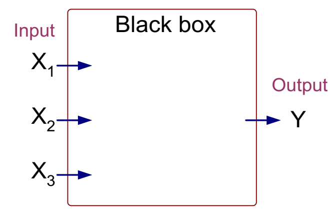
Introduction to Machine Learning

Ranga Raju Vatsavai, Ph.D.
Chancellors Faculty Excellence Associate Professor in Geospatial Analytics
Department of Computer Science, North Carolina State University (NCSU)

Feb. 25-27, 2019

Artificial Neural Networks (ANN)

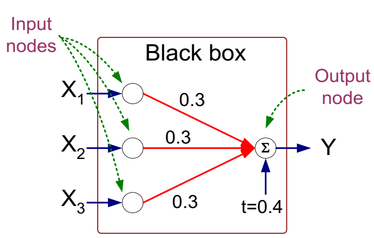
X_1	X_2	X_3	Y
1	0	0	-1
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	-1
0	1	0	-1
0	1	1	1
0	0	0	-1



Output Y is 1 if at least two of the three inputs are equal to 1.

Artificial Neural Networks (ANN)

X_1	X_2	X_3	Y
1	0	0	-1
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	-1
0	1	0	-1
0	1	1	1
0	0	0	-1

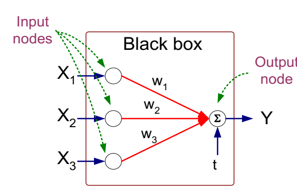


$$Y = sign(0.3X_1 + 0.3X_2 + 0.3X_3 - 0.4)$$

$$\text{where } sign(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$$

Artificial Neural Networks (ANN)

- Model is an assembly of inter-connected nodes and weighted links
- Output node sums up each of its input value according to the weights of its links
- Compare output node against some threshold t

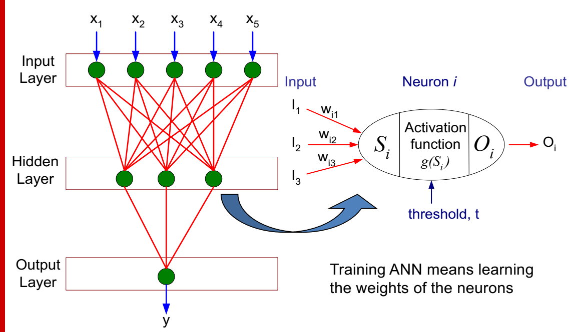


Perceptron Model

$$Y = sign(\sum_{i=1}^{d} w_i X_i - t)$$

$$= sign(\sum_{i=0}^{d} w_i X_i)$$

General Structure of ANN

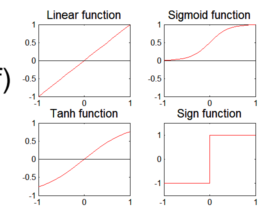


Training ANN means learning the weights of the neurons

Artificial Neural Networks (ANN)

- Various types of neural network topology
  - single-layered network (perceptron) versus multi-layered network
  - Feed-forward versus recurrent network
- Various types of activation functions (f)

$$Y = f(\sum_i w_i X_i)$$



Activation Functions

- Various types of activation functions (f)

$$Y = f(\sum_i w_i X_i)$$

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}, w = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}$$

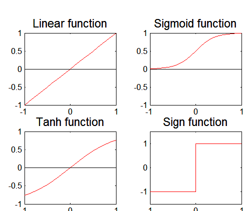
Sigmoid

$$f(x) = \frac{1}{1 + e^{-w^T x}}$$

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

Tanh

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



Perceptron

- Single layer network
  - Contains only input and output nodes
- Activation function: f = sign(**w•x**)
- Applying model is straightforward

$$Y = sign(0.3X_1 + 0.3X_2 + 0.3X_3 - 0.4)$$

$$\text{where } sign(x) = \begin{cases} 1 & \text{if } x \ge 0 \\ -1 & \text{if } x < 0 \end{cases}$$

  - $X_1 = 1$, $X_2 = 0$, $X_3 = 1$ => y = sign(0.2) = 1

Perceptron Learning Rule

- Initialize the weights (w_0, w_1, …, w_d)
- Repeat
  - For each training example (x_i, y_i)
    - Compute f(w, x_i)
    - Update the weights:

$$w^{(k+1)} = w^{(k)} + \lambda\left[y_i - f(w^{(k)}, x_i)\right]x_i$$

- Until stopping condition is met

Perceptron Learning Rule

- Weight update formula:

$$w^{(k+1)} = w^{(k)} + \lambda\left[y_i - f(w^{(k)}, x_i)\right]x_i \ ; \ \lambda : \text{learning rate}$$

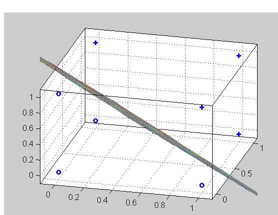
- Intuition:

$$e = \left[y_i - f(w^{(k)}, x_i)\right]$$

  - Update weight based on error:
  - If y=f(x,w), e=0: no update needed
  - If y>f(x,w), e=2: weight must be increased so that f(x,w) will increase
  - If y<f(x,w), e=-2: weight must be decreased so that f(x,w) will decrease

Perceptron Learning Rule

- Since f(w,x) is a linear combination of input variables, decision boundary is linear

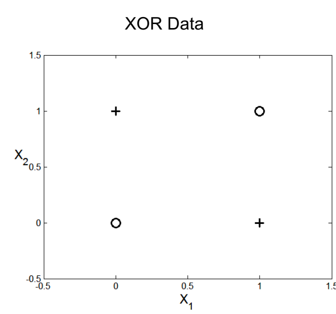


- For nonlinearly separable problems, perceptron learning algorithm will fail because no linear hyperplane can separate the data perfectly

Nonlinearly Separable Data

$$y = x_1 \oplus x_2$$

x_1	x_2	y
0	0	-1
1	0	1
0	1	1
1	1	-1

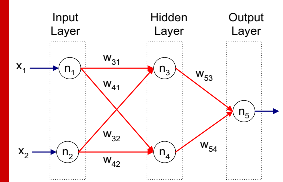


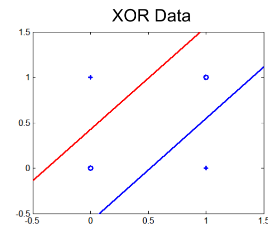
Multilayer Neural Network

- Hidden layers
  - intermediary layers between input & output layers
- Hidden units (nodes)
  - Nodes embedded in hidden layers
- More general activation functions (sigmoid, linear, etc)

Multi-layer Neural Network

- Multi-layer neural network can solve any type of classification task involving nonlinear decision surfaces





Learning Multi-layer Neural Network

- Can we apply perceptron learning rule to each node, including hidden nodes?
  - Perceptron learning rule computes error term e = y-f(w,x) and updates weights accordingly
    - Problem: how to determine the true value of y for hidden nodes?
  - Approximate error in hidden nodes by error in the output nodes
    - Problem:
      - Not clear how adjustment in the hidden nodes affect overall error
      - No guarantee of convergence to optimal solution

Outline of Back Propagation

- Given: Inputs, Outputs, Initial Weights and Biases, and Network
  - Forward pass the inputs
    - Calculate the output for every neuron starting from input layer, through hidden layers, and all the way to the output layer
    - At each neuron, compute W^TX, pass through activation function (e.g., sigmoid), pass output to each neuron in the next layer
  - Back-propagate and adjust weights and biases
    - Compute error for each output unit
    - Then layer by layer compute error (local gradient) for each hidden neuron by backpropagating errors
  - Repeat forward pass and backpropagation until convergence

Design Issues in ANN

- Number of nodes in input layer
  - One input node per binary/continuous attribute
  - k or $\log_2$ k nodes for each categorical attribute with k values
- Number of nodes in output layer
  - One output for binary class problem
  - k for k-class problem
- Number of nodes in hidden layer
- Initial weights and biases

Characteristics of ANN

- Multilayer ANN are universal approximators but could suffer from overfitting if the network is too large
- Gradient descent may converge to local minimum
- Model building can be very time consuming, but testing can be very fast
- Can handle redundant attributes because weights are automatically learnt
- Sensitive to noise in training data
- Difficult to handle missing attributes

Recent Noteworthy Developments in ANN

- Use in deep learning and unsupervised feature learning
  - Seek to automatically learn a good representation of the input from unlabeled data
- Google Brain project
  - Learned the concept of a 'cat' by looking at unlabeled pictures from YouTube
  - One billion connection network

Additional Slides (Math Behind)

- Training <x,t>, $\lambda = 0.05$, w_i = small random weights
- Until termination condition is met, repeat
  - Initialize each $\Delta w_i = 0$
  - For each training example in <x,t>, repeat
    - Compute output o
    - For each weight w_i $\Delta w_i = \Delta w_i + \lambda(t-o)x_i$
  - Update each weight w_i $w_i = w_i + \Delta w_i$

Gradient Descent for Multilayer NN

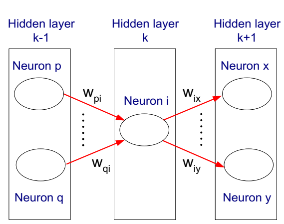
- Training <x,t>, $\lambda = 0.05$, w_i = small random weights
- Until termination condition is met, repeat
  - Initialize each $\Delta w_i = 0$
  - For each training example in <x,t>, repeat
    - Compute output o
    - For each weight w_i $\Delta w_i = \Delta w_i + \lambda(t-o)x_i$
  - Update each weight w_i $w_i = w_i + \Delta w_i$

Gradient Descent for Multilayer NN

- Weight update: $w_j^{(k+1)} = w_j^{(k)} - \lambda \frac{\partial E}{\partial w_j}$
- Error function: $E = \frac{1}{2}\sum_{i=1}^{N}\left(t_i - f(\sum_j w_j x_{ij})\right)$
- Activation function f must be differentiable
- For sigmoid function:
  $$w_j^{(k+1)} = w_j^{(k)} + \lambda \sum_i (t_i - o_i) o_i (1-o_i) x_{ij}$$
- Stochastic gradient descent (update the weight immediately)

Gradient Descent for Multilayer NN

- For output neurons, weight update formula is the same as before (gradient descent for perceptron)
- For hidden neurons:



$$w_{pi}^{(k+1)} = w_{pi}^{(k)} + \lambda o_i(1-o_i)\sum_{j \in \Phi_i} \delta_j w_{ij} x_{pi}$$

Output neurons: $\delta_j = o_j(1-o_j)(t_j - o_j)$

Hidden neurons: $\delta_j = o_j(1-o_j)\sum_{k \in \Phi_j} \delta_k w_{jk}$