Lecture 15: Neural Networks

CS 486/686: Introduction to Artificial Intelligence

Plan for Today

- Supervised learning
  - Neural Networks
  - Deep Learning

Limits of DTs and Linear Models

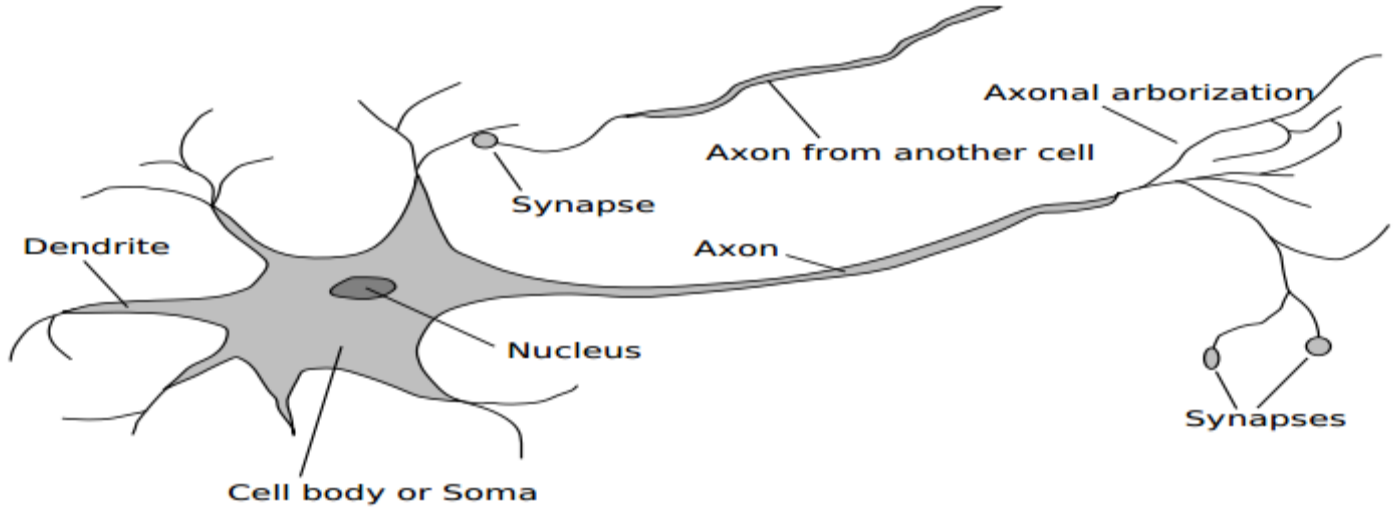
- Linear models
  - Can handle large number of input variables
  - But computation from each input to output is very short
    - No interaction b/w input variables
    - Can only represent linear functions and boundaries in input space
- Decision trees
  - Can allow long computation paths that depend on many input variables
    - But only for a small fraction of input vectors
    - Otherwise exponentially large path in number of input variables

Introduction

- The relationship between input and output can be arbitrarily complex
- We want
  - Arbitrary relationships
  - Ensure methods are efficient and do not overfit
- Idea: Mimic the human brain to do computation!
  - (in reality, this is not actually how brains work, but people like the metaphor)

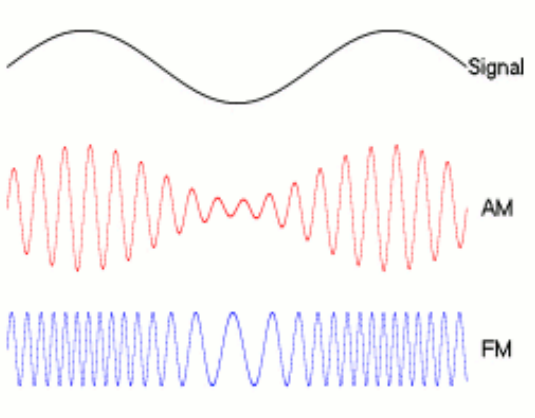
Human Brains

- A brain is a set of densely connected neurons.
- A neuron has several parts:
  - Dendrites: Receive inputs from other cells
  - Soma: Controls activity of the neuron
  - Axon: Sends output to other cells
  - Synapse: Links between neurons



Human Brains

- Neurons have two states
  - Firing, not firing
- All firings are the same



- Rate of firing communicates information (FM)
- Activation passed via chemical signals at the synapse between firing neuron's axon and receiving neuron's dendrite
- Learning causes changes in how efficiently signals transfer across specific synaptic junctions.

Artificial Neural Networks

- Idea: mimic the brain to do computation
- Artificial Neural Networks:
  - Nodes (aka units) correspond to neurons
  - Links correspond to synapses
- Computation:
  - Numerical signals transmitted between nodes corresponds to chemical signals between neurons
  - Nodes modifying numerical signal correspond to neurons firing rate

Artificial Brains?

An artificial neuron (McCulloch and Pitts 1943)

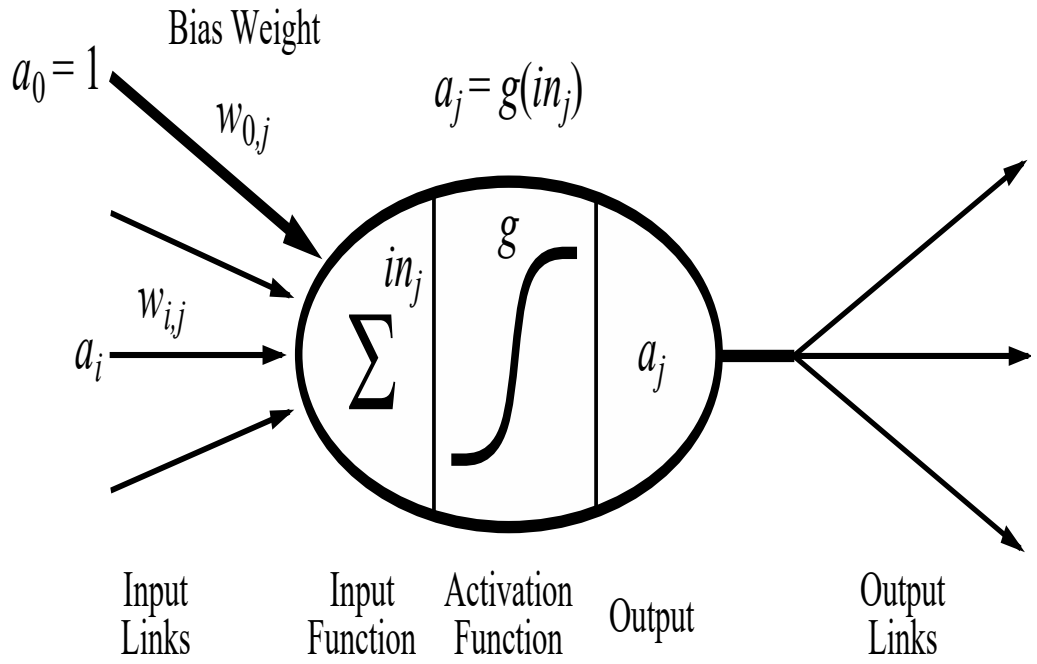
Link~ Synapse

Weight ~ Efficiency

Input Fun.~ Dendrite

Activation Fun.~ Soma

Output = Fire or not



ANN Unit

- For each unit i:

- Weights: W
  - Strength of the link from unit i to unit j
  - Input signals x_i weighted by W_{ji} and linearly combined
  - $A_j = \sum_i W_{ji}\, x_i + W_{j0} = \mathbf{W_j}\bar{\mathbf{x}}$
- Activation function: g
  - Numerical signal produced by $y_j = g(a_j)$

Activation Functions

- Should be non-linear
  - Otherwise the network is linear
- Often chosen to mimic firing neurons
  - Unit should be ”active” (close to 1) when fed the right input
  - Unit should be “inactive” (output near 0) when fed the wrong input

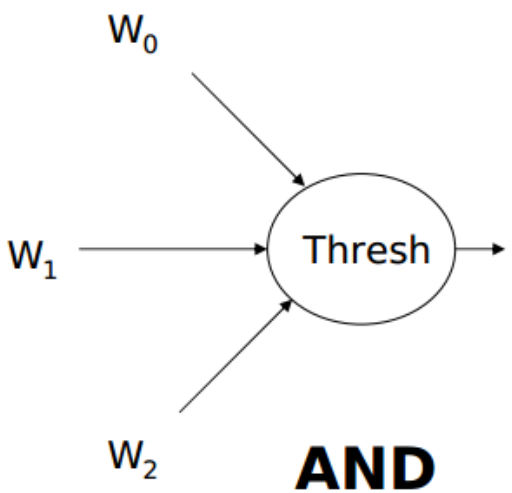
Activation Functions

- Rectified Linear Unit (ReLU): $g(x)=\max\{0,x\}$
- Sigmoid Functions: $g(x)=1/(1+e^{x})$
- Hyperbolic Tangent: $g(x)=\tanh(x)=(e^{2x}-1)/(e^{2x}+1)$

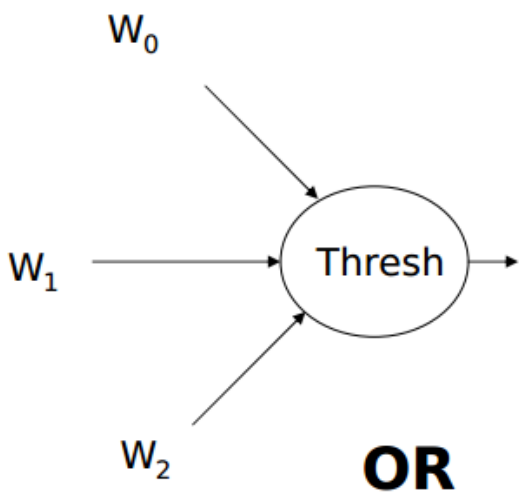
- Threshold Function: $g(x)=1$ if $x \geq b$, 0 otherwise
  - (not really used in practice often but useful to explain concepts)

Logic Gates

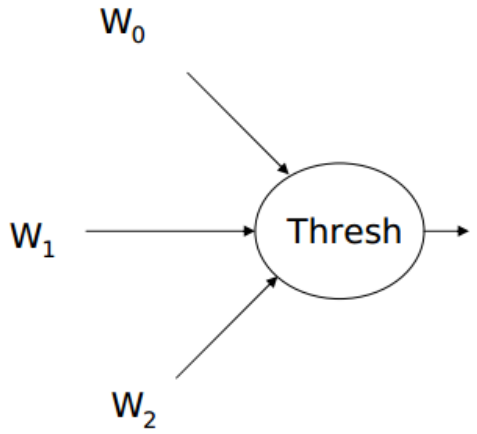
- It is possible to construct a universal set of logic gates using the neurons described (McCulloch and Pitts 1943)



AND



OR



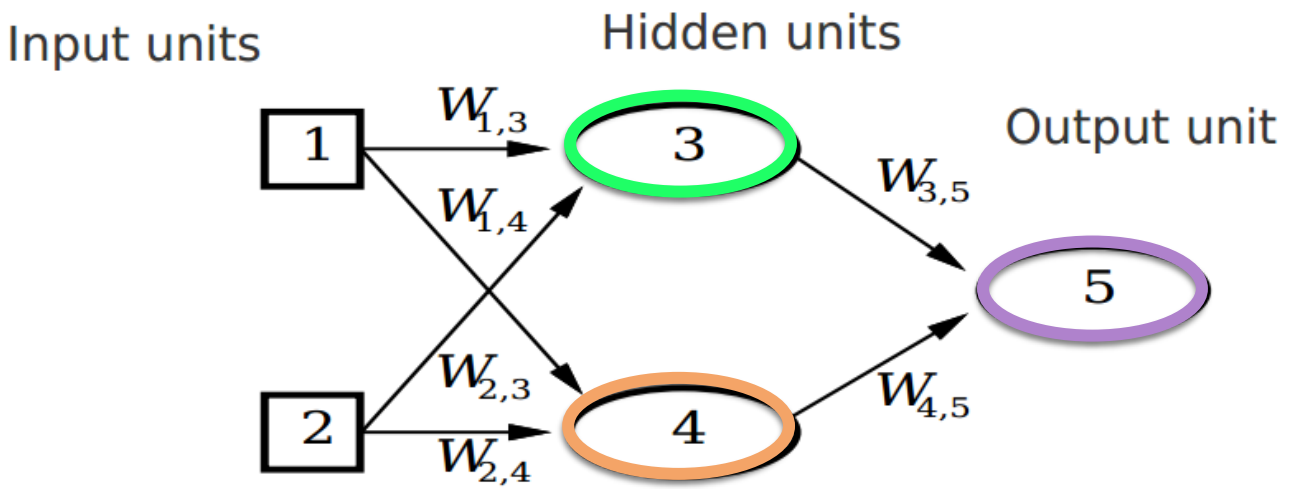
NOT

Simple Feedforward Networks

- FFN structure
  - *Directed acyclic graph* with designated input and output nodes
  - Node: *unit* of the neural net
    - Calculates weighted sum of inputs from predecessor nodes
    - Applies nonlinear function to produce output
    - a_j: output of unit j
    - w_{ij}: weight attached to link from unit i to unit j
    - g_j: nonlinear activation function

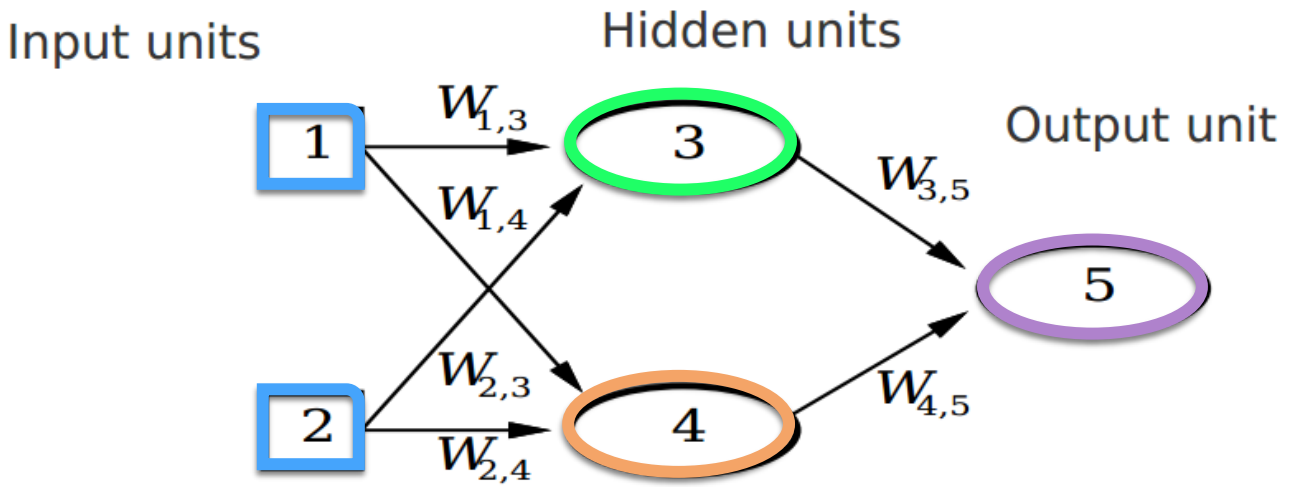
$$a_j = g_j(\sum_i w_{i,j} a_i) \equiv g_j(in_j), \qquad a_j = g_j(\mathbf{w}^\top \mathbf{x}) \qquad (22.1)$$

Example



$$a_5 = g(W_{3,5} \cdot a_5 + W_{4,5} \cdot a_4)$$

Example

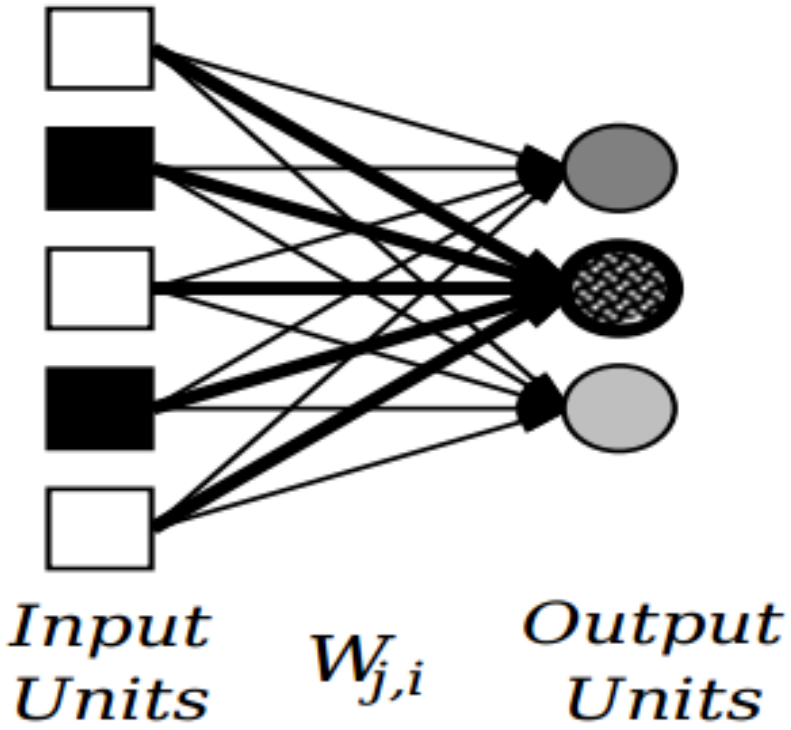


$$a_5 = g(W_{3,5} \cdot a_5 + W_{4,5} \cdot a_4)$$

$$a_5 = g(W_{3,5} \cdot g(W_{1,3} \cdot a_1 + W_{2,3} \cdot a_2) + W_{4,5} \cdot g(W_{1,4} \cdot a_1 + W_{2,4} \cdot a_2))$$

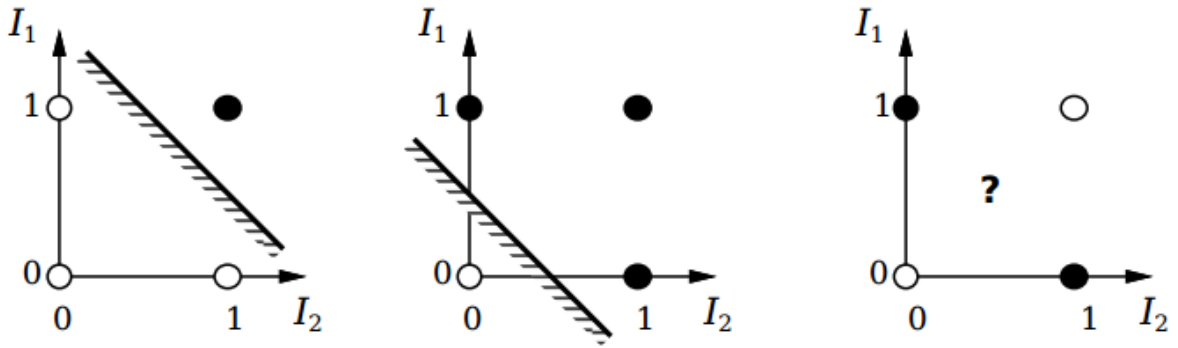
Perceptrons

- Single layer feed-forward networks



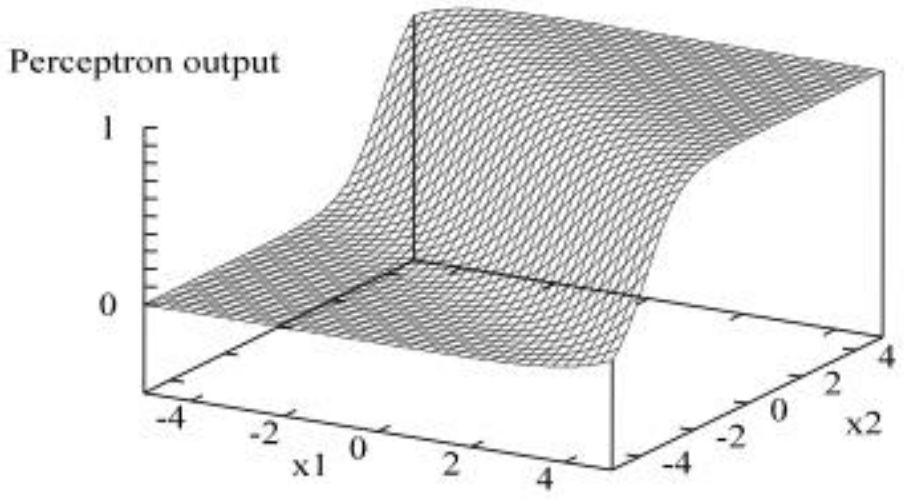
Perceptrons

- Hypothesis space $h_{\mathbf{w}}$
  - All binary classifications with parameters **w** such that
    - $\mathbf{w}^T \bar{x} > 0 \rightarrow +1$
    - $\mathbf{w}^T \bar{x} < 0 \rightarrow -1$
  - Since $\mathbf{w}^T \bar{x}$ is linear in w, perceptron is called a linear separator



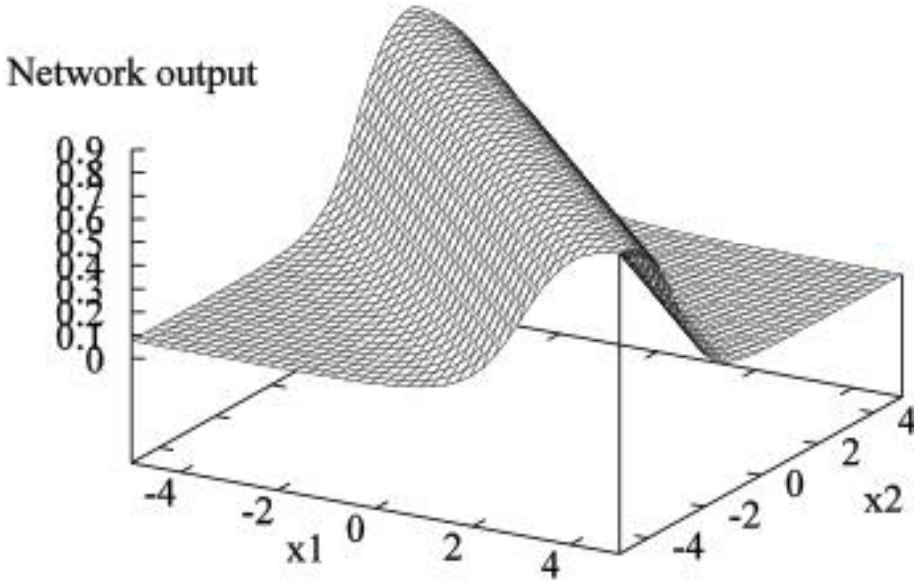
Sigmoid Perceptrons

- Represent “soft” linear separators



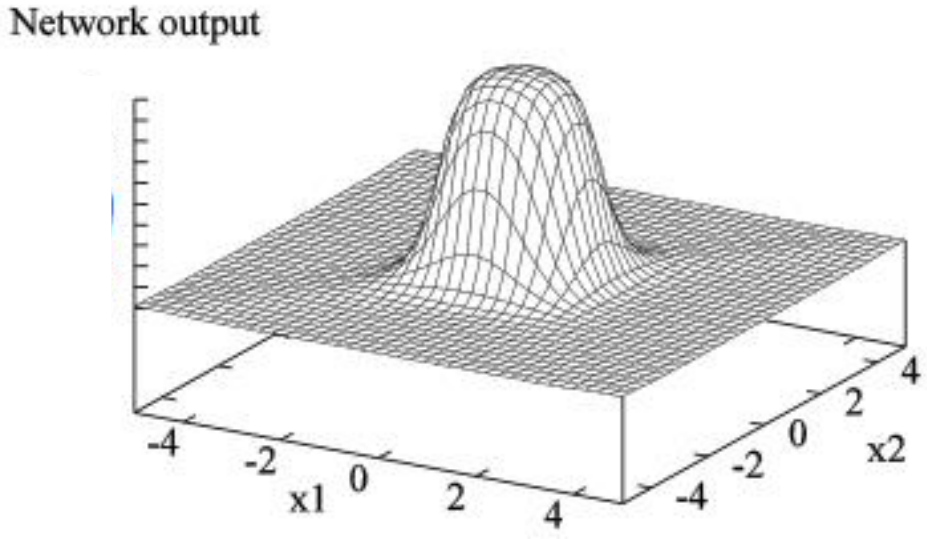
Multilayer Networks

- Adding two sigmoid units with parallel but opposite “cliffs” produces a ridge



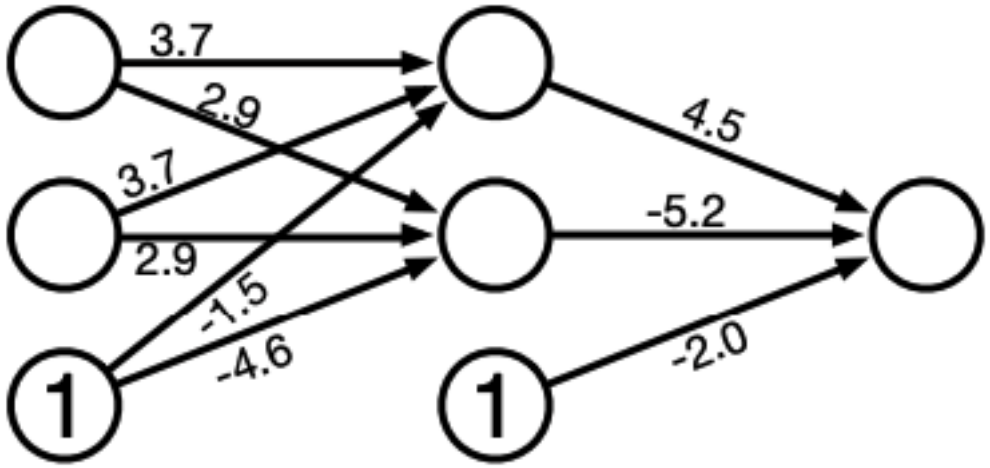
Multilayer Networks

- Adding two interesting ridges (and thresholding) produces a bump



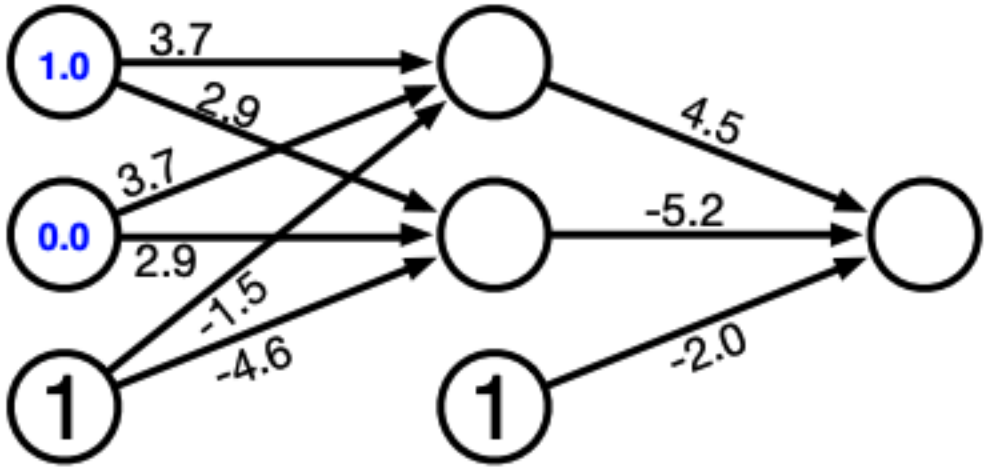
Theorem: Neural networks with at least one hidden layer of sufficiently many sigmoid units can approximate any function arbitrarily closely.

Computing using Neural Nets



slides adapted from
Philipp Koehn

Computing using Neural Nets

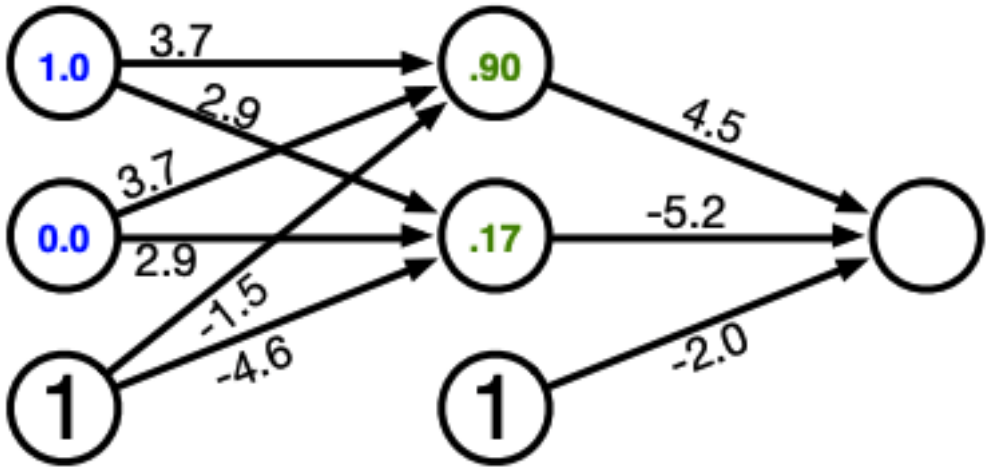


- Suppose have 2 input values
- Hidden units

$$\text{sigmoid}(1.0 \times 3.7 + 0.0 \times 3.7 + 1 \times -1.5) = \text{sigmoid}(2.2) = \frac{1}{1 + e^{-2.2}} = 0.90$$

$$\text{sigmoid}(1.0 \times 2.9 + 0.0 \times 2.9 + 1 \times -4.5) = \text{sigmoid}(-1.6) = \frac{1}{1 + e^{1.6}} = 0.17$$

Computing using Neural Nets

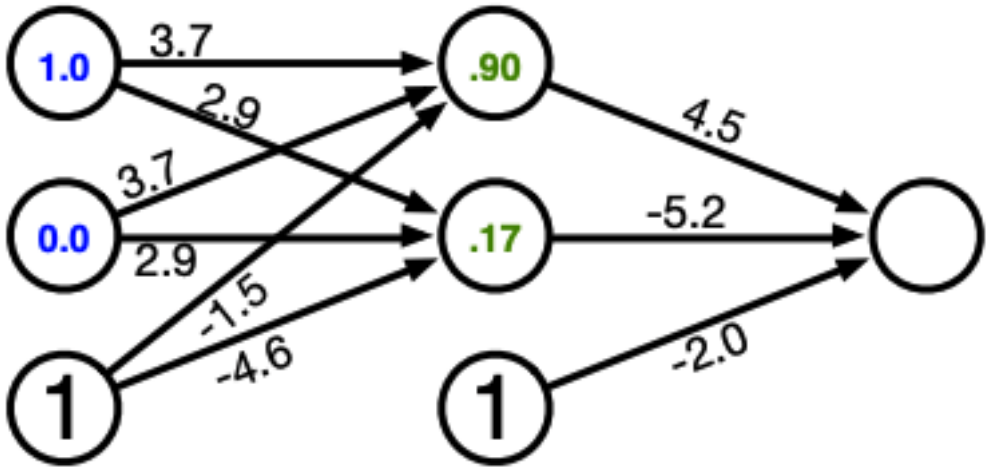


- Suppose have 2 input values
- Hidden units

$$\text{sigmoid}(1.0 \times 3.7 + 0.0 \times 3.7 + 1 \times -1.5) = \text{sigmoid}(2.2) = \frac{1}{1 + e^{-2.2}} = 0.90$$

$$\text{sigmoid}(1.0 \times 2.9 + 0.0 \times 2.9 + 1 \times -4.5) = \text{sigmoid}(-1.6) = \frac{1}{1 + e^{1.6}} = 0.17$$

Computing using Neural Nets



- Hidden units

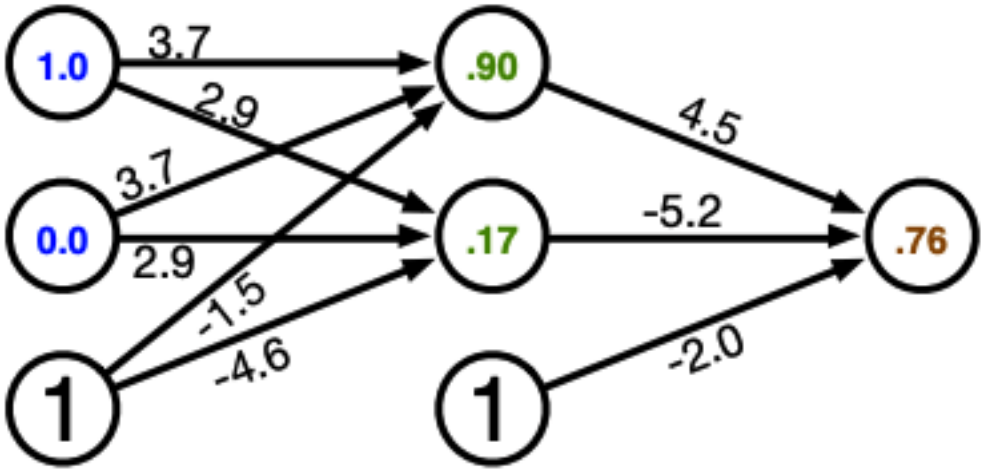
$$\text{sigmoid}(1.0 \times 3.7 + 0.0 \times 3.7 + 1 \times -1.5) = \text{sigmoid}(2.2) = \frac{1}{1+e^{-2.2}} = 0.90$$

$$\text{sigmoid}(1.0 \times 2.9 + 0.0 \times 2.9 + 1 \times -4.5) = \text{sigmoid}(-1.6) = \frac{1}{1+e^{1.6}} = 0.17$$

- Output unit

$$\text{sigmoid}(.90 \times 4.5 + .17 \times -5.2 + 1 \times -2.0) = \text{sigmoid}(1.17) = \frac{1}{1+e^{-1.17}} = 0.76$$

Computing using Neural Nets



- Hidden units

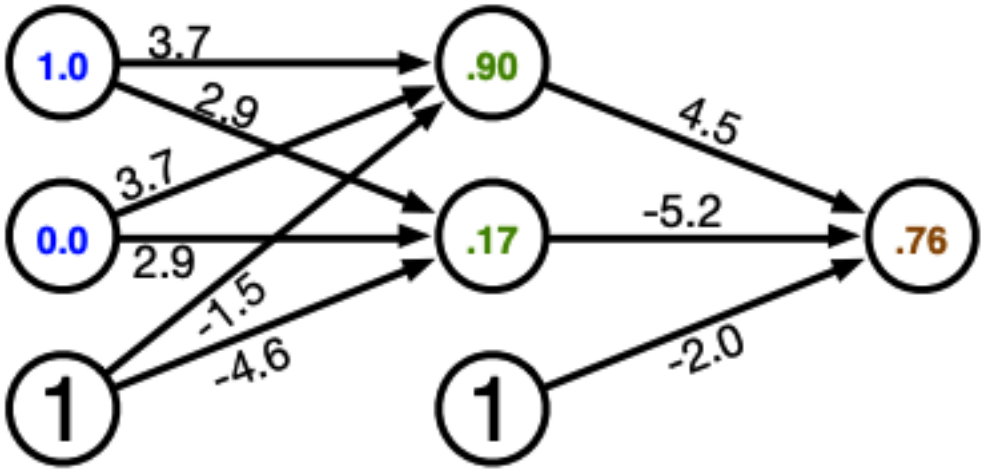
$$\text{sigmoid}(1.0 \times 3.7 + 0.0 \times 3.7 + 1 \times -1.5) = \text{sigmoid}(2.2) = \frac{1}{1+e^{-2.2}} = 0.90$$

$$\text{sigmoid}(1.0 \times 2.9 + 0.0 \times 2.9 + 1 \times -4.5) = \text{sigmoid}(-1.6) = \frac{1}{1+e^{1.6}} = 0.17$$

- Output unit

$$\text{sigmoid}(.90 \times 4.5 + .17 \times -5.2 + 1 \times -2.0) = \text{sigmoid}(1.17) = \frac{1}{1+e^{-1.17}} = 0.76$$

How can we train the network?



- Suppose correct output *t=1*
- Computed output *y=0.76*
- How can we adjust the weights to learn?
  - Adjust arc annotations *w* just like we did w/ Linear Models
  - Minimize "loss" *L* just like we did w/ Linear Models

Key concepts

- Gradient descent
  - Error is a function of weights
  - Adjust weights towards direction of lower error (gradient)
- Backpropagation
  - First adjust last set of weights
  - Propagate error back to each previous layer
  - Adjust their weights according to *their contribution* to final error
    - This will look complicated but is very mechanical

Sigmoid and derivative

- Sigmoid

$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$$

- Quotient rule

$$\left(\frac{f(x)}{g(x)}\right)' = \frac{g(x)f'(x) - f(x)g'(x)}{g(x)^2}$$

- Derivative of sigmoid

$$\begin{aligned}
\frac{d\,\text{sigmoid}(x)}{dx} &= \frac{d}{dx}\frac{1}{1+e^{-x}} \\
&= \frac{0 \times (1-e^{-x}) - (-e^{-x})}{(1+e^{-x})^2} \\
&= \frac{1}{1+e^{-x}}\left(\frac{e^{-x}}{1+e^{-x}}\right) \\
&= \frac{1}{1+e^{-x}}\left(1 - \frac{1}{1+e^{-x}}\right) \\
&= \text{sigmoid}(x)(1 - \text{sigmoid}(x))
\end{aligned}$$

Backprop: final layer

- Computes linear combination of weights $s = \sum_k w_k h_k$
- Activation function $y = \text{sigmoid}(s)$
- Error (L_2 norm) $E = \frac{1}{2}(t - y)^2$
- Derivative of error wrt one weight w_k

$$\frac{dE}{dw_k} = \frac{dE}{dy}\frac{dy}{ds}\frac{ds}{dw_k}$$

Backprop: final layer

- Computes linear combination of weights $s = \sum_k w_k h_k$
- Activation function $y = \text{sigmoid}(s)$
- Error (L_2 norm) $E = \frac{1}{2}(t - y)^2$
- Derivative of error wrt one weight w_k

$$\frac{dE}{dw_k} = \frac{dE}{dy}\frac{dy}{ds}\frac{ds}{dw_k}$$

- Error *E* is defined wrt *y*

$$\frac{dE}{dy} = \frac{d}{dy}\frac{1}{2}(t - y)^2 = -(t - y)$$

Backprop: final layer

- Computes linear combination of weights $s = \sum_k w_k h_k$
- Activation function $y = \text{sigmoid}(s)$
- Error (L_2 norm) $E = \frac{1}{2}(t - y)^2$
- Derivative of error wrt one weight w_k

$$\frac{dE}{dw_k} = \frac{dE}{dy}\frac{dy}{ds}\frac{ds}{dw_k}$$

- Output y is defined wrt *preactivation* s

$$\frac{dy}{ds} = \frac{d\,\text{sigmoid}(s)}{ds} = \text{sigmoid}(s)(1 - \text{sigmoid}(s)) = y(1 - y)$$

Backprop: final layer

- Computes linear combination of weights $s = \sum_k w_k h_k$
- Activation function $y = \text{sigmoid}(s)$
- Error (L_2 norm) $E = \frac{1}{2}(t - y)^2$
- Derivative of error wrt one weight w_k

$$\frac{dE}{dw_k} = \frac{dE}{dy}\frac{dy}{ds}\frac{ds}{dw_k}$$

- Preactivation s is defined wrt w_k

$$\frac{ds}{dw_k} = \frac{d}{dw_k}\sum_k w_k h_k = h_k$$

Backprop: final layer

- Computes linear combination of weights $s = \sum_k w_k h_k$
- Activation function $y = \text{sigmoid}(s)$
- Error (L_2 norm) $E = \frac{1}{2}(t - y)^2$
- Derivative of error wrt one weight w_k

$$\frac{dE}{dw_k} = \frac{dE}{dy}\frac{dy}{ds}\frac{ds}{dw_k}$$

$$= -(t - y) \quad y(1 - y) \quad h_k$$

  - Blue: error
  - Green: derivative of sigmoid: *y'*
- Weight adjustment scaled by learning rate μ

$$\Delta w_k = \mu \, (t - y) \, y' \, h_k$$

Backprop: final layer

- If have multiple output nodes
- Error is computed over all outputs nodes
- Weights $w_{j \leftarrow k}$ connect to output node j

$$E = \sum_j \frac{1}{2}(t_j - y_j)^2$$

$$\Delta w_{j \leftarrow k} = \mu (t_j - y_j)\, y'_j\, h_k$$

Backprop: hidden layer

- In a hidden layer, we do not have target output value
  - Instead ask: how much does each node contribute to downstream error?
- Definition of error term of each node (y_i': derivative of activation)

$$\delta_j = (t_j - y_j)\, y_j'$$

- Backprop error term from *next layer* (not necessarily the output!)

$$\delta_i = \left(\sum_j w_{j \leftarrow i} \delta_j \right) y_i'$$

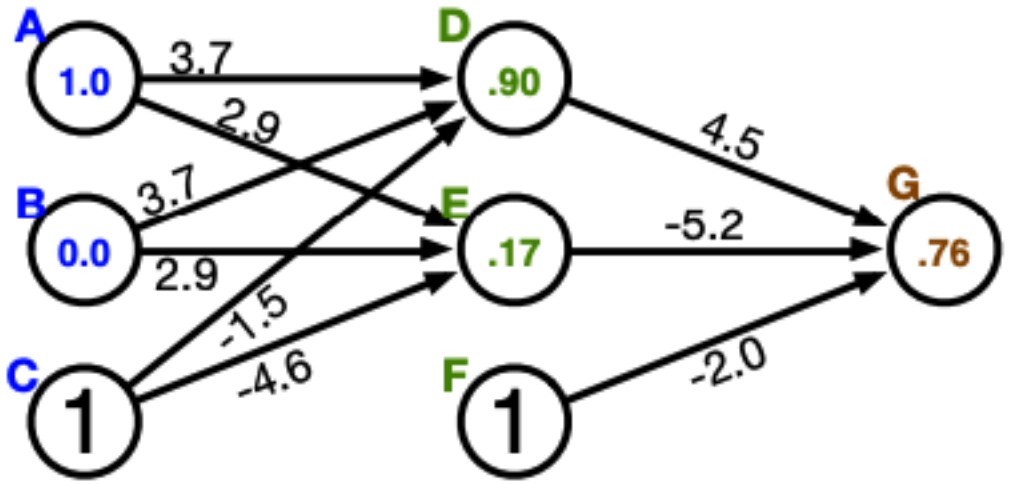
- Universal update formula

$$\Delta w_{j \leftarrow k} = \mu\, \delta_j\, h_k$$

Backprop: our example

$$\delta_j = (t_j - y_j)\, y'_j$$

$$\Delta w_{j \leftarrow k} = \mu\, \delta_j\, h_k$$



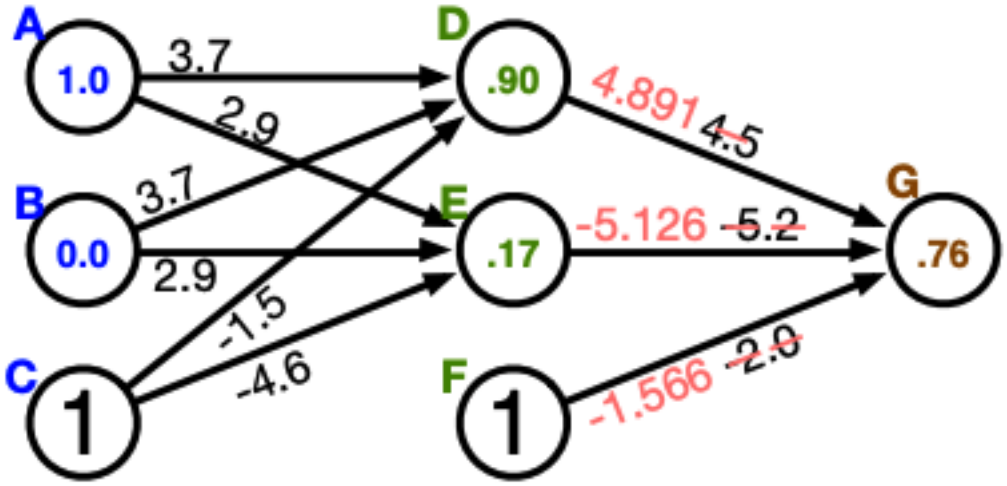
- Suppose correct output *t=1,* Computed output *y=0.76*
- Final layer updates (learning rate = 10 for illustration)

$$\delta_G = (t - y)\, y' = (1 - .76)\, 0.181 = .0434$$
$$\Delta w_{GD} = \mu\, \delta_G\, h_D = 10 \times .0434 \times .90 = .391$$
$$\Delta w_{GE} = \mu\, \delta_G\, h_E = 10 \times .0434 \times .17 = .074$$
$$\Delta w_{GF} = \mu\, \delta_G\, h_F = 10 \times .0434 \times 1 = .434$$

Backprop: our example

$$\delta_j = (t_j - y_j)\, y'_j$$

$$\Delta w_{j \leftarrow k} = \mu\, \delta_j\, h_k$$



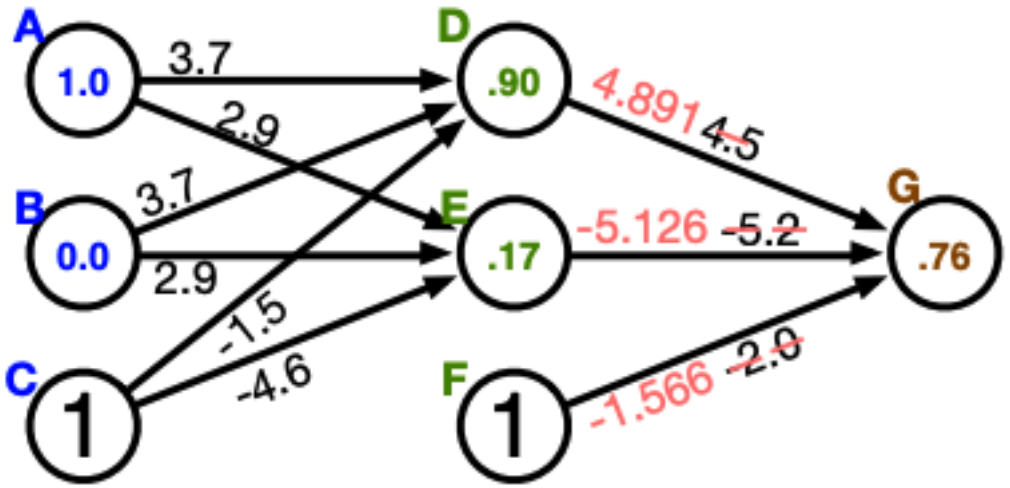
- Suppose correct output *t=1,* Computed output *y=0.76*
- Final layer updates (learning rate = 10 for illustration)

$$\delta_G = (t - y)\, y' = (1 - .76)\, 0.181 = .0434$$
$$\Delta w_{GD} = \mu\, \delta_G\, h_D = 10 \times .0434 \times .90 = .391$$
$$\Delta w_{GE} = \mu\, \delta_G\, h_E = 10 \times .0434 \times .17 = .074$$
$$\Delta w_{GF} = \mu\, \delta_G\, h_F = 10 \times .0434 \times 1 = .434$$

Backprop: our example

$$\delta_i = \left(\sum_j w_{j \leftarrow i} \delta_j \right) y'_i$$

$$\Delta w_{j \leftarrow k} = \mu \, \delta_j \, h_k$$



- Hidden node D

$$\delta_D = \left(\sum_j w_{j \leftarrow i} \delta_j \right) y'_D = w_{GD} \, \delta_G \, y'_D = 4.5 \times .0434 \times .0898 = .0175$$
$$\Delta w_{DA} = \mu \, \delta_D \, h_A = 10 \times .0175 \times 1.0 = .175$$
$$\Delta w_{DB} = \mu \, \delta_D \, h_B = 10 \times .0175 \times 0.0 = 0$$
$$\Delta w_{DC} = \mu \, \delta_D \, h_C = 10 \times .0175 \times 1 = .175$$

- Hidden node E

$$\delta_E = \left(\sum_j w_{j \leftarrow i} \delta_j \right) y'_E = w_{GE} \, \delta_G \, y'_E = -5.2 \times .0434 \times 0.1411 = -.0318$$
$$\Delta w_{EA} = \mu \, \delta_E \, h_A = 10 \times -.0318 \times 1.0 = -.318$$

etc.

Analysis

- Efficiency
  - Fast gradient computation: linear in number of weights
- Convergence
  - Slow convergence (linear rate)
  - May get trapped in local optima
- Prone to overfitting
  - Solutions: early stopping, regularization (add penalty term to objective), dropout

Dropout

- Idea is to randomly “drop” some units from the network when training
- Training: at each iteration of gradient descent
  - Each input unit is dropped with probability p1
  - Each hidden unit is dropped with probability p2
- Prediction (testing):
  - Multiply each input unit by 1-p1
  - Multiply each hidden until by 1-p2