Neural Networks

Aarti Singh

Machine Learning 10-315
Sept 16, 2019

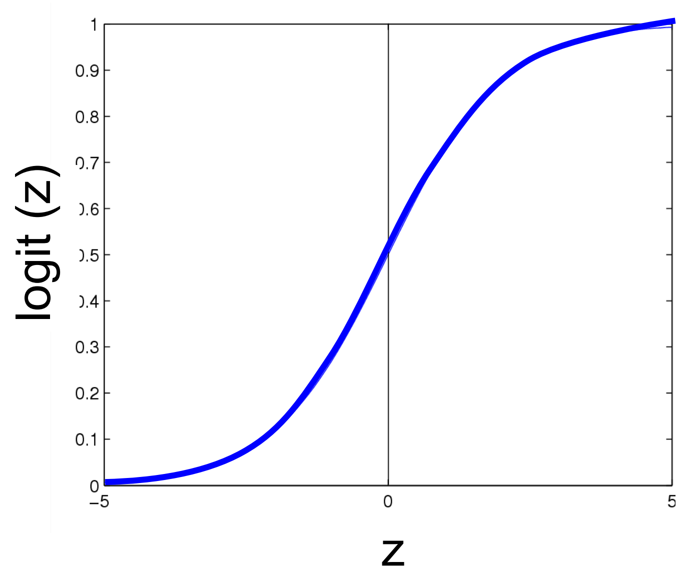
Logistic Regression

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): $\frac{1}{1+exp(-z)}$



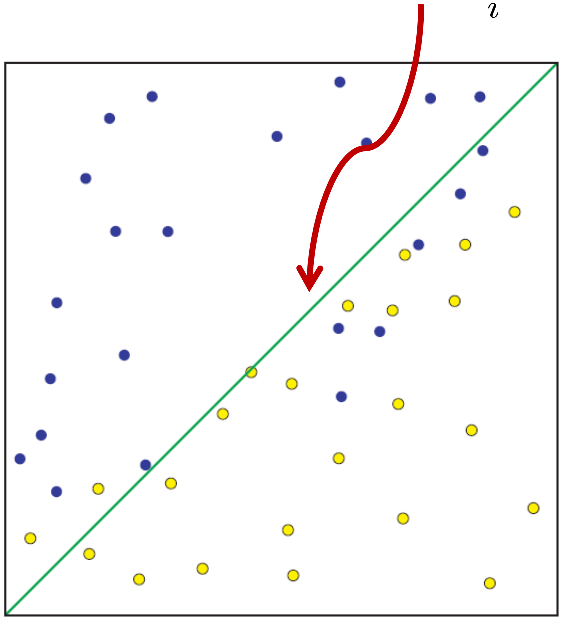
Logistic Regression is a Linear Classifier!

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:

$$w_0 + \sum_i w_i X_i = 0$$



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

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

(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 (<u>Conditional</u>) Likelihood Estimates

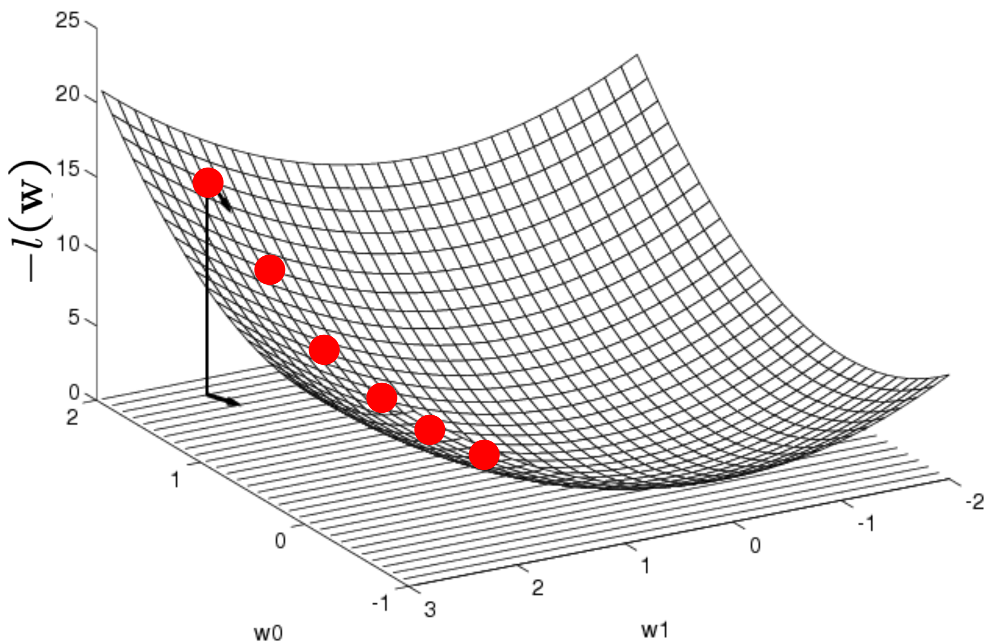
$$\widehat{\mathbf{w}}_{MCLE} = \arg\max_{\mathbf{w}} \prod_{j=1}^{n} P(Y^{(j)} \mid X^{(j)}, \mathbf{w})$$

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:

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

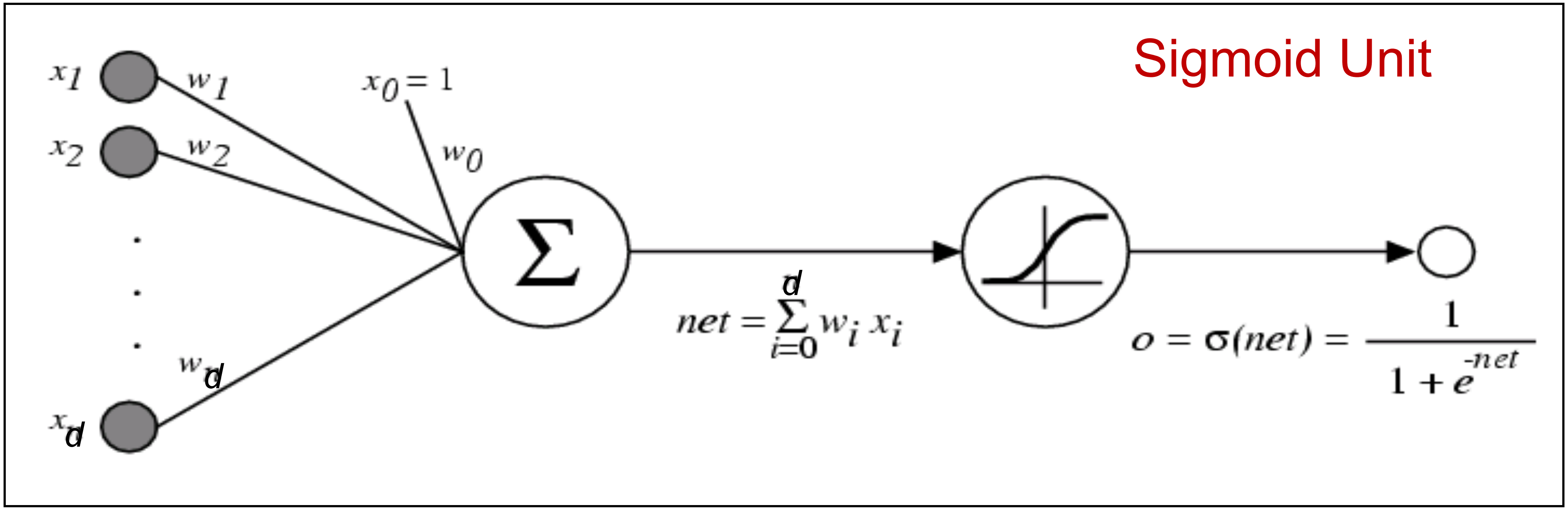
Update rule: Learning rate, η>0

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

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

Logistic function as a Graph

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

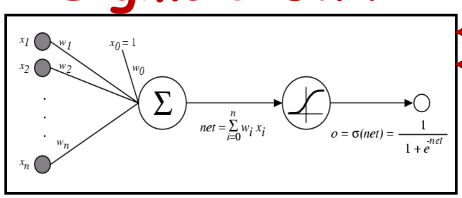


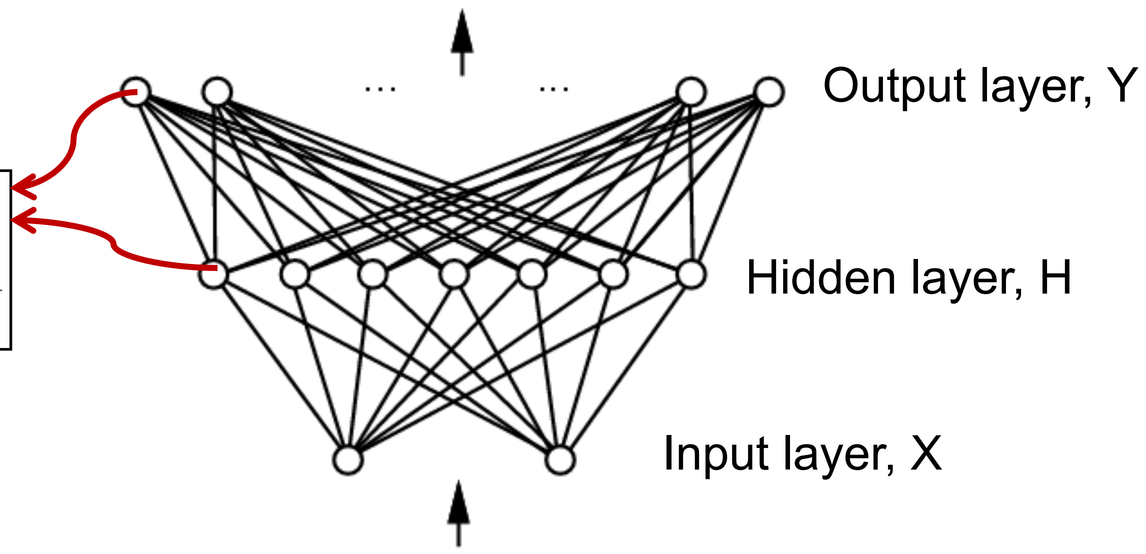
Neural Networks to learn f: X → Y

- 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:

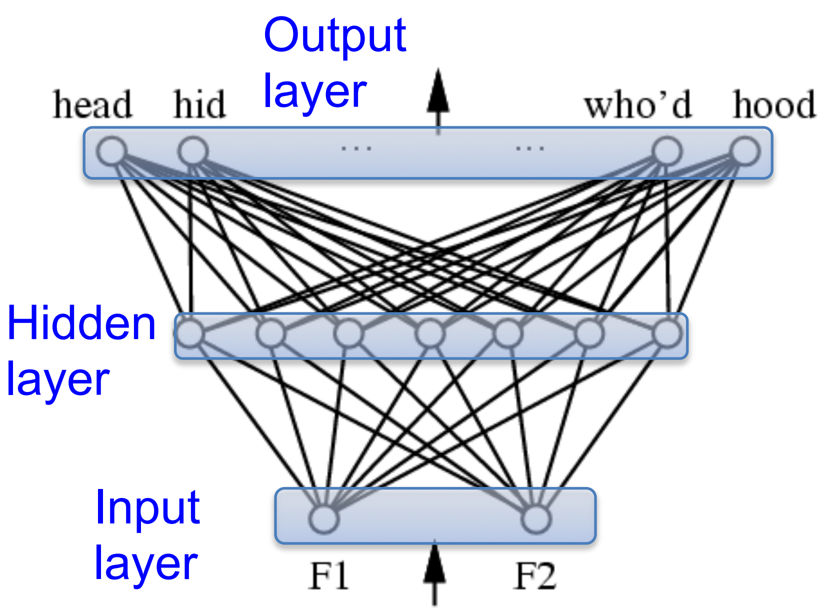
Sigmoid Unit

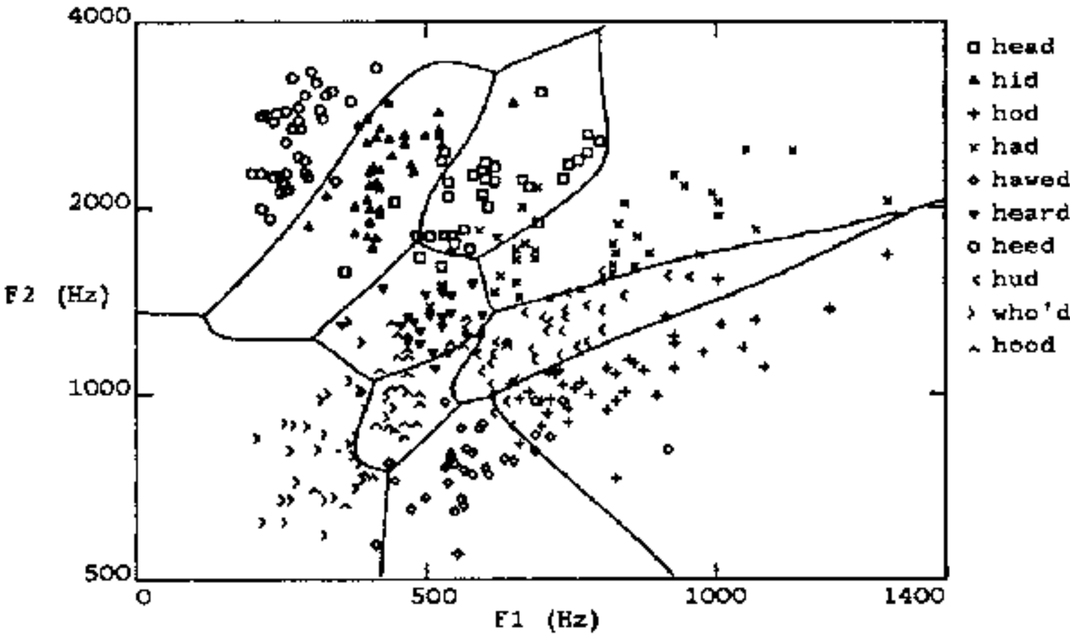




Multilayer Networks of Sigmoid Units

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

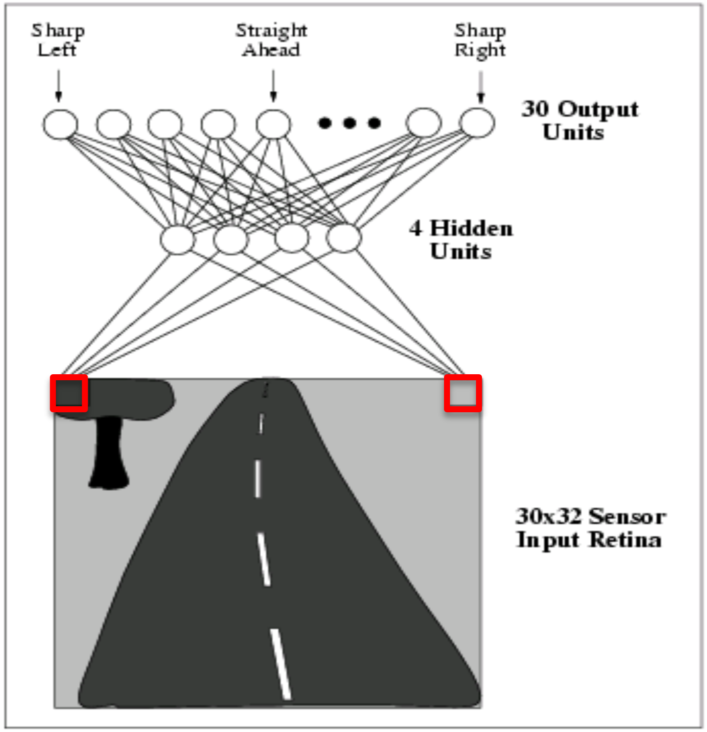




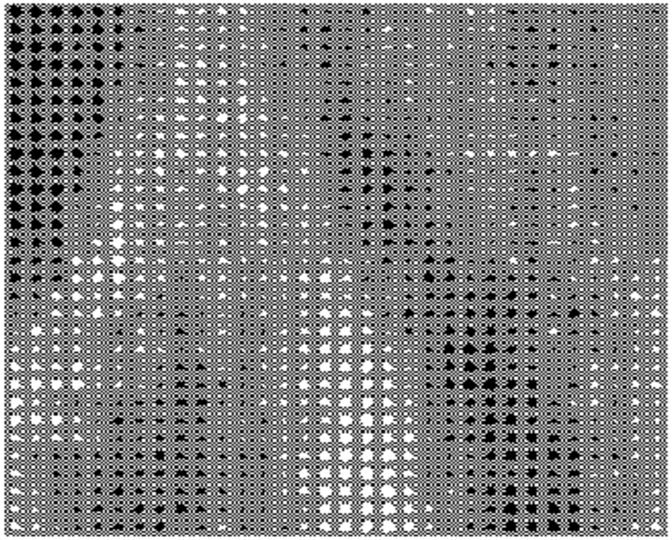
Two layers of logistic units

Highly non-linear decision surface

Neural Network trained to drive a car!



Weights to output units from one 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

Prediction using Neural Networks

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:

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

1-Hidden layer,
1 output NN:

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

M(C)LE Training for Neural Networks

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

$y = f(x) + \varepsilon$ ← assume noise $N(0,\sigma^2_\varepsilon)$, iid

deterministic

- Let's maximize the conditional data likelihood

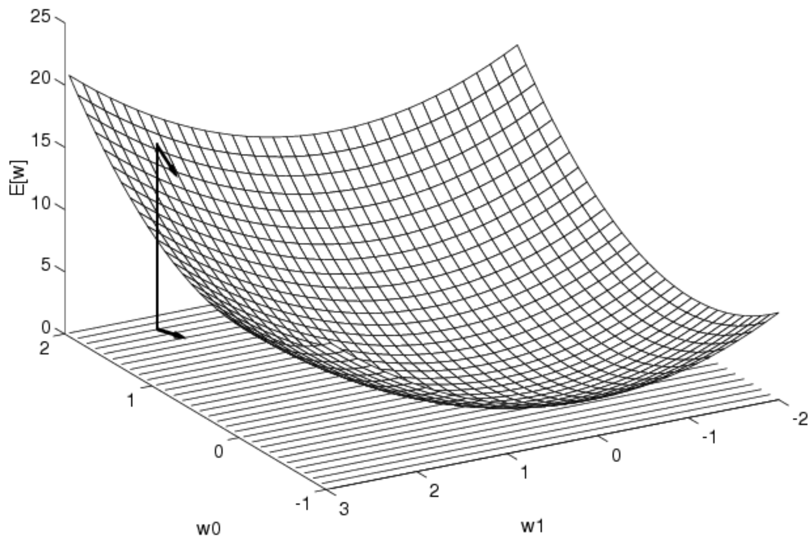
$$W \leftarrow \arg\max_W \ \ln \prod_l P(Y^l | X^l, W)$$

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

Learned neural network

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

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:

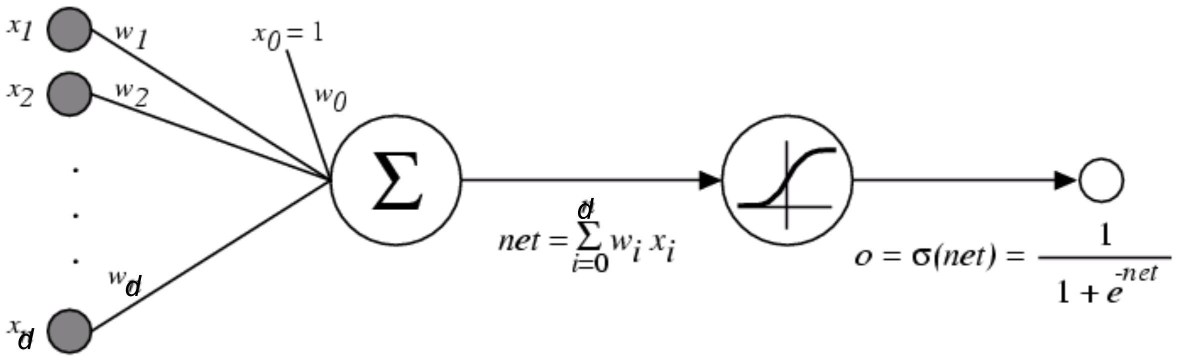
$$\Delta \vec{w} = -\eta \nabla E[\vec{w}]$$

i.e.,

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

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

Training Neural Networks



$\sigma(x)$ is the sigmoid function

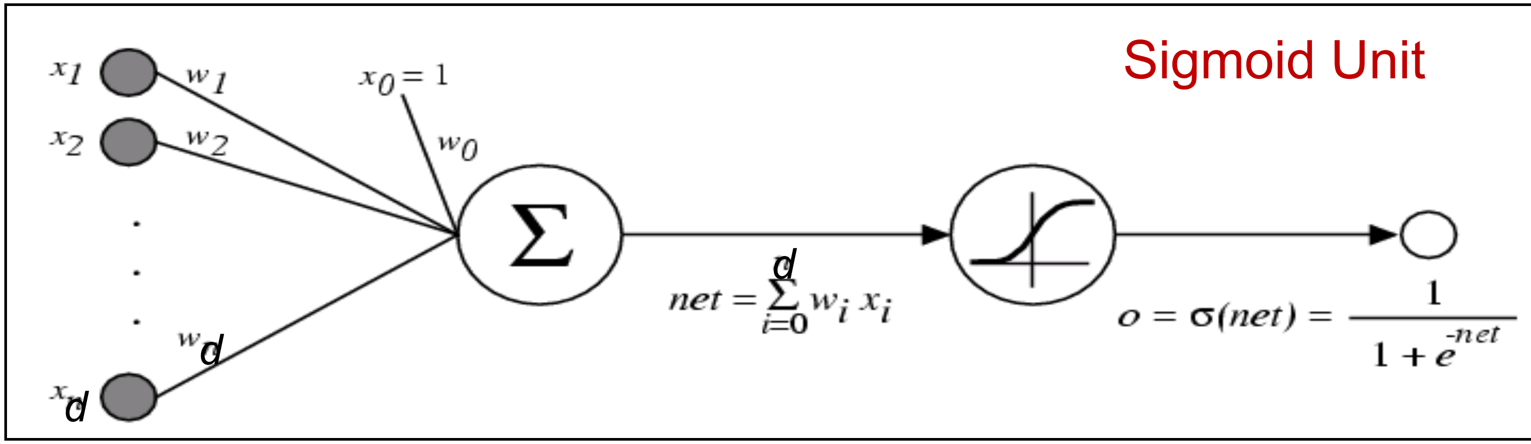
$$\frac{1}{1 + e^{-x}}$$

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 $\rightarrow$ Backpropagation

Error Gradient for a Sigmoid Unit



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

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

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

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

$$= - \sum_{d} (y^d - o^d) \frac{\partial o^d}{\partial net^d} \frac{\partial net^d}{\partial w_i}$$

But we know:

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

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

So:

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

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. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_l[\vec{w}]$

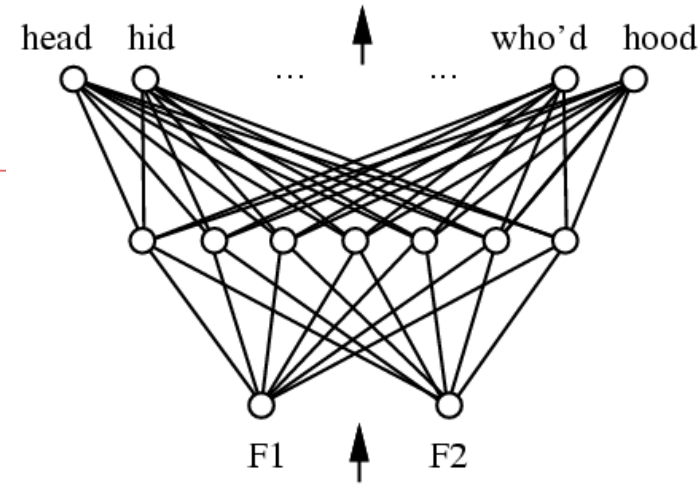
$$E_l[\vec{w}] \equiv \frac{1}{2} (y^l - o^l)^2$$

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

Error Gradient for 1-Hidden layer, 1-output neural network

http://www.cs.cmu.edu/~aarti/Class/10701/readings/Notes.pdf

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

l = training example

y_k = target output (label) of output unit k

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

w_{ij} = wt from i to j

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

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

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].