ANALYTIXLABS

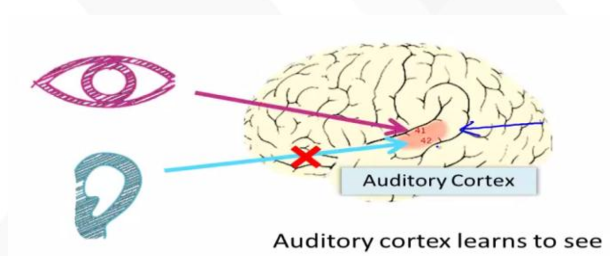
Machine Learning: Artificial Neural Network

About Human Brain

Human Brain

✓ Does loads of crazy things
- Hypothesis is that the brain has a single learning algorithm

✓ Evidence for hypothesis
- Auditory cortex --> takes sound signals
  - If you cut the wiring from the ear to the auditory cortex
  - Re-route optic nerve to the auditory cortex
  - Auditory cortex learns to see

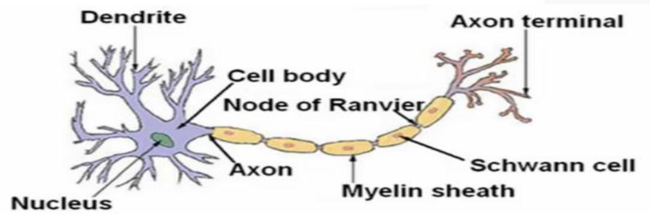


Human echolocation: https://www.youtube.com/watch?v=A8lztr1tu4o

ANALYTIXLABS

Neural Networks

- ✓ How do we represent neural networks (NNs)?
  - – Neural networks were developed as a way to simulate networks of neurons
- ✓ How does a neuron look like



A neural network is a set of connected input/output units (neurons) where each connection has a weight associated with it.

In an artificial neural network, a neuron is a logistic unit

- – Feed input via input wires
- – Logistic unit does computation
- – Sends output down output wires that logistic computation is just like our previous logistic regression hypothesis calculation

Neural Networks

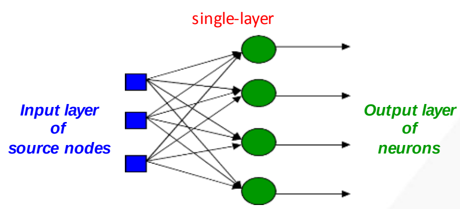
- ✓ Why do we need neural networks?
  - • Complex Non Linear Hypothesis
  - • Good way to build classifiers when N is large
- ✓ **Neural networks (NNs)** were originally motivated by looking at machines which replicate the brain's functionality looked at here as a machine learning technique
- ✓ **Origins**
  - • To build learning systems, why not mimic the brain?
  - • Used a lot in the 80s and 90s
  - • Popularity diminished in late 90s
- ✓ **Recent major resurgence**
  - • NNs are computationally expensive, so only recently large scale neural networks became computationally feasible

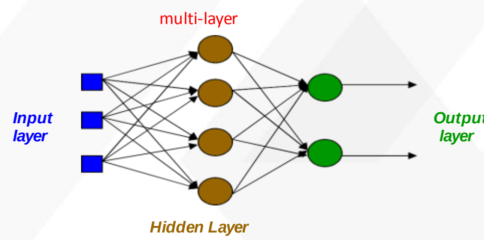
Network Architectures

- Three different classes of network architectures
  - single-layer feed-forward
  - multi-layer feed-forward
  - recurrent

neurons are organized in acyclic layers

- The architecture of a neural network is linked with the learning algorithm used to train



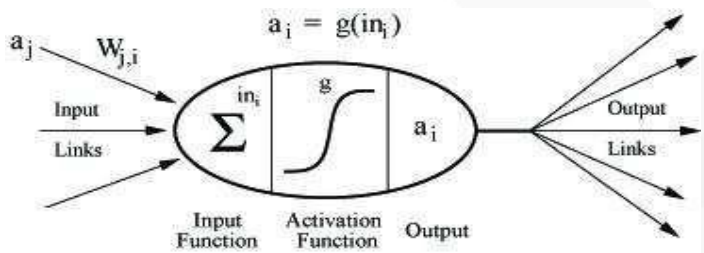


Neural Networks

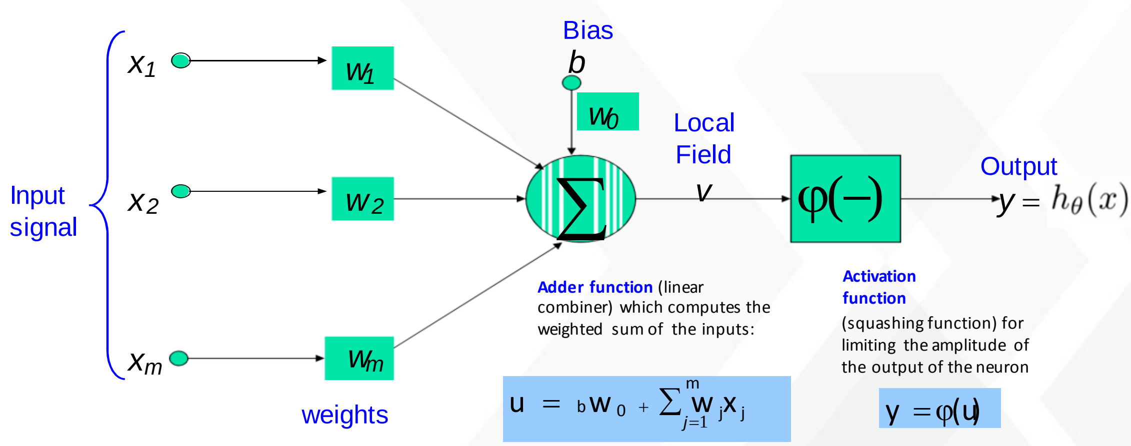
- **Advantages**
  - prediction accuracy is generally high
  - robust, works when training examples contain errors or noisy data
  - output may be discrete, real-valued, or a vector of several discrete or real- valued attributes
  - fast evaluation of the learned target function
- **Criticism**
  - parameters are best determined empirically, such as the network topology or structure
  - long training time
  - difficult to understand the learned function (weights)
  - not easy to incorporate domain knowledge

Neurons

- Neural networks are built out of a densely interconnected set of simple units (neurons)
  - Each neuron takes a number of real-valued inputs
  - Produces a single real-valued output
  - Inputs to a neuron may be the outputs of other neurons.
  - A neuron's output may be used as input to many other neurons



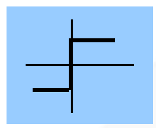
The Neuron



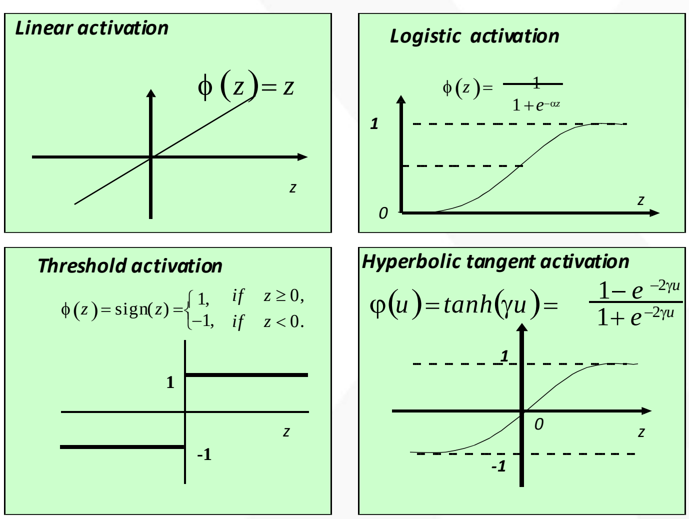
Bias: serves to vary the activity of the unit

How does it Works?

- Assign weights to each input-link
- Multiply each weight by the input value (0 or 1)
- Sum all the weight-firing input combinations
- Apply squash function, e.g.:
  - If sum > threshold for the Neuron then
  - Output = +1
  - Else Output = -1



Popular activation functions



How Are Neural Networks Trained?

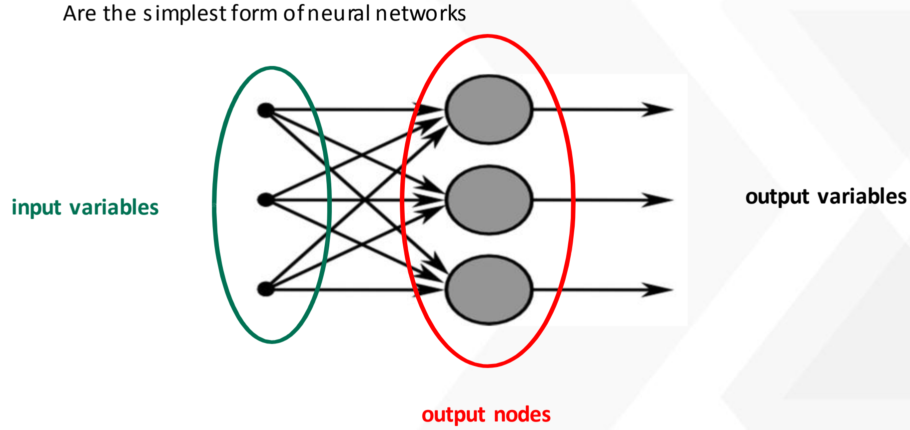
- Initially
  - choose small random weights (w_i)
  - Set threshold = 1 (step function)
  - Choose small *learning rate* (r)

- Apply each member of the *training set* to the neural net model using a *training rule* to adjust the weights
  - For each unit
    - Compute the net input to the unit as a linear combination of all the inputs to the unit
    - Compute the output value using the activation function
    - Compute the error
    - Update the weights and the bias

ANALYTIXLABS

Single Layer Perceptron

Are the simplest form of neural networks



ANALYTIXLABS

Single layer perceptron: training rule

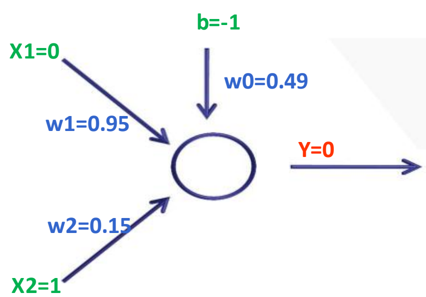
- Modify the ***weights*** (w_i) according to the *Training Rule:*

$$w_i = w_i + r \cdot (t - a) \cdot x_i$$

  - r is the *learning rate* (eg. 0.2)
  - t = target output
  - a = actual output
  - x_i = i-th input value

Learning rate: if too small learning occurs at a small pace, if too large it may stuck in local minimum in the decision space

Example



x1	x2	Y
0	0	0
1	0	1
0	1	1
1	1	1

threshold = 0.5
r=0.05

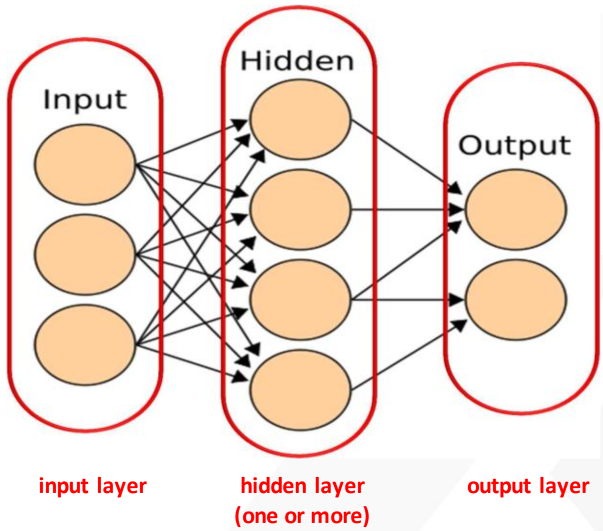
Compute output for the input
u = -1 x 0.49 + 0 x 0.95 + 1 x 0.15=-0.34 < t
thus, y=0

Compute the error
target output = 1
actual output (y) = 0
error = (1-0) = 1
correction factor = error x r = 0.05

Compute the new weights
w0 = 0.49 + 0.05 x (1-0) x (-1) = 0.44
w1 = 0.95 + 0.05 x (1-0) x 0 = 0.95
w2 = 0.15 + 0.05 x (1-0) x 1 = 0.20

Repeat the process with the new weigths for a given number of iterations

Multi layer network

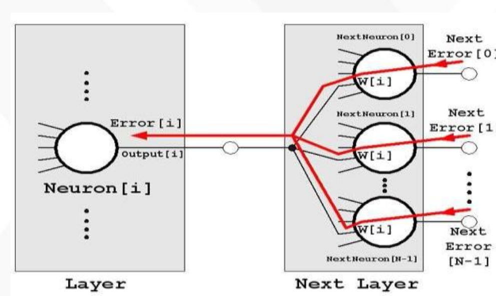


ANALYTIXLABS

Training multi layer networks

back-propagation algorithm

- **Phase 1: Propagation**
- Forward propagation of a training input
- Back propagation of the propagation's output activations.
- **Phase 2: Weight update**
- For each weight-synapse:
- Multiply its output delta and input activation to get the gradient of the weight.
- Bring the weight in the opposite direction of the gradient by subtracting a ratio of it from the weight.
- This ratio influences the speed and quality of learning. The sign of the gradient of a weight indicates where the error is increasing, this is why the weight must be updated in the opposite direction.
- Repeat the phase 1 and 2 until the performance of the network is good enough.



ANALYTIXLABS

Multi-Layer network of sigmoid units

Problem: what is the desired output for a hidden node? => Backpropagation algorithm

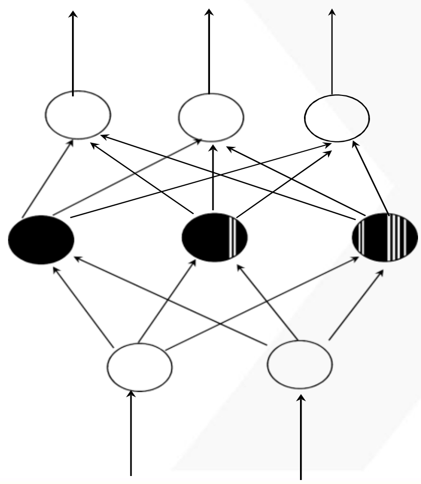
Output vector

Output nodes

Hidden nodes

Input nodes

Input vector: x_i



$\theta_j = \theta_j + (r) Err_j$

to update the bias

$w_{ij} = w_{ij} + (r) Err_j O_i$

to update the weights

$Err_j = O_j (1 - O_j) \sum_k Err_k w_{jk}$

error for a node in the hidden layer

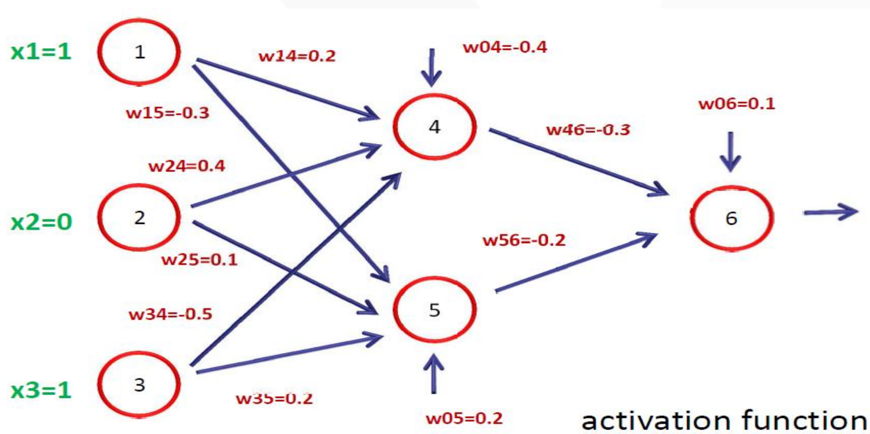
$Err_j = O_j (1 - O_j)(T_j - O_j)$

error for a node in the output layer

$$O_j = \frac{1}{1 + e^{-I_j}}$$

$$I_j = \sum_i w_{ij} O_i + \theta_j$$

Example



xi – input variables (1,0,1) whose class is 1

wij – randomly assigned weights

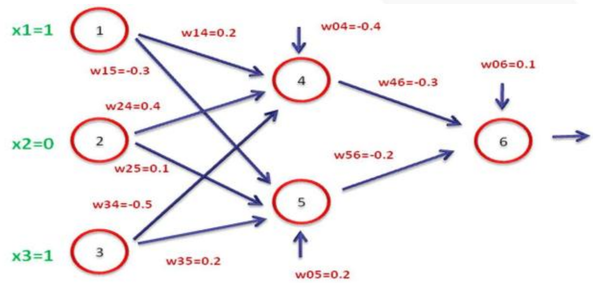
activation function

Oj = 1 / (1+e^{-Ij})

and

learning rate = 0.9

Propagation



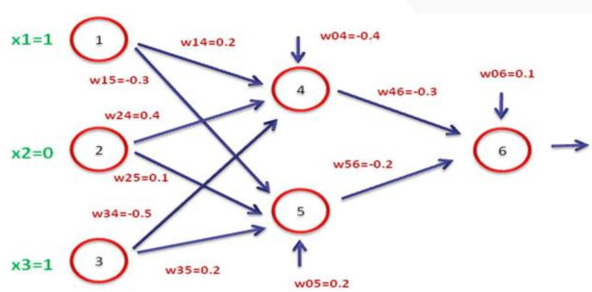
$$I_j = \sum_i w_{ij} O_i + \theta_j$$

$$O_j = \frac{1}{1+e^{-I_j}}$$

neuron	input	output
4	0.2x1+0.4x0-0.5x1-0.4=-0.7	$1/(1+e^{0.7})$=0.332
5	-0.3x1+0.1x0+0.2x1+0.2=0.1	$1/(1+e^{-0.1})$=0.525
6	-0.3x0.332-0.2x0.525+0.1=-0.105	$1/(1+e^{0.105})$=0.474

ANALYTIXLABS

Calculation of the neuron



error for a node in the output layer

$$Err_j = O_j(1-O_j)(T_j - O_j)$$

error for a node in the hidden layer

$$Err_j = O_j(1-O_j)\sum_k Err_k w_{jk}$$

neuron	output
4	0.332
5	0.525
6	0.474

neuron	error
6	0.474 x (1 - 0.474) x (1 - 0.474) = 0.1311
5	0.525 x (1 - 0.525) x (-0.2) x 0.1311 = -0.0065
4	0.332 x (1 − 0.332) x (-0.3) x 0.1311 = -0.0087

ANALYTIXLABS

Updating weights

$\theta_j = \theta_j + (r)Err_j$

to update the bias

$w_{ij} = w_{ij} + (r)Err_j O_i$

to update the weights

$Err_j = O_j(1-O_j)\sum_k Err_k w_{jk}$

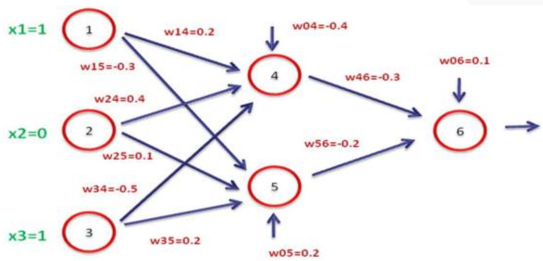
error for a node in the hidden layer

$Err_j = O_j(1-O_j)(T_j - O_j)$

error for a node in the output layer

$O_j = \frac{1}{1+e^{-I_j}}$

$I_j = \sum_i w_{ij} O_i + \theta_j$



to update the weights

$w_{ij} = w_{ij} + (r)Err_j O_i$

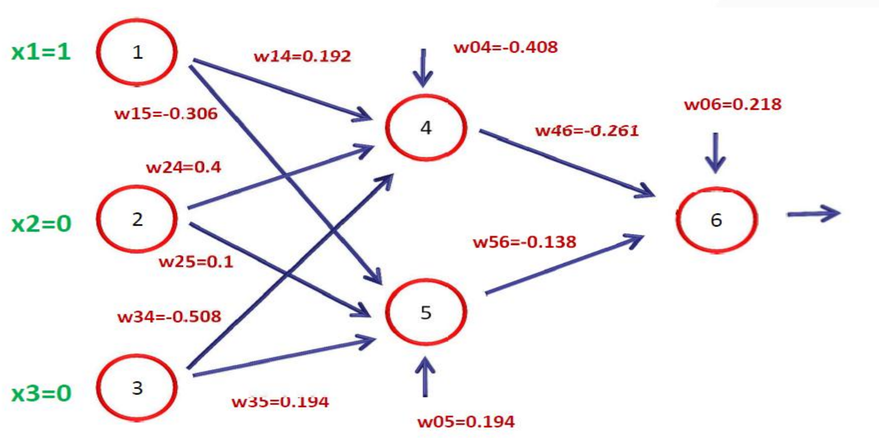
to update the bias

$\theta_j = \theta_j + (r)Err_j$

neuron	output	error
4	0.332	-0.0087
5	0.525	-0.0065
6	0.474	0.1311

weight	New value
w46	-0.3 + 0.9 x 0.1311 x 0.332 = -0.261
w56	-0.2 + 0.9 x 0.1311 x 0.525 = -0.138
w14	0.2 + 0.9 x -0.0087 x 1 = 0.192
w15	-0.3 + 0.9 x -0.0065 x 1 = -0.306
w24	0.4 + 0.9 x -0.0087 x 0 = 0.4
w25	0.1 + 0.9 x -0.0065 x 0 = 0.1
w34	-0.5 + 0.9 x -0.0087 x 1 = -0.508
w35	0.2 + 0.9 x -0.0065 x 1 = 0.194
w06	0.1 + 0.9 x 0.1311 = 0.218
w05	0.2 + 0.9 x -0.0065 = 0.194
w04	-0.4 + 0.9 x -0.0087 = -0.408

Example



This is the resulting network after the first iteration. We now have to process another training example until the overall error is low or we run out of examples.

Neural Networks (Cost Function)

The (regularized) logistic regression cost function is as follows;

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

For neural networks our cost function is a generalization of this equation above, so instead of one output we generate *k* outputs

$$h_\Theta(x) \in \mathbb{R}^K \quad (h_\Theta(x))_i = i^{th} \text{ output}$$

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_\Theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2$$

ANALYTIXLABS

Minimize Neural Network's Cost Function (Back Propagation)

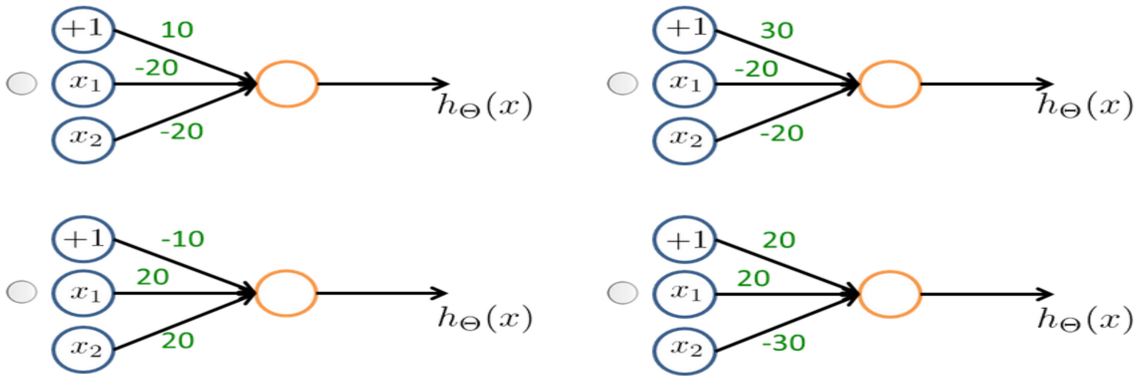
Back propagation

- ✓ Back propagation basically takes the output you got from your network, compares it to the real value (y) and calculates how wrong the network was (i.e. how wrong the parameters were)
- ✓ It then, using the error you've just calculated, back-calculates the error associated with each unit from the preceding layer (i.e. layer *L* - 1)
- ✓ This goes on until you reach the input layer (where obviously there is no error, as the activation is the input)
- ✓ These "error" measurements for each unit can be used to calculate the **partial derivatives**
- ✓ We use the partial derivatives with gradient descent to try minimize the cost function and update all the Θ values
- ✓ This repeats until gradient descent reports convergence

ANALYTIXLABS

Quiz

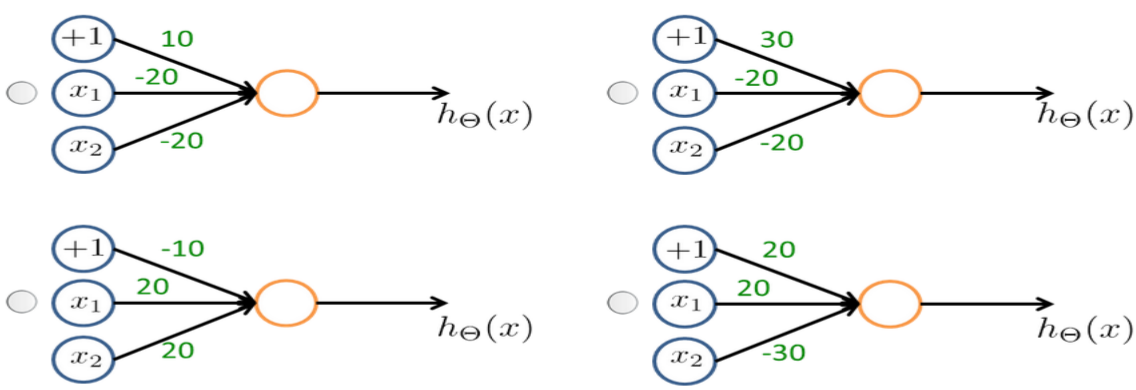
Suppose that x_1 and x_2 are binary valued (0 or 1). Which of the following networks (approximately) computes the boolean function (NOT x_1) AND (NOT x_2)?



ANALYTIXLABS

Quiz

Suppose that x_1 and x_2 are binary valued (0 or 1). Which of the following networks (approximately) computes the boolean function (NOT x_1) AND (NOT x_2)?



The Answer: C

ANALYTIXLABS