Neural Networks

Aarti Singh

Machine Learning 10-601
Nov 3, 2011

Slides Courtesy: Tom Mitchell

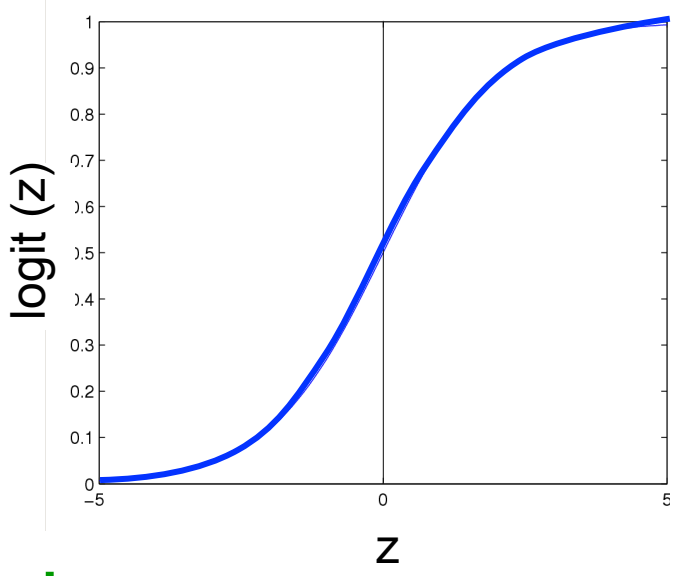
Logistic Regression

Assumes the following functional form for P(Y|X):

$$P(Y=1|X) = \frac{1}{1+\exp(-(w_0 + \sum_i w_i X_i))}$$

Logistic function applied to a linear function of the data

Logistic function (or Sigmoid): $\frac{1}{1+exp(-z)}$



Features can be discrete or continuous!

Logistic Regression is a Linear Classifier!

Assumes the following functional form for P(Y|X):

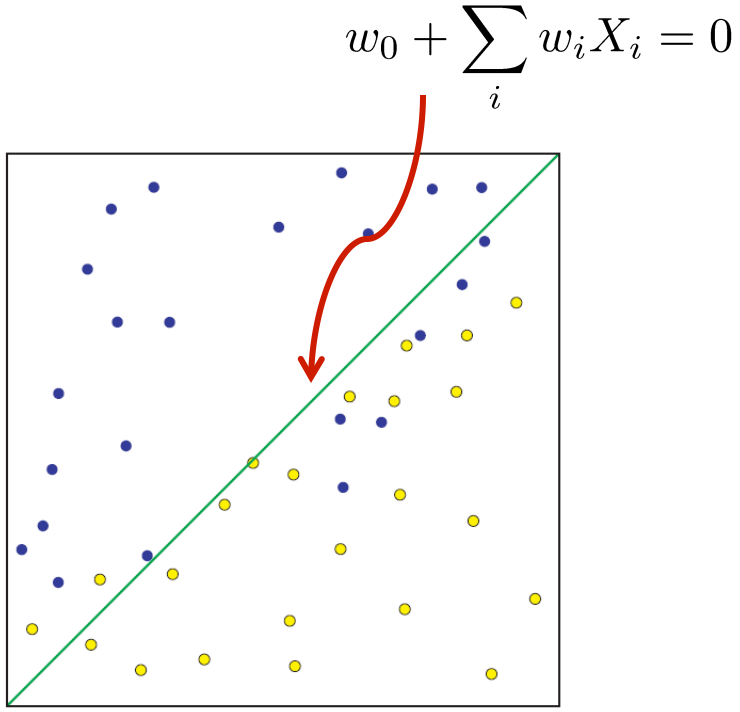
$$P(Y=1|X) = \frac{1}{1+\exp(-(w_0+\sum_i w_i X_i))}$$

Decision boundary:

$$P(Y=0|X) \underset{1}{\overset{0}{\gtrless}} P(Y=1|X)$$

$$0 \underset{1}{\overset{0}{\gtrless}} w_0 + \sum_i w_i X_i$$

(Linear Decision Boundary)



Training Logistic Regression

How to learn the parameters w_0, w_1, … w_d?

Training Data $\{(X^{(j)}, Y^{(j)})\}_{j=1}^{n}$ $X^{(j)} = (X_1^{(j)}, \ldots, X_d^{(j)})$

Maximum (Conditional) Likelihood Estimates

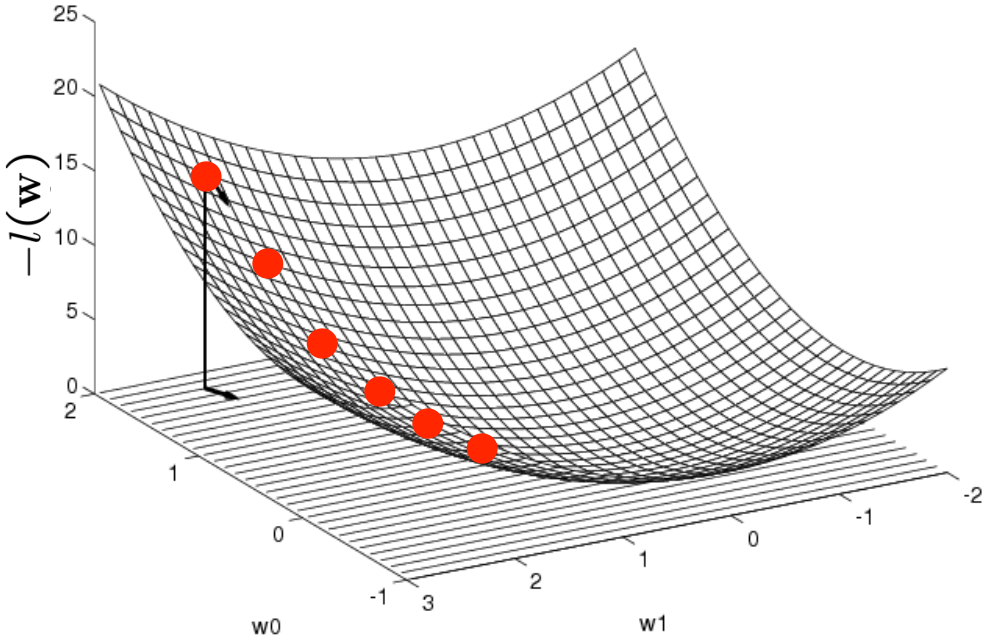
$$\widehat{\mathbf{w}}_{MCLE} = \arg\max_{\mathbf{w}} \prod_{j=1}^{n} P(Y^{(j)} \mid X^{(j)}, \mathbf{w})$$

Discriminative philosophy – Don't waste effort learning P(X), focus on P(Y|X) – that's all that matters for classification!

Optimizing convex function

- Max Conditional log-likelihood = Min Negative Conditional log-likelihood
- Negative Conditional log-likelihood is a convex function

Gradient Descent (convex)



Gradient:

$$\nabla_{\mathbf{w}} l(\mathbf{w}) = [\frac{\partial l(\mathbf{w})}{\partial w_0}, \ldots, \frac{\partial l(\mathbf{w})}{\partial w_d}]'$$

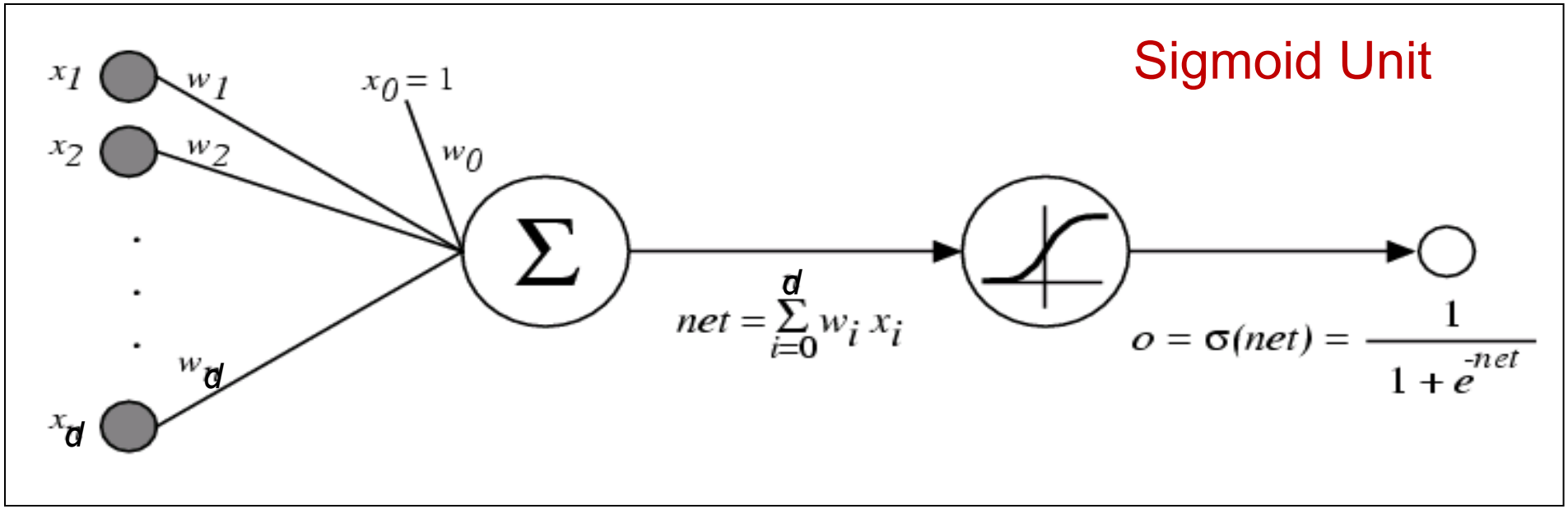
Update rule: **Learning rate, η>0**

$$\Delta \mathbf{w} = \eta \nabla_{\mathbf{w}} l(\mathbf{w})$$

$$w_i^{(t+1)} \leftarrow w_i^{(t)} + \eta \left. \frac{\partial l(\mathbf{w})}{\partial w_i} \right|_t$$

Logistic function as a Graph

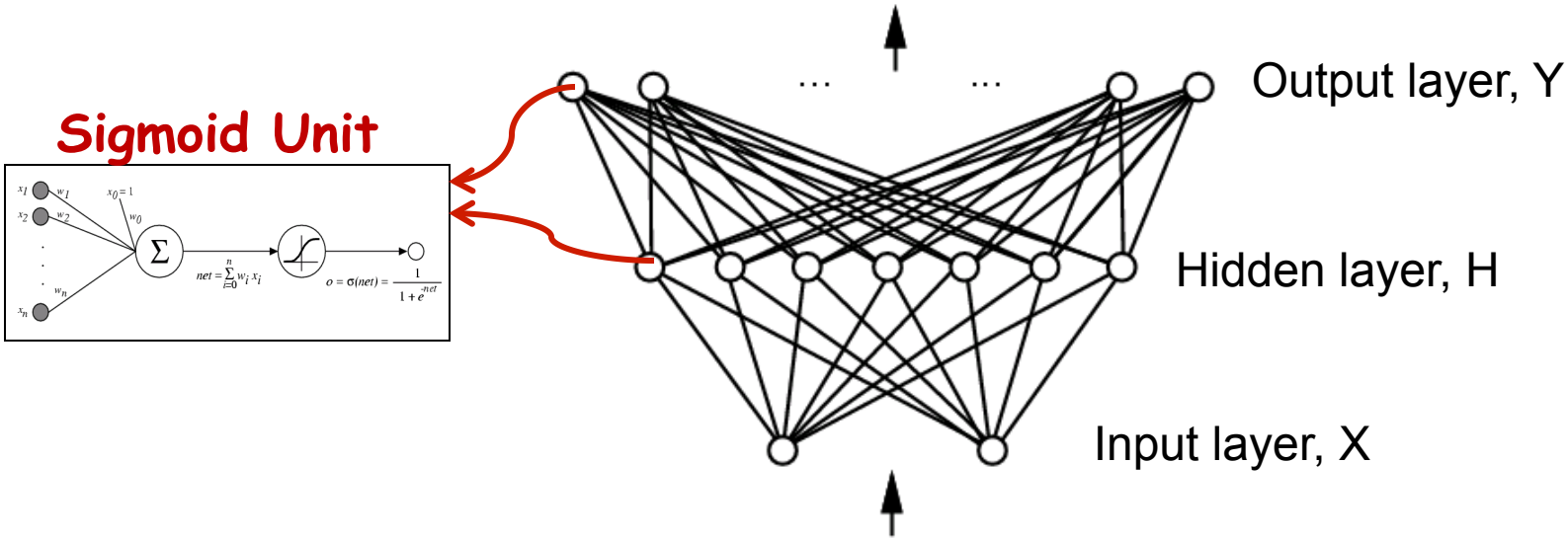
$$\text{Output, } o(\mathbf{x}) = \sigma(w_0 + \sum_i w_i X_i) = \frac{1}{1 + \exp(-(w_0 + \sum_i w_i X_i))}$$



Neural Networks to learn f: X → Y

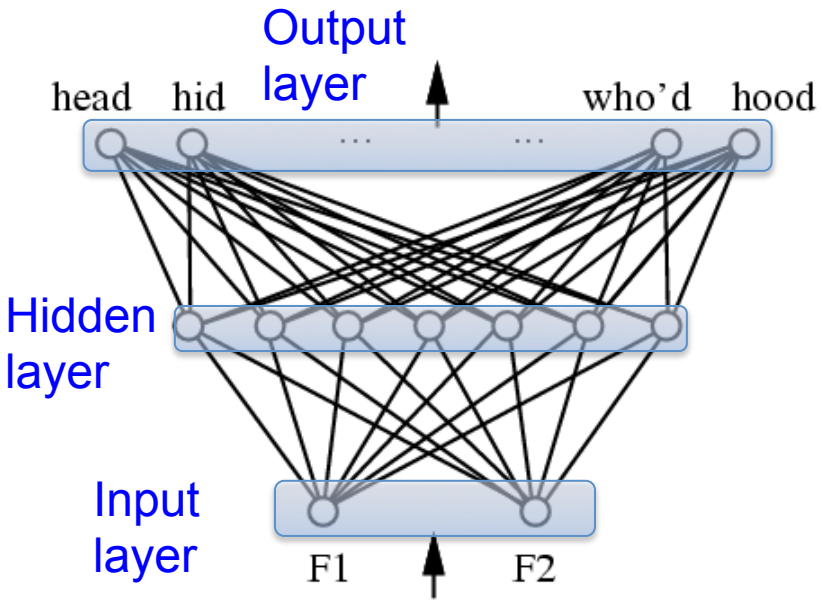
- f can be a **non-linear** function
- X (vector of) continuous and/or discrete variables
- Y (**vector** of) continuous and/or discrete variables

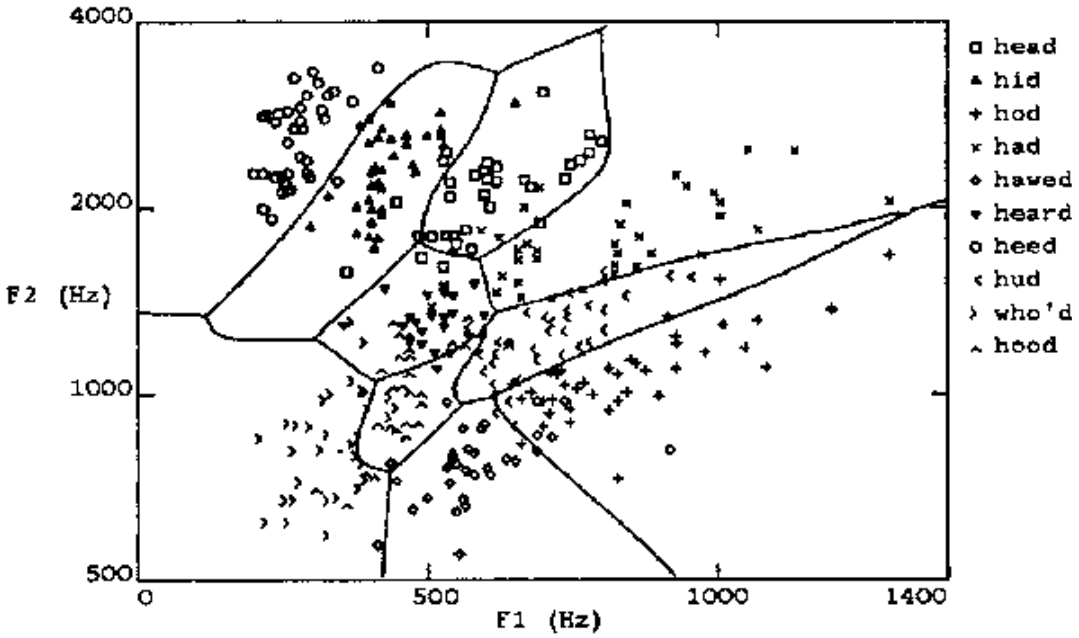
- Neural networks - Represent f by *network* of logistic/sigmoid units:



Multilayer Networks of Sigmoid Units

Neural Network trained to distinguish vowel sounds using 2 formants (features)

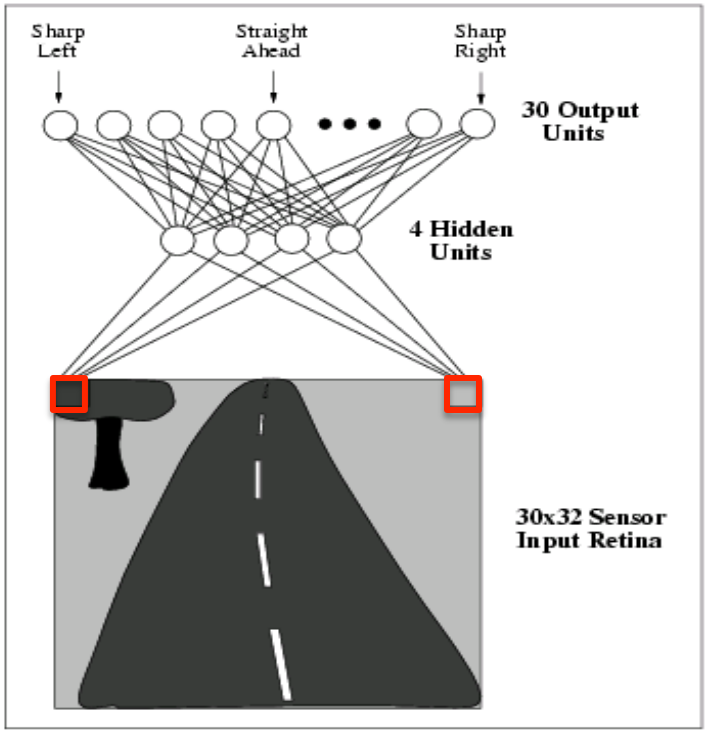




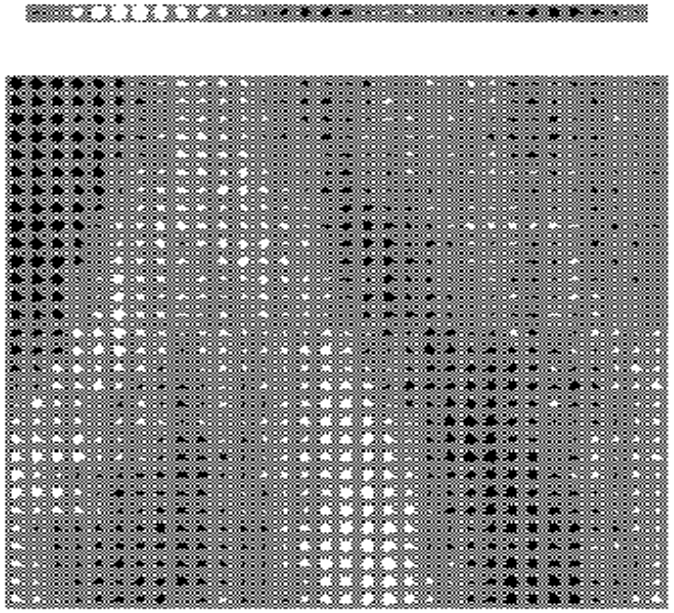
Two layers of logistic units

Highly non-linear decision surface

Neural Network trained to drive a car!



Weights to output units from the hidden unit



Weights of each pixel for one hidden unit

Connectionist Models

Consider humans:

- Neuron switching time ~ .001 second
- Number of neurons ~ 10^{10}
- Connections per neuron ~ 10^{4-5}
- Scene recognition time ~ .1 second
- 100 inference steps doesn't seem like enough

$\rightarrow$ much parallel computation

Properties of artificial neural nets (ANN's):

- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process

Forward Propagation for prediction

Prediction – Given neural network (hidden units and weights), use it to predict the label of a test point

Forward Propagation –

Start from input layer

For each subsequent layer, compute output of sigmoid unit

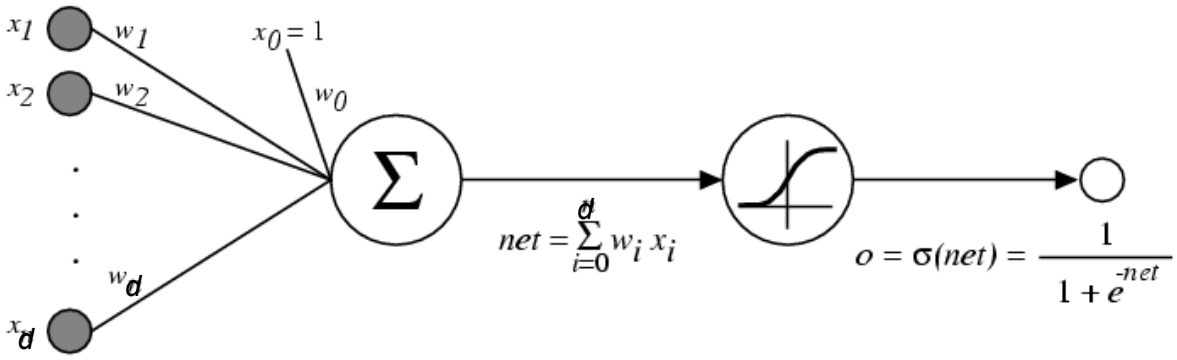
Sigmoid unit:

$$o(\mathbf{x}) = \sigma(w_0 + \sum_i w_i x_i)$$

1-Hidden layer, 1 output NN:

$$o(\mathbf{x}) = \sigma\left(w_0 + \sum_h w_h \underbrace{\sigma(w_0^h + \sum_i w_i^h x_i)}_{o_h}\right)$$

Training Neural Networks



$\sigma(x)$ is the sigmoid function

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

Nice property: $\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$ Differentiable

We can derive gradient decent rules to train

- One sigmoid unit
- *Multilayer networks* of sigmoid units → Backpropagation

M(C)LE Training for Neural Networks

- Consider regression problem f:X→Y , for scalar Y

$y = f(x) + \varepsilon$ ← assume noise $N(0,\sigma_\varepsilon)$, iid

deterministic

- Let's maximize the conditional data likelihood

$$W \leftarrow \arg\max_W \ \ln \prod_l P(Y^l | X^l, W)$$

$$W \leftarrow \arg\min_W \ \sum_l (y^l - \hat{f}(x^l))^2$$

Learned neural network

Train weights of all units to minimize sum of squared errors of predicted network outputs

MAP Training for Neural Networks

- Consider regression problem f:X→Y , for scalar Y

$y = f(x) + \varepsilon$ ← noise $N(0,\sigma_\varepsilon)$

deterministic

Gaussian $P(W) = N(0,\sigma I)$

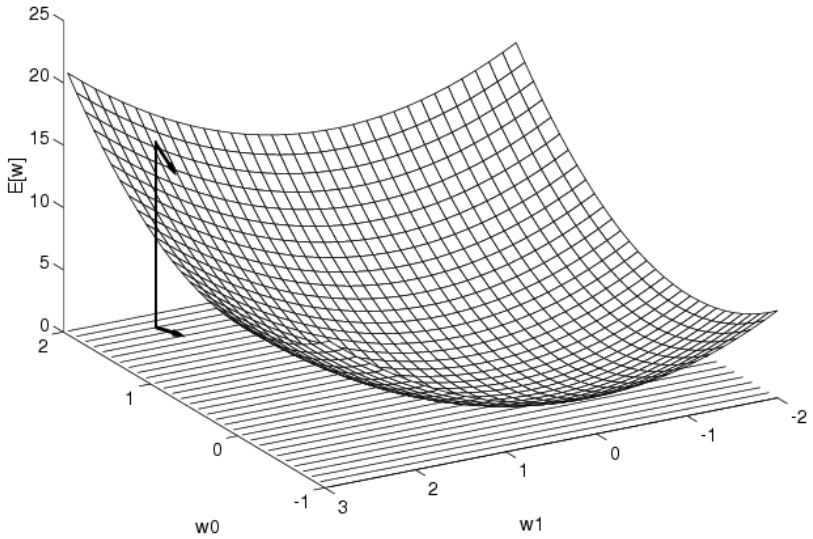
$$W \leftarrow \arg\max_W \ln P(W) \prod_l P(Y^l|X^l, W)$$

$$W \leftarrow \arg\min_W \left[c \sum_i w_i^2\right] + \left[\sum_l (y^l - \hat{f}(x^l))^2\right]$$

$\ln P(W) \leftrightarrow c \sum_i w_i^2$

Train weights of all units to minimize sum of squared errors of predicted network outputs plus weight magnitudes

Gradient Descent



E – Mean Square Error

Gradient

$$\nabla E[\vec{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots \frac{\partial E}{\partial w_d}\right]$$

Training rule:

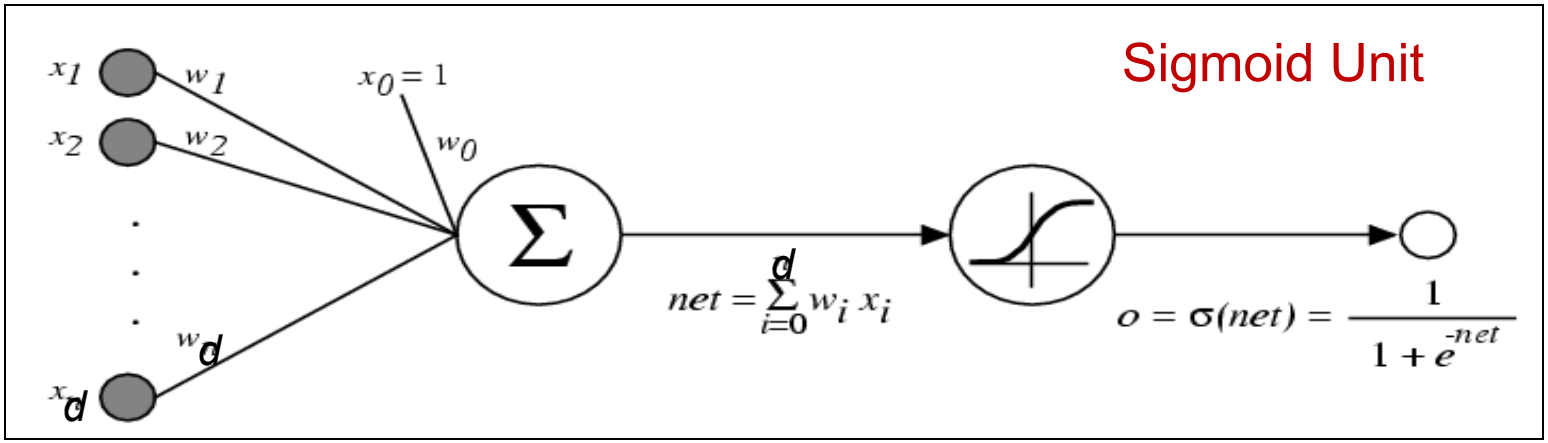
$$\Delta \vec{w} = -\eta \nabla E[\vec{w}]$$

i.e.,

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

For Neural Networks,
***E*[*w*] no longer convex in w**

Error Gradient for a Sigmoid Unit



$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{l \in D} (y^l - o^l)^2$$

$$= \frac{1}{2} \sum_l \frac{\partial}{\partial w_i} (y^l - o^l)^2$$

$$= \frac{1}{2} \sum_l 2(y^l - o^l) \frac{\partial}{\partial w_i} (y^l - o^l)$$

$$= \sum_l (y^l - o^l) \left(-\frac{\partial o^l}{\partial w_i} \right)$$

$$= -\sum_l (y^l - o^l) \frac{\partial o^l}{\partial net^l} \frac{\partial net^l}{\partial w_i}$$

But we know:

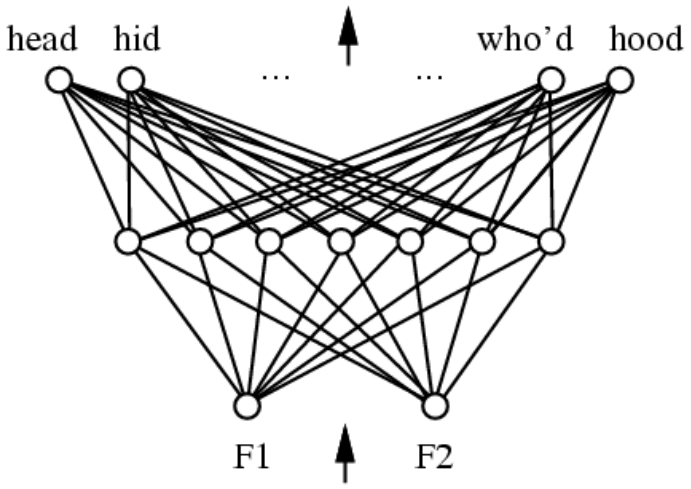
$$\frac{\partial o^l}{\partial net^l} = \frac{\partial \sigma(net^l)}{\partial net^l} = o^l (1 - o^l)$$

$$\frac{\partial net^l}{\partial w_i} = \frac{\partial (\vec{w} \cdot \vec{x}^l)}{\partial w_i} = x_i^l$$

So:

$$\frac{\partial E}{\partial w_i} = -\sum_{l \in D} (y^l - o^l) o^l (1 - o^l) x_i^l$$

Backpropagation Algorithm (MLE)



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 → Using Forward propagation
  2. For each output unit k

$$\delta_k^l \leftarrow o_k^l(1 - o_k^l)(y_k^l - o_k^l)$$

  3. For each hidden unit h

$$\delta_h^l \leftarrow o_h^l(1 - o_h^l) \sum_{k \in outputs} w_{h,k}\delta_k^l$$

  4. Update each network weight $w_{i,j}$

$$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}^l$$

  where

$$\Delta w_{i,j}^l = \eta \delta_j^l o_i^l$$

y_k = target output (label)

$o_{k/h}$ = unit output (obtained by forward propagation)

w_{ij} = wt from i to j

Note: if i is input variable, $o_i = x_i$

Incremental (Stochastic) Gradient Descent

Batch mode Gradient Descent:
Do until satisfied

1. Compute the gradient $\nabla E_D[\vec{w}]$ Using all training data *D*
2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_D[\vec{w}]$

$$E_D[\vec{w}] \equiv \frac{1}{2} \sum_{l \in D} (y^l - o^l)^2$$

Incremental mode Gradient Descent:
Do until satisfied

- For each training example l in D
  1. Compute the gradient $\nabla E_l[\vec{w}]$
  2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_l[\vec{w}]$

$$E_l[\vec{w}] \equiv \frac{1}{2}(y^l - o^l)^2$$

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

More on Backpropagation

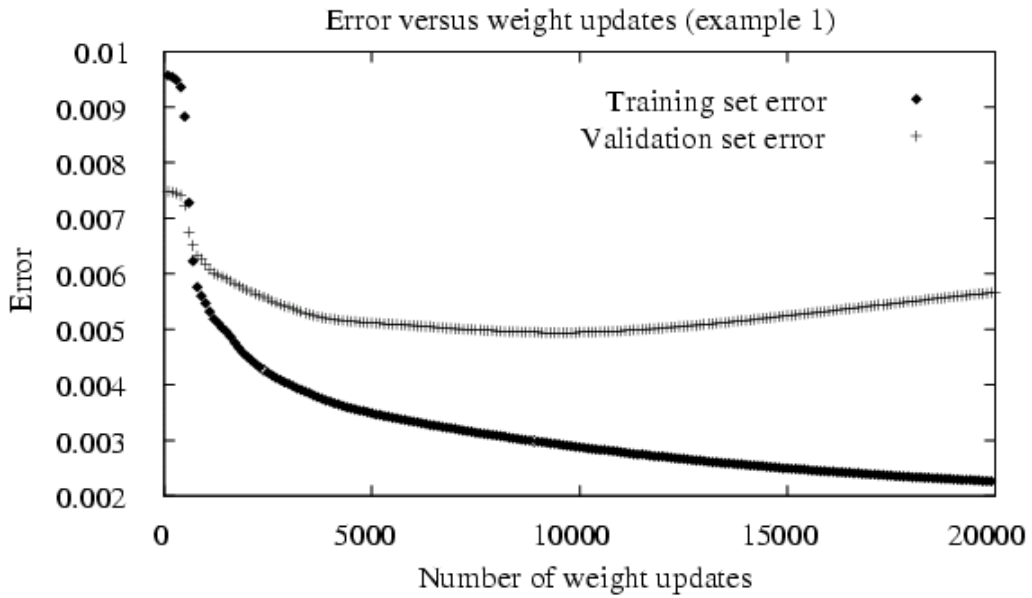
- Gradient descent over entire *network* weight vector
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum
  - In practice, often works well (can run multiple times)
- Often include weight *momentum* α

$$\Delta w_{i,j}(n) = \eta \delta_j x_{i,j} + \alpha \Delta w_{i,j}(n-1)$$

- Minimizes error over *training* examples
  - Will it generalize well to subsequent examples?
- Training can take thousands of iterations $\rightarrow$ slow!
- Using network after training is very fast

Objective/Error no longer convex in weights

Dealing with Overfitting



Our learning algorithm involves a parameter

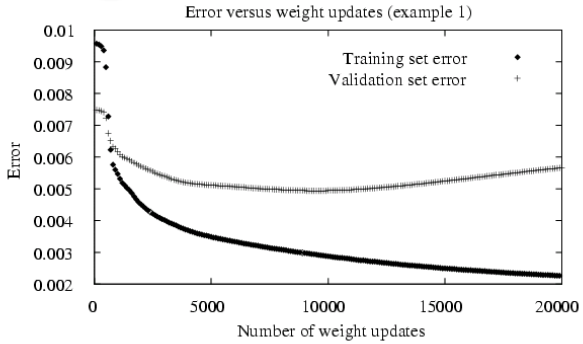
n=number of gradient descent iterations

How do we choose n to optimize future error?

(note: similar issue for logistic regression, decision trees, …)

e.g. the n that minimizes error rate of neural net over future data

Dealing with Overfitting



Our learning algorithm involves a parameter

n=number of gradient descent iterations

How do we choose n to optimize future error?

- Separate available data into training and validation set
- Use training to perform gradient descent
- n ← number of iterations that optimizes validation set error

K-fold Cross-validation

Idea: train multiple times, leaving out a disjoint subset of data each time for test. Average the test set accuracies.

Partition data into K disjoint subsets

For k=1 to K

testData = kth subset

h ← classifier trained* on all data except for testData

accuracy(k) = accuracy of h on testData

end

FinalAccuracy = mean of the K recorded testset accuracies

* might withhold some of this to choose number of gradient decent steps

Leave-one-out Cross-validation

This is just k-fold cross validation leaving out one example each iteration

---

Partition data into K disjoint subsets, *each containing one example*

For k=1 to K

testData = kth subset

h ← classifier trained* on all data except for testData

accuracy(k) = accuracy of h on testData

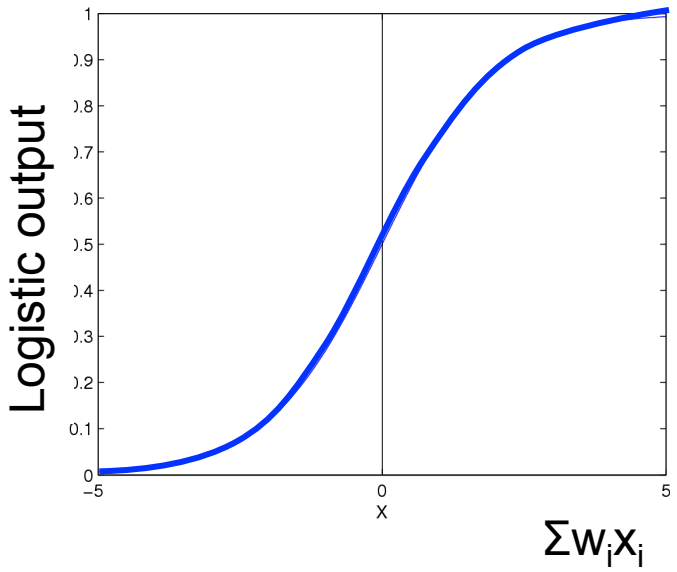
end

FinalAccuracy = mean of the K recorded testset accuracies

* might withhold some of this to choose number of gradient decent steps

Dealing with Overfitting

- Cross-validation
- Regularization – small weights imply NN is linear (low VC dimension)



- Control number of hidden units – low complexity

Expressive Capabilities of ANNs

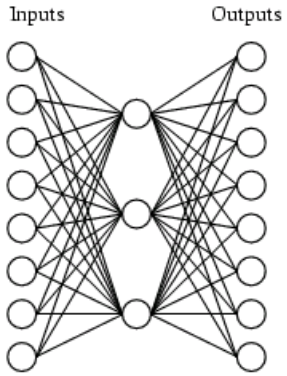
Boolean functions:

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

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].

Learning Hidden Layer Representations



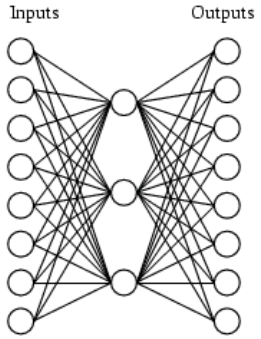
A target function:

Input		Output
10000000	$\rightarrow$	10000000
01000000	$\rightarrow$	01000000
00100000	$\rightarrow$	00100000
00010000	$\rightarrow$	00010000
00001000	$\rightarrow$	00001000
00000100	$\rightarrow$	00000100
00000010	$\rightarrow$	00000010
00000001	$\rightarrow$	00000001

Can this be learned??

Learning Hidden Layer Representations

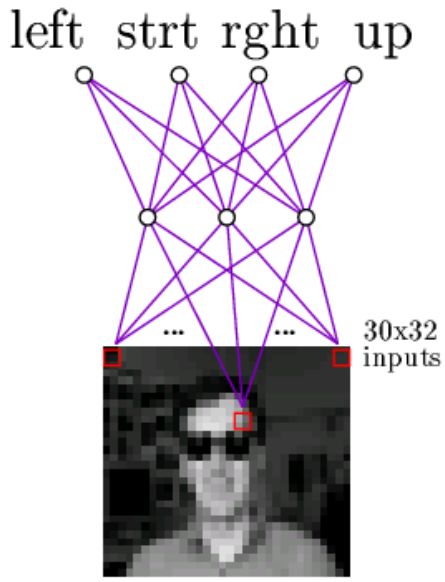
A network:



Learned hidden layer representation:

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

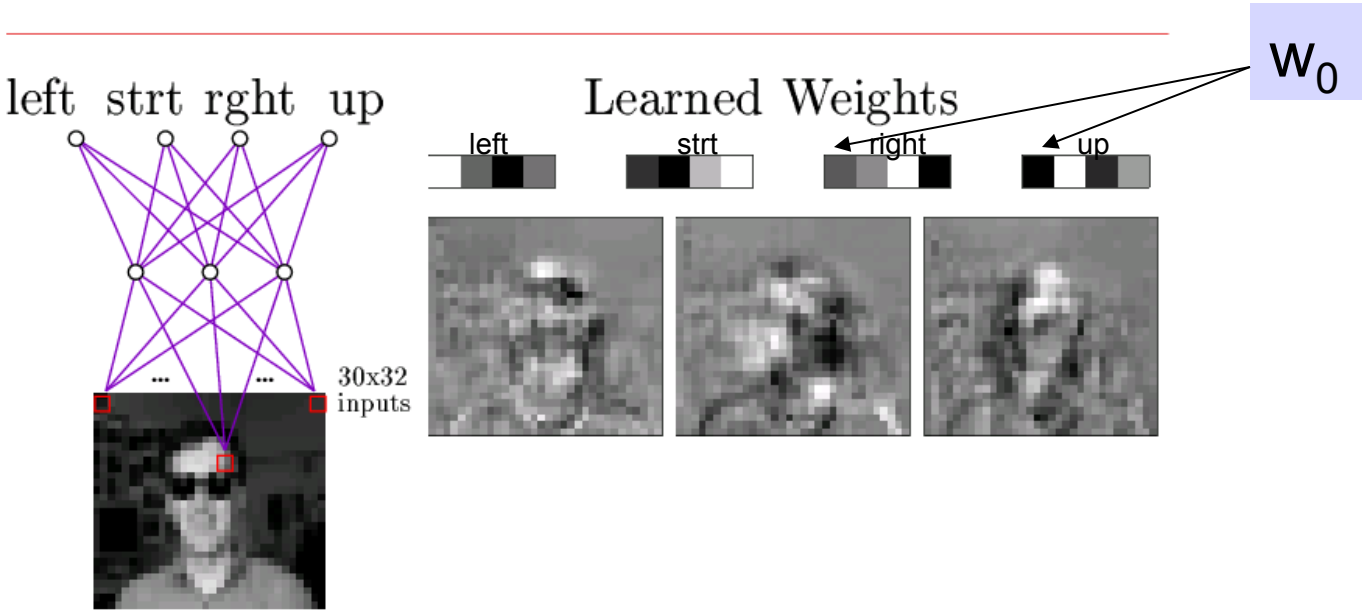
Neural Nets for Face Recognition



Typical input images

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

Learned Hidden Unit Weights

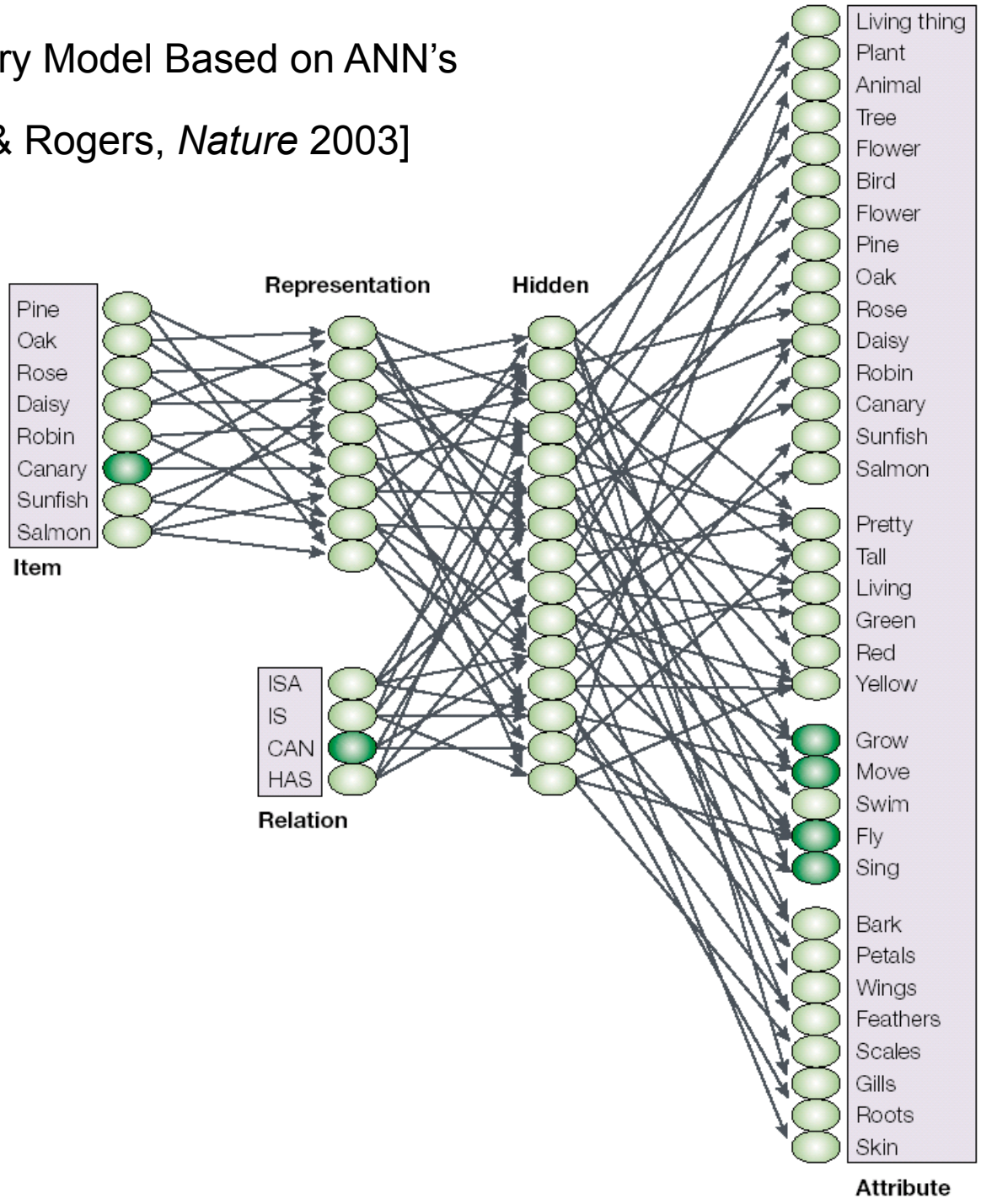


Typical input images

http://www.cs.cmu.edu/~tom/faces.html

Semantic Memory Model Based on ANN's

[McClelland & Rogers, *Nature* 2003]



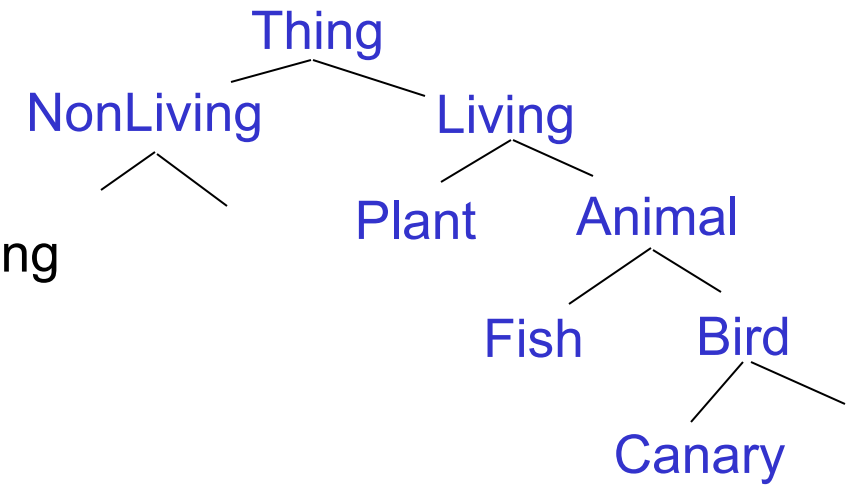
No hierarchy given.

Train with assertions,
e.g., Can(Canary,Fly)

Humans act as though they have a hierarchical memory organization

1. Victims of Semantic Dementia progressively lose knowledge of objects
   But they lose specific details first, general properties later, suggesting hierarchical memory organization

2. Children appear to learn general categories and properties first, following the same hierarchy, top down*.



Question: What learning mechanism could produce this emergent hierarchy?

* some debate remains on this.

Memory deterioration follows semantic hierarchy

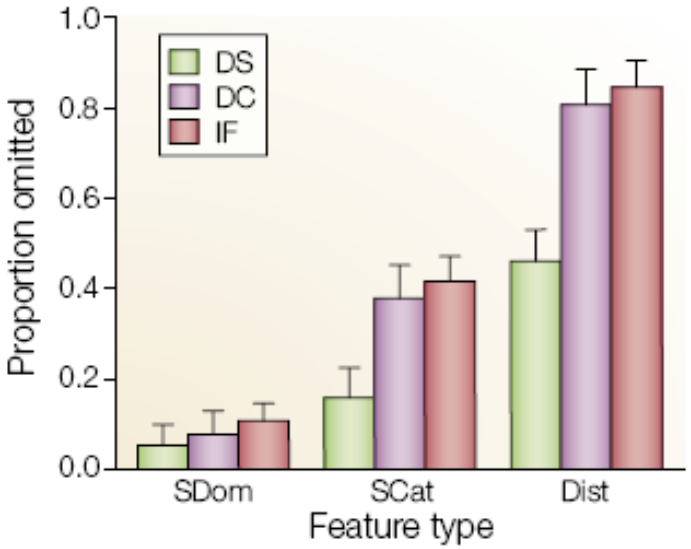
[McClelland & Rogers, *Nature* 2003]

a

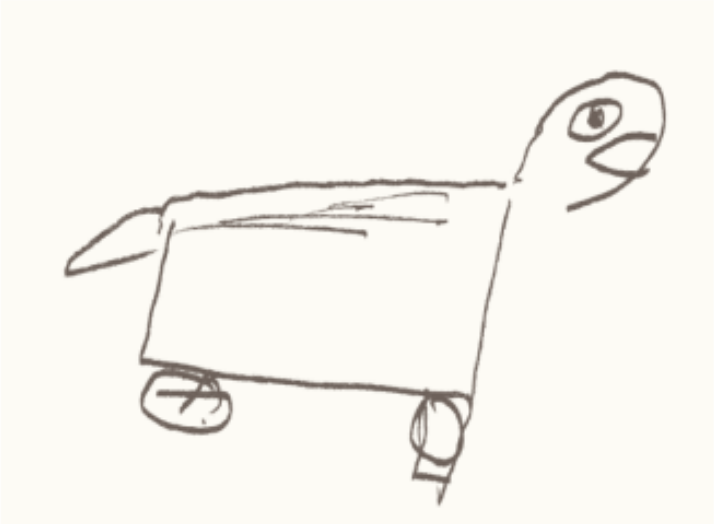
Picture naming responses for JL

Item	Sept. 91	March 92	March 93
Bird	+	+	Animal
Chicken	+	+	Animal
Duck	+	Bird	Dog
Swan	+	Bird	Animal
Eagle	Duck	Bird	Horse
Ostrich	Swan	Bird	Animal
Peacock	Duck	Bird	Vehicle
Penguin	Duck	Bird	Part of animal
Rooster	Chicken	Chicken	Dog

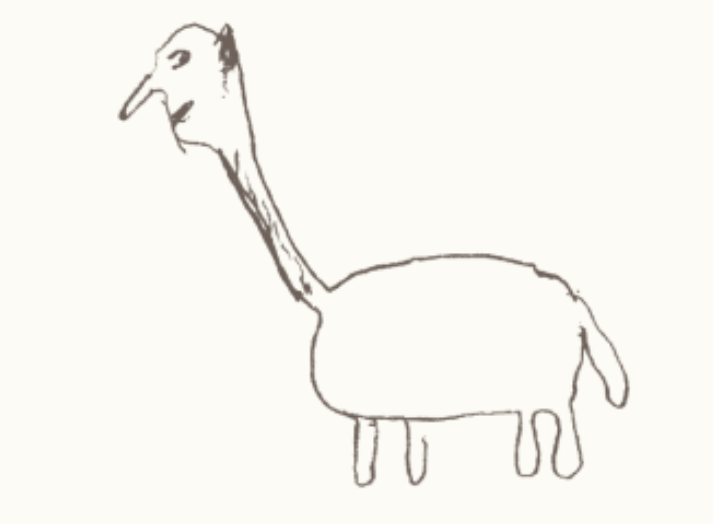
b



c IF's delayed copy of a camel



d DC's delayed copy of a swan



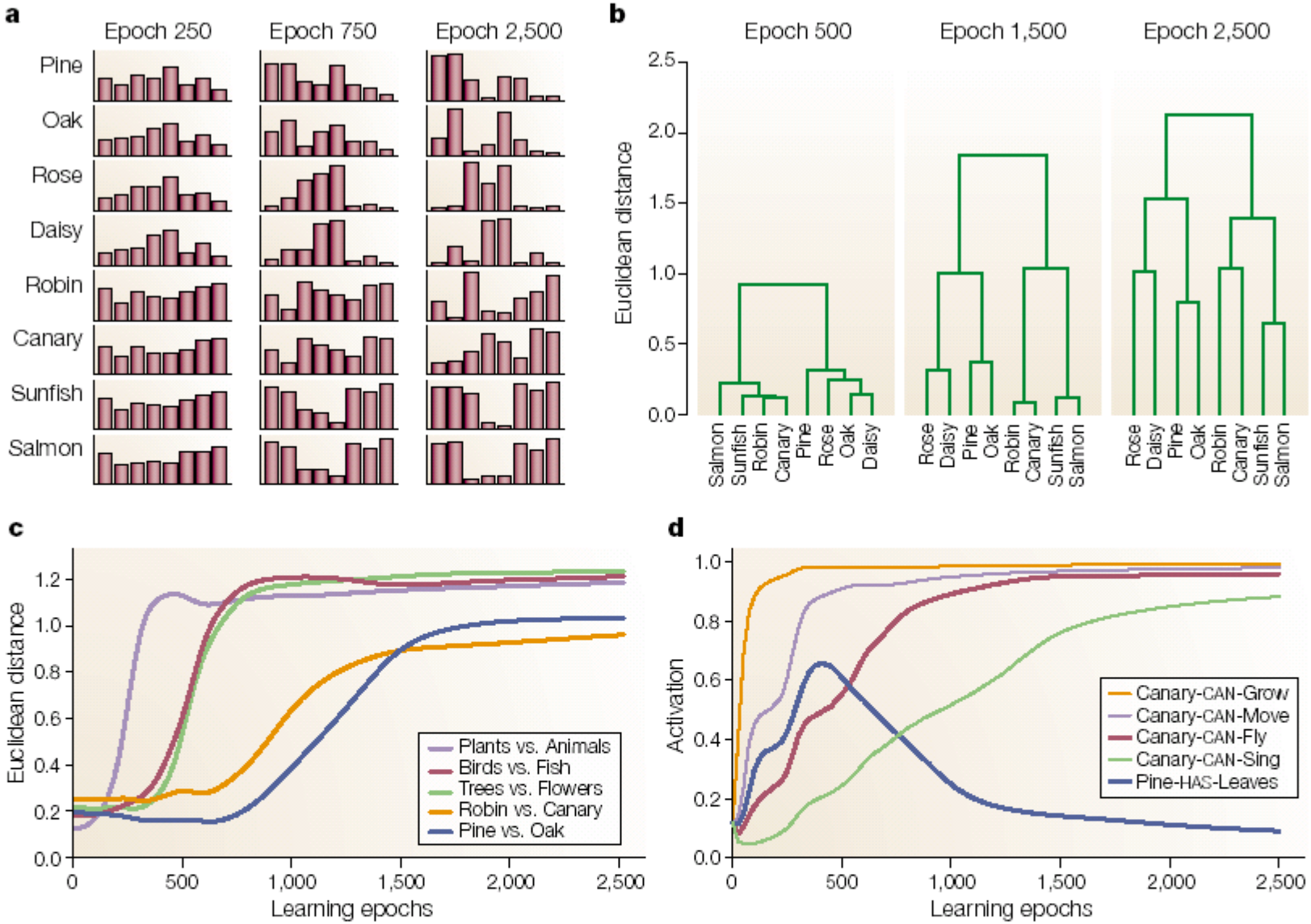


Figure 4 | **The process of differentiation of conceptual representations.** The representations are those seen in the feedforward network model shown in FIG. 3. **a** | Acquired patterns of activation that represent the eight objects in the training set at three points in the learning process (epochs 250, 750 and 2,500). Early in learning, the patterns are undifferentiated; the first difference to appear is between plants and animals. Later, the patterns show clear differentiation at both the superordinate (plant–animal) and intermediate (bird–fish/tree–flower) levels. Finally, the individual concepts are differentiated, but the overall hierarchical organization of the similarity structure remains. **b** | A standard hierarchical clustering analysis program has been used to visualize the similarity structure in the

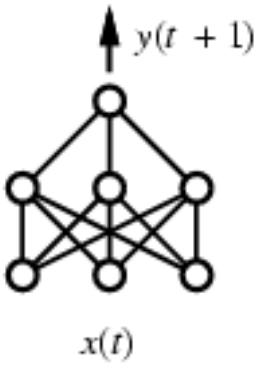
Training Networks on Time Series

- Suppose we want to predict next state of world
  - and it depends on history of unknown length
  - e.g., robot with forward-facing sensors trying to predict next sensor reading as it moves and turns

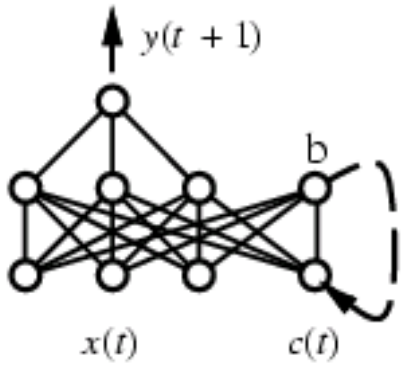
Training Networks on Time Series

- Suppose we want to predict next state of world
  - and it depends on history of unknown length
  - e.g., robot with forward-facing sensors trying to predict next sensor reading as it moves and turns

- Idea: use hidden layer in network to capture state history



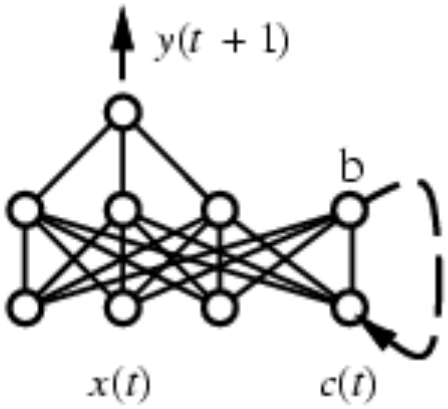
(a) Feedforward network

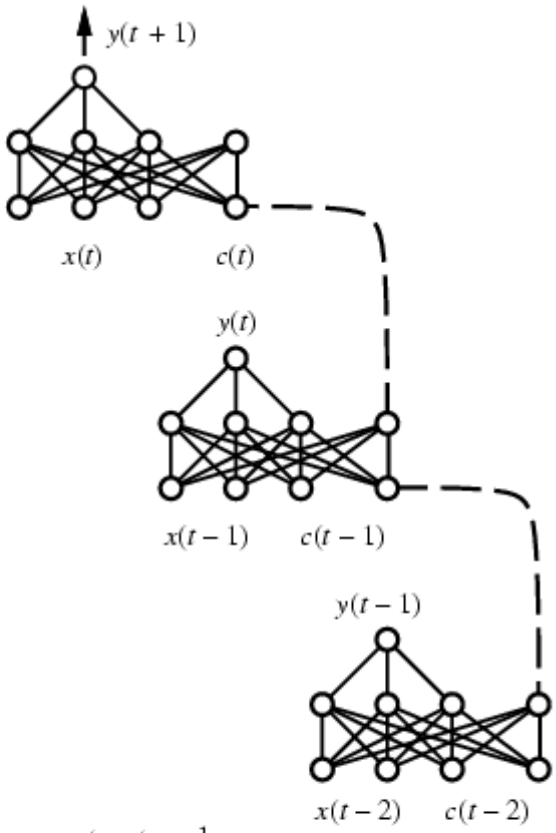


(b) Recurrent network

Training Networks on Time Series

How can we train recurrent net??





(c) Recurrent network unfolded in time

Artificial Neural Networks: Summary

- Actively used to model distributed computation in brain
- Highly non-linear regression/classification
- Vector-valued inputs and outputs
- Potentially millions of parameters to estimate - overfitting
- Hidden layers learn intermediate representations – how many to use?

- Prediction – Forward propagation
- Gradient descent (Back-propagation), local minima problems

- Mostly obsolete – kernel tricks are more popular, but coming back in new form as deep belief networks (probabilistic interpretation)