
Chapter 4 (Part 2): Artificial Neural Networks

CS 536: Machine Learning
Littman (Wu, TA)

Administration

iCML-03: instructional Conference on
Machine Learning

<http://www.cs.rutgers.edu/~mlittman/courses/ml03/iCML03/>

Weka exercise

<http://www.cs.rutgers.edu/~mlittman/courses/ml03/hw1.pdf>

Grading Components

- HWs (not handed in): 0%
- Project paper (iCML): 25%
 - document (15%), review (5%), revision(5%)
- Project presentation: 20%
 - oral (10%), slides (10%)
- Midterm (take home): 20%
- Final: 35%

Artificial Neural Networks

[Read Ch. 4]

[Review exercises 4.1, 4.2, 4.5, 4.9, 4.11]

- Threshold units [ok]
- Gradient descent [today]
- Multilayer networks [today]
- Backpropagation [today?]
- Hidden layer representations
- Example: Face Recognition
- Advanced topics

Gradient Descent

To understand, consider simpler *linear unit*, where

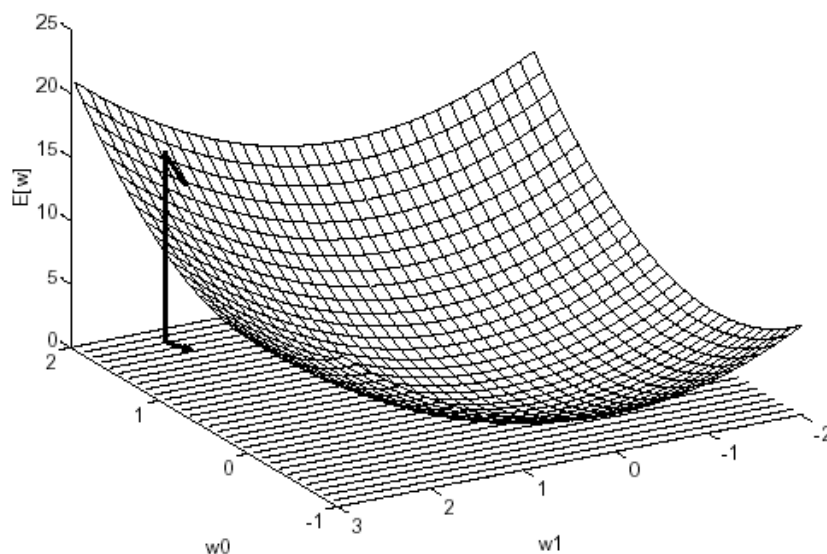
$$o = w_0 + w_1x_1 + \dots + w_nx_n$$

Let's learn w_i 's to minimize squared error

$$E[\mathbf{w}] \equiv 1/2 \sum_{d \in D} (t_d - o_d)^2$$

Where D is set of training examples

Error Surface



Gradient Descent

Gradient

$$\nabla E[\mathbf{w}] = [\partial E/\partial w_0, \partial E/\partial w_1, \dots, \partial E/\partial w_n]$$

Training rule:

$$\Delta \mathbf{w} = -\eta \nabla E[\mathbf{w}]$$

in other words:

$$\Delta w_i = -\eta \partial E/\partial w_i$$

Gradient of Error

$$\partial E/\partial w_i$$

$$= \partial/\partial w_i \frac{1}{2} \sum_d (t_d - o_d)^2$$

$$= \frac{1}{2} \sum_d \partial/\partial w_i (t_d - o_d)^2$$

$$= \frac{1}{2} \sum_d 2(t_d - o_d) \partial/\partial w_i (t_d - o_d)$$

$$= \sum_d (t_d - o_d) \partial/\partial w_i (t_d - \mathbf{w} \cdot \mathbf{x}_d)$$

$$= \sum_d (t_d - o_d) (-x_{i,d})$$

Gradient Descent Code

GRADIENT-DESCENT(training examples, η)

Each training example is a pair of the form $\langle x, t \rangle$, where x is the vector of input values, and t is the target output value. η is the learning rate (e.g., .05).

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
 - Initialize each Δw_i to zero.
 - For each $\langle x, t \rangle$ in training examples, Do
 - Input the instance x to the unit and compute the output o
 - For each linear unit weight w_i , Do
$$\Delta w_i \leftarrow \Delta w_i + \eta (t - o)x_i$$
 - For each linear unit weight w_i , Do
$$w_i \leftarrow w_i + \Delta w_i$$

Summary

Perceptron training rule will succeed if

- Training examples are linearly separable
- Sufficiently small learning rate η

Linear unit training uses gradient descent

- Guaranteed to converge to hypothesis with minimum squared error
- Given sufficiently small learning rate η
- Even when training data contains noise
- Even when training data not H separable

Stochastic Gradient Descent

Batch mode Gradient Descent:

Do until satisfied

1. Compute the gradient $\nabla E_D[\mathbf{w}]$
2. $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla E_D[\mathbf{w}]$

Incremental mode Gradient Descent:

Do until satisfied

- For each training example d in D
 1. Compute the gradient $\nabla E_d[\mathbf{w}]$
 2. $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla E_d[\mathbf{w}]$

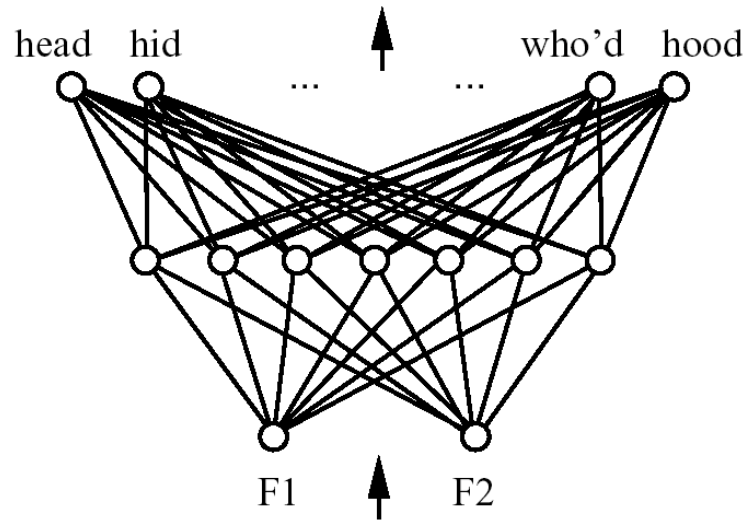
More Stochastic Grad. Desc.

$$E_D[\mathbf{w}] \equiv 1/2 \sum_{d \in D} (t_d - o_d)^2$$

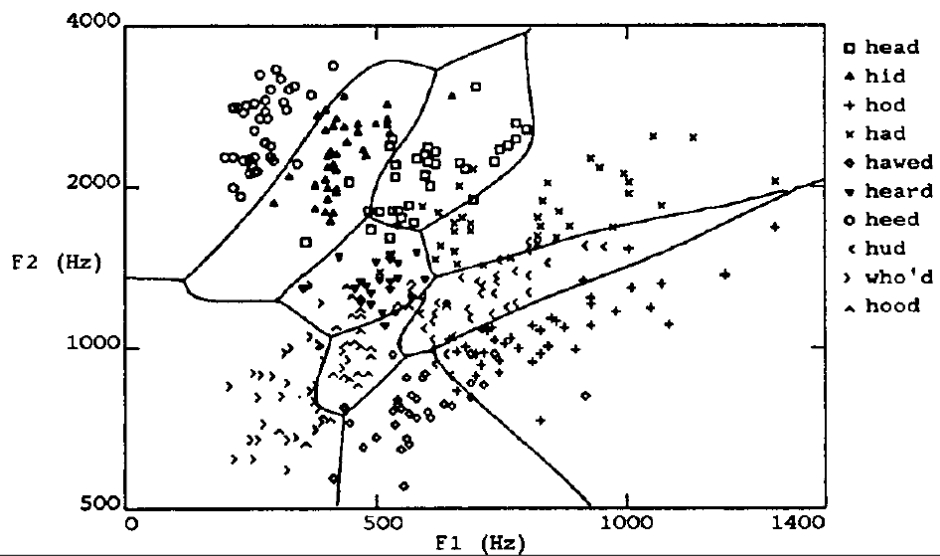
$$E_d[\mathbf{w}] \equiv 1/2 (t_d - o_d)^2$$

Incremental Gradient Descent can approximate *Batch Gradient Descent* arbitrarily closely if η set small enough

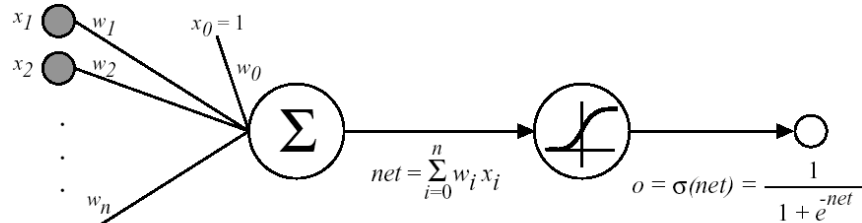
Multilayer Networks



Decision Boundaries



Sigmoid Unit



$\sigma(x)$ is the sigmoid (s-like) function
 $1/(1 + e^{-x})$

Derivatives of Sigmoids

Nice property:

$$d\sigma(x)/dx = \sigma(x)(1-\sigma(x))$$

We can derive gradient decent rules to train

- One sigmoid unit
- *Multilayer networks* of sigmoid units \square *Backpropagation*

Error Gradient for Sigmoid

$$\begin{aligned}\partial E / \partial w_i &= \partial / \partial w_i \frac{1}{2} \sum_d (t_d - o_d)^2 \\ &= \frac{1}{2} \sum_d \partial / \partial w_i (t_d - o_d)^2 \\ &= \frac{1}{2} \sum_d 2 (t_d - o_d) \partial / \partial w_i (t_d - o_d) \\ &= \sum_d (t_d - o_d) (-\partial o_d / \partial w_i) \\ &= - \sum_d (t_d - o_d) (\partial o_d / \partial net_d \partial net_d / \partial w_i)\end{aligned}$$

Even more...

But we know:

$$\begin{aligned}\partial o_d / \partial net_d &= \partial \sigma(net_d) / \partial net_d = o_d (1 - o_d) \\ \partial net_d / \partial w_i &= \partial (\mathbf{w} \cdot \mathbf{x}_d) / \partial w_i = x_{i,d}\end{aligned}$$

So:

$$\partial E / \partial w_i = - \sum_d (t_d - o_d) o_d (1 - o_d) x_{i,d}$$

Backpropagation Algorithm

Initialize all weights to small random numbers.

Until satisfied, Do

- For each training example, Do
 1. Input the training example to the network and compute the network outputs
 2. For each output unit k
$$\delta_k = o_k(1 - o_k)(t_k - o_k)$$
 3. For each hidden unit h
$$\delta_h = o_h(1 - o_h) \sum_{k \text{ in outputs}} w_{h,k} \delta_k$$
 4. Update each network weight $w_{i,j}$
$$w_{i,j} \leftarrow w_{i,j} + \delta w_{i,j} \text{ where } \delta w_{i,j} = \delta \delta_j x_{i,j}$$

More on Backpropagation

- Gradient descent over entire *network* weight vector
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum
 - In practice, often works well (can run multiple times)

More more

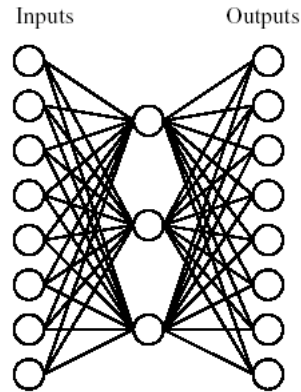
- Often include weight *momentum* η
$$\eta w_{i,j}(n) = \eta \sum_j x_{i,j} + \eta \eta w_{i,j}(n-1)$$
- Minimizes error over training examples
 - Will it generalize well to subsequent examples?
- Training can take thousands of iterations
 η slow!
- Using network after training is very fast

Hidden Layer Reps

Simple target function:

- | Input | Output |
|------------|-----------------|
| • 10000000 | η 10000000 |
| • 01000000 | η 01000000 |
| • 00100000 | η 00100000 |
| • 00010000 | η 00010000 |
| • 00001000 | η 00001000 |
| • 00000100 | η 00000100 |
| • 00000010 | η 00000010 |
| • 00000001 | η 00000001 |

Autoencoder

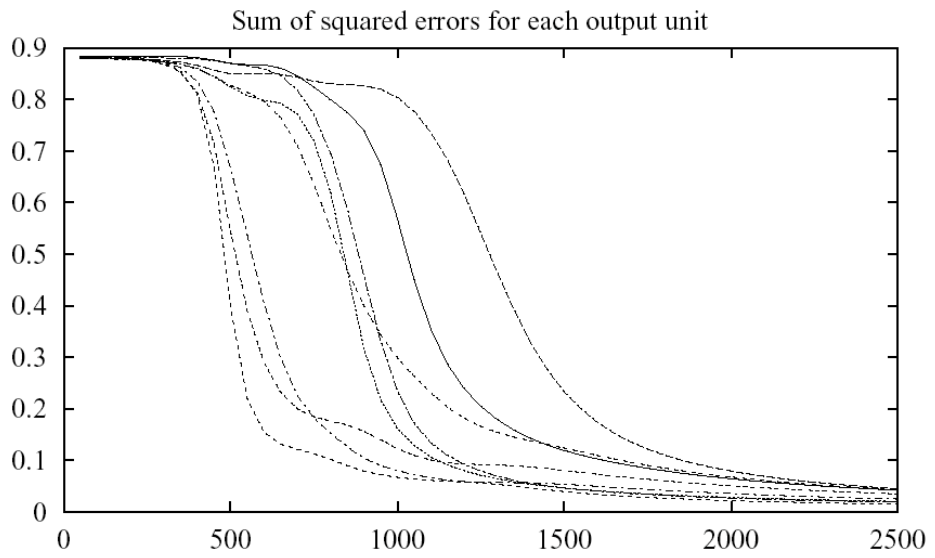


Can the mapping be learned with this network??

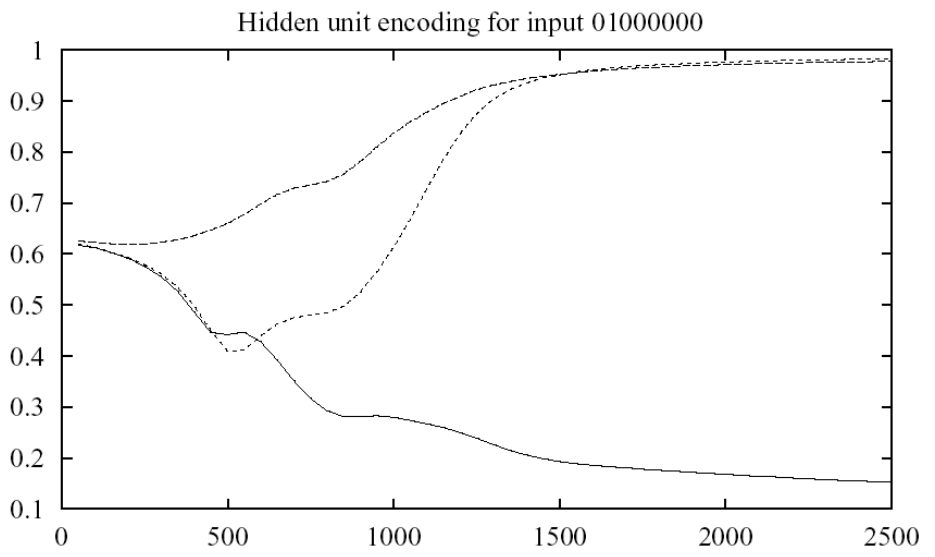
Hidden Layer Rep.

Input	Hidden Values	Output
• 10000000	□ .89 .04 .08 □	10000000
• 01000000	□ .01 .11 .88 □	01000000
• 00100000	□ .01 .97 .27 □	00100000
• 00010000	□ .99 .97 .71 □	00010000
• 00001000	□ .03 .05 .02 □	00001000
• 00000100	□ .22 .99 .99 □	00000100
• 00000010	□ .80 .01 .98 □	00000010
• 00000001	□ .60 .94 .01 □	00000001

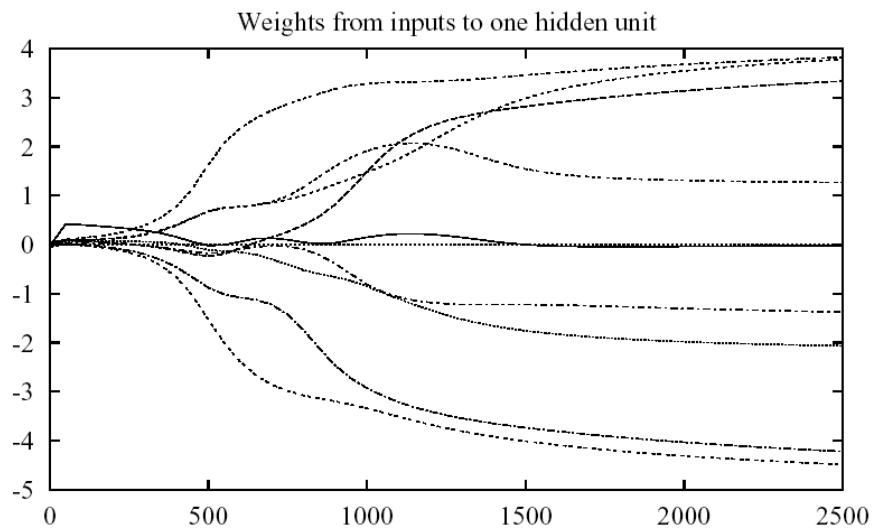
Training



Training



Training



Convergence of Backprop

Gradient descent to some local minimum

- Perhaps not global minimum...
- Add momentum
- Stochastic gradient descent
- Train multiple nets with different initial weights

More on Convergence

Nature of convergence

- Initialize weights near zero
- Therefore, initial networks are near-linear
- Increasingly non-linear functions possible as training progresses

Expressiveness of ANNs

Boolean functions:

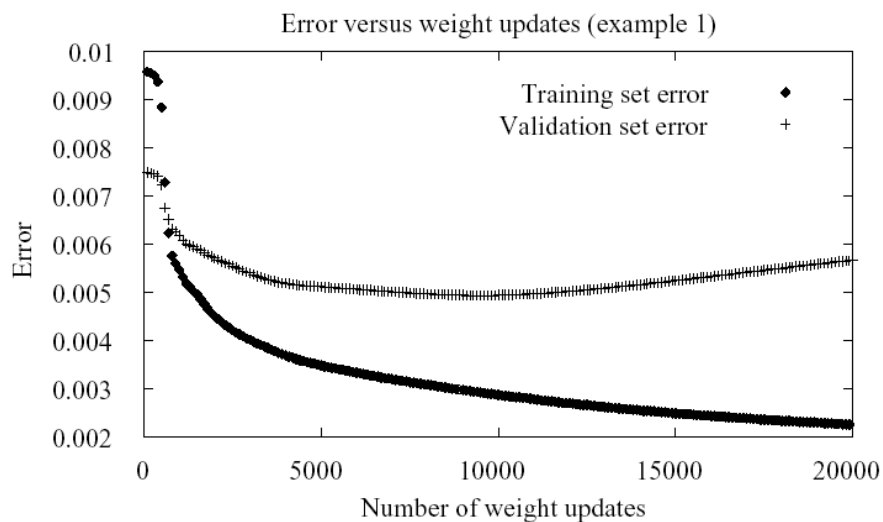
- Every Boolean function can be represented by network with single hidden layer
- but might require exponential (in number of inputs) hidden units

Real-valued Functions

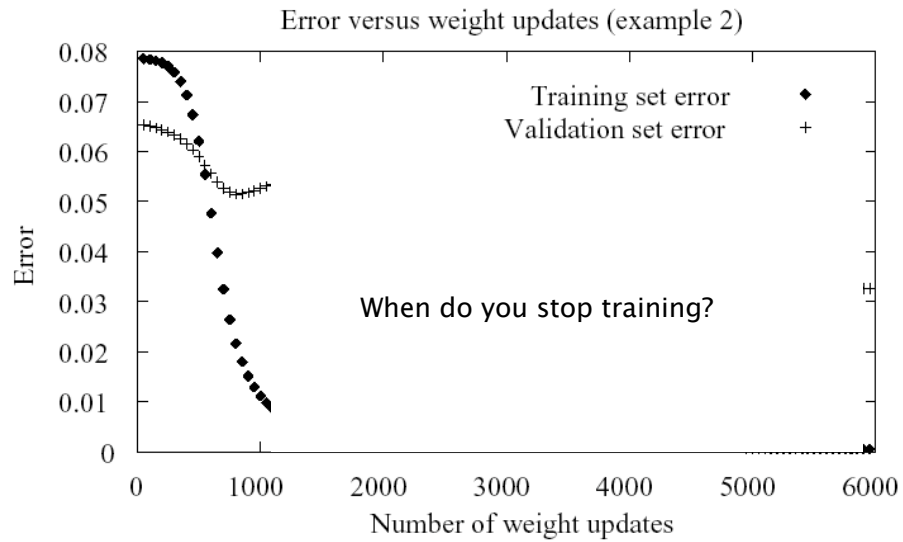
Continuous functions:

- Every bounded continuous function can be approximated, with arbitrarily small error, by network with one hidden layer [Cybenko 1989; Hornik et al. 1989]
- *Any* function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988].

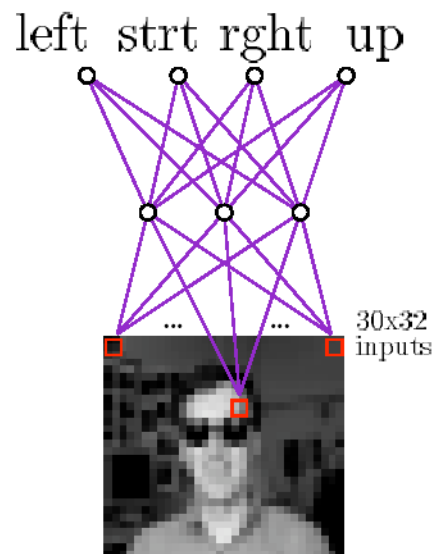
Overfitting in ANNs (Ex. 1)



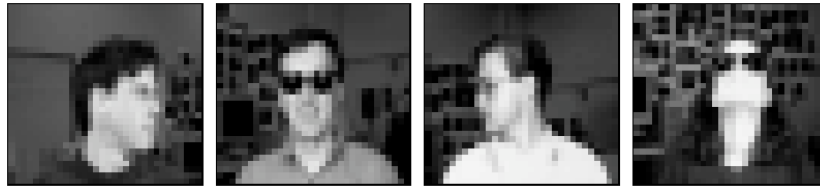
Overfitting in ANNs (Ex. 2)



Face Recognition

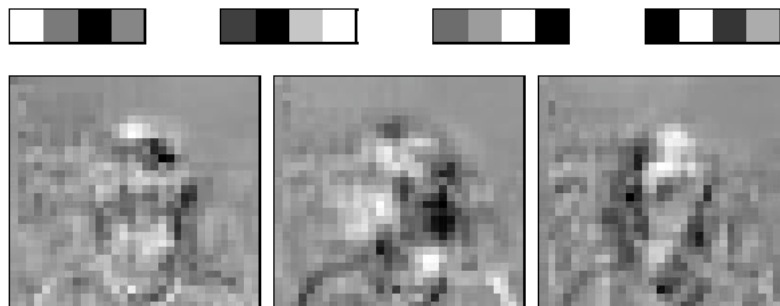


Typical input images



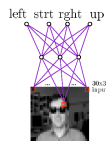
- 90% accurate learning head pose, and recognizing 1-of-20 faces

Learned Weights



<http://www.cs.cmu.edu/tom/faces.html>

Bias first?



Alternative Error Functions

Penalize large weights:

$$E(\mathbf{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \text{ in outputs}} (t_{kd} - o_{kd})^2 + \lambda \sum_{ij} w_{ji}^2$$

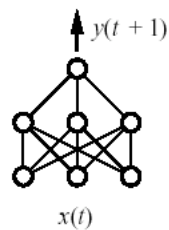
Train on target slopes as well as values:

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in \text{outputs}} \left[(t_{kd} - o_{kd})^2 + \mu \sum_{j \in \text{inputs}} \left(\frac{\partial t_{kd}}{\partial x_d^j} - \frac{\partial o_{kd}}{\partial x_d^j} \right)^2 \right]$$

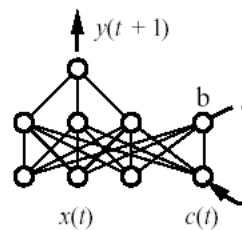
Tie together weights:

- e.g., in phoneme recognition network

Recurrent Networks



(a) Feedforward network



(b) Recurrent network

Unfolding: BPTT

