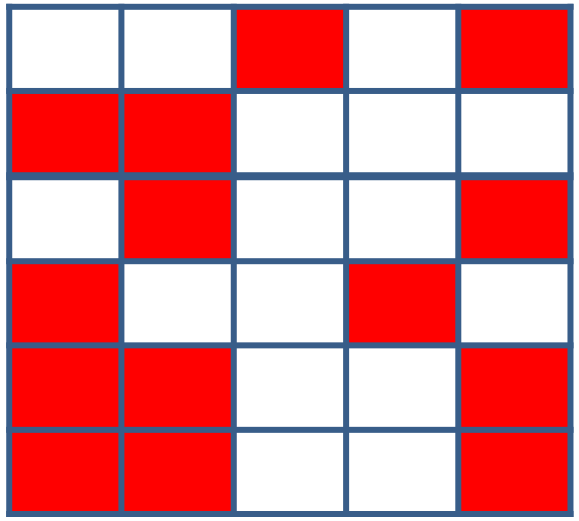
input pattern matches weight pattern closely enough

Perceptron … a Correlation Filter!

W

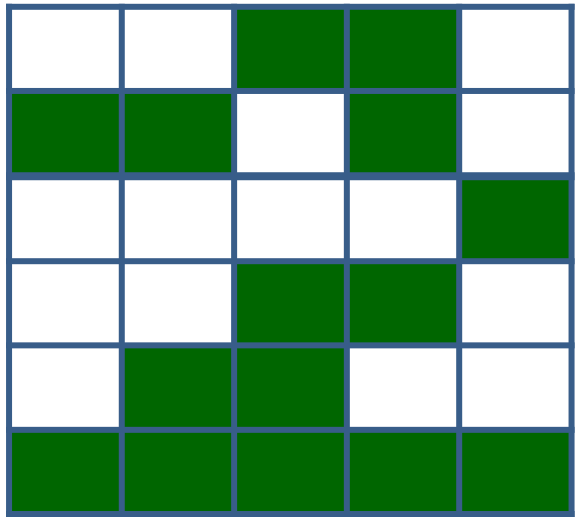


X



Correlation = 0.57

x

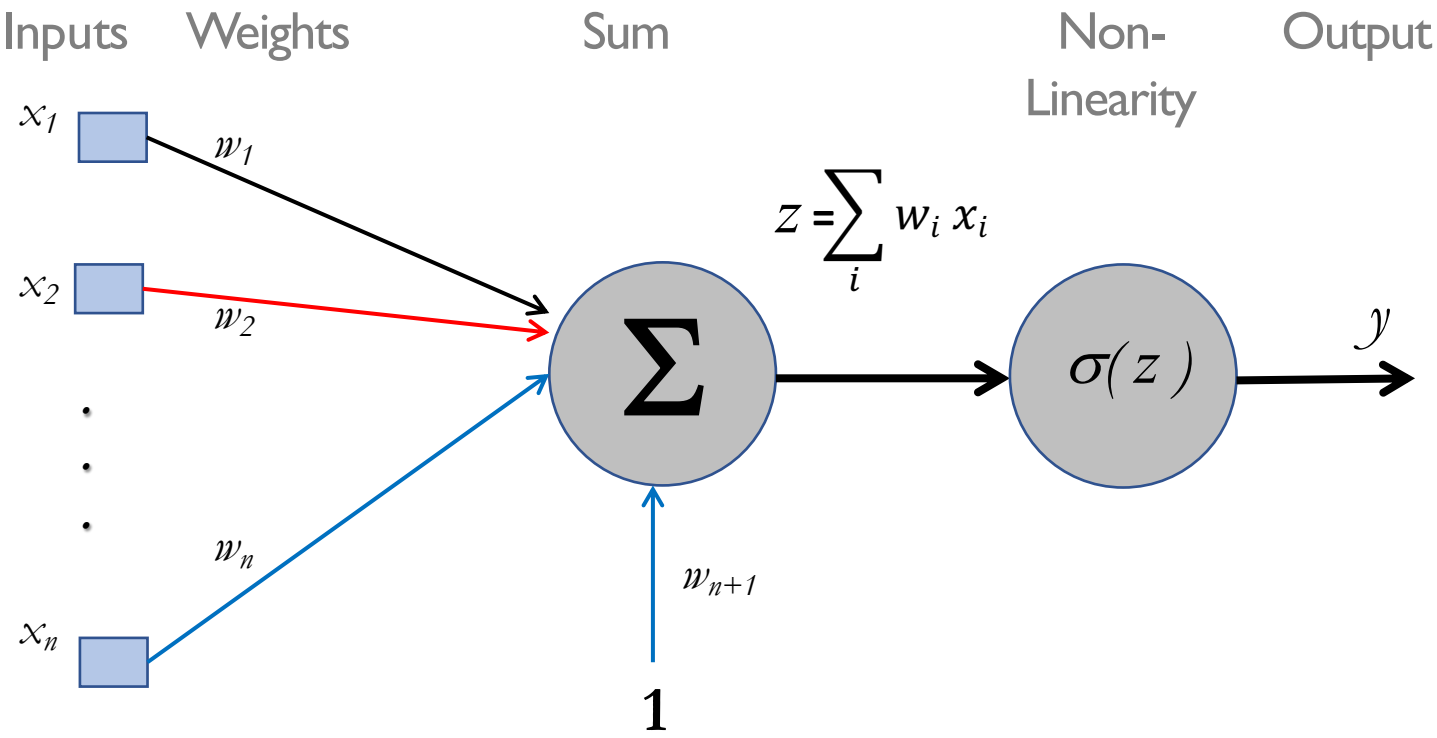


Correlation = 0.82

$$y = \begin{cases} 1 \text{ if } \sum_{i=1}^{n} w_i \, x_i - \mathrm{T} > 0 \\ 0 \text{ otherwise} \end{cases}$$

FIRE: if correlation between weight pattern & inputs exceeds a threshold.

Perceptron.. again

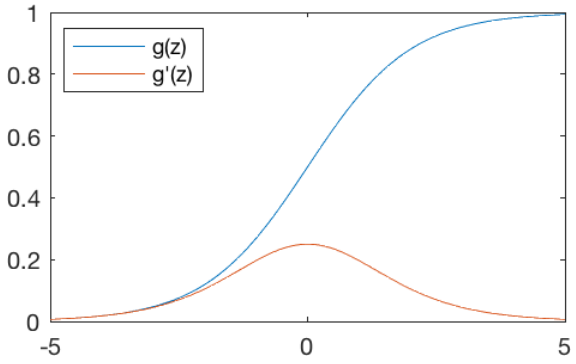


Bias viewed as weight of an input set to 1..

Non-linear activation Function $\sigma(z)$ ..

Common Activation Functions

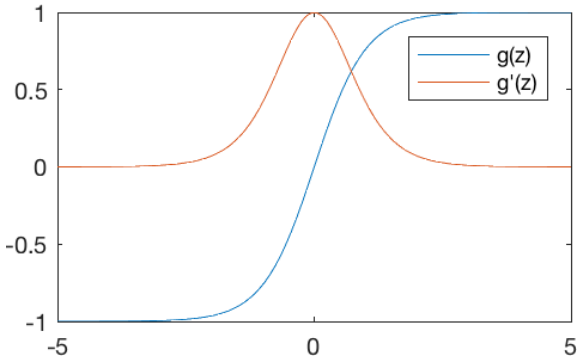
Sigmoid Function



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

$$g'(z) = g(z)\,(1 - g(z))$$

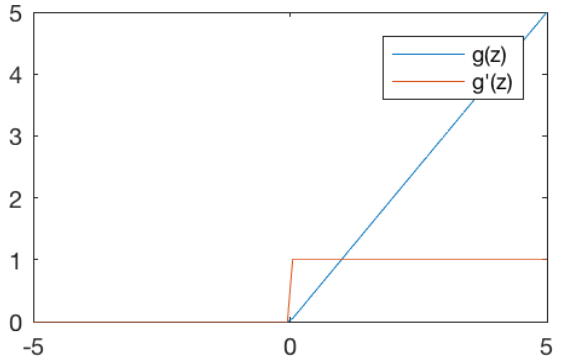
Hyperbolic Tangent



$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$$g'(z) = 1 - g(z)^2$$

Rectified Linear Unit (ReLU)



$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1 & z > 0 \\ 0 & \text{otherwise} \end{cases}$$

Sigmoids produce a value bet 0 & 1 are commonly used .. considered to represent probabilities.

ReLU are popular as simple to implement .. piece-wise linear..

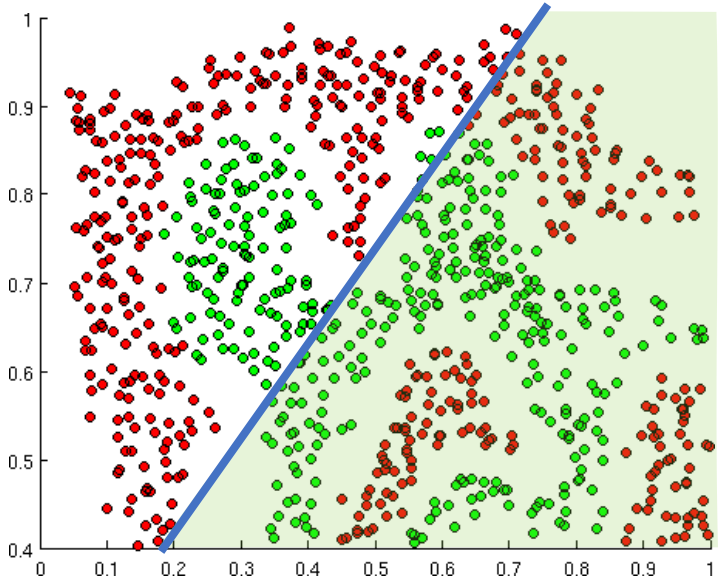
Importance of Activation Functions

Activation functions introduce non-linearities into the network.

How to design a Neural Network to distinguish green vs red points?

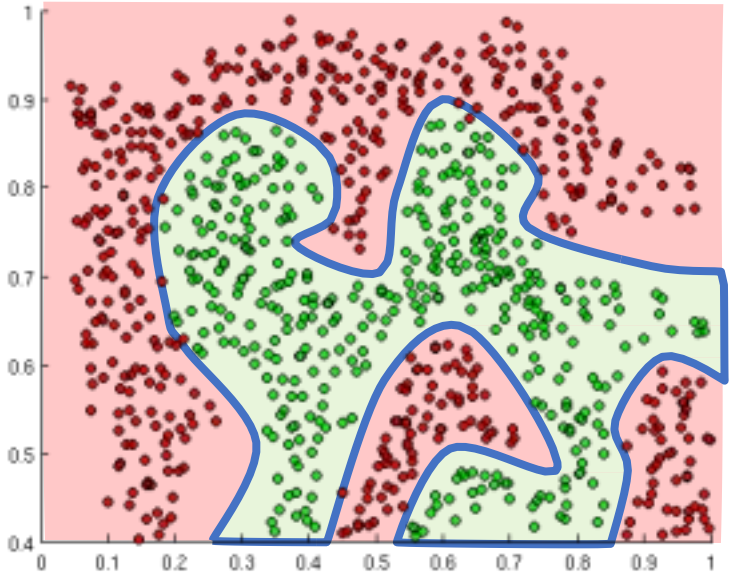
Importance of Activation Functions

Linear Activation functions produce **linear** decisions irrespective of network size.



$$f(W_1)f(W_2)f(W_3) = \tilde{f}\ (W_1W_2W_3)$$

Non-linearities allows approximation of arbitrarily complex functions

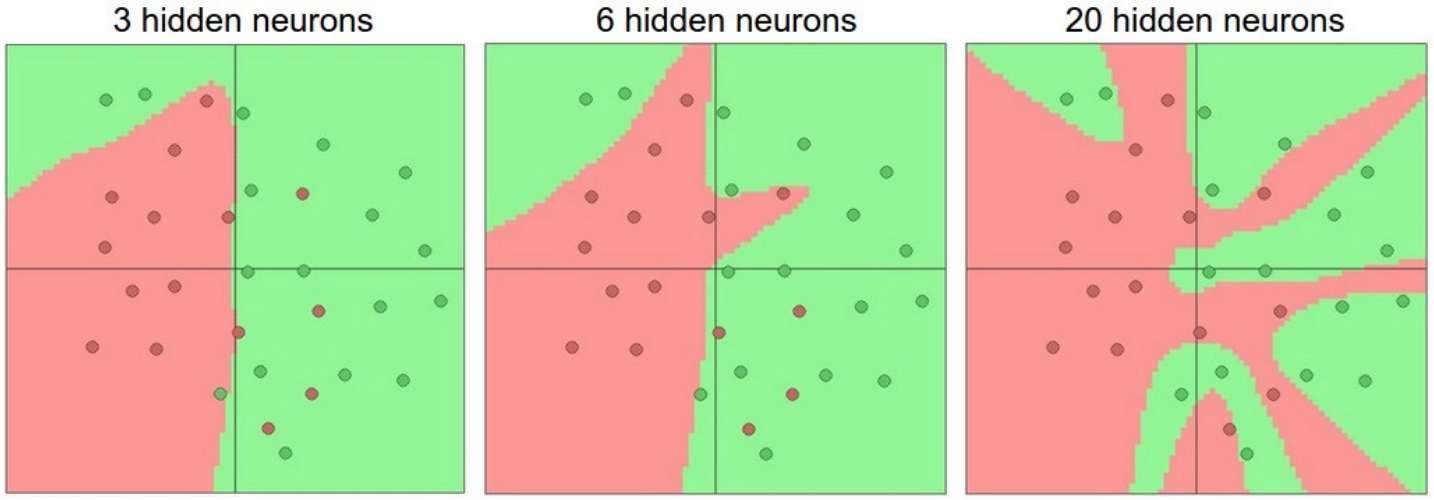


Single Hidden Layer Neural Network

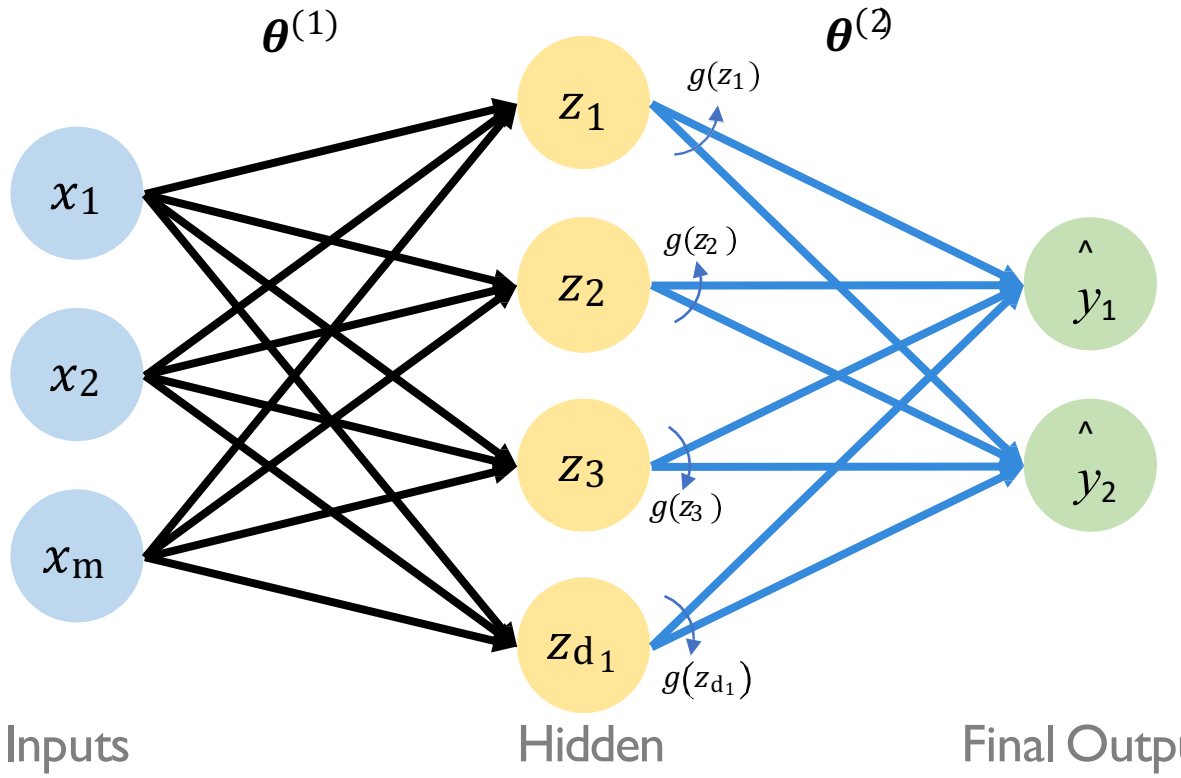
Introducing a ***hidden layer*** of perceptrons computing linear combinations of inputs followed by a *nonlinearity:*

→ gives a *universal function approximator*

but hidden layer may need to be huge.



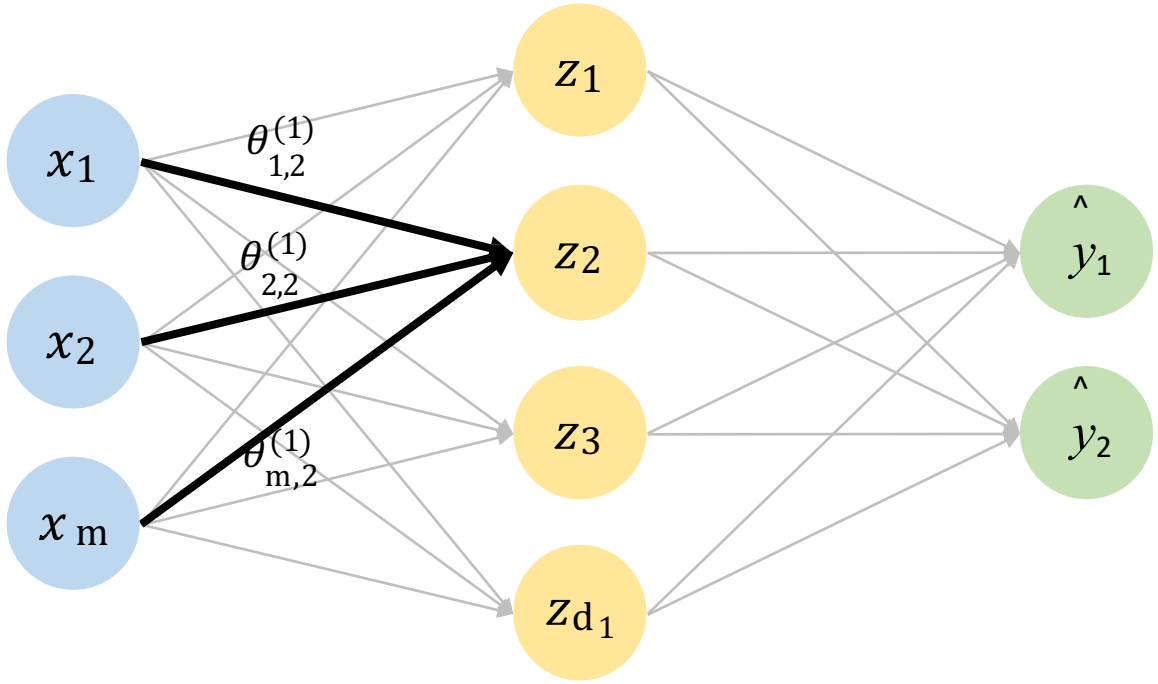
Single Hidden Layer Neural Network



$$z_i = \theta_{0,i}^{(1)} + \sum_{j=1}^{m} x_j \theta_{j,i}^{(1)}$$

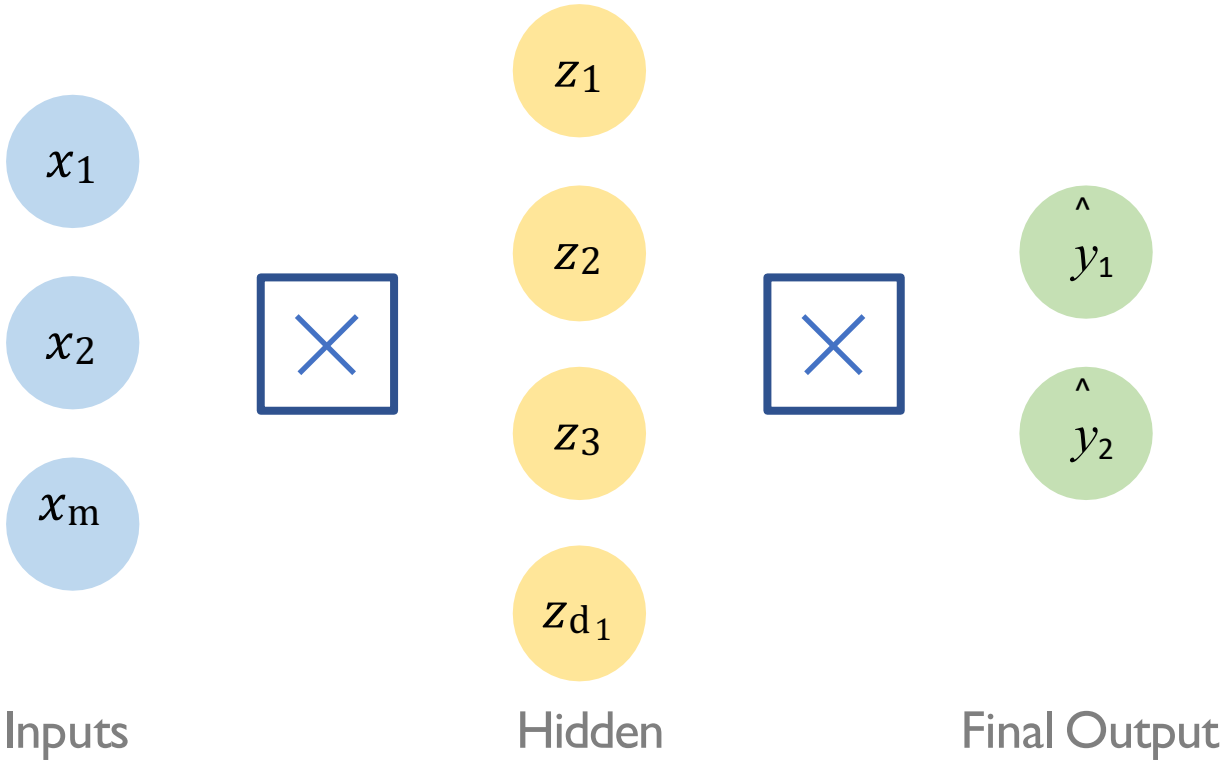
$$\hat{y}_i = \theta_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j \theta_{j,i}^{(2)}$$

Single Hidden Layer Neural Network

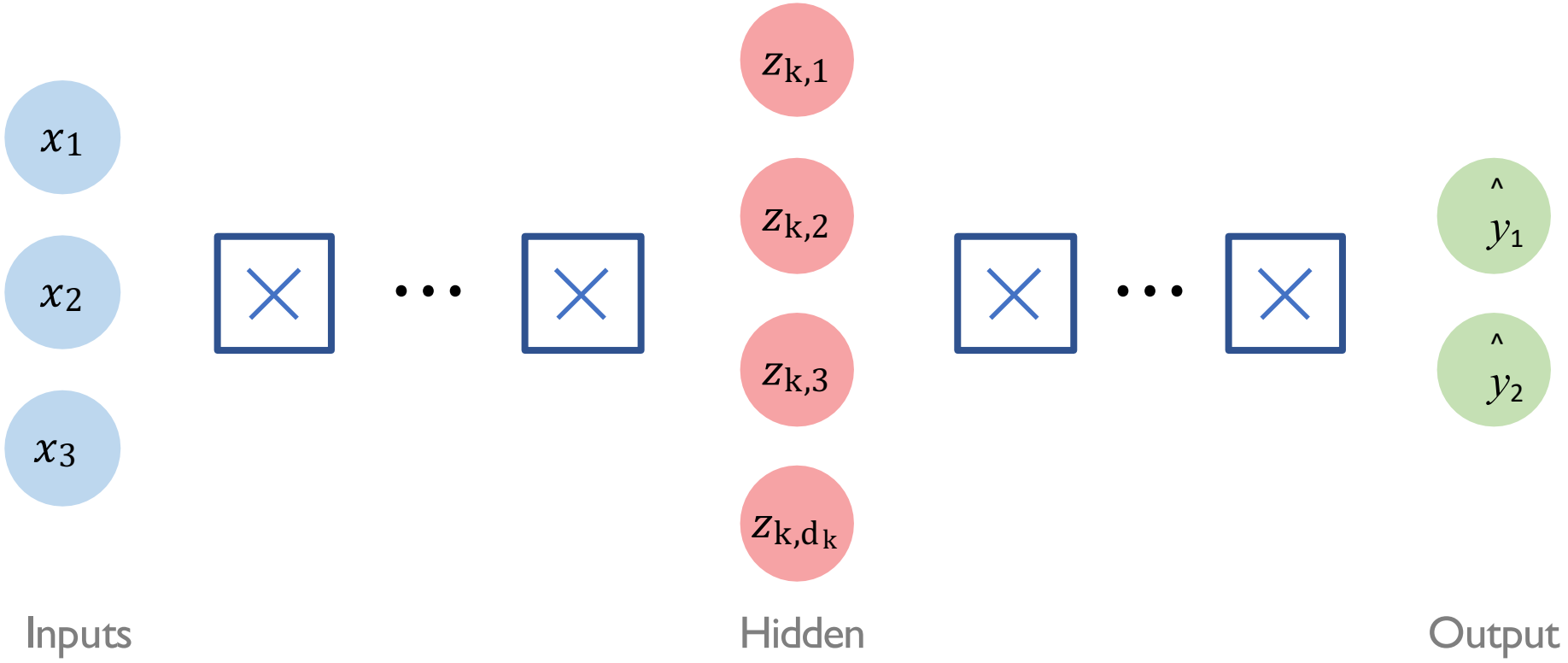


$$z_2 = \theta^{(1)}_{0,2} + \sum_{j=1}^{m} x_j \theta^{(1)}_{j,2} = \theta^{(1)}_{0,2} + x_1 \theta^{(1)}_{1,2} + x_2 \theta^{(1)}_{2,2} + x_m \theta^{(1)}_{m,2}$$

Multi-Output Perceptron



Deep Neural Network



$$z_{k,i} = \theta_{0,i}^{(k)} + \sum_{j=1}^{d_{k-1}} g(z_{k-1,j})\theta_{j,i}^{(k)}$$

Deep Neural Network

Learns a ***feature hierarchy***.

Each layer extracts features from output of previous layer

All layers are trained jointly.

Training
Neural Networks

Loss Optimization

Objective: *to find network weights that achieve the lowest **loss***

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{argmin} \frac{1}{n} \sum_{i=0}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{\theta}\right), y^{(i)}\right)$$

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, J(\boldsymbol{\theta})$$

$$\boldsymbol{\theta} = \left\{\theta^{(0)}, \theta^{(1)}, \cdots\right\}$$

"Loss" informs how the network is performing…
*Objective is minimize the loss **over entire training set**..*
*i.e., find the **set of weights** θ that minimizes the loss!*

imagine a "2D" landscape for two weights θ

…

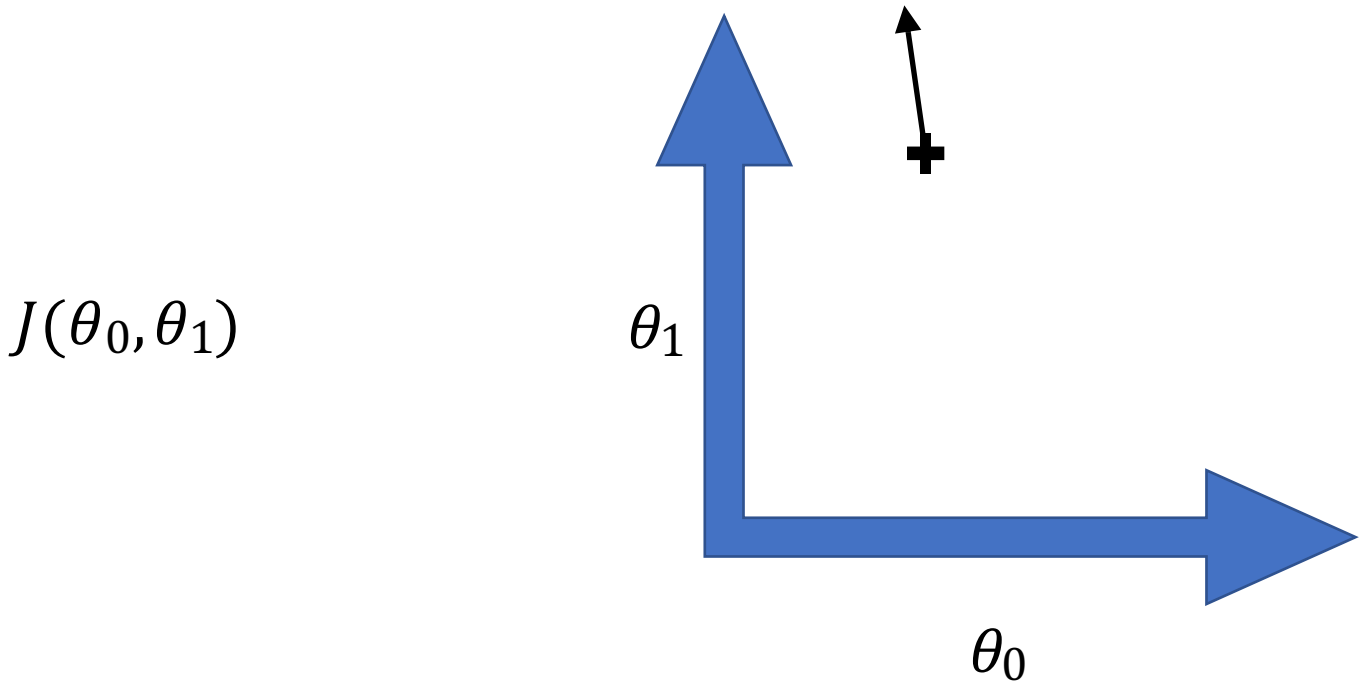
Loss Optimization

"Loss" is a function of the network weights!

Randomly pick an initial (θ_0, θ_1)

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} J(\boldsymbol{\theta})$$

Compute gradient $\frac{\delta J(\boldsymbol{\theta})}{\delta \boldsymbol{\theta}}$ *shows how "loss" is changing for each of the weights over training set..*



Loss Optimization

"Loss" is a function of the network weights!

Randomly pick an initial (θ_0, θ_1)

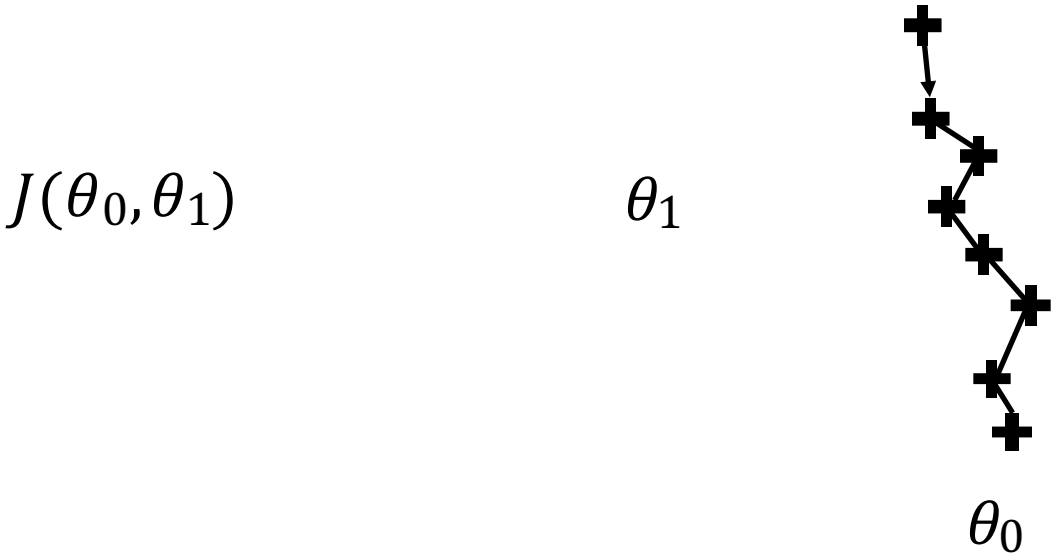
$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} J(\boldsymbol{\theta})$$

Compute gradient $\frac{\delta J(\boldsymbol{\theta})}{\delta \boldsymbol{\theta}}$

Take small step in opposite direction of gradient

Gradient Descent!

Repeat until convergence *when lowest point is reached*



Convergence of Perceptron update rule

Linearly separable data: converges to a perfect solution.

is solution unique?

Non-separable data: converges to **a minimum-error** solution.

Gradient Descent

Algorithm

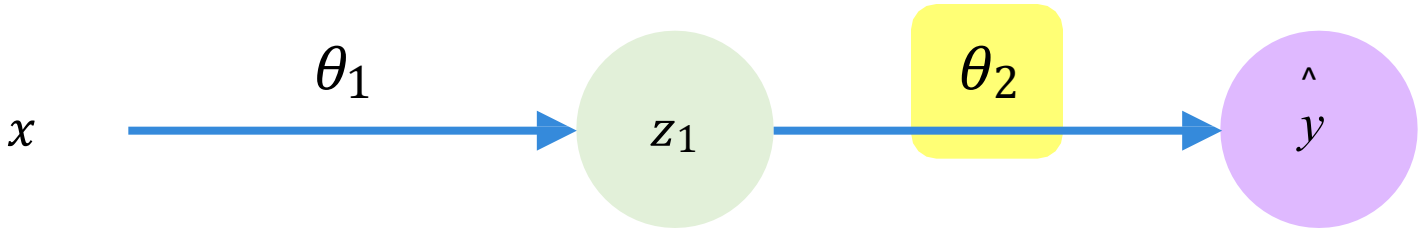
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
   3. Compute gradient $\frac{\delta J(\boldsymbol{\theta})}{\delta \boldsymbol{\theta}}$ *η : learning rate ..*
   4. Update weights $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \frac{\delta J(\boldsymbol{\theta})}{\delta \boldsymbol{\theta}}$ *..update rule: follow negative gradient..*
5. Return weights..

*computing the gradient… **a major issue** in NNs.*
Consider a simple NN to illustrate.. *1 hidden unit.. 1 output unit*

Gradient Descent.. Backpropagation

"Loss" is a function of the network weights!

$J(\boldsymbol{\theta})$

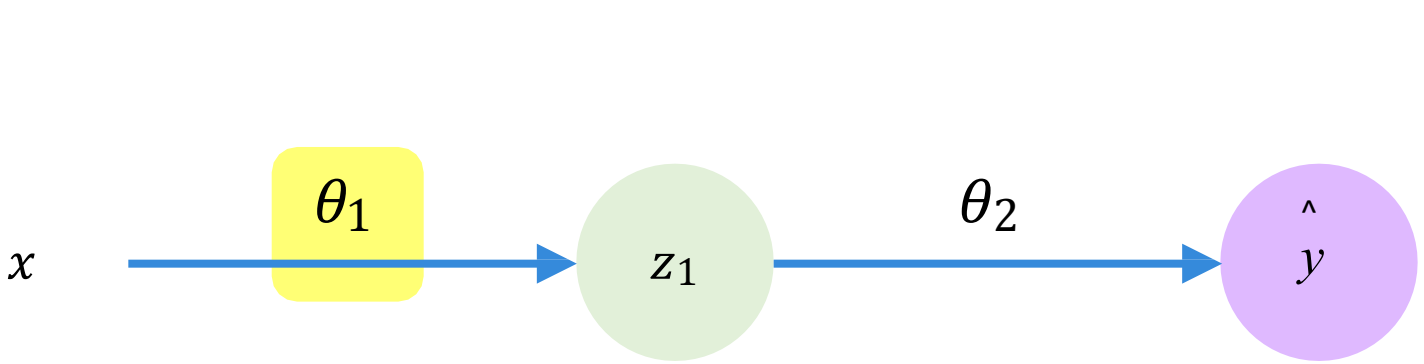


how do small changes in a weight say θ_2 affects final loss $J(\boldsymbol{\theta})$…

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_2} = \frac{\partial J(\boldsymbol{\theta})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial \theta_2}$$

..chain rule..

Gradient Descent.. Backpropagation



This is repeated for every weight using gradients from later layers.

..now the previous layer >>

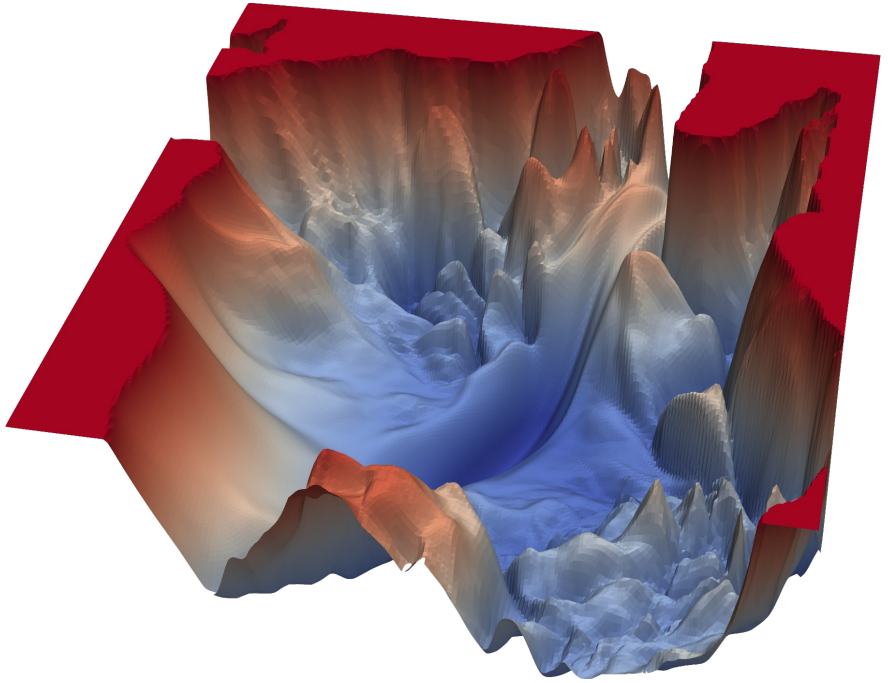
$$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_1} = \frac{\partial J(\boldsymbol{\theta})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial \theta_1}$$

$$= \frac{\partial J(\boldsymbol{\theta})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial \theta_1}$$

Learning Rate..

In real situations, training NNs is very complex.

The "Loss landscape" is usually extremely non-convex and so probability of getting stuck in local minima is very high…



"Visualizing the loss landscape of neural nets". Dec 2017.

Learning Rate..

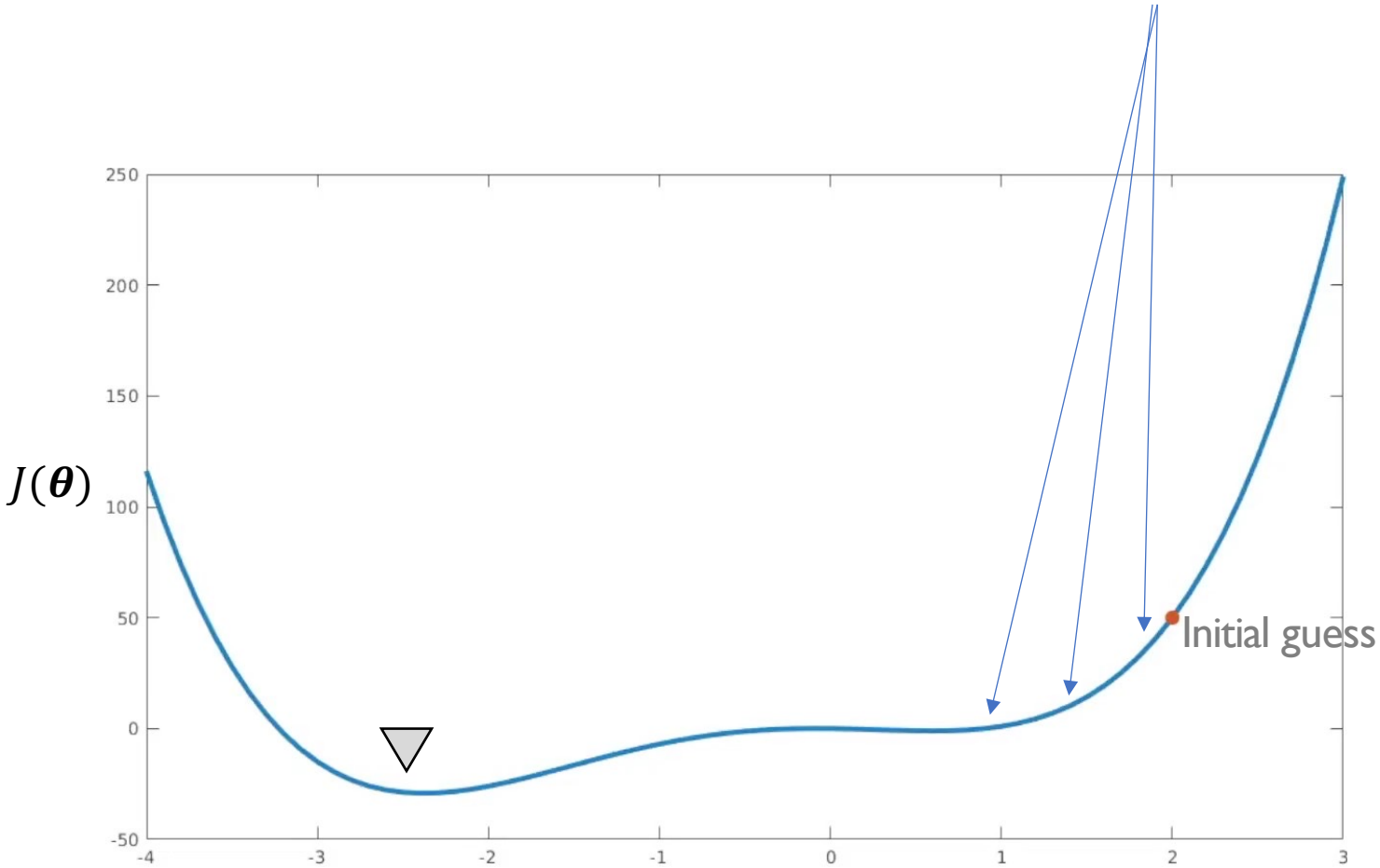
In real situations, training NNs is very complex.

The "Loss landscape" is usually extremely non-convex and so probability of getting stuck in local minima is very high…

Adjusting the learning rate η : $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \frac{\delta J(\boldsymbol{\theta})}{\delta \boldsymbol{\theta}}$ determines how large a step to take in the direction of the gradient .. but challenging to set…

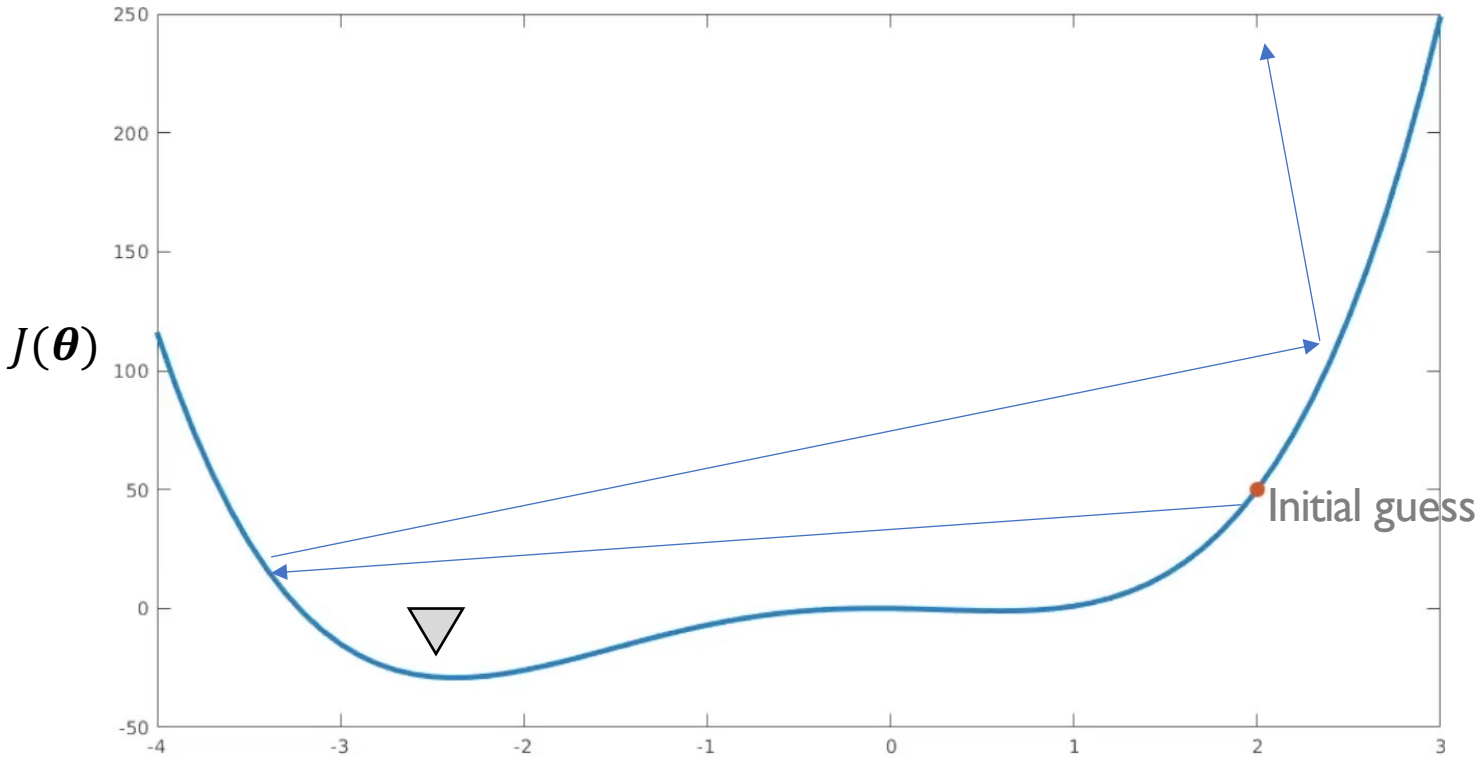
Learning Rate.. η

Small learning rate converges slowly … gets stuck in false local minima



Learning Rate.. η

Large learning rates overshoot,become unstable .. diverge

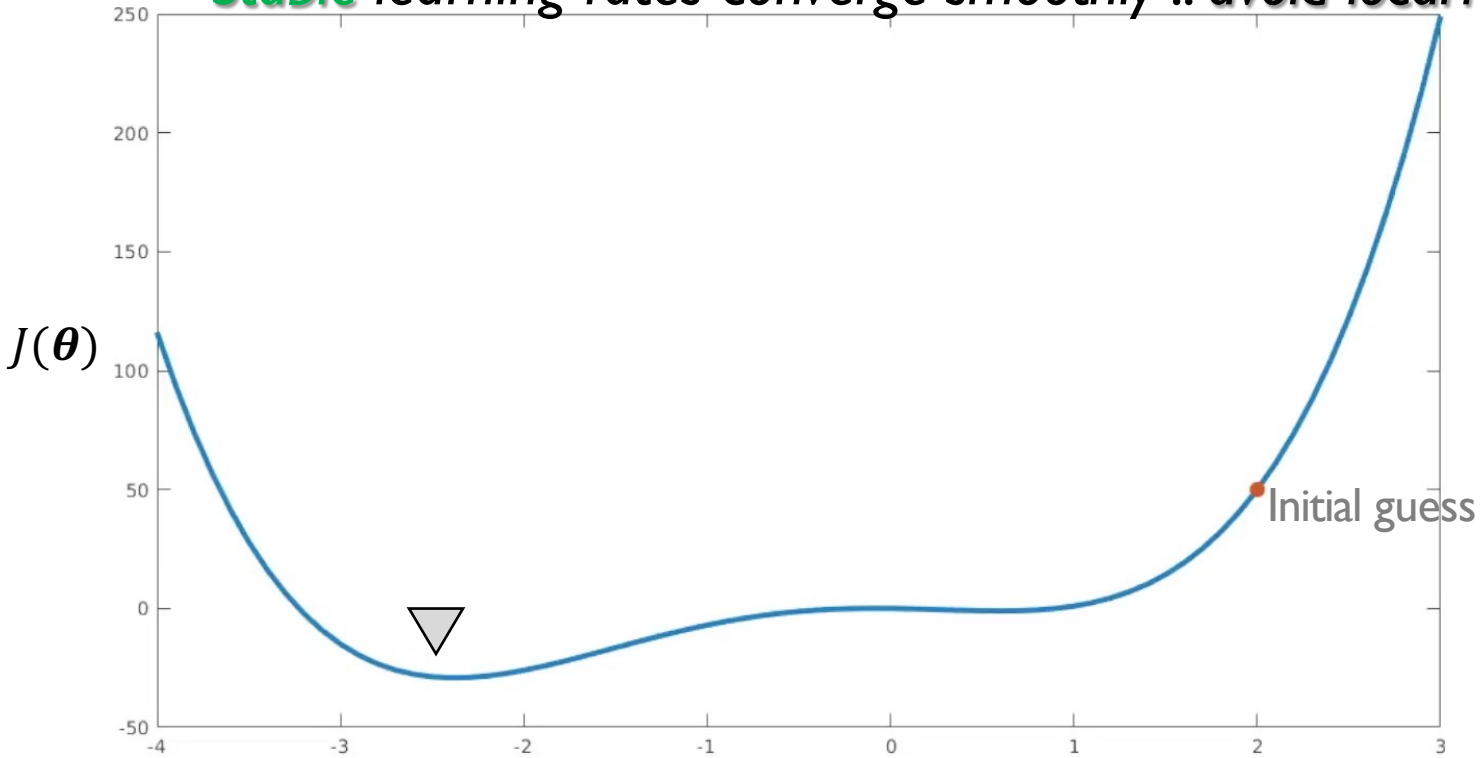


Learning Rate.. η

Small learning rate converges slowly … gets stuck in false local minima

Large learning rates overshoot,become unstable .. diverge

Stable learning rates converge smoothly .. avoid local minima

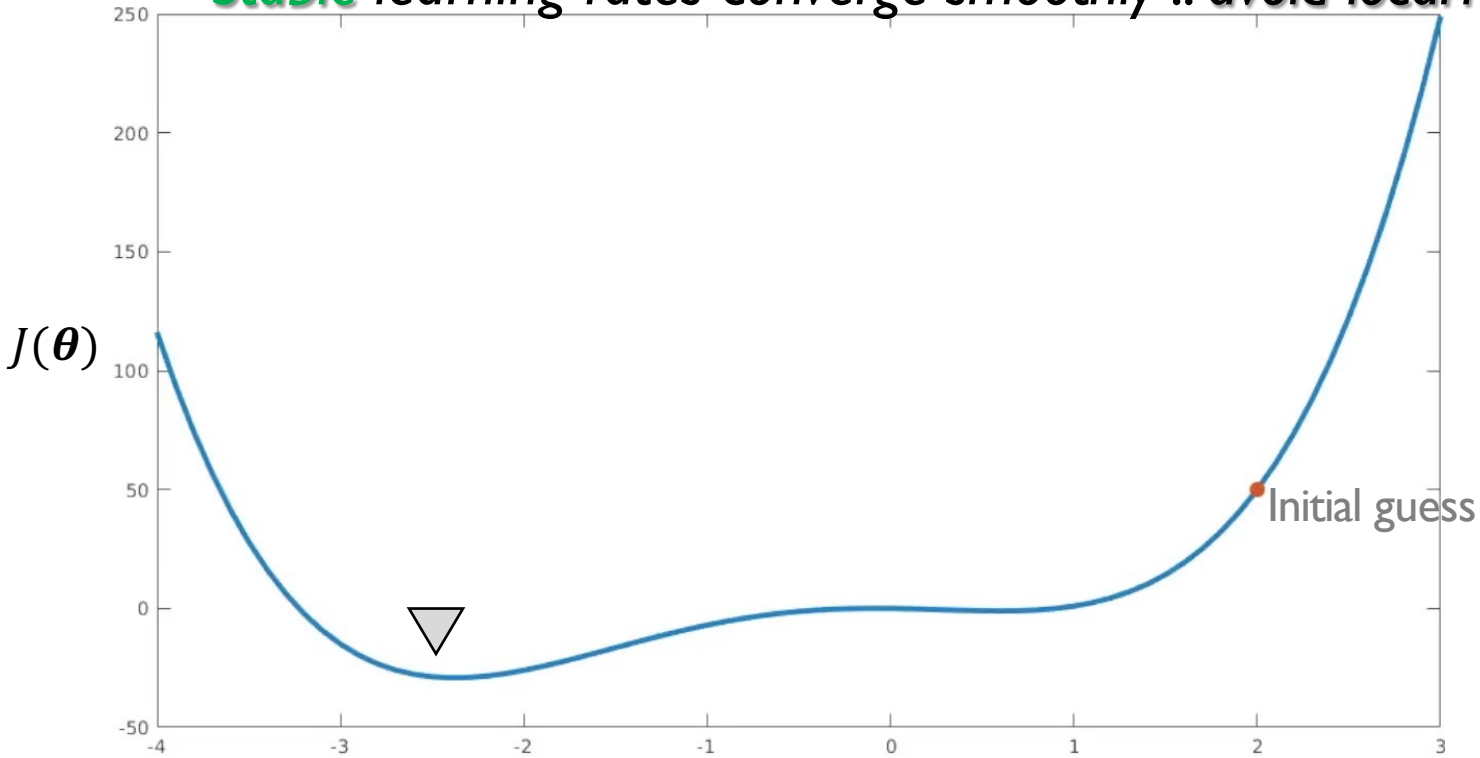


Learning Rate.. η

Small learning rate converges slowly … gets stuck in false local minima

Large learning rates overshoot,become unstable .. diverge

Stable learning rates converge smoothly .. avoid local minima



Learning Rate.. η

Vary learning rates (i.e. no longer fixed) and observe performance.. OR

Use **Adaptive Learning Rate** .. "adapts" to landscape … made *larger* or *smaller* depending on size of gradient / weights; speed of learning *(how fast learning is taking place)* etc. :

Qian et al."On the momentum term in gradient descent learning algorithms." 1999.

Duchi et al."Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." 2011.

Zeiler et al."ADADELTA:An Adaptive Learning Rate Method." 2012.

Kingma et al."Adam:A Method for Stochastic Optimization." 2014.

Adaptive Learning Rate.. η

- Decrease learning rate over time
- At the beginning, we are far from the destination, so we use larger learning rate
- After several epochs, we are close to the destination, so we reduce the learning rate
- **Adagrad**

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

$\frac{\partial L}{\partial w}$ in the ith update

- Smaller derivatives, larger learning rate, and vice versa

Neural Networks in Practise

Mini-bathing the data….

Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
   3. Compute gradient $\frac{\delta J(\boldsymbol{\theta})}{\delta \boldsymbol{\theta}}$ *can be computationally intensive ..because many data points!*
   4. Update weights $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \frac{\delta J(\boldsymbol{\theta})}{\delta \boldsymbol{\theta}}$ *..update rule: follow negative gradient..*
5. Return weights..

*computing the gradient… **a major issue** in NNs.*
Consider a simple NN to illustrate.. *1 hidden unit.. 1 output unit*

Stochastic Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
   3. Pick single data point i
   4. Compute gradient $\frac{\delta J_i(\boldsymbol{\theta})}{\delta \boldsymbol{\theta}}$
   5. Update weights $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \frac{\delta J(\boldsymbol{\theta})}{\delta \boldsymbol{\theta}}$
6. Return weights..

..noisy... as its only a single point!

*computing the gradient at single point... **Easier..***

Stochastic Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
   3. Pick batch of M data points
   4. Compute gradient $\frac{\delta J(\boldsymbol{\theta})}{\delta \boldsymbol{\theta}} = \frac{1}{M} \sum_{\boldsymbol{k=1}}^{M} \frac{\delta J_k(\boldsymbol{\theta})}{\delta \boldsymbol{\theta}}$
   5. Update weights $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \frac{\delta J(\boldsymbol{\theta})}{\delta \boldsymbol{\theta}}$
6. Return weights..

Mini-batch GD *-- only use a small portion of training set to compute the gradient. .. fast training!*

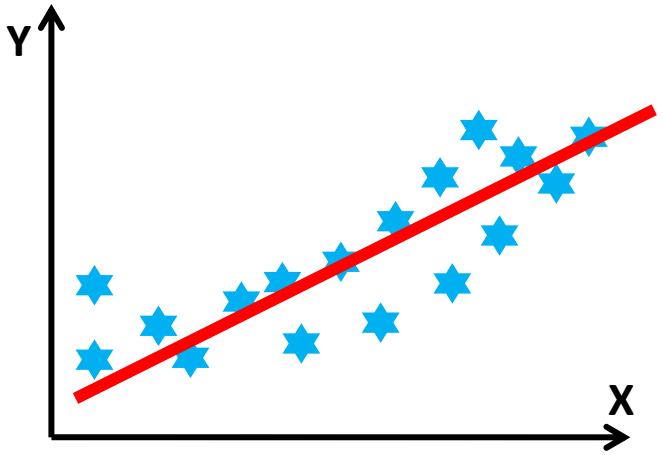
more accurate than with single data

Parallelizable .. GPU's

Using
Neural Networks

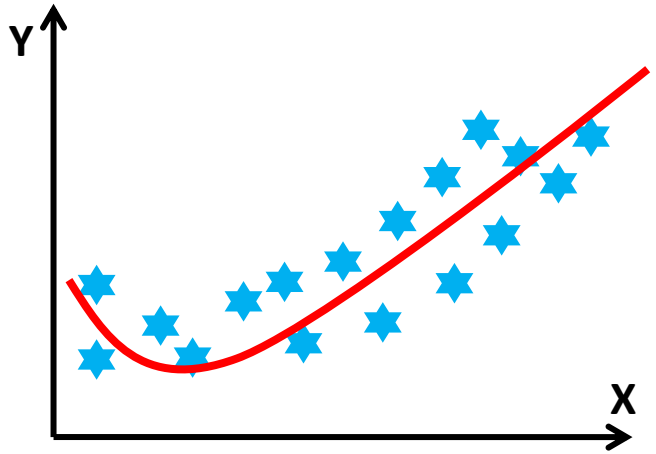
Overfitting ..

Learned hypothesis may **fit** the **training data** very well, even for outliers (**noise**) but fail to **generalize** to new examples (TEST data!)..

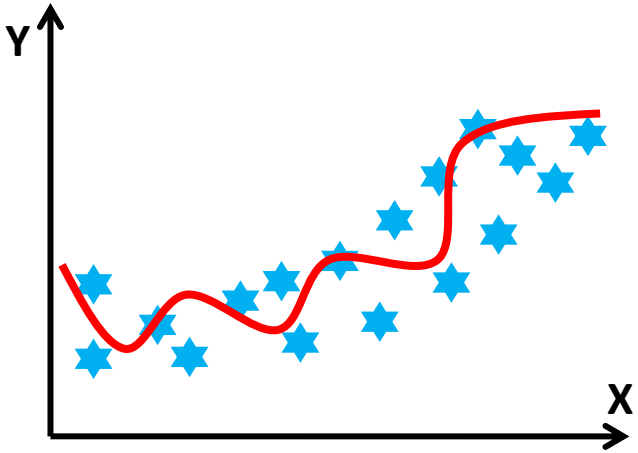


Underfitting

No capacity to fully learn the data



Ideal fit



Overfitting

Does not generalize well.

too complex, extra parameters

Regularization ..

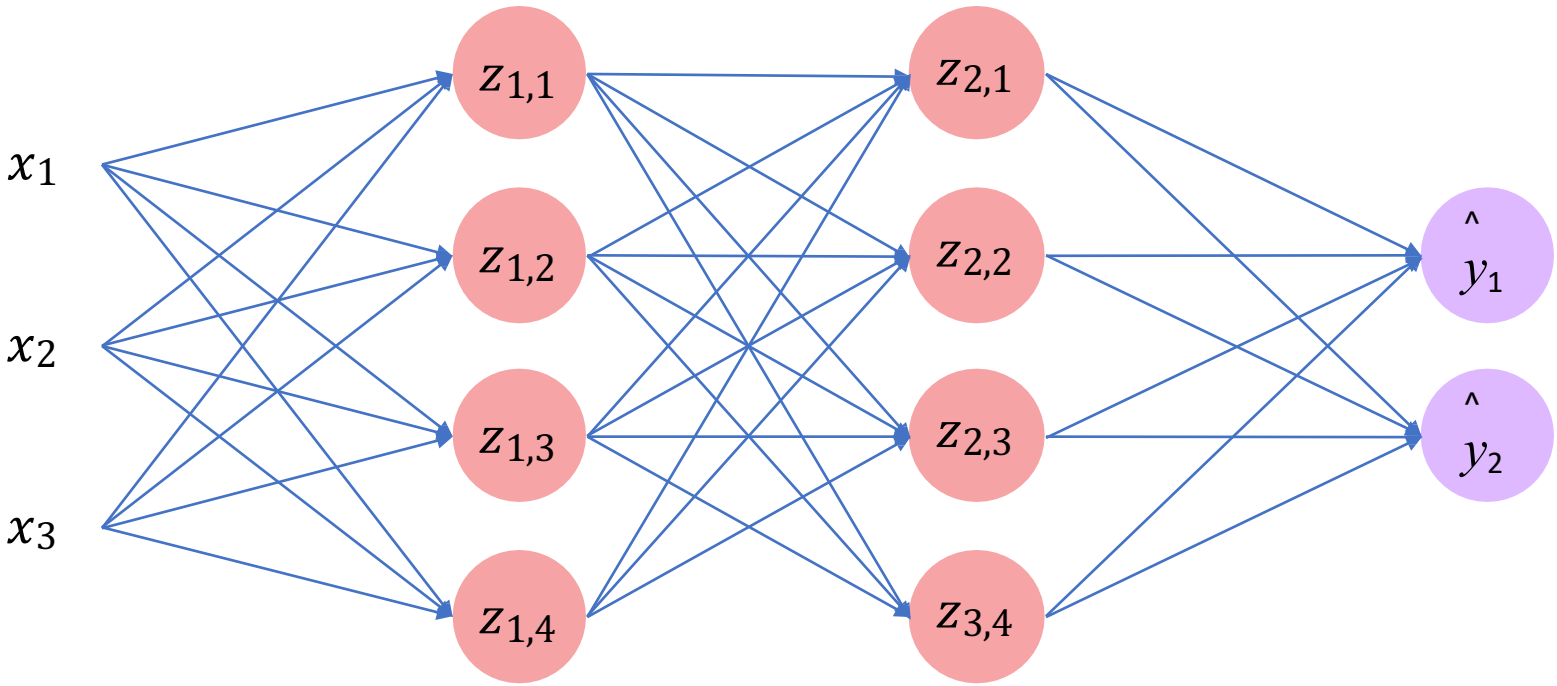
Constrains the optimization problem to **DISCOURAGE** complex models.

to avoid a situation that model closely follows training data

Improves generalization of models on unseen data.

Regularization.. drop out

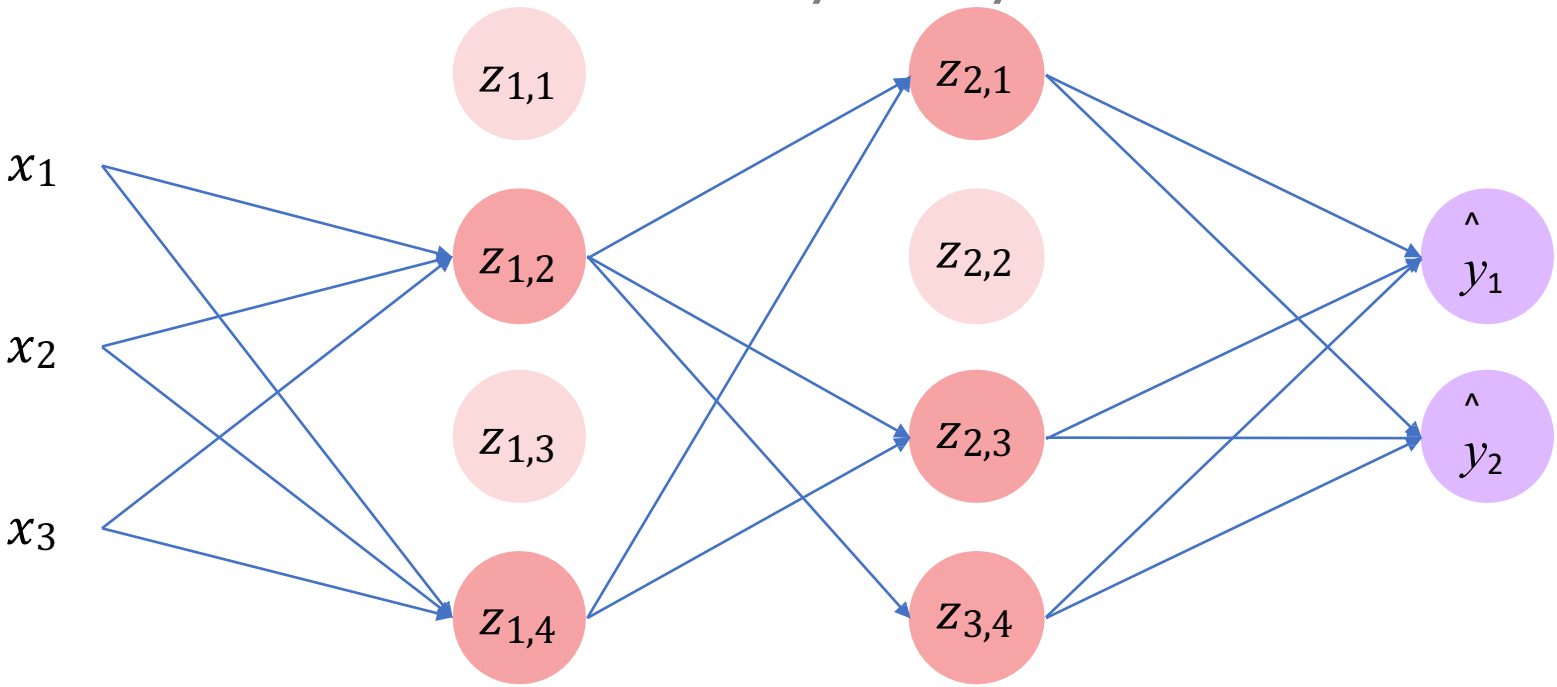
Randomly set some activations to 0 during training.



Regularization.. drop out

Randomly set some activations to 0 during training.

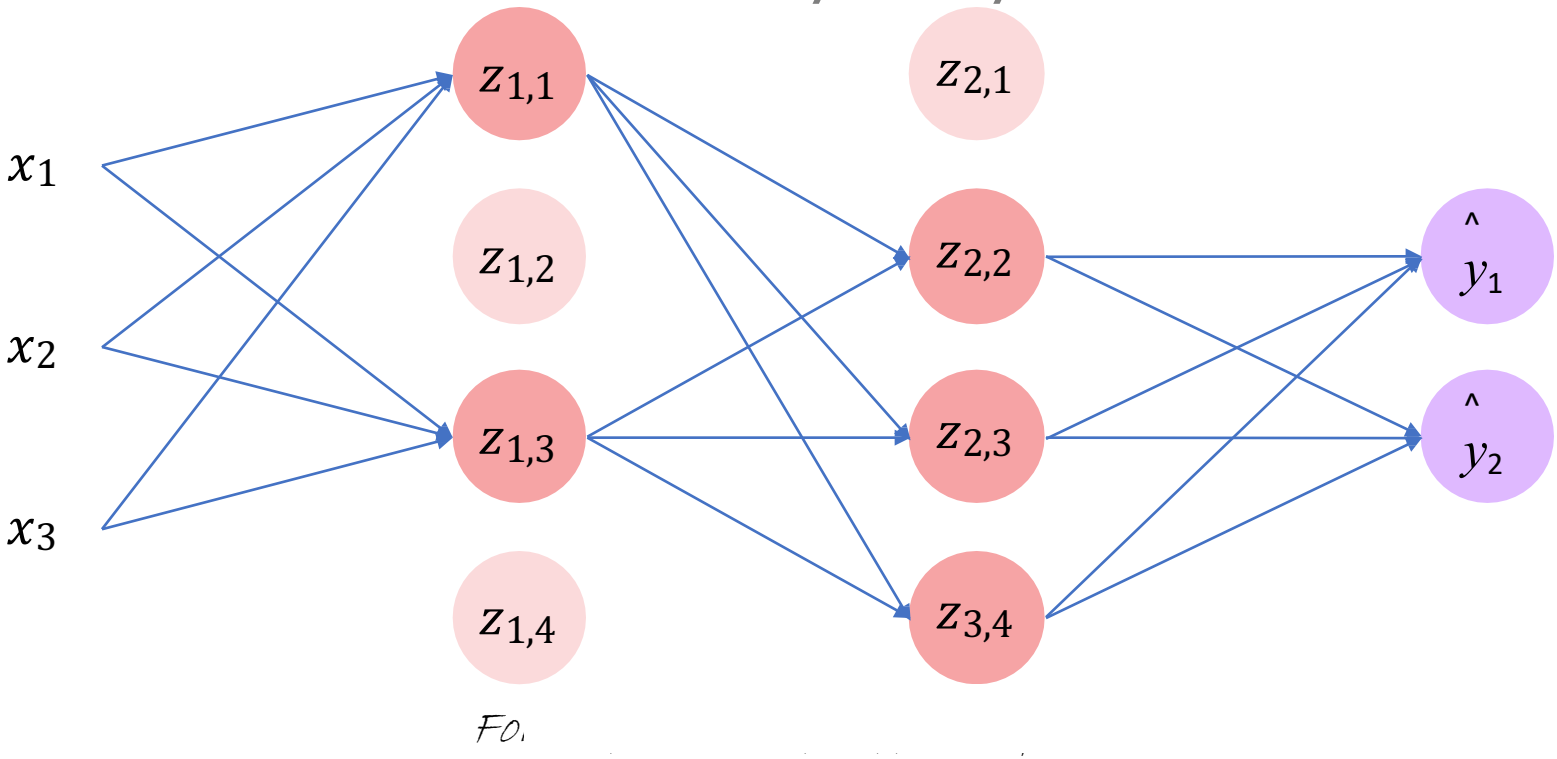
- 'drop' 50% of activations in layer at each iteration!
- forces network to not rely on any one node



Regularization.. drop out

Randomly set some activations to 0 during training.

- 'drop' 50% of activations in layer at each iteration!
- forces network to not rely on any one node



FO

Regularization.. "weight decay"

Regularization term that **penalizes big weights..**

Weight decay value determines how dominant the Regularization is during gradient computation.

Big weight decay coefficient big penalty for big weights.

So let's say that we have a cost or error function $E(\mathbf{w})$ that we want to minimize.

$$w_i \leftarrow w_i - \eta \frac{\partial E}{\partial w_i}$$

Regularize the cost function

$$\widetilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^2$$

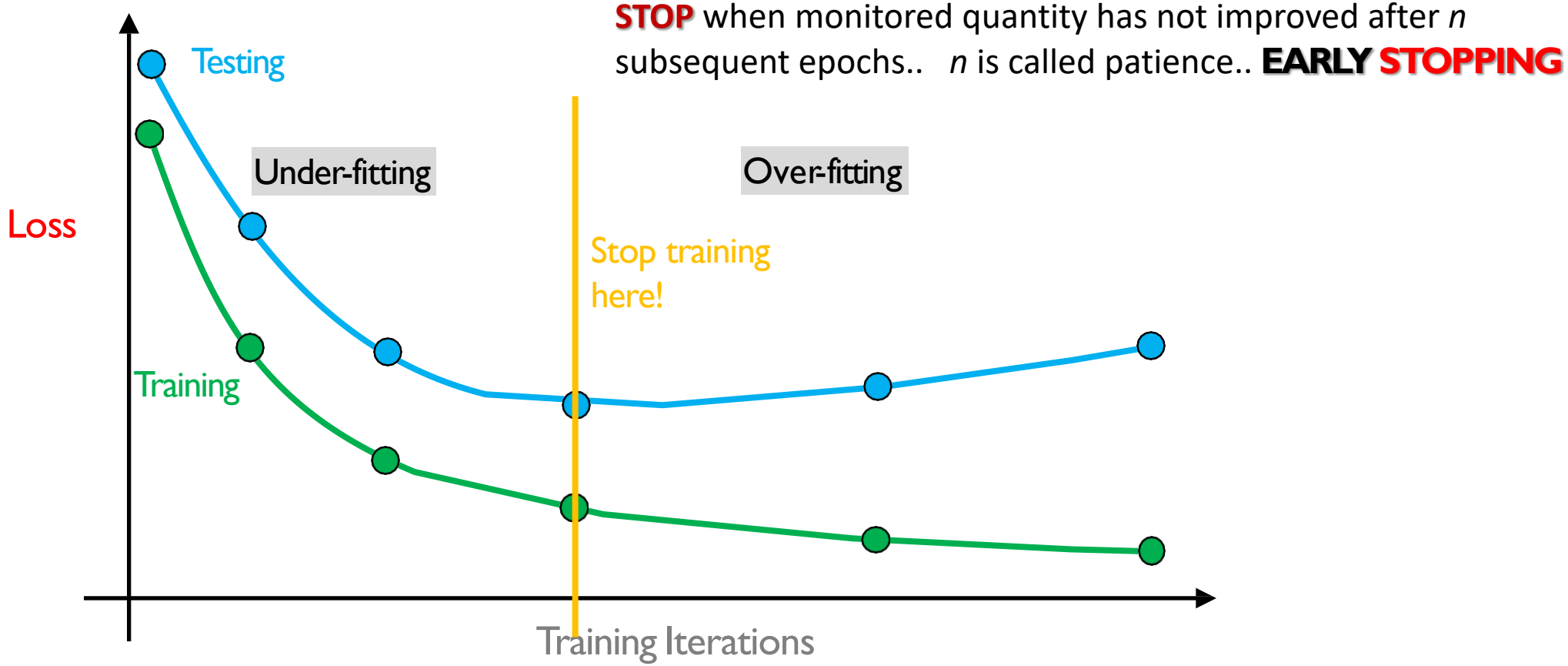
$$w_i \leftarrow w_i - \eta \frac{\partial E}{\partial w_i} - \eta\lambda w_i.$$

The new term $-\eta\lambda w_i$coming from the regularization causes the weight to decay in proportion to its size.

Regularization.. early Stopping

STOP training before overfitting takes place….

STOP when monitored quantity has not improved after *n* subsequent epochs.. *n* is called patience.. **EARLY STOPPING**



SUMMARY

The Perceptron

- Structural building blocks
- Nonlinear activation functions

Neural Networks

- Stacking Perceptrons to form neural networks
- Optimization through backpropagation

NN Practice

- Adaptive learning
- Batching
- Regularization