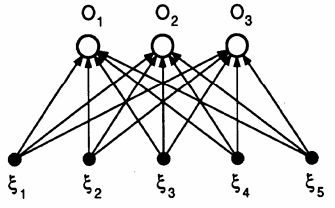
CSE 473

Chapter 20

Machine Learning Algorithms: Neural Networks

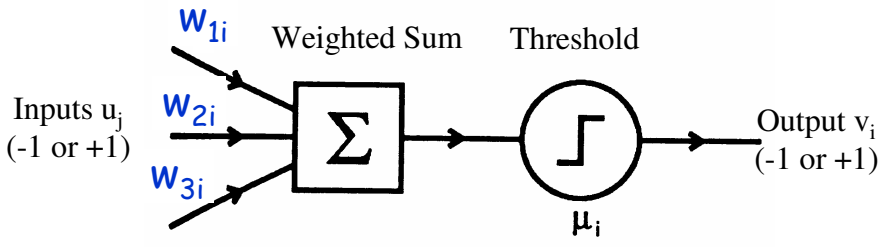


Recap: Neurons as "Threshold Units"

- Artificial neuron:
  - m binary inputs (-1 or 1) and 1 output (-1 or 1)
  - Synaptic weights w_{ji}
  - Threshold μ_i

$$v_i = \Theta(\sum_j w_{ji} u_j - \mu_i)$$

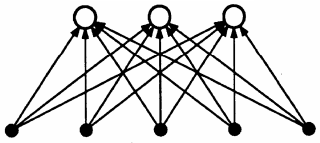
$\Theta(x) = 1$ if $x > 0$ and -1 if $x \leq 0$



"Perceptrons" for Classification

- Fancy name for a single layer "feed-forward" network
- Uses artificial neurons ("units") with binary inputs and outputs

Single-layer



Perceptrons and Classification

- Consider a single-layer perceptron
  Weighted sum forms a *linear hyperplane*

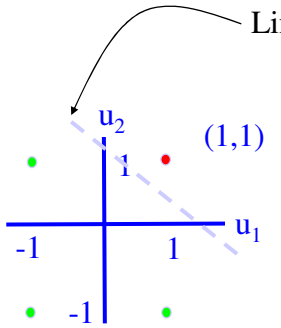
$$\sum_j w_{ji} u_j - \mu_i = 0$$

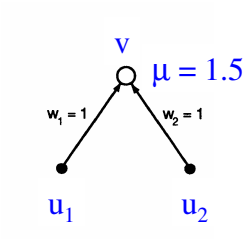
Everything *on one side* of this hyperplane is in class 1 (output = +1) and everything *on other side* is class 2 (output = -1)

Example: AND function

- Example: AND is linearly separable

u_1	u_2	AND
-1	-1	-1
1	-1	-1
-1	1	-1
1	1	1





$v = 1$ iff $u_1 + u_2 - 1.5 > 0$

How do we learn the appropriate weights given only examples of (input,output)?

Idea: Change the weights to decrease the error in ouput

Perceptron Learning Rule

- Given input pair (**u**, v^d) where $v^d \in \{+1,-1\}$ is the desired output, adjust **w** and μ as follows:

1. Calculate current output v of neuron

$$v = \Theta(\sum_j w_j u_j - \mu) = \Theta(\mathbf{w}^T\mathbf{u} - \mu)$$

2. Compute error signal $e = (v^d - v)$

Perceptron Learning Rule

3. Change **w** and μ according to error:
   If input is positive and error is positive,
   then w not large enough $\Rightarrow$ increase **w**
   If input is positive and error is negative,
   then w too large $\Rightarrow$ decrease **w**

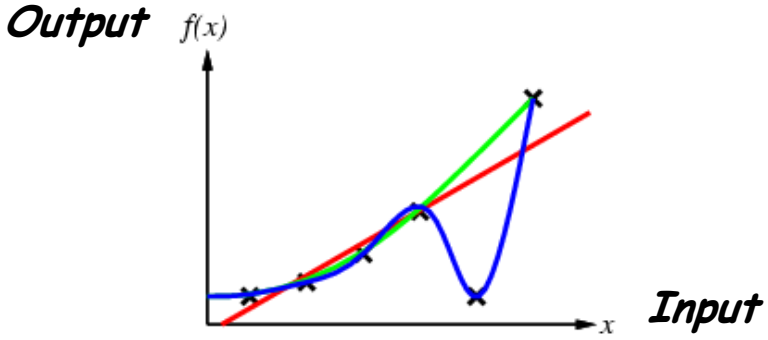
   Similar reasoning for other cases yields:

$$\mathbf{w} \rightarrow \mathbf{w} + \varepsilon(v^d - v)\mathbf{u} \qquad A \rightarrow B \text{ means replace } A \text{ with } B$$

$$\mu \rightarrow \mu - \varepsilon(v^d - v)$$

ε is the "learning rate" (a small positive number, e.g., 0.2)

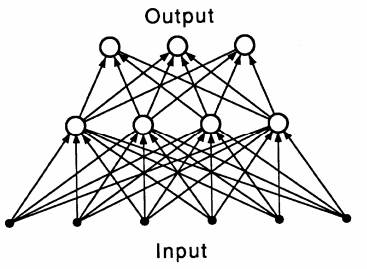
What if we want to learn continuous-valued functions?



Function Approximation

- We want networks that can learn a function
  Network maps real-valued inputs to real-valued output

  Idea: Given data, *minimize errors* between network's output and desired output by changing weights

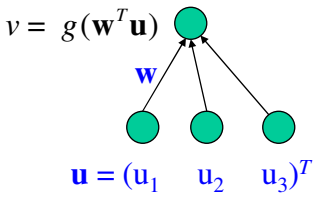


Continuous output values Can't use binary threshold units anymore

To minimize errors, a *differentiable* output function is desirable

Sigmoidal Networks

The most common activation function:

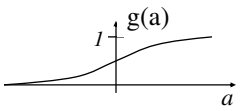


Output

Input nodes

Sigmoid function:

$$g(a) = \frac{1}{1 + e^{-\beta a}}$$



Non-linear "squashing" function: Squashes input to be between 0 and 1. The parameter β controls the slope.

Gradient-Descent Learning ("Hill-Climbing")

- Given training examples ($\mathbf{u}^m, d^m$) (m = 1, …, N), define an error function (cost function or "energy" function)

$$E(\mathbf{w}) = \frac{1}{2}\sum_m (d^m - v^m)^2$$

where $v^m = g(\mathbf{w}^T\mathbf{u}^m)$

Gradient-Descent Learning ("Hill-Climbing")

- Would like to change **w** so that $E(\mathbf{w})$ is minimized

  Gradient Descent: Change **w** in proportion to -dE/d**w** (why?)

$$\mathbf{w} \rightarrow \mathbf{w} - \varepsilon \frac{dE}{d\mathbf{w}}$$

$$\frac{dE}{d\mathbf{w}} = -\sum_m (d^m - v^m) \frac{dv^m}{d\mathbf{w}} = -\sum_m (d^m - v^m) g'(\mathbf{w}^T \mathbf{u}^m) \mathbf{u}^m$$

Derivative of sigmoid

"Stochastic" Gradient Descent

- What if the inputs only arrive one-by-one?
- Stochastic gradient descent approximates sum over all inputs with an "on-line" running sum:

$$\mathbf{w} \rightarrow \mathbf{w} - \varepsilon \frac{dE_1}{d\mathbf{w}}$$

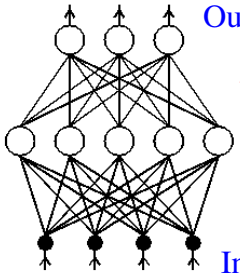
$$\frac{dE_1}{d\mathbf{w}} = -(d^m - v^m) g'(\mathbf{w}^T \mathbf{u}^m) \mathbf{u}^m$$

delta = error

Also known as the "delta rule" or "LMS (least mean square) rule"

But wait….

- Delta rule tells us how to modify the connections from input to output (one layer network)
  One layer networks are not that interesting (remember XOR?)
- What if we have multiple layers?



Output $\mathbf{v} = (v_1\ v_2\ \ldots\ v_J)^T$; Desired = $\mathbf{d}$

Delta rule can be used to adapt these weights

How do we adapt these?

Input $\mathbf{u} = (u_1\ u_2\ \ldots\ u_K)^T$

Next Time

- Learning by Backpropagating
- Reinforcement Learning