CSCE 496/896 Lecture 5: Artificial Neural Networks

Stephen D. Scott
(Adapted from Tom Mitchell's slides)

September 24, 1999

Outline

- Threshold units: Perceptron, Winnow
- Gradient descent/exponentiated gradient
- Multilayer networks
- Backpropagation
- VC dimension of neural networks
- Example: Face Recognition
- Advanced topics

Connectionist Models

Consider humans:

- Total number of neurons $\approx 10^{10}$
- Neuron switching time $\approx 10^{-3}$ second (vs. 10^{-10})
- Connections per neuron $\approx 10^4$–10^5
- Scene recognition time ≈ 0.1 second
- 100 inference steps doesn't seem like enough

$\Rightarrow$ much parallel computation

Properties of artificial neural nets (ANNs):

- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process
- Emphasis on tuning weights automatically

Strong differences between ANNs for ML and ANNs for biological modeling

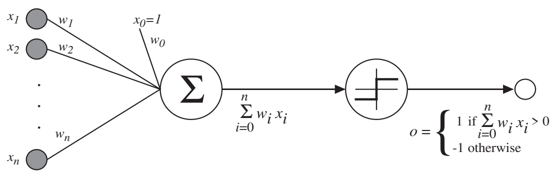
When to Consider Neural Networks

- Input is high-dimensional discrete or real-valued (e.g. raw sensor input)
- Output is discrete or real valued
- Output is a vector of values
- Possibly noisy data
- Form of target function is unknown
- Human readability of result is unimportant
- Long training times acceptable

Examples:

- Speech phoneme recognition [Waibel]
- Image classification [Kanade, Baluja, Rowley]
- Financial prediction

The Perceptron & Winnow



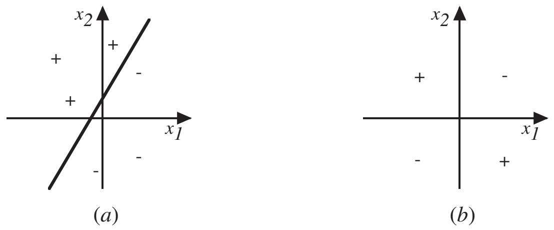
$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise} \end{cases}$$

(sometimes use 0 instead of -1)

Sometimes we'll use simpler vector notation:

$$o(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} > 0 \\ -1 & \text{otherwise} \end{cases}$$

Decision Surface of Perceptron/Winnow



Represents some useful functions

- What weights represent $g(x_1, x_2) = AND(x_1, x_2)$?

But some functions not representable

- I.e. those not linearly separable
- Therefore, we'll want networks of neurons

Perceptron Training Rule

$$w_i \leftarrow w_i + \Delta w_i^{add}, \text{ where } \Delta w_i^{add} = \eta(t - o)x_i$$

and

- $t = c(\vec{x})$ is target value
- o is perceptron output
- η is small constant (e.g. 0.1) called learning rate

I.e. if $(t - o) > 0$ then increase w_i w.r.t. x_i, else decrease

Can prove rule will converge if training data is linearly separable and η sufficiently small

Winnow Training Rule

$$w_i \leftarrow w_i \cdot \Delta w_i^{mult}, \text{ where } \Delta w_i^{mult} = \alpha^{(t-o)x_i}$$

and $\alpha > 1$

I.e. use multiplicative updates vs. additive updates

Problem: Sometimes negative weights are required

- Maintain two weight vectors $\vec{w}^+$ and $\vec{w}^-$ and replace $\vec{w} \cdot \vec{x}$ with $\left(\vec{w}^+ - \vec{w}^-\right) \cdot \vec{x}$
- Update $\vec{w}^+$ and $\vec{w}^-$ independently as above, using $\Delta w_i^+ = \alpha^{(t-o)x_i}$ and $\Delta w_i^- = 1/\Delta w_i^+$

Can also guarantee convergence

Perceptron vs. Winnow

Winnow works well when most attributes are irrelevant, i.e. when the optimal weight vector $\vec{w}^*$ is sparse (many 0 entries)

E.g. let examples $\vec{x} \in \{0,1\}^n$ be labeled by a k-disjunction over n attributes, $k \ll n$

- Remaining $n-k$ are irrelevant
- E.g. $x_5 \vee x_9 \vee \neg x_{12}$, $n = 150$, $k = 3$
- For disjunctions, number of prediction mistakes (in on-line model) is $O(k \log n)$ for Winnow and (in worst case) $\Omega(kn)$ for Perceptron
- So in worst case, need exponentially fewer updates for learning with Winnow than Perceptron

Bound is only for disjunctions, but improvement for learning with irrelevant attributes is often true

When $\vec{w}^*$ not sparse, sometimes advantage goes to Perceptron

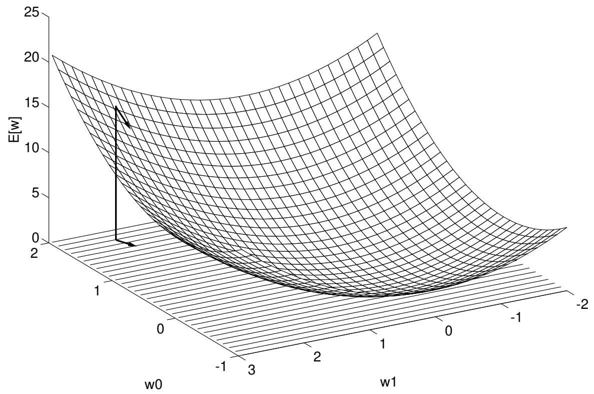
Gradient Descent and Exponentiated Gradient

- Useful when linear separability impossible but still want to minimize training error
- Consider simpler *linear unit*, where
$$o = w_0 + w_1 x_1 + \cdots + w_n x_n$$
(i.e. no threshold)
- For moment, assume that we update weights after seeing each example $\vec{x}_d$
- For each example, want to compromise between correctiveness and conservativeness
  - Correctiveness: Tendency to improve on $\vec{x}_d$
  - Conservativeness: Tendency to keep $\vec{w}_{d+1}$ close to $\vec{w}_d$
- Use cost function that measures both:

$$U(\vec{w}) = conserv\left(\vec{w}_{d+1}, \vec{w}_d\right) + \eta \cdot correc\left(t_d, \overbrace{\vec{w}_{d+1} \cdot \vec{x}_d}^{\text{curr ex, new wts}}\right)$$

Gradient Descent and Exponentiated Gradient

(cont'd)



$$\frac{\partial U}{\partial \vec{w}} = \left[\frac{\partial U}{\partial w_0}, \frac{\partial U}{\partial w_1}, \cdots, \frac{\partial U}{\partial w_n}\right]$$

Gradient Descent

$$U(\vec{w}) = \overbrace{\|\vec{w}_{d+1} - \vec{w}_d\|_2^2}^{conserv.} + \overbrace{\eta}^{coef.} \overbrace{(t_d - \vec{w}_{d+1} \cdot \vec{x}_d)^2}^{corrective}$$
$$= \sum_{i=1}^{n} \left(w_{i,d+1} - w_{i,d}\right)^2 + \eta \left(t_d - \sum_{i=1}^{n} w_{i,d+1}\, x_{i,d}\right)^2$$

Take gradient w.r.t. $\vec{w}_{d+1}$ and set to $\vec{0}$:

$$0 = 2\left(w_{i,d+1} - w_{i,d}\right) - 2\eta \left(t_d - \sum_{i=1}^{n} w_{i,d+1}\, x_{i,d}\right) x_{i,d}$$

Approximate with

$$0 = 2\left(w_{i,d+1} - w_{i,d}\right) - 2\eta \left(t_d - \sum_{i=1}^{n} w_{i,d}\, x_{i,d}\right) x_{i,d},$$

which yields

$$w_{i,d+1} = w_{i,d} + \overbrace{\eta\,(t_d - o_d)\, x_{i,d}}^{\Delta w_{i,d}^{add}}$$

Exponentiated Gradient

Conserv. portion uses unnormalized relative entropy:

$$U(\vec{w}) = \overbrace{\sum_{i=1}^{n} \left(w_{i,d} - w_{i,d+1} + w_{i,d+1} \ln \frac{w_{i,d+1}}{w_{i,d}} \right)}^{conserv.} + \overbrace{\eta}^{coef.} \overbrace{(t_d - \vec{w}_{d+1} \cdot \vec{x}_d)^2}^{corrective}$$

Take gradient w.r.t. $\vec{w}_{d+1}$ and set to $\vec{0}$:

$$0 = \ln \frac{w_{i,d+1}}{w_{i,d}} - 2\eta \left(t_d - \sum_{i=1}^{n} w_{i,d+1}\, x_{i,d} \right) x_{i,d}$$

Approximate with

$$0 = \ln \frac{w_{i,d+1}}{w_{i,d}} - 2\eta \left(t_d - \sum_{i=1}^{n} w_{i,d}\, x_{i,d} \right) x_{i,d},$$

which yields (for $\eta = \ln \alpha / 2$)

$$w_{i,d+1} = w_{i,d} \exp\left(2\eta \left(t_d - o_d\right) x_{i,d} \right) = w_{i,d}\, \overbrace{\alpha^{(t_d - o_d) x_{i,d}}}^{\Delta w_{i,d}^{mult}}$$

Implementation Approaches

- Can use rules on previous slides on an example-by-example basis, sometimes called incremental, stochastic, or on-line GD/EG
  - Has a tendency to "jump around" more in searching, which helps avoid getting trapped in local minima
- Alternatively, can use standard or batch GD/EG, in which the classifier is evaluated over all training examples, summing the error, and then updates are made
  - I.e. sum up Δw_i for all examples, but don't update w_i until summation complete (p. 93, Table 4.1)
  - This is an inherenet averaging process and tends to give better estimate of the gradient

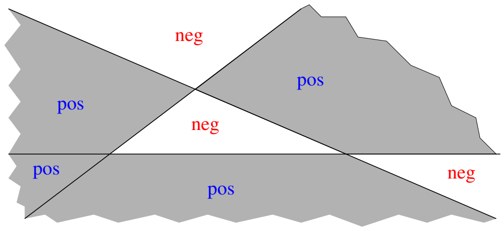
Remarks

- Perceptron and Winnow update weights based on thresholded output, while GD and EG use unthresholded outputs
- P/W converge in finite number of steps to perfect hyp if data linearly separable; GD/EG work on non-linearly separable data, but only converge asymptotically (to wts with minimum squared error)
- As with P vs. W, EG tends to work better than GD when many attributes are irrelevant
  - Allows the addition of attributes that are nonlinear combinations of original ones, to work around linear sep. problem (perhaps get linear separability in new, higher-dimensional space)
  - E.g. if two attributes are x_1 and x_2, use as EG inputs

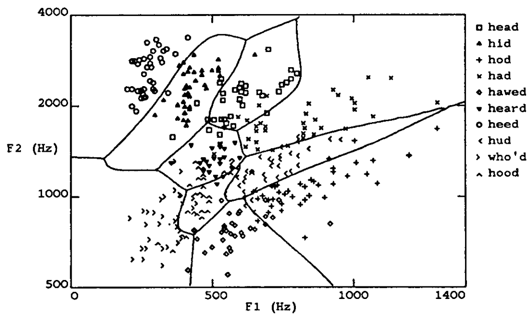
$$\vec{x} = \left[x_1, x_2, x_1 x_2, x_1^2, x_2^2 \right]$$

Handling Nonlinearly Separable Data

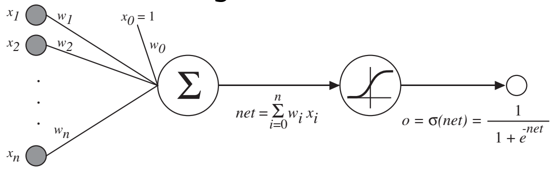
- By adding up to 2 hidden layers of perceptrons, can represent any union of intersection of halfspaces



- Problem: Still defined linearly



Sigmoid Unit



$\sigma(x)$ is the logistic function

$$\frac{1}{1+e^{-x}}$$

(a type of sigmoid function)

Squashes net into $[0,1]$ range

Nice property:

$$\frac{d\sigma(x)}{dx} = \sigma(x)(1-\sigma(x))$$

We can derive GD/EG rules to train

- One sigmoid unit
- *Multilayer networks* of sigmoid units $\Rightarrow$ Backpropagation

GD/EG for Sigmoid Unit

- First note that conservativeness and correctiveness are only additively related $\Rightarrow$ derivatives always independent
- Thus in general get

$$w_{i,d+1} = w_{i,d} - \frac{\eta}{2}\frac{\partial\, correc}{\partial w_{i,d}} \quad \text{for GD}$$

$$w_{i,d+1} = w_{i,d} \exp\left(-\eta \frac{\partial\, correc}{\partial w_{i,d}}\right) \quad \text{for EG}$$

- So all we have to do is define an error function, take its gradient, and substitute into the equations

GD/EG for Sigmoid Unit
(cont'd)

Return to book notation, where correctiveness is:

$$E(\vec{w}_d) = \frac{1}{2}(t_d - o_d)^2$$

(folding 1/2 of correctiveness into error func)

Thus $\frac{\partial E}{\partial w_{i,d}} = \frac{\partial}{\partial w_{i,d}} \frac{1}{2}(t_d - o_d)^2$

$$= \frac{1}{2}\, 2\,(t_d - o_d)\, \frac{\partial}{\partial w_{i,d}}(t_d - o_d) = (t_d - o_d)\left(-\frac{\partial o_d}{\partial w_{i,d}}\right)$$

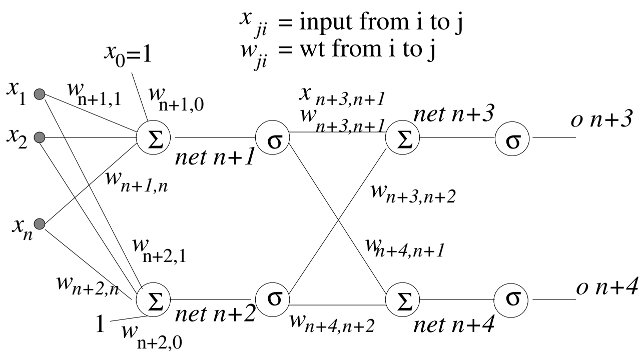
Since o_d is a function of net_d,

$$\begin{aligned}\frac{\partial E}{\partial w_{i,d}} &= -\,(t_d - o_d)\,\frac{\partial o_d}{\partial net_d}\,\frac{\partial net_d}{\partial w_{i,d}} \\ &= -\,(t_d - o_d)\,\frac{\partial \sigma\,(net_d)}{\partial net_d}\,\frac{\partial net_d}{\partial w_{i,d}} \\ &= -\,(t_d - o_d)\, o_d\,(1 - o_d)\, x_{i,d}\end{aligned}$$

$$\boxed{w_{i,d+1} = w_{i,d} + \eta\, o_d\,(1-o_d)\,(t_d - o_d)\, x_{i,d} \quad \text{for GD}}$$

$$\boxed{w_{i,d+1} = w_{i,d} \exp\left(2\eta\, o_d\,(1-o_d)\,(t_d - o_d)\, x_{i,d}\right) \quad \text{for EG}}$$

Multilayer Networks



Use sigmoid units since continuous and differentiable

Error:

$$E_d = E(\vec{w}_d) = \frac{1}{2}\sum_{k \in outputs}\left(t_{k,d} - o_{k,d}\right)^2$$

Training
Output Units

- Adjust wt $w_{ji,d}$ according to E_d as before

- For output units, this is easy since contribution of $w_{ji,d}$ to E_d when j is an output unit is the same as for single neuron case*, i.e.

$$\frac{\partial E_d}{\partial w_{ji,d}} = -\left(t_{j,d} - o_{j,d}\right) o_{j,d} \left(1 - o_{j,d}\right) x_{ji,d} = -\delta_j x_{ji,d}$$

where $\delta_j = \frac{\partial E_d}{\partial net_j}$ = error term of unit j

*This is because all other outputs are constants w.r.t. $w_{ji,d}$

Training
Hidden Units

- How can we compute the error term for hidden layers when there is no target output $\vec{t}$ for these layers?

- Instead propagate back error values from output layer toward input layers, scaling with the weights

- Scaling with the weights characterizes how much of the error term each hidden unit is "responsible for"

Training
Hidden Units (cont'd)

The impact that $w_{ji,d}$ has on E_d is only through net_j and units immediately "downstream" of j:

$$\frac{\partial E_d}{\partial w_{ji,d}} = \frac{\partial E_d}{\partial net_j}\frac{\partial net_j}{\partial w_{ji,d}} = x_{ji} \sum_{k \in down(j)} \frac{\partial E_d}{\partial net_k}\frac{\partial net_k}{\partial net_j}$$

$$= x_{ji} \sum_{k \in down(j)} -\delta_k \frac{\partial net_k}{\partial net_j} = x_{ji} \sum_{k \in down(j)} -\delta_k \frac{\partial net_k}{\partial o_j}\frac{\partial o_j}{\partial net_j}$$

$$= x_{ji} \sum_{k \in down(j)} -\delta_k w_{kj} \frac{\partial o_j}{\partial net_j} = x_{ji} \sum_{k \in down(j)} -\delta_k w_{kj} o_j \left(1 - o_j\right)$$

Works even when node j not a hidden layer, so this works for arbitrary number of hidden layers

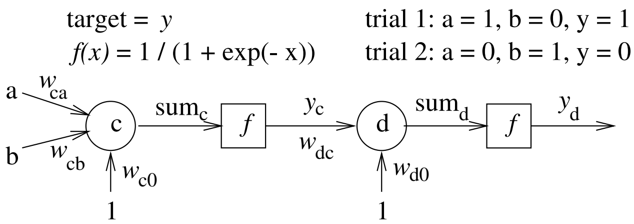
Backpropagation Algorithm

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
     $$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$
  3. For each hidden unit h
     $$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in down(h)} w_{h,k}\delta_k$$
  4. Update each network weight $w_{i,j}$
     $$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$
     where
     $$\Delta w_{i,j} = \eta \delta_j x_{i,j}$$

The Backpropagation Algorithm
Example

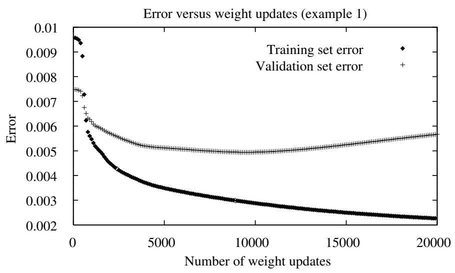


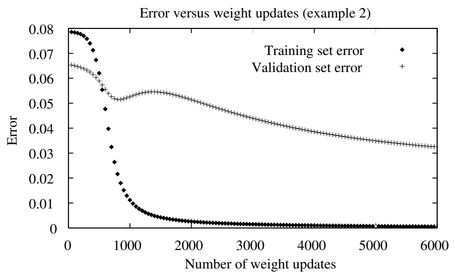
eta	0.3		
	trial 1	trial 2	
w_ca	0.1	0.1008513	0.1008513
w_cb	0.1	0.1	0.0987985
w_c0	0.1	0.1008513	0.0996498
a	1	0	
b	0	1	
const	1	1	
sum_c	0.2	0.2008513	
y_c	0.5498340	0.5500447	
w_dc	0.1	0.1189104	0.0964548
w_d0	0.1	0.1343929	0.0935679
sum_d	0.1549834	0.1997990	
y_d	0.5386685	0.5497842	
target	1	0	
delta_d	0.1146431	-0.136083	
delta_c	0.0028376	-0.004005	
delta_d(t) = y_d(t) * (y(t) - y_d(t)) * (1 - y_d(t))			
delta_c(t) = y_c(t) * (1 - y_c(t)) * delta_d(t) * w_dc(t)			
w_dc(t+1) = w_dc(t) + eta * y_c(t) * delta_d(t)			
w_ca(t+1) = w_ca(t) + eta * a * delta_c(t)			

Remarks on Backprop

- When to stop training? When weights don't change much, error rate sufficiently low, etc. (be aware of overfitting: use validation set)
- Cannot ensure convergence to global minimum due to myriad local minima, but tends to work well in practice (can re-run with new random weights)
- Generally training very slow (thousands of iterations), use is very fast
- Setting η: Small values slow convergence, large values might overshoot minimum, can adapt it over time
- Can add momentum term α that tends to keep the updates moving in the same direction as previous trials:
$$\Delta w_{ji,d+1} = \eta\, \delta_{j,d+1}\, x_{ji,d+1} + \alpha\, \Delta w_{ji,d}$$
Can help move through small local minima to better ones & move along flat surfaces

Overfitting





Danger of stopping too soon!

Remarks on Backprop
(cont'd)

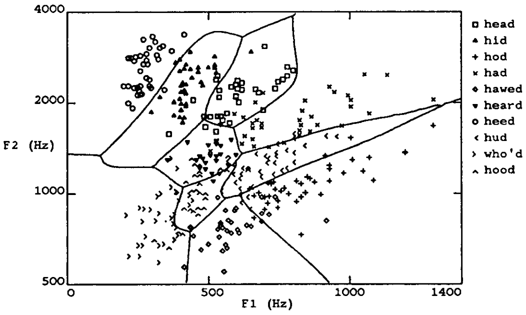
- Alternative error function: cross entropy
$$E_d = \sum_{k \in outputs} \left(t_{k,d} \ln o_{k,d} + \left(1 - t_{k,d}\right) \ln \left(1 - o_{k,d}\right)\right)$$
"blows up" if $t_{k,d} \approx 1$ and $o_{k,d} \approx 0$ or vice-versa (vs. square error, which is always in $[0, 1]$)
- Can penalize large weights to make space more linear and reduce risk of overfitting:
$$E_d = \frac{1}{2} \sum_{k \in outputs} (t_{kd} - o_{ok})^2 + \gamma \sum_{i,j} w_{ji,d}^2$$
- Representational power: Any boolean func. can be represented with 2 layers, any bounded, continuous func. can be rep. with arbitrarily small error with 2 layers, any func. can be rep. with arb. small error with 3 layers
  - Number of required units may be large
  - GD/EG may not be able to find the right weights

Hypothesis Space

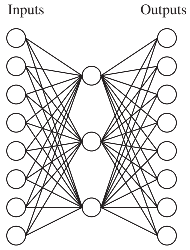
1. Hyp. space is set of all weight vectors (continuous vs. discrete of decision trees)

2. Search via GD/EG: Possible because error function and output functions are continuous & differentiable

3. Inductive bias: (Roughly) smooth interpolation between data points



Learning Hidden Layer Representations



Learned hidden layer representation:

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

Thus the hidden layer has the ability to learn new representations of inputs (i.e. new features) to minimize error

VC Dimension of NNs

- Theorem: Let C be a concept class with VCD d and G be a directed acyclic graph with $s \geq 2$ nodes from C (each with $\leq r$ inputs), and C_G be the set of functions representable by G. Then $VC(C_G) \leq 2ds \log(es)$

- The VCD of a single perceptron (halfspace) in r dimensions is $r+1$

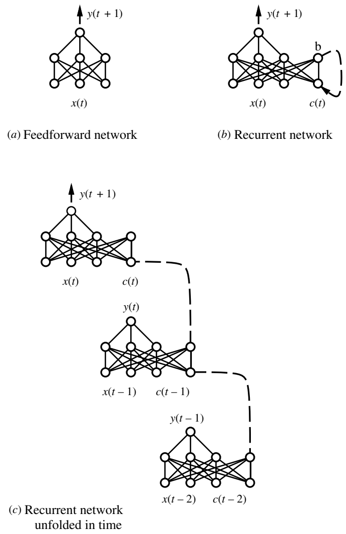
- Thus $VC\left(C_G^{percep}\right) \leq 2(r+1)s \log(es)$

- Can now compute sample complexity

- Problems:
  - Backprop works with sigmoid units, not perceptrons
  - Finding a consistent hypothesis is NP-Complete!

Advanced Topics

Recurrent Networks to handle time series data (i.e. label of current ex. depends on past exs.)



(a) Feedforward network (b) Recurrent network

(c) Recurrent network unfolded in time

Other optimization procedures

Dynamically modifying network structure