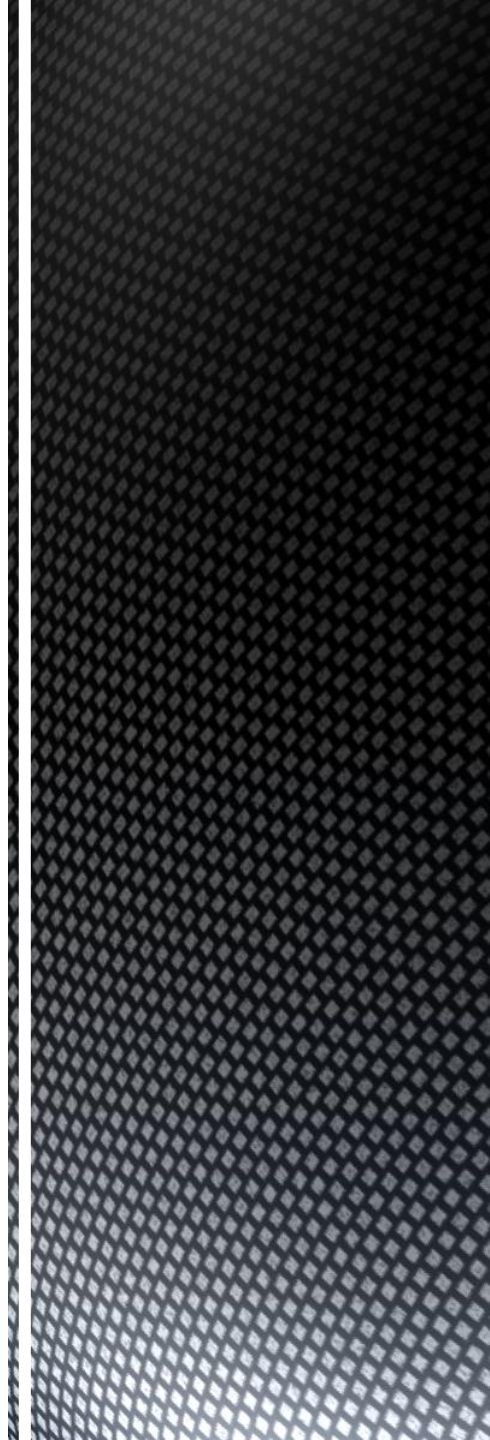


Artificial Neural Networks

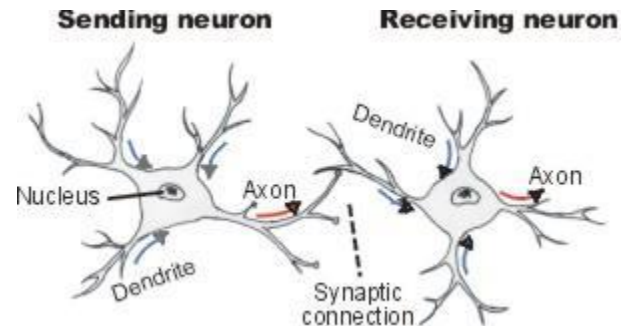


- Brains
- Neural networks
- Perceptrons
- Multilayer perceptrons
- Applications of neural networks

Outline

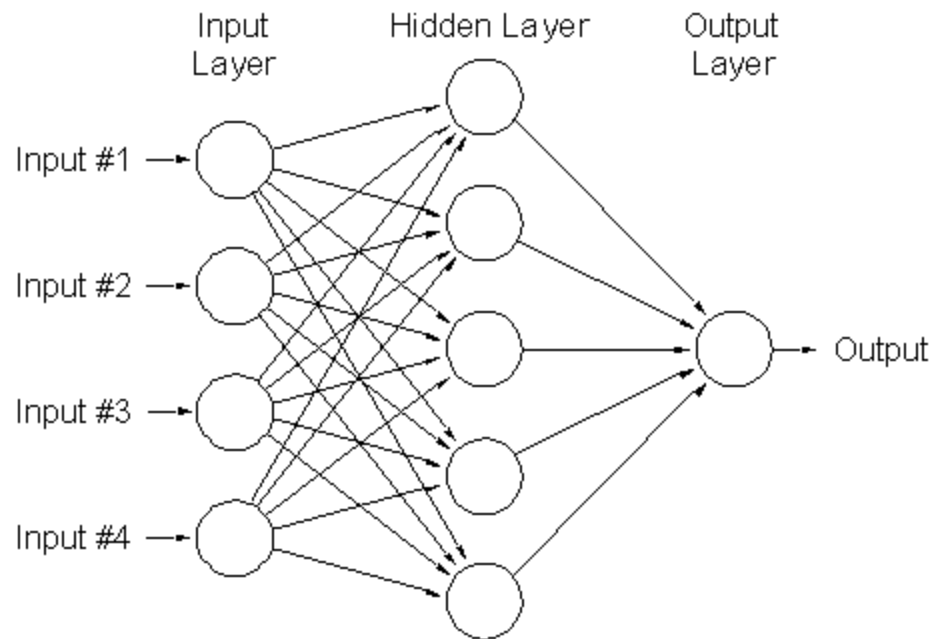
- Artificial neural networks (ANNs) are patterned after the structure and function of the brain
- When a neuron fires, it sends an electro-chemical signal along its axon to the synapses which connect it to other neurons
 - If this signal is strong enough, the next neuron may also fire, resulting in a spreading activation pattern
 - The strength of the connections between neurons can change over time, and this is the basis for learning
 - Connections leading to a “good answer” are strengthened while those leading to a “bad answer” are weakened
 - Humans have about 10 billion neurons and 60 trillion synapses

Introduction



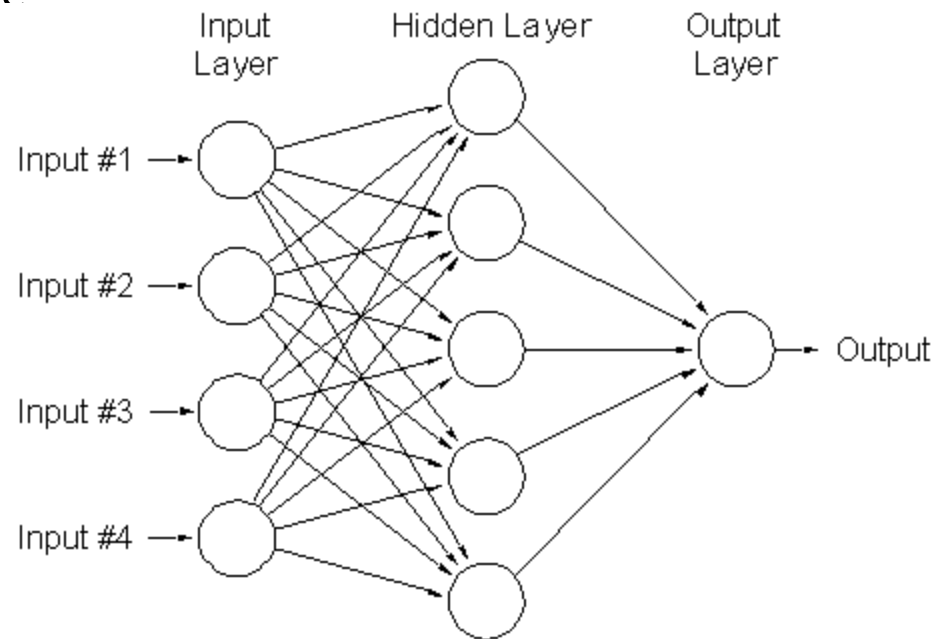
Introduction

- Artificial neural networks are patterned after the brain
 - Neurodes (or just nodes) represent neurons
 - Connections represent synapses
 - Weights on the connections change in order to produce learning



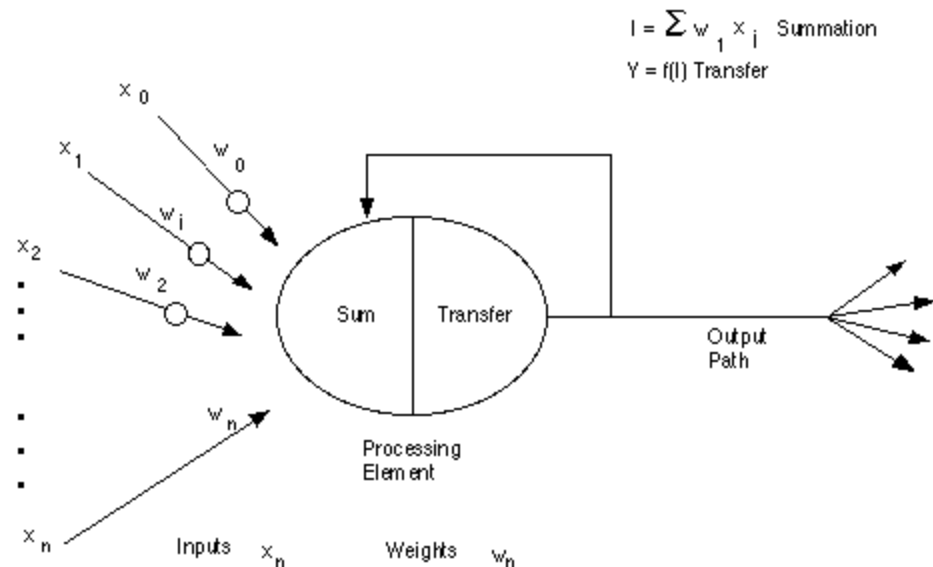
Introduction

- Architecture:
- In most cases, we use a fully connected model
 - All neurodes at one layer are connected to each of the neurodes at the next layer
- This picture shows a fully connected model



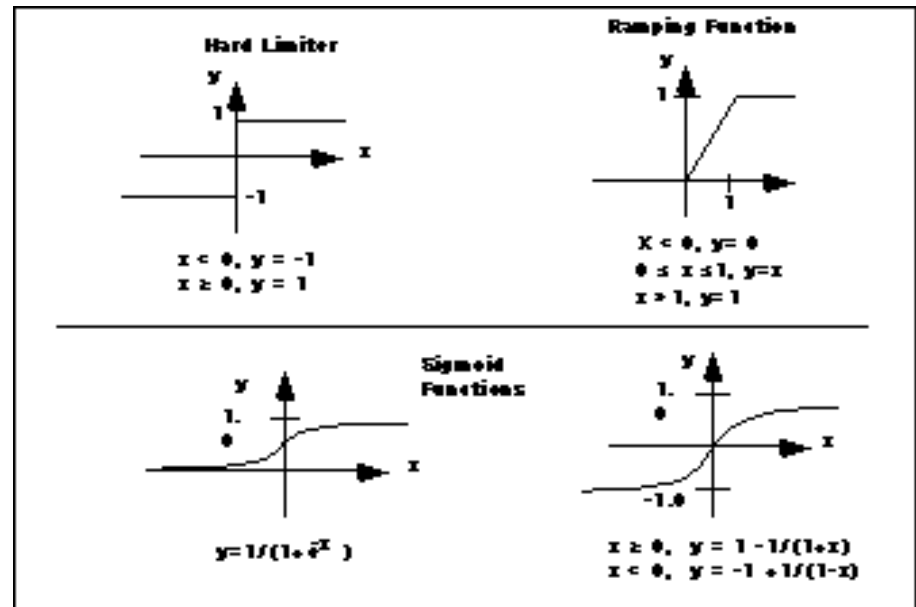
Introduction

- Each neurode sums the input signals coming into it
 - Actually, multiply the connection weight and the incoming signal, and sum each of these
- Output or “transfer” function could be:
 - Step function
 - Sign function
 - Sigmoid function
 - Linear function



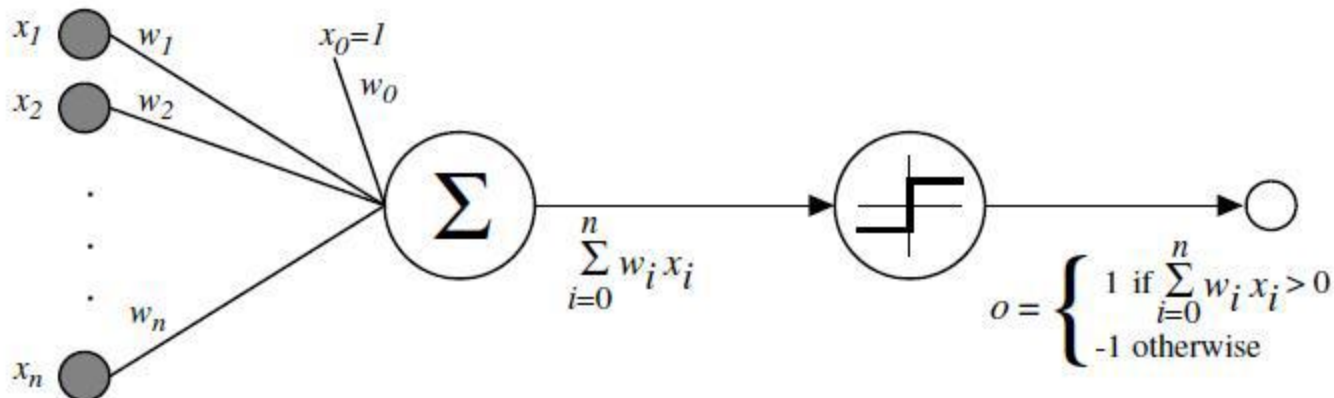
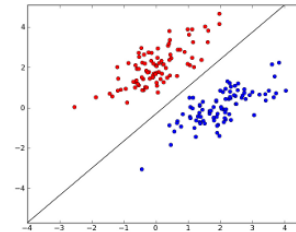
Introduction

- Transfer functions
 - Step (or sign) function
 - “Hard Limiter”
 - Linear (ramping) function
 - Sigmoid function
 - Most common because it's continuous
 - Usually used in backpropagation networks



- With two inputs, the decision boundary takes on the form of a straight line
 - So if you had a problem like this one, the perceptron could learn to solve it
 - “Linearly separable” (which extends beyond two dimensions)

Perceptron



- However, even very simple problems that are not linearly separable cannot be solved by a perceptron
 - e.g. Exclusive Or (XOR)

	x_2	0	1
x_1	0	1	0
	1	0	1

Perceptron

- Perceptron can't solve problems that are not linearly separable, but a multilayer network can
- A multilayer network has one or more hidden layers between the input and output layers
- Usually a feed-forward, backpropagation architecture

Multilayer Networks

- Feed Forward:

- Input to neuron is still

$$x_j = \sum_{i=1}^n x_i w_{i,j}$$

- n= number of connected inputs
 - x_i = the input on connection i
 - $w_{i,j}$ = the weight on the connection between neurode i and neurode j

- Transfer function is sigmoid

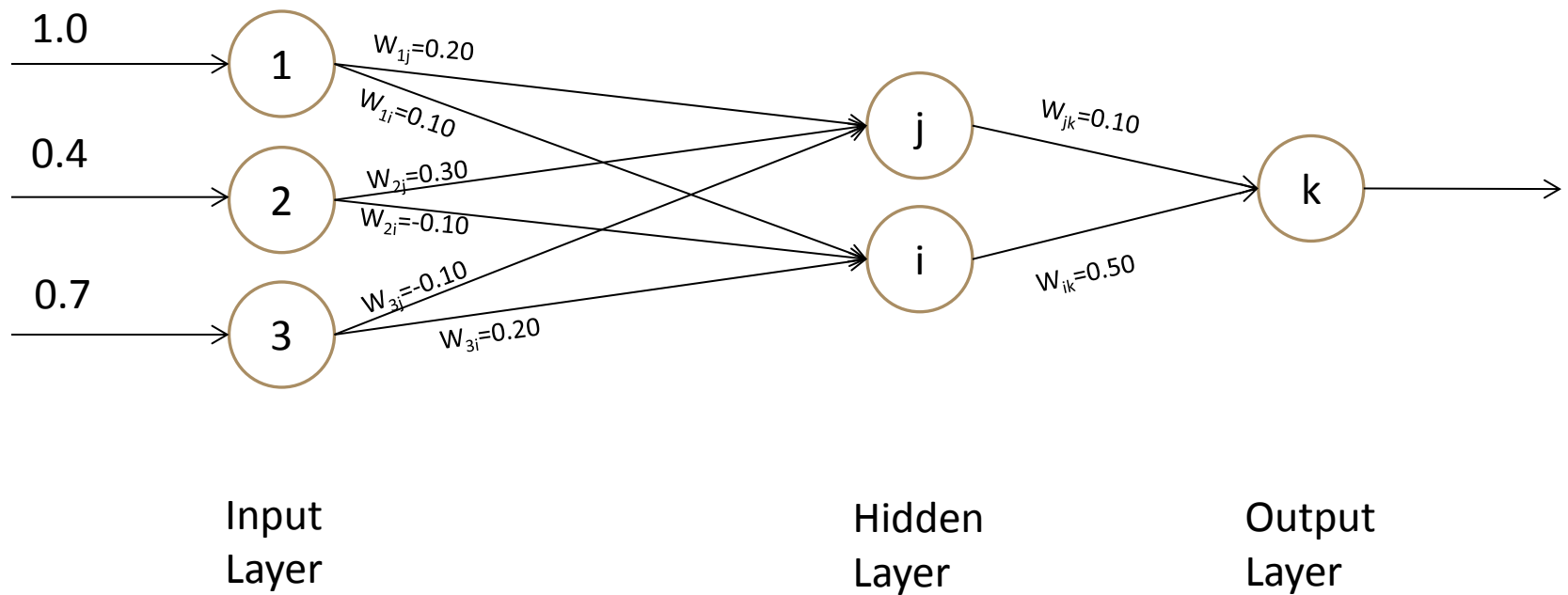
$$y_j = \frac{1}{1+e^{-x_j}}$$

- This bounds the output between 0 and 1 and is continuously differentiable

Multilayer Networks

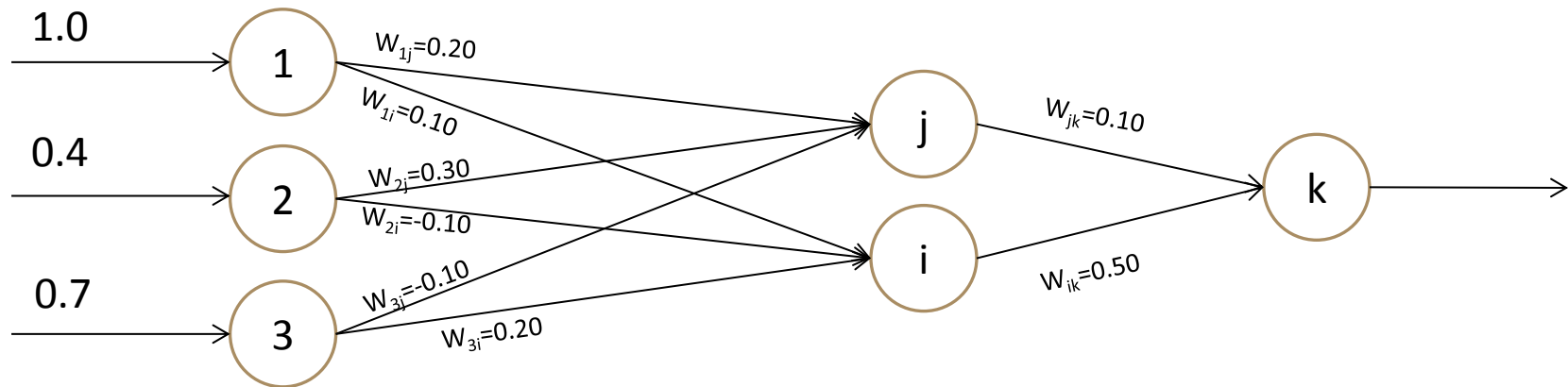
- Feed forward example

Multilayer Networks



- Feed forward example

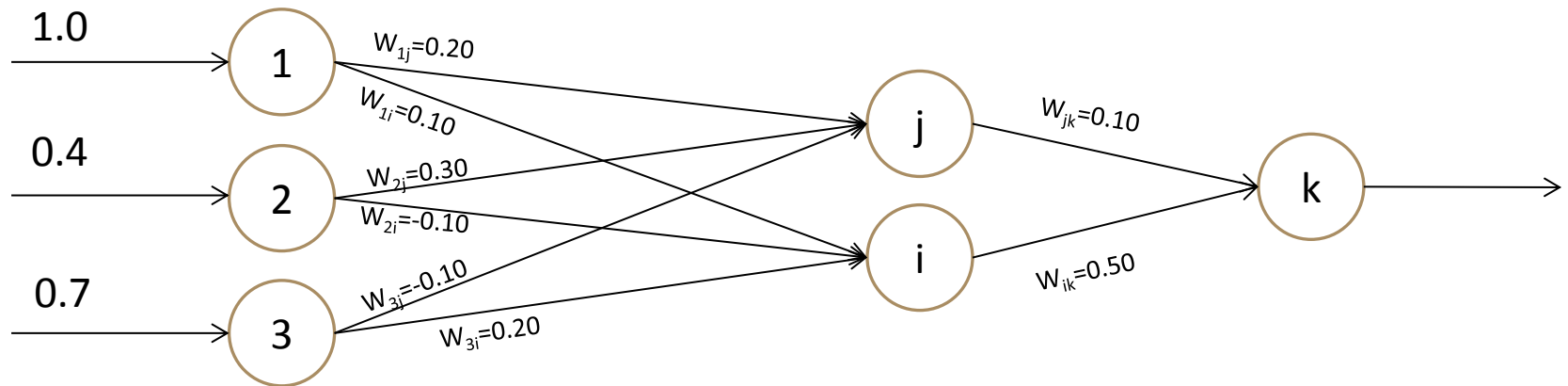
Multilayer Networks



- Input to node j = $\sum_{n=1}^3 w_{n,j} o_n$ where o_n = output of node n
- Input to node j = $w_{1j} * 1.0 + w_{2j} * 0.4 + w_{3j} * 0.7$
 - $= 0.2 * 1.0 + 0.3 * 0.4 + -0.1 * 0.7$
 - $= 0.2 + 0.12 + -0.07 = 0.25$

Multilayer Networks

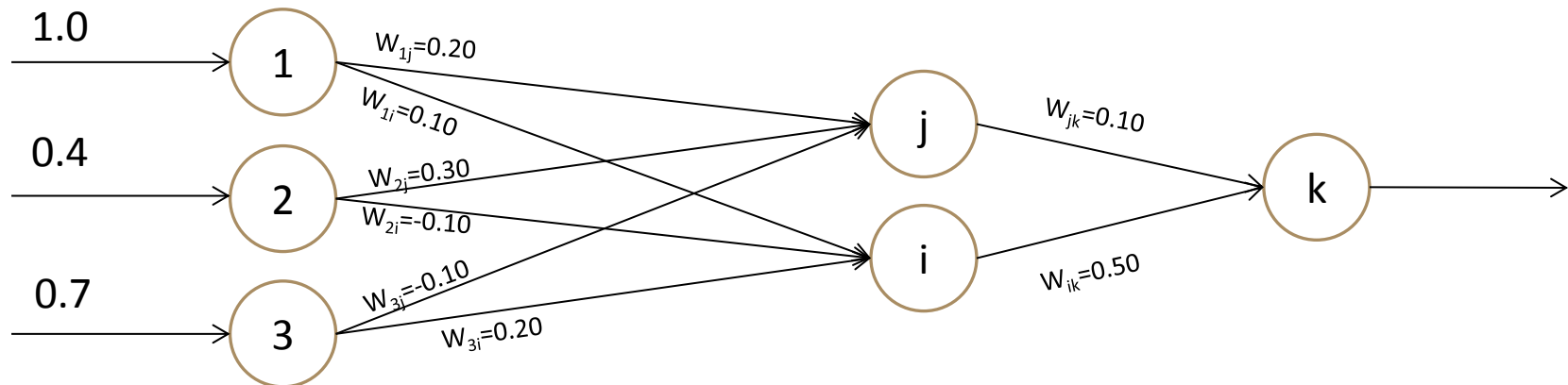
- Feed forward example



- Output from node j = $\frac{1}{1+e^{-input}} = 0.562177$

- Feed forward example

Multilayer Networks



- Input to node j = 0.25, Output from node j = 0.562
- Input to node i = 0.20, Output from node i = 0.550
- Input to node k = 0.331, Output from node k = 0.582

- Backpropagation:
 - Error at node j:
 - $Error(j) = (\sum_k Error(k) * w_{j,k}) * f'(x_j)$
 - Error(k) = output error at node k
 - w_{jk} = weight of connection between nodes j and k
 - $f'(x) = O_j (1-O_j)$
 - O_j = output at node j

Multilayer Networks

Multilayer Networks

- Backpropagation:
 - The Delta Rule:
 - $w_{jk}(\text{new}) = w_{jk}(\text{current}) + \Delta w_{jk}$
 - $\Delta w_{jk} = r * \text{Error}(k) * O_j$
 - $r = \text{learning rate}, 0 < r < 1$
 - $\text{Error}(k) = \text{error at node } k$
 - $O_j = \text{output of node } j$

Multilayer Networks

■ Backpropagation Example:

- $Error(j) = (\sum_k Error(k) * w_{j,k}) * f'(x_j)$
- Let's say we want 0.599 as our output, so Error(k) is 0.017
- $Error(j) = 0.017 * 0.10 * 0.25 = 0.00042$
 - $w_{jk}(new) = w_{jk}(current) + \Delta w_{jk}$
 - $\Delta w_{jk} = r * Error(k) * O_j$
- Let's say our learning rate, $r = 0.5$
- $\Delta w_{jk} = 0.5 * 0.017 * 0.562 = 0.0048$
- $w_{jk}(new) = 0.10 + 0.0048 = 0.1048$

- Initialization
 - Randomly initialize weights between [-0.5, 0.5]
- Activation
 - Apply inputs $x_1 \dots x_n$ and calculate the output
 - First, summation function
 - Then, transfer function – step function or sign function for perceptron, sigmoid most likely for multilayer
- Weight Adjustment
 - If the output is not what was desired, go back and adjust each weight
 - First, error function
 - Then, Delta rule
- Iterate until the error rate is acceptable (or we reach some other stopping condition)

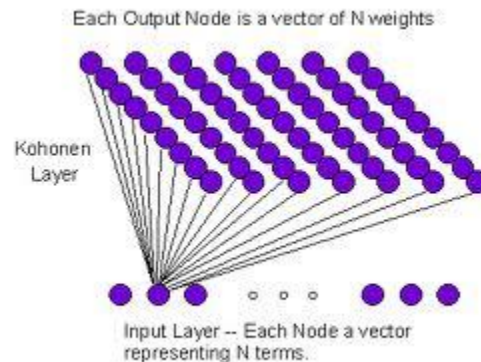
Steps in Training a Network:

- Unsupervised (!) neural network
- Competitive learning
 - Only a single output node is active for a given input
 - Winner takes all
- Kohonen's "principle of topographic map formation"
 - The spatial location of an active output neurode in the topographic map corresponds to a specific feature of the input pattern

Kohonen Self-Organizing Maps

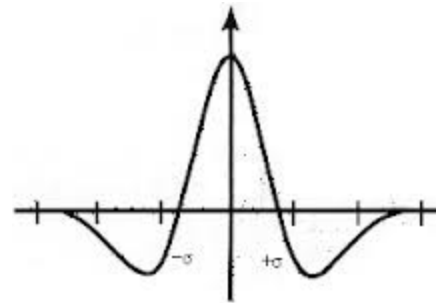
- Architecture / Behavior
 - Two layers – input and output (Kohonen layer)
 - Many more nodes in output layer than in input
 - Input layer is fully connected to the output layer
 - One input node for each input feature (attribute)

Kohonen Self-Organizing Maps



- Training / Learning
 - Input instances are presented to the input layer and fed through to the output layer
 - The single output node whose weights most closely match those of the input is the one that “wins”
 - The winner is rewarded by having its weights changed to match the input even more closely
 - Initially, those output neurodes near the winner are also rewarded
 - Size of “neighborhood” decreased as number of iterations increase
 - Mexican hat function
 - Neighborhood defined by city block or Euclidean distance
 - Output nodes winning the most instances during the last pass of the data through the network are saved
 - The number of output nodes eventually saved corresponds to the number of “classes” found by the network

Kohonen Self-Organizing Maps



- Training and Testing
 - “Epoch” is one pass of all of the training instances through the neural network
 - Rule of thumb in supervised learning is to use 80% of the data for training and 20% for testing
 - Can apply similar rule to Kohonen maps
 - Build clustering / classification network with 80% of cases and then see how remaining 20% are classified
 - Usually use root mean squared (rms) error but could also use:
 - Absolute error
 - Mean squared error

General Considerations (for all ANNs)

- Conditioning the Input
 - Input must be numeric
 - Works best if in the range of [0, 1]

General Considerations (for all ANNs)

- Categorical Input Data:
 - Divide interval range into equal sized units
 - red -> 0.00
 - green -> 0.33
 - blue -> 0.67
 - yellow -> 1.00
 - Pitfall here is it implies some sort of ordering on the data that is just not true (red < green?)
 - Use additional input nodes
 - red -> 0, 0
 - green -> 0, 1
 - blue -> 1, 0
 - yellow -> 1, 1

General Considerations (for all ANNs)

- Numeric Input Data:
 - Normalize into [0, 1] range
 - $$\text{new_value} = \frac{\text{original_value} - \text{min}}{\text{max} - \text{min}}$$
- Output Strategies
 - Reverse numeric range to scale output to original (non-normalized) input

General Considerations (for all ANNs)

- Architecture
 - Input Layer
 - Number of nodes is equal to number of inputs
 - But, may vary these to get at your data better, particularly categorical data

General Considerations (for all ANNs)

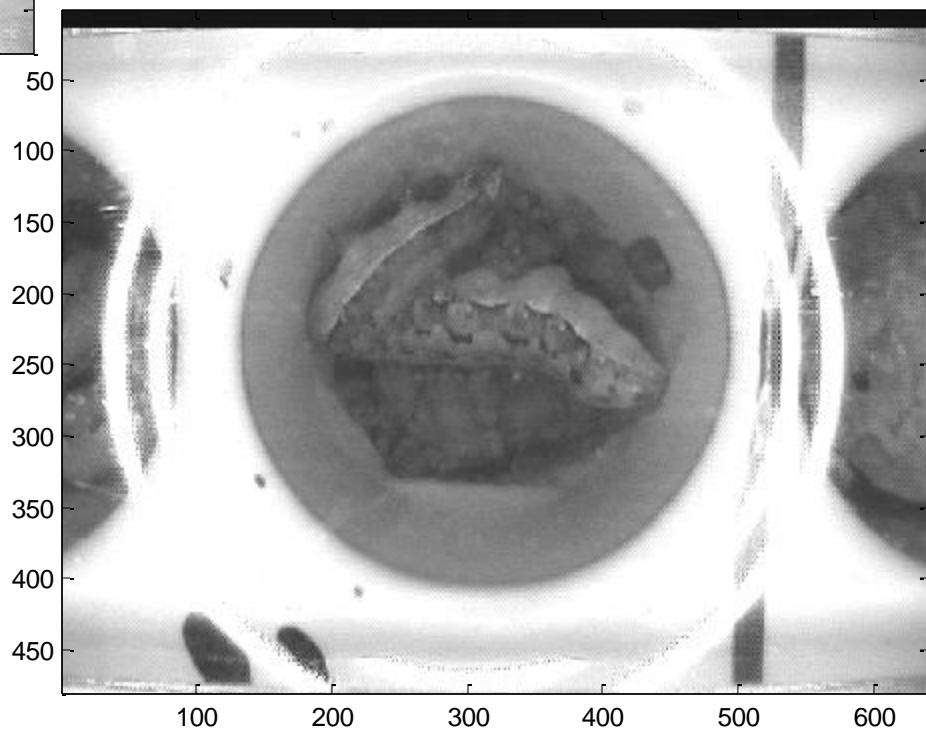
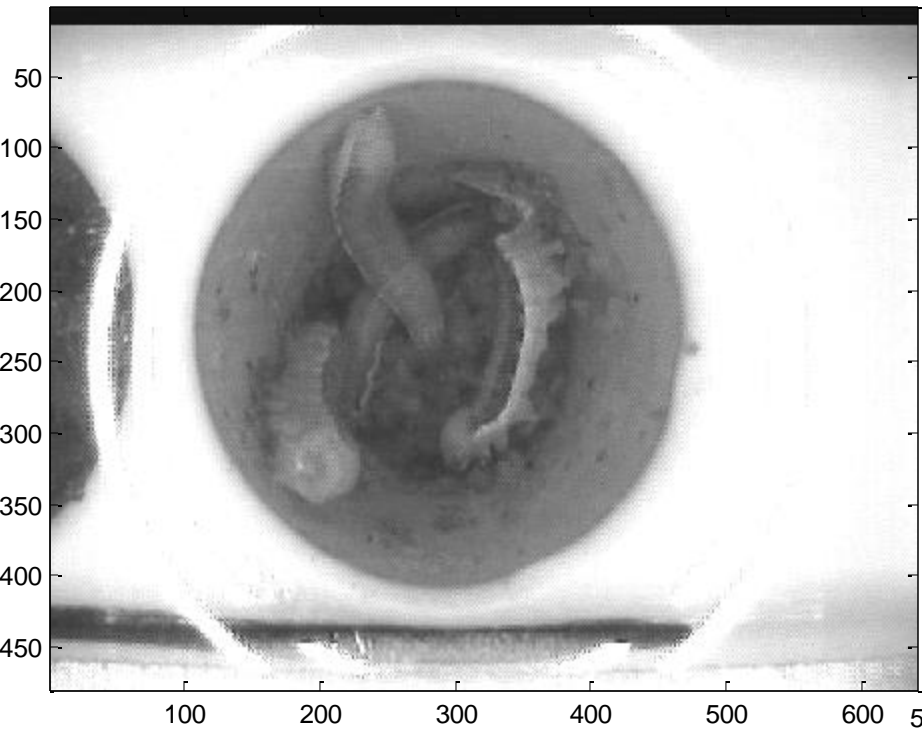
■ Architecture

- Hidden Layers
 - Need to experiment with number of layers and number of nodes in each layer
 - Best is to use the least of each and still get convergence, but you need to figure out what “least” is
 - Too many nodes/layers, network will learn training data perfectly
 - Memorizes the training examples and doesn't generalize
 - Overtraining
 - Does poorly on test data
 - Too few, won't reach convergence
 - Can get oscillatory behavior on weight adjustments

General Considerations (for all ANNs)

- Architecture
 - Output Layer
 - Depends on what you want from the output
 - May choose to add more nodes for categorical output

General Considerations (for all ANNs)



- Most brains have lots of neurons
- Perceptrons (one-layer networks) insufficiently expressive
- Multi-layer networks are sufficiently expressive; can be trained by gradient descent, i.e., error back-propagation
- Many applications: speech, driving, handwriting, fraud detection, etc.

Summary