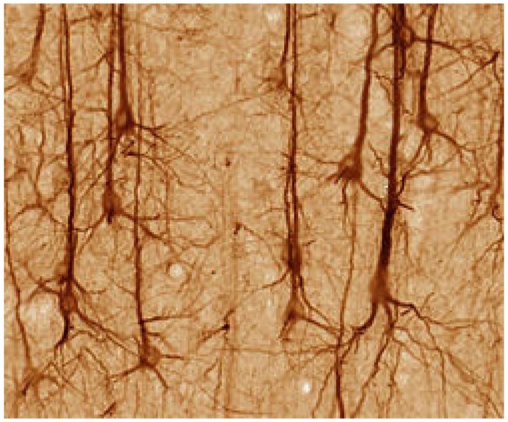
Neural Networks



CS4881 Artificial Intelligence

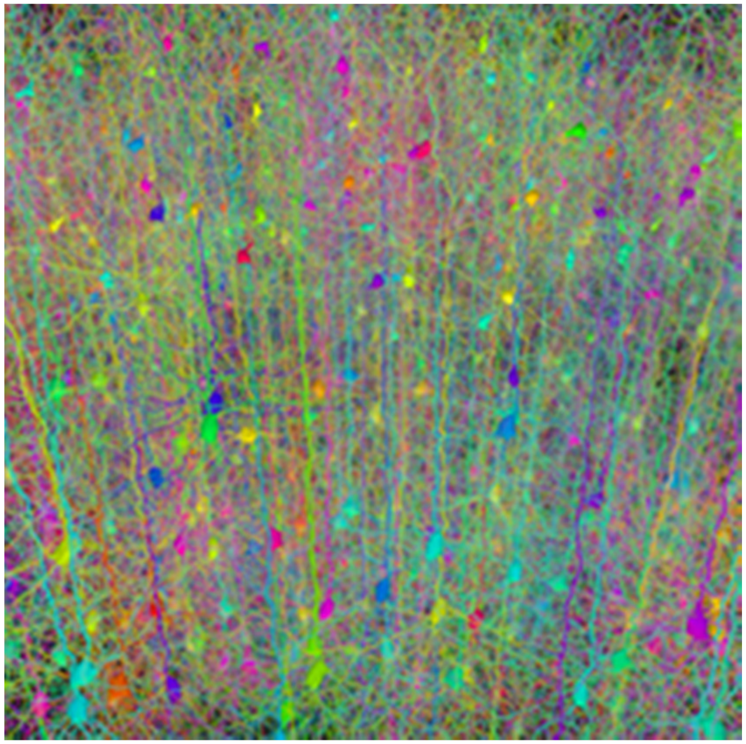
Jay Urbain, PhD

Credits:

Machine Learning, Tom Mitchell

Nazli Goharian, Georgetown; David Grossman, IIT

Dendrites of Pyramidal Neurons



The Scientific American Book of the Brain. New York: Scientific American, 1999: 3.

"An adult human brain has more than 100 billion neurons"

1000 trillion connections

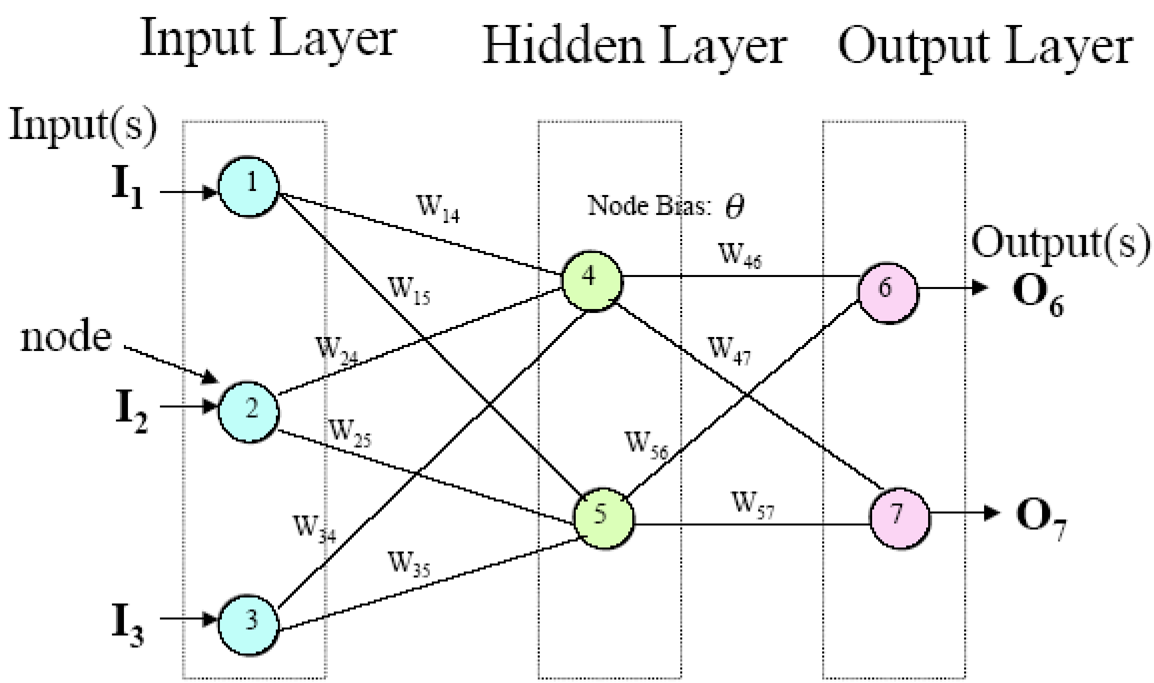
(Artificial) Neural Network Classification

- Loosely based on concept of neurons in the brain.
- Set of connected nodes with weights for nodes and arcs.
- Weights are initially assigned randomly.
- Typically involves a long learning process.
- Tolerance to noisy data.
- Subject to *overfitting*.

Neural Network Components

- Input Layer
- Hidden Layer
- Output Layer
- Connections
- Weights
- Activation Functions
- Training, test sets
- Learning algorithms

Neural Network Components



Input Layer

- Input layer has feature attributes to be used for classification.
- Select attributes by examining the data and utilizing domain knowledge.
- Inputs are the attribute values for each tuple:
  - {age=30, education=MS, salary=90,000}
- Input values should be normalized and discrete values are numerically encoded.

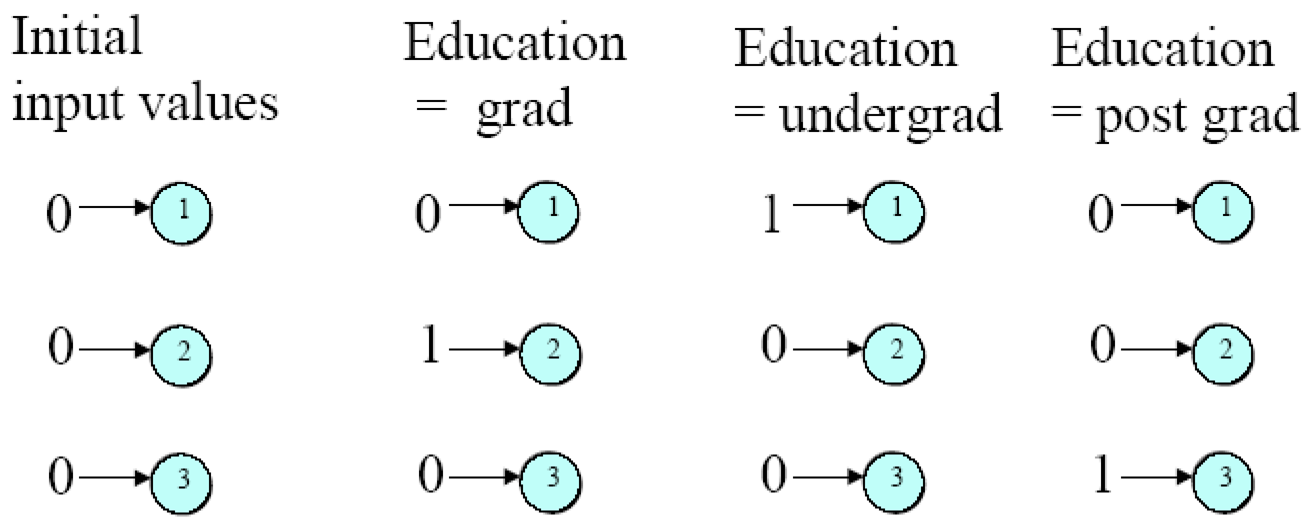
Input Layer cont.

- Number of nodes in input layer is typically defined by the number of attributes and the number of attribute types.
  - *Continuous attributes* like salary are typically normalized between {0,1} and fed into one node.
  - *Boolean attributes* are assigned {0,1} and fed into one node.
  - *Ordinal attributes* like {*freshman, sophomore, junior, senior*} can be scaled to between 0 and 1, encoded, or assigned individual nodes for each element.
  - *Categorical attributers* (or continuous attributes transformed into categories) are first encoded and one node is created for each category.

Note: Z-score can be substituted for {0,1} normalization.

Example input layer

Education: {undergrad, grad, post grad}



Hidden Layer

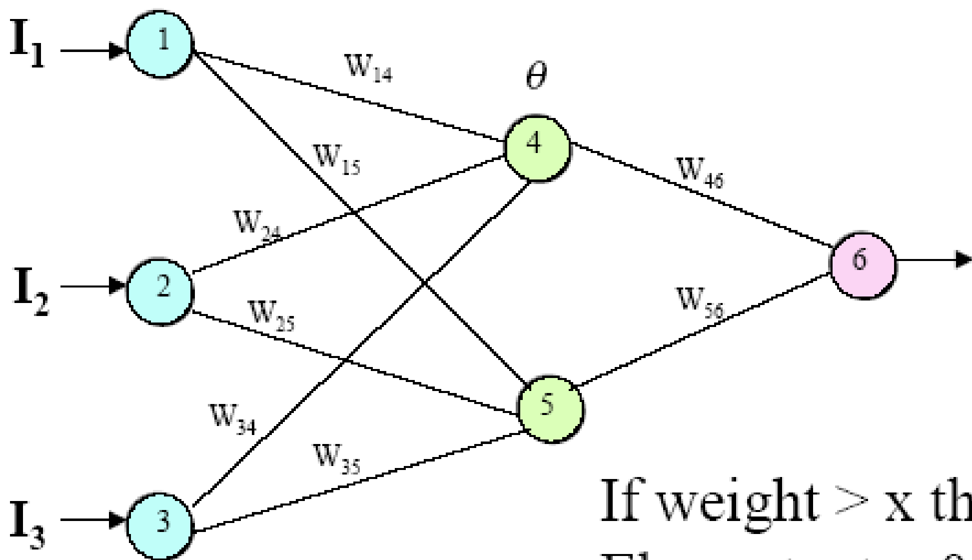
- Hidden layer allows networks to solve complex *nonlinear* problems.
- A network can have one or more hidden layers.
- The number of nodes in the hidden layer(s) is determined via experimentation. *~6 is a good start, or some number between #inputs & outputs.*
- Too many nodes => over-fitting
- Too few nodes => reduced classification accuracy.

Output Layer

- Result of the classification is the output of a node in the output layer.
- Weights and activation functions determine the output.
- Output layer may have one or more nodes.
- There is typically one output node for each class:
  - E.g., 3 output nodes for high-income, mid-income, and low-income classes.
  - If two classes, i.e., binary classification, e.g., {lucky, not lucky} you can use one node {1=lucky, 0=not lucky}.
- Can also train for continuous output value

Example Output Layer

Class labels: lucky, not lucky

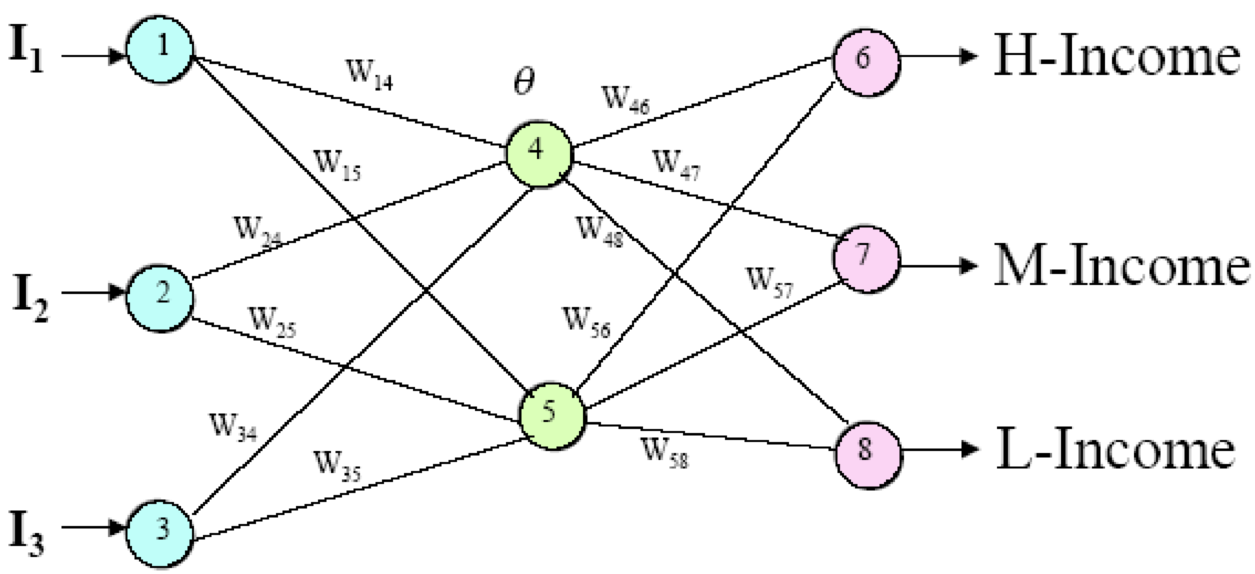


1= lucky
0= not lucky

If weight > x then output= 1
Else output = 0

Example Output Layer cont.

Class labels: H-Income, M-Income, L-Income



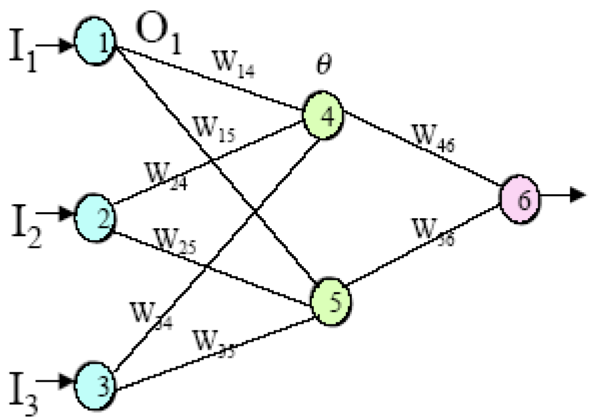
Example: Arcs and weights

In input layer: $O_i = I_i$

In hidden or output layer:

- Input to node j:

$$I_j = \left(\sum_i w_{ij} . O_i \right) + \theta_j$$



- Output from node j using *sigmoid* activation function:

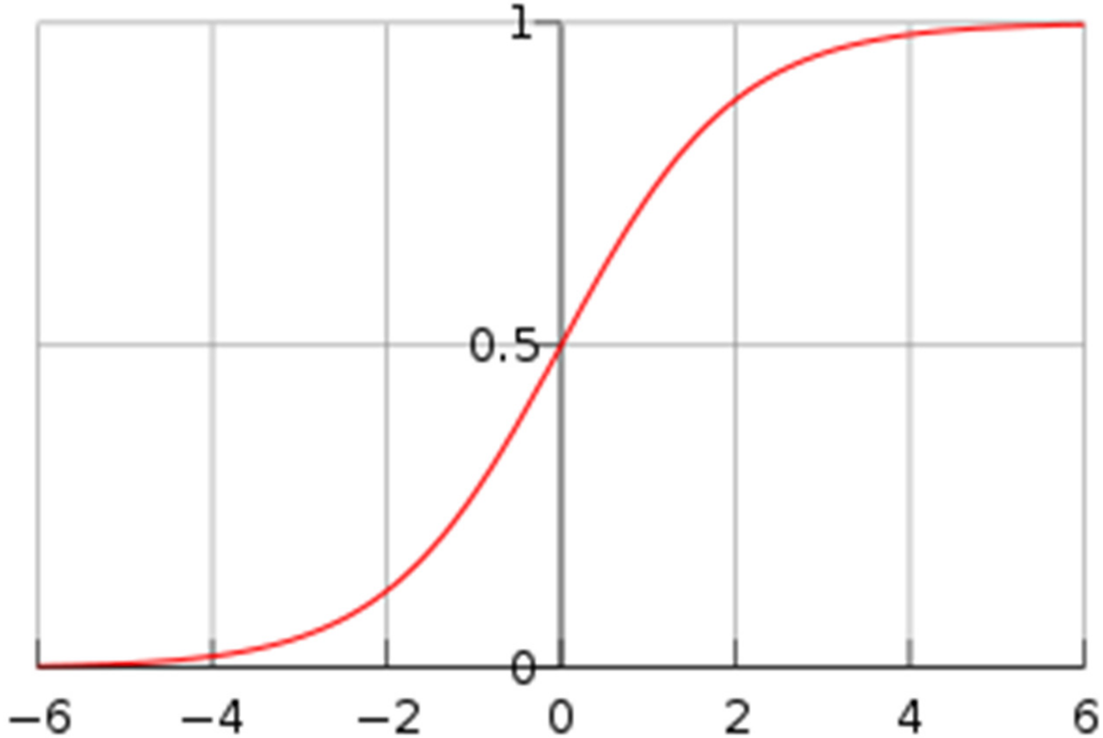
$$O_j = \frac{1}{1 + e^{-I_j}}$$

Activation Function

- Different activation functions can be used, *sigmoid* is typical.
- Also called a *squashing function*
  - as it squashes the output value into a range of {0 to 1} to reduce the weak or gray area by pushing it toward one class or another.
  - Mimic all or nothing principal of neuronal firing.

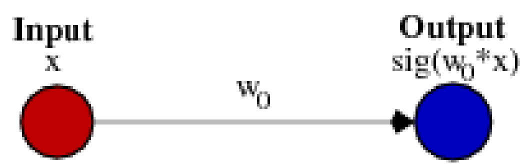
Sigmoid Activation Function

- Logistic Curve



Role of Bias

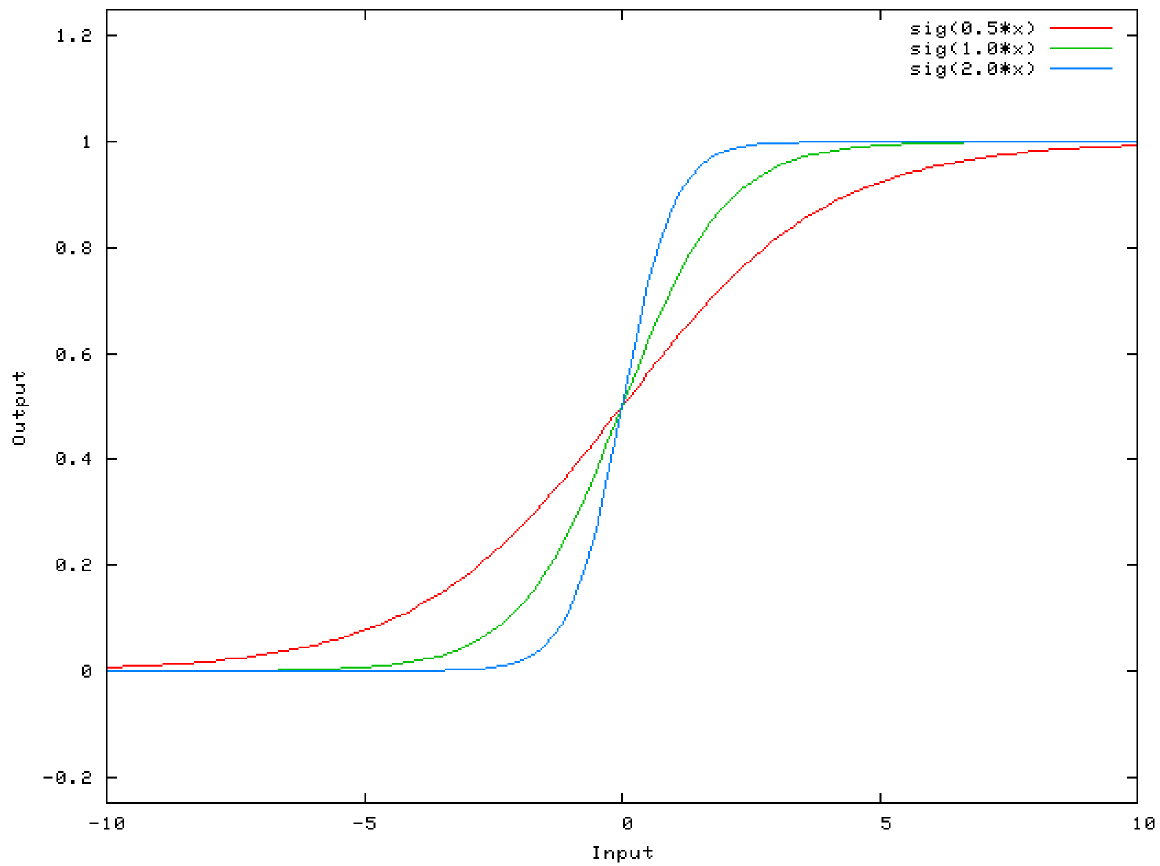
- A bias value allows you to shift the activation function to the left or right, which may be critical for successful learning.
- Consider this 1-input, 1-output network that has no bias:



- The output of the network is computed by multiplying the input (x) by the weight (w_0) and passing the result through some kind of activation function.

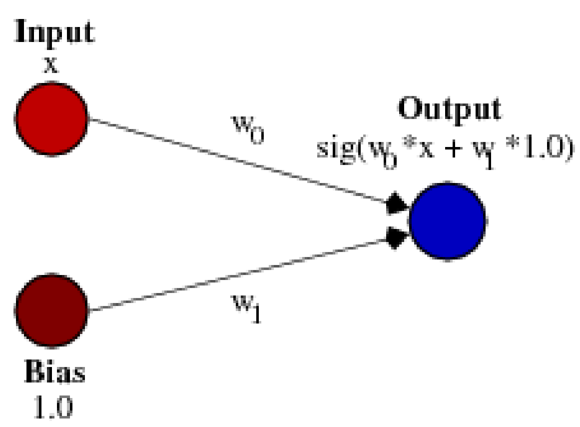
Role of Bias

- Here is the function that this network computes, for various values of w_0:



Role of Bias

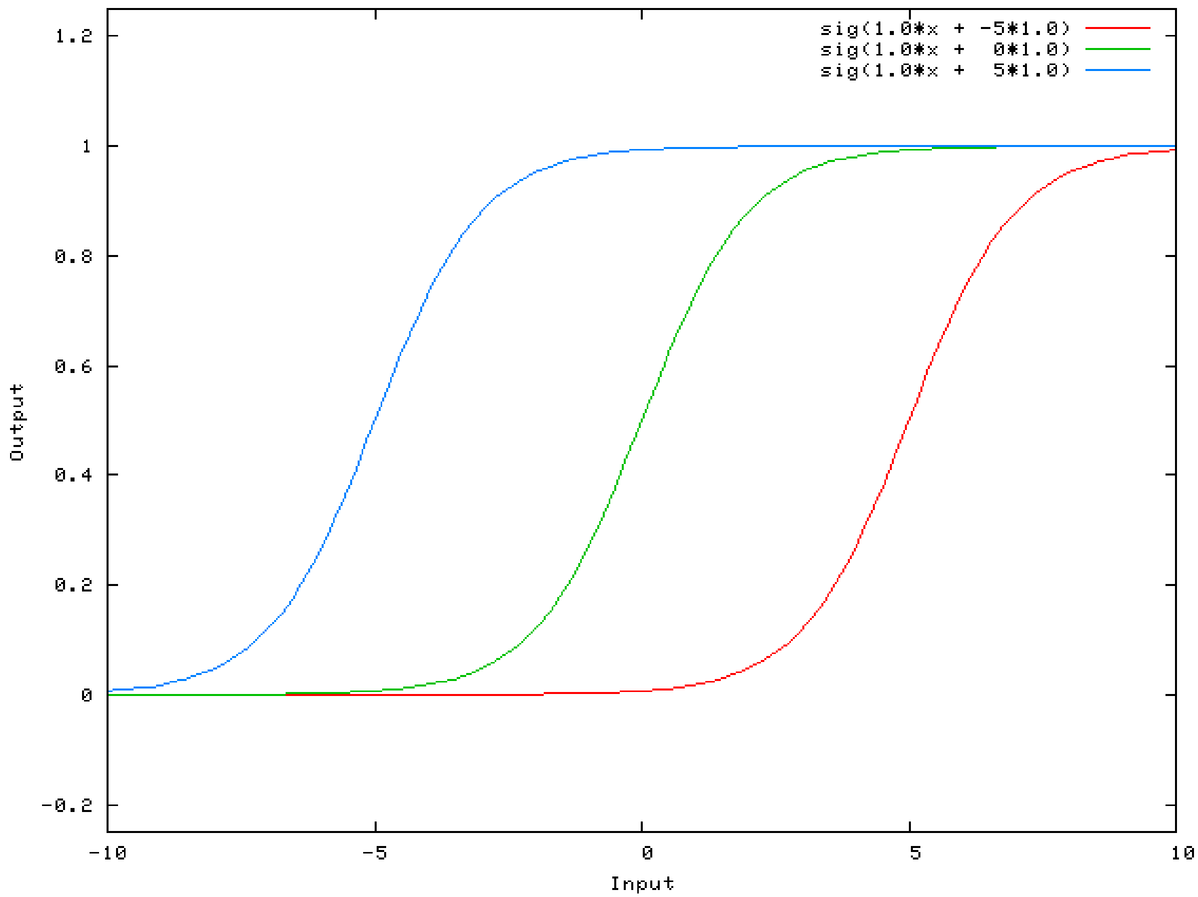
- Changing the weight w_0 changes the "*steepness*" of the sigmoid.
- That's useful, but what if you wanted the network to output 0 when x is 2?
- Just changing the steepness of the sigmoid won't really work -- **you want to be able to shift the entire curve to the right**.
- That's exactly what the bias allows you to do. If we add a bias to that network, like so:



Role of Bias

- ...then the output of the network becomes sig(w_0*x + w_1*1.0). Here is what the output of the network looks like for various values of w_1:

Having a weight of -5 for w_1 shifts the curve to the right, which allows us to have a network that outputs 0 when x is 2.



Training a Neural Network

- Run an example from the training set, by giving its attribute values as input (normalized of course!).

- *Feed-forward* process
  - Summation of weights and activation functions are applied at each node of hidden and output layers, until an output is generated.

- *Back-propagation* process
  - If the output does not match, go back from the output layer to each hidden layer, (layer by layer) and modify the arc weights and biases of nodes.

- Eventually the weights will (*should*) converge and processing stops.

Feed-Forward Process

- Process starts from input nodes to hidden nodes:

For each training sample X do

For each hidden or output layer node j

Calculate input I_J to that node: $I_j = \left(\sum_i w_{ij} . O_i \right) + \theta_j$

Calculate output O_J from that node: $O_j = \frac{1}{1 + e^{-I_j}}$

- At this point, the final output is generated.

Back Propagation Process

- Calculate *error* and update weights

  For each node j in the *output layer* do

  Calculate the error:

$$Err_j = \underbrace{O_j\left(1-O_j\right)}_{\text{Derivative of squashing function}}\left(T-O_j\right)$$

Expected result

- For each node j in (each) *hidden layer* (last to first), and for each output k

  Calculate the error:

$$Err_j = O_j\left(1-O_j\right)\left(\sum_k Err_k \,.\, w_{jk}\right)$$

Derivative of Sigmoid

$$\frac{ds(x)}{dx} = \frac{1}{1+e^{-x}}$$

$$= \left(\frac{1}{1+e^{-x}}\right)^2 \frac{d}{dx}(1+e^{-x})$$

$$= \left(\frac{1}{1+e^{-x}}\right)^2 e^{-x}(-1)$$

$$= \left(\frac{1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right)(-e^{-x})$$

$$= \left(\frac{1}{1+e^{-x}}\right)\left(\frac{-e^{-x}}{1+e^{-x}}\right)$$

$$= s(x)(1-s(x))$$

Back Propagation Process cont.

- For each weight w_{ij}

Calculate weight increment: $\Delta w_{ij} = \underset{\text{Learning rate}}{\underset{\downarrow}{l}} \, . \, Err_j \, . \, O_i$

Update weight: $w_{ij} = w_{ij} + \Delta w_{ij}$

- For each node, calculate total error, update bias

Calculate the error: $Err_j = O_j\left(1 - O_j\right)\left(\sum_k Err_k \, . \, w_{jk}\right)$

Epochs

- Iterations of forward and backward propagation continue until "convergence."
- These iterations are referred to as training "epochs."
- Too many epochs can contribute to over fitting.
- Too many hidden layer nodes can also lead to over fitting.
- Experimentation is key.

Current Research

- Cognitive science – simulate brain.
- Computer science – improve machine learning.
- Cortex has 4 to 6 layers.
- Gregory Hinton and Yuan Lucan are working on learning algorithms to train NN with many hidden layers.
- Jeff Hawkins working on hierarchical temporal memory.

Hierarchical temporal memory (HTM)

- Models some of the structural and algorithmic properties of the *neocortex* using an approach somewhat similar to Bayesian networks.
- HTM model is based on the memory-prediction theory of brain function described by Jeff Hawkins in his book *On Intelligence*.
- HTMs are claimed to be biomimetic models of cause inference in intelligence

Summary

- Set of connected nodes along with the weights for nodes and arcs.
- Different network topologies based on trial and error, though there has been considerable research into which topologies are optimal for different classes of problems.
- Strengths:
  - Tolerance to noise
  - Works well with complex nonlinear problems that are difficult to characterize.
  - *Can be highly accurate.*
- Weaknesses
  - Not intuitive, difficult to extract human understandable description from learned weights
  - Can have a long learning process
  - Prone to *overfitting*

Example Neural Network

Following are codes snippets showing the details of:

- *Forward propagation*
- *Backward propagation*
- *Weight updating*
- *Bias updating*

1) // initialize weights

```
weightsIJ = new double [numInputLayerNodes][numHiddenLayerNodes];
weightsJK = new double [numHiddenLayerNodes][numOutputLayerNodes];

for(int i=0; i< numInputLayerNodes; i++) {
    for(int j=0; j< numHiddenLayerNodes; j++) {
        weightsIJ[i][j] = random.nextDouble()-0.5;
    }
}
for(int j=0; j< numHiddenLayerNodes; j++) {
    for(int k=0; k< numOutputLayerNodes; k++) {
        weightsJK[j][k] = random.nextDouble()-0.5;
    }
}
```

2) // forward propagation

```
// propagate to hidden layer
// input = sum(Wik * Oi)

for(int j=0; j< numHiddenLayerNodes; j++) {
    double weightedSum = 0;
    for(int i=0; i< numInputLayerNodes; i++) {
        weightedSum += weightsIJ[i][j] * ((Node) inputLayerI.get(i)).getOutput();
    }
    currentNode = (Node) hiddenLayerJ.get(j);
    currentNode.setInput(weightedSum + currentNode.getBias() );
    currentNode.setOutput( 1/(1+ Math.exp(-currentNode.getInput()) ) );
}
```

2) // forward propagation

// <u>propogate to output layer</u>
// input = sum(<u>Wjk * Oj)</u>

```
for(int k=0; k< numOutputLayerNodes; k++) {
    double weightedSum = 0;
    for(int j=0; j< numHiddenLayerNodes; j++) {
        weightedSum += weightsJK[j][k] * ((Node) hiddenLayerJ.get(j)).getOutput();
    }
    currentNode = (Node) outputLayerK.get(k);
    currentNode.setInput(weightedSum + currentNode.getBias() );
    currentNode.setOutput( 1/(1+ Math.exp(-currentNode.getInput()) ) );
}
```

3) // Back propagation

```
// calc output & output layer Error
// Errk = Ok(1 - Ok)(Tk - Ok)

double Tk = classIndex; // need to expand this for multiple output nodes
double Ok = 0;
for(int k=0; k< numOutputLayerNodes; k++) {
    currentNode = (Node) outputLayerK.get(k);
    Ok = currentNode.getOutput();// Ouput value for this output node
    currentNode.setError(Ok*(1 - Ok)*(Tk - Ok));
}
```

3) // back propagation

```
// calc hidden layer Error
// Errj = Oj(1 - Ok)*sum(Errk*Wjk)
double Oj = 0;
for(int j=0; j< numHiddenLayerNodes; j++) {
    double weightedSum = 0;
    for(int k=0; k< numOutputLayerNodes; k++) {
        weightedSum += weightsJK[j][k] * ((Node) outputLayerK.get(k)).getError();
    }
    currentNode = (Node) hiddenLayerJ.get(j);
    Oj = currentNode.getOutput();// Ouput value for this output node
    currentNode.setError(Oj*(1 - Oj)*weightedSum);
}
```

3) // back propagation

```
// calc input layer Error
// Erri = Oi(1 - Oj)*sum(Errj*Wij)
double Oi = 0;
for(int i=0; i< numInputLayerNodes; i++) {
    double weightedSum = 0;
    for(int j=0; j< numHiddenLayerNodes; j++) {
        weightedSum += weightsIJ[i][j] * ((Node) hiddenLayerJ.get(j)).getError();
    }
    currentNode = (Node) inputLayerI.get(i);
    Oi = currentNode.getOutput();// Ouput value for this output node
    currentNode.setError(Oi*(1 - Oi)*weightedSum);
}
```

3) // update weights

```
// update weights jk
double dWeight, dBias = 0;
for(int j=0; j< numHiddenLayerNodes; j++) {
    Node hiddenNode = (Node) hiddenLayerJ.get(j);
    for(int k=0; k< numOutputLayerNodes; k++) {
        Node outputNode = (Node) outputLayerK.get(k);
        dWeight = learningRate * outputNode.getError() * hiddenNode.getOutput();
        weightsJK[j][k] = weightsJK[j][k] + dWeight;
    }
}
for(int i=0; i< numInputLayerNodes; i++) {
    Node inputNode = (Node) inputLayerI.get(i);
    for(int j=0; j< numHiddenLayerNodes; j++) {
        Node hiddenNode = (Node) hiddenLayerJ.get(j);
        dWeight = learningRate * hiddenNode.getError() * inputNode.getOutput();
        weightsIJ[i][j] = weightsIJ[i][j] + dWeight;
    }
}
```

3) // update bias

```
// update output layer bias
for(int k=0; k< numOutputLayerNodes; k++) {
    Node outputNode = (Node) outputLayerK.get(k);
    outputNode.setBias( outputNode.getBias() + (learningRate * outputNode.getError()) );
}

// update hidden layer bias
for(int j=0; j< numHiddenLayerNodes; j++) {
    Node hiddenNode = (Node) hiddenLayerJ.get(j);
    hiddenNode.setBias( hiddenNode.getBias() + (learningRate * hiddenNode.getError()) );
}
```