CSE/NEUBEH 528

Lecture 14: Supervised Learning

(Chapter 8)

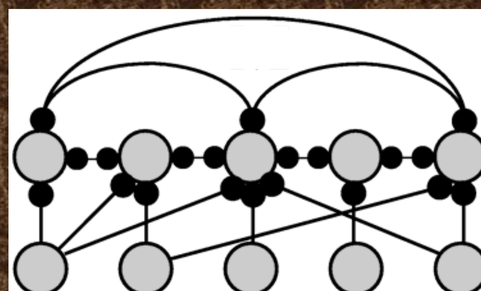
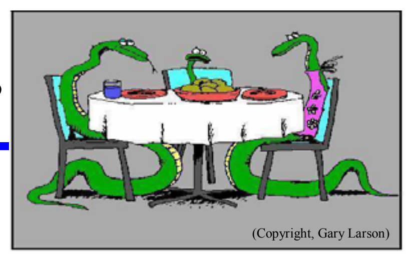


Image from http://clasdean.la.asu.edu/news/images/ubep2001/neuron3.jpg

Lecture figures are from Dayan & Abbott's book http://people.brandeis.edu/~abbott/book/index.html

What's on the menu today?



"Oh, brother! . . . Not hamsters again!"

- Supervised Learning
  - Why supervised learning?
    - Classification
    - Function Approximation
  - Perceptrons & Learning Rule
  - Linear Separability: Minsky-Papert deliver the bad news
  - Multilayer networks to the rescue
  - Function Approximation
  - Backpropagating (errors)
  - Radial Basis Function Networks
  - Recurrent Networks
  - Demos

Why Supervised Learning?

- Two Primary Tasks

1. Classification

- Inputs u_1, u_2, … and discrete classes C_1, C_2, …, C_k
- Training examples: (u_1, C_2), (u_2, C_7), etc.
- Learn the mapping from an arbitrary input to its class
- Example: Inputs = images, output classes = face, not a face

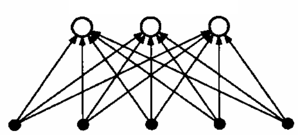
2. Function Approximation (regression)

- Inputs u_1, u_2, … and continuous outputs v_1, v_2, …
- Training examples: (input, desired output) pairs
- Learn to map an arbitrary input to its corresponding output
- Example: Highway driving
  Input = road image, output = steering angle

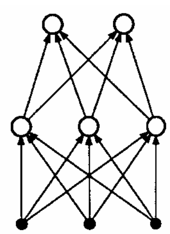
Perceptrons

- Fancy name for a type of layered feedforward networks
- Uses artificial neurons ("units") with binary inputs and outputs

Single-layer



Multilayer



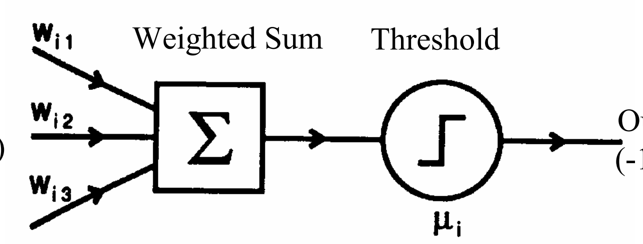
Perceptron uses "Threshold Units"

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

$$v_i = \Theta\left(\sum_j w_{ij} u_j - \mu_i\right)$$

$\Theta(x) = 1$ if $x \geq 0$ and -1 if $x < 0$

Inputs u_j (-1 or +1)

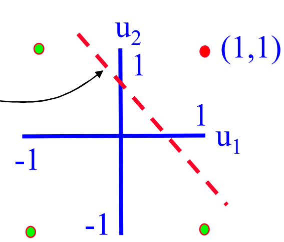


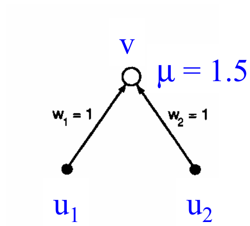
Output v_i (-1 or +1)

Perceptrons and Classification

- Consider a single-layer perceptron $\sum_j w_{ij} u_j - \mu_i = 0$
  - Weighted sum forms a *linear hyperplane*
  - Everything *on one side* of this hyperplane is in class 1 (output = +1) and everything *on other side* is class 2 (output = -1)
  - Any function that is linearly separable can be computed by a perceptron
- Example: AND is linearly separable
  - a AND b = 1 if and only if a = 1 and b = 1

Linear hyperplane





Perceptron Learning Rule

- Given inputs **u** and desired output v^d, adjust **w** and μ as follows:
  1. Compute error signal $e = (v^d - v)$ where v is the current output:
     $$v = \Theta(\sum_j w_j u_j - \mu) = \Theta(\mathbf{w}^T \mathbf{u} - \mu)$$
  2. Change weights and threshold according to e
     - ⇒ For positive inputs, increase weights if error is positive and decrease if error is negative
     - ⇒ For positive inputs, decrease threshold if error is positive, increase if error is negative

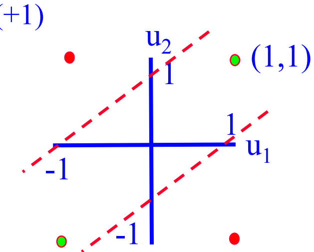
$$\mathbf{w} \to \mathbf{w} + \varepsilon(v^d - v)\mathbf{u}$$

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

$A \to B$ means replace A with B

Linear Inseparability

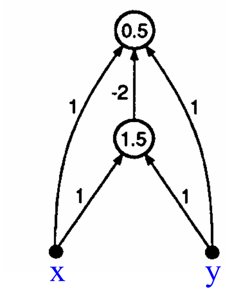
- Single-layer perceptron with threshold units fails if classification task is not linearly separable
  - Example: XOR
  - a XOR b = 1 iff (a = -1, b = 1) or (a = 1, b = -1)
  - No single line can separate the "yes" (+1) outputs from the "no" (-1) outputs!



- Minsky and Papert's book showing such negative results was very influential – put a damper on neural networks research for over a decade!

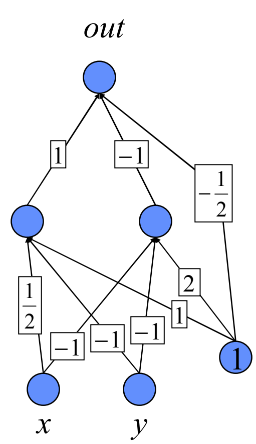
Solution in 1980s: Multilayer perceptrons

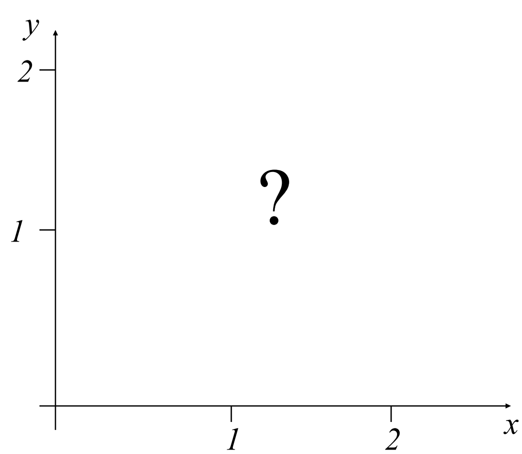
- Removes limitations of single-layer networks
  - Can solve XOR
- An example of a two-layer perceptron that computes XOR



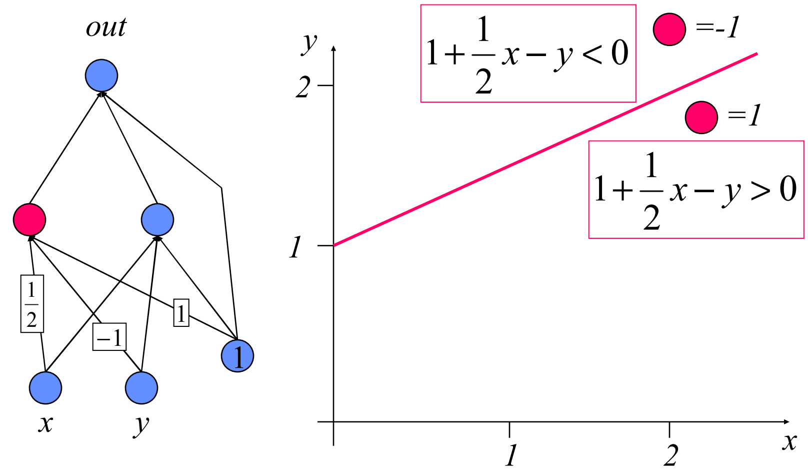
- Output is 1 if and only if $x + y - 2(x + y - 1.5 > 0) - 0.5 > 0$

Multilayer Perceptron: What does it do?

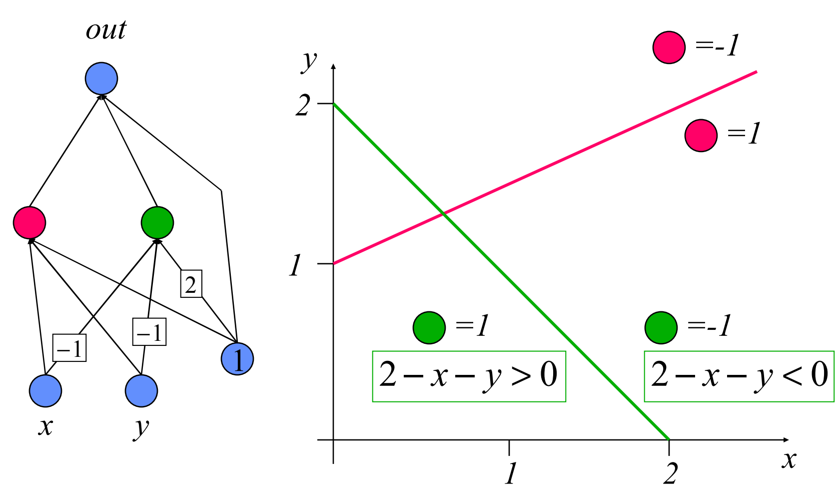




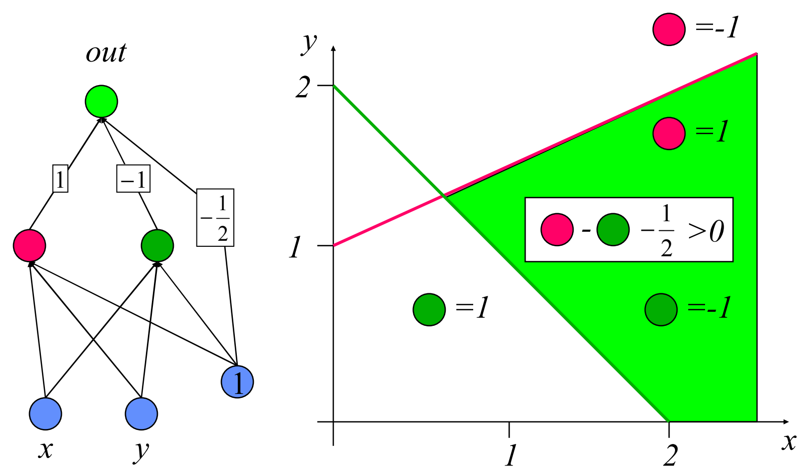
Example: Perceptrons as Constraint Satisfaction Networks



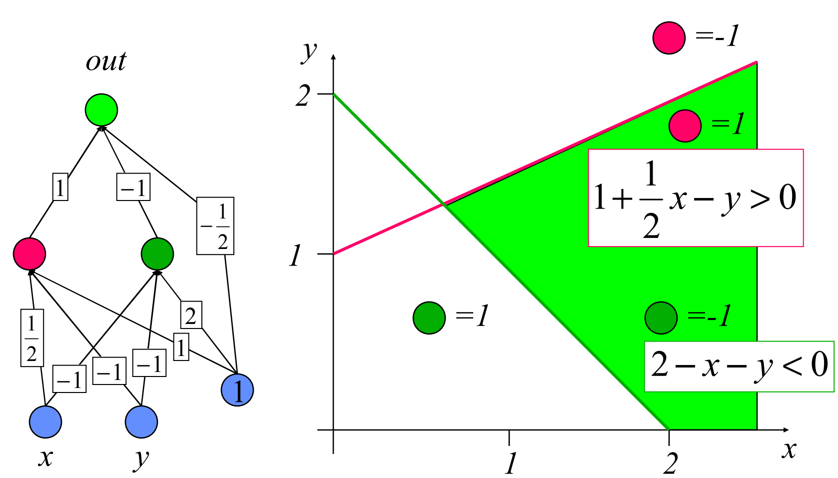
Example: Perceptrons as Constraint Satisfaction Networks



Example: Perceptrons as Constraint Satisfaction Networks

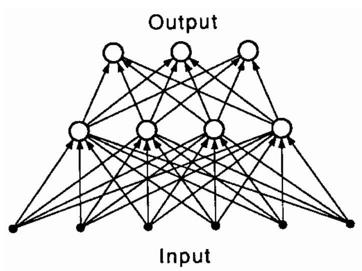


Perceptrons as Constraint Satisfaction Networks



Function Approximation

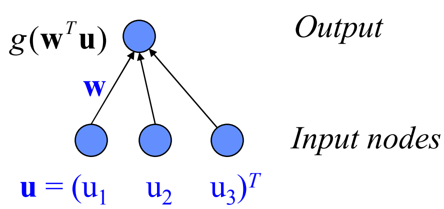
- We want networks that can learn a function
  - Network maps real-valued inputs to real-valued output
  - Want to generalize to predict outputs for new inputs
  - Idea: Given input data, *minimize errors* between network's output and desired output by changing weights



Continuous output values → Can't use binary threshold units anymore

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

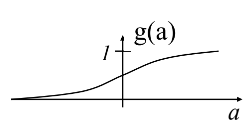
Sigmoidal Networks



The most common activation function:

Sigmoid function:

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



(non-linear "squashing" function)

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)

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

- Would like to change $\mathbf{w}$ so that $E(\mathbf{w})$ is minimized
  - Gradient Descent: Change $\mathbf{w}$ in proportion to $-dE/d\mathbf{w}$

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

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

"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:

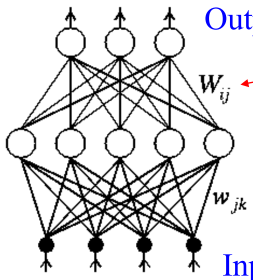
$$\mathbf{w} \to \mathbf{w} - \varepsilon \frac{dE_1}{d\mathbf{w}}$$

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

Also known as the "delta rule" or "LMS 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?

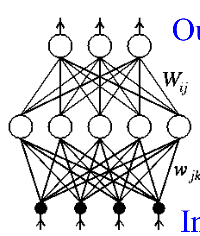
Input $\mathbf{u} = (u_1\ u_2 \ldots u_K)^T$

Let's Backpropagate (Errors)

- Backpropagation = gradient-descent learning for multilayer feedforward networks
- Idea: Propagate credit/blame for errors back to internal nodes
  - Use delta rule to change weights for output layer
  - Use chain rule (from calculus) to change weights for internal "hidden" nodes

error = $(\mathbf{d} - \mathbf{v})$

Backpropagate this to correct all weights



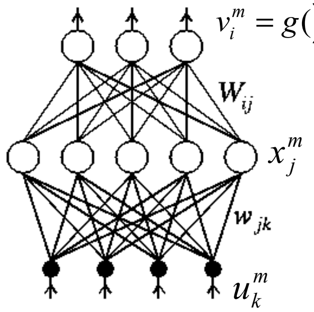
Output $\mathbf{v} = (v_1\ v_2 \ldots v_J)^T$

Delta rule

Backprop rule

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

Notation for Backprop



$$v_i^m = g(\sum_j W_{ij} g(\underbrace{\sum_k w_{jk} u_k^m}_{x_j^m}))$$

Find **W** and **w** that minimize total squared output error:

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

$$= \frac{1}{2}\sum_{m,i} (d_i^m - v_i^m)^2$$

Backpropagation (for Math lovers' eyes only!)

- Learning rule for hidden-output connection weights:

$$W_{ij} \to W_{ij} - \varepsilon \frac{\partial E}{\partial W_{ij}}$$

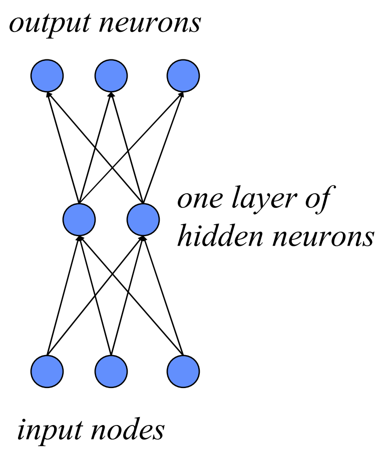
$$\frac{dE}{dW_{ij}} = -\sum_m (d_i^m - v_i^m) g'(\sum_j W_{ij} x_j^m) x_j^m$$

- Learning rule for input-hidden connection weights:

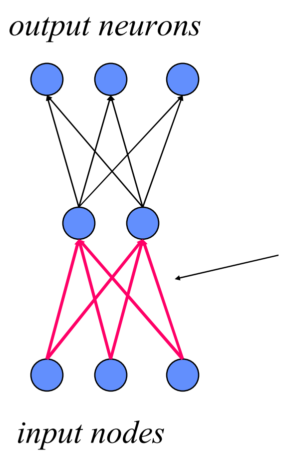
$$w_{jk} \to w_{jk} - \varepsilon \frac{\partial E}{\partial w_{jk}} \quad \text{But}: \frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial x_j^m} \cdot \frac{\partial x_j^m}{\partial w_{jk}} \qquad \{\textit{chain rule}\}$$

$$\frac{dE}{dw_{jk}} = -\sum_{m,i} (d_i^m - v_i^m) g'(\sum_j W_{ij} x_j^m) W_{ij} \cdot g'(\sum_k w_{jk} u_k^m) u_k^m$$

Alternate Method: Radial Basis Function Networks



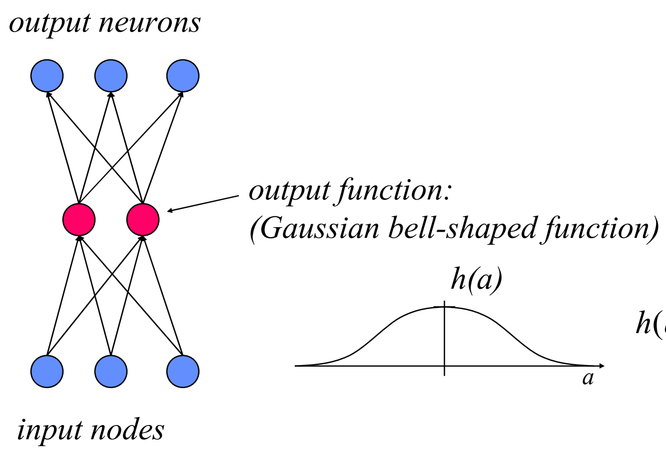
Radial Basis Function Networks



"activation" function:

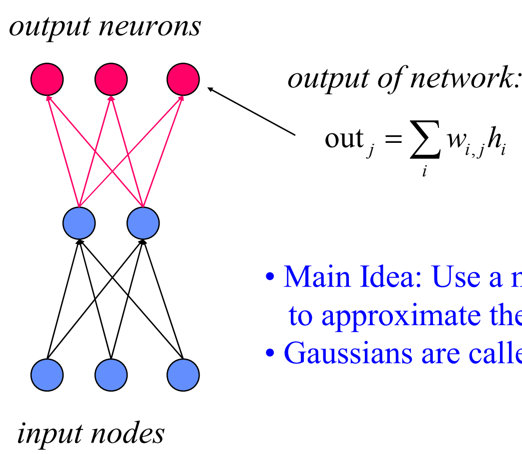
$$a_j = \sqrt{\sum_{i=1}^{n} (x_i - \mu_{i,j})^2}$$

Radial Basis Function Networks



$$h(a) = e^{-\frac{a^2}{2\sigma^2}}$$

Radial Basis Function Networks

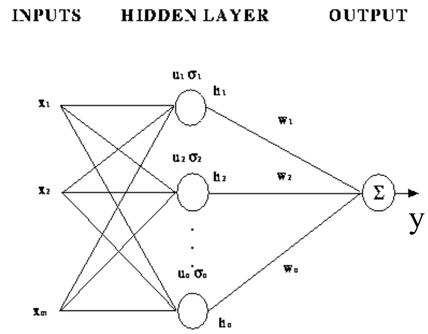


$$\text{out}_j = \sum_i w_{i,j} h_i$$

- Main Idea: Use a mixture of Gaussians to approximate the output
- Gaussians are called "basis functions"

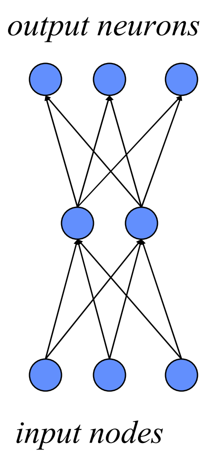
RBF networks

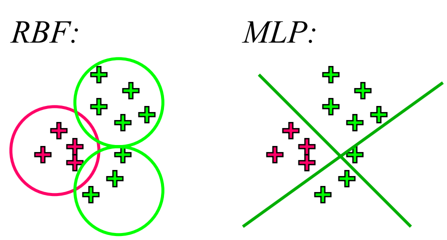
- Radial basis functions
  - Hidden units store means and variances
  - Hidden units compute a Gaussian function of inputs $x_1,\ldots x_n$ that constitute the input vector $\mathbf{x}$
- Learn weights w_i, means μ_i, and variances σ_i by minimizing squared error function (gradient descent learning)



$$h_i = exp[-\frac{(\mathbf{x}-\mathbf{u}_i)^T(\mathbf{x}-\mathbf{u}_i)}{2\sigma^2}],\ y = \sum_i h_i w_i$$

RBF Networks and Multilayer Perceptrons





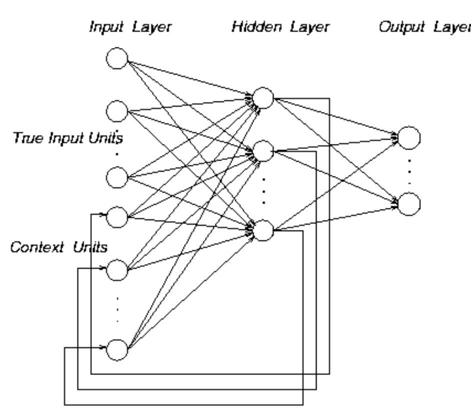
Recurrent Supervised Networks

- Why use recurrent networks?
  - To keep track of recent history and context
  - Can learn temporal patterns (time series or oscillations)
- Examples
  - Hopfield network (see previous lecture and textbook)
  - Recurrent backpropagation networks: for small sequences, *unfold network in time dimension* and use backpropagation learning
  - Partially recurrent networks E.g. Elman net

Partially Recurrent Networks

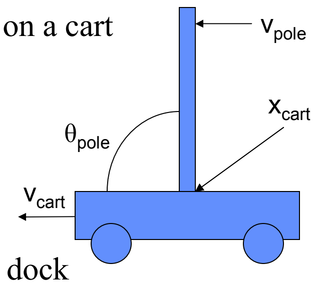
- Example
  - Elman net
    - Partially recurrent
    - Context units keep *internal memory of past inputs*
    - Fixed context weights
    - Backpropagation for learning
    - E.g. Can disambiguate A→B→C and C→B→A

Elman network

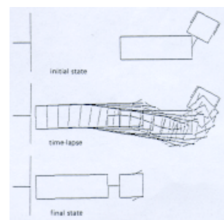


Demos (by Keith Grochow, CSE 599, 2001)

- Neural network learns to balance a pole on a cart
  - System:
    - 4 state variables: x_{cart}, v_{cart}, θ_{pole}, v_{pole}
    - 1 input: Force on cart
  - Backprop Network:
    - Input: State variables
    - Output: New force on cart



- NN learns to back a truck into a loading dock
  - System (Nyugen and Widrow, 1989):
    - State variables: x_{cab}, y_{cab}, θ_{cab}
    - 1 input: new $\theta_{steering}$
  - Backprop Network:
    - Input: State variables
    - Output: Steering angle $\theta_{steering}$



Next Class: Reinforcement Learning

- Things to do:
  - Read Chapter 9
  - Finish Last Homework (due this Friday, 5pm)
  - Work on mini-project