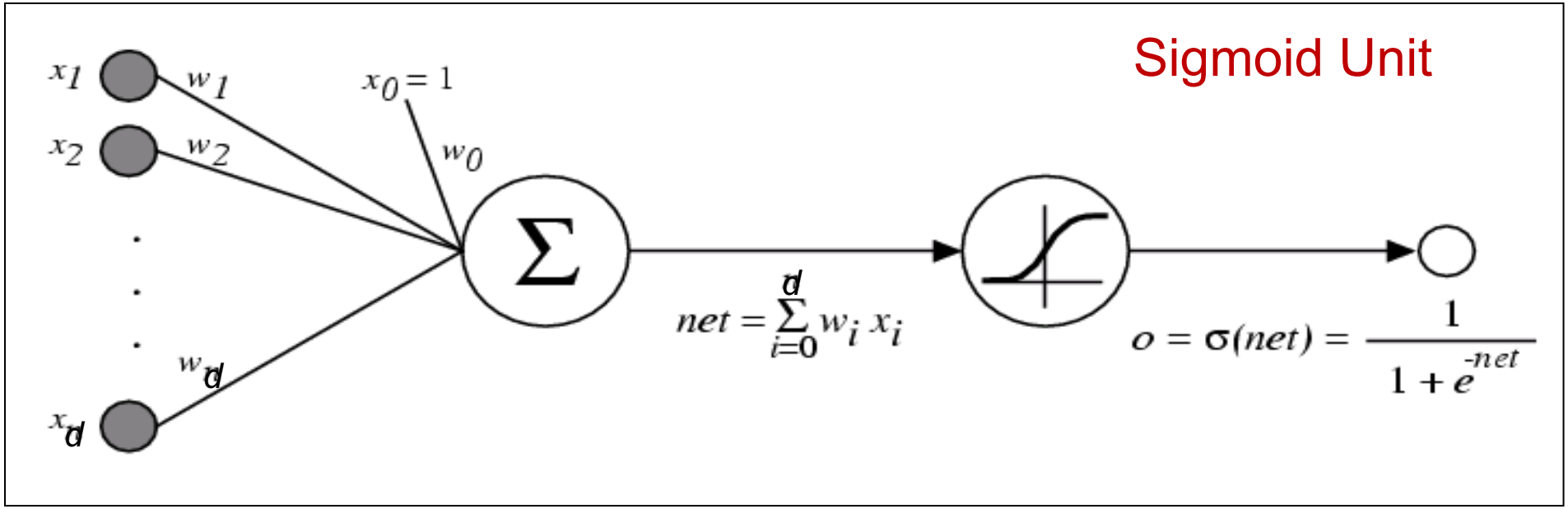
Neural Networks

Aarti Singh

Machine Learning 10-701
Feb 13, 2023

Logistic function as a Graph

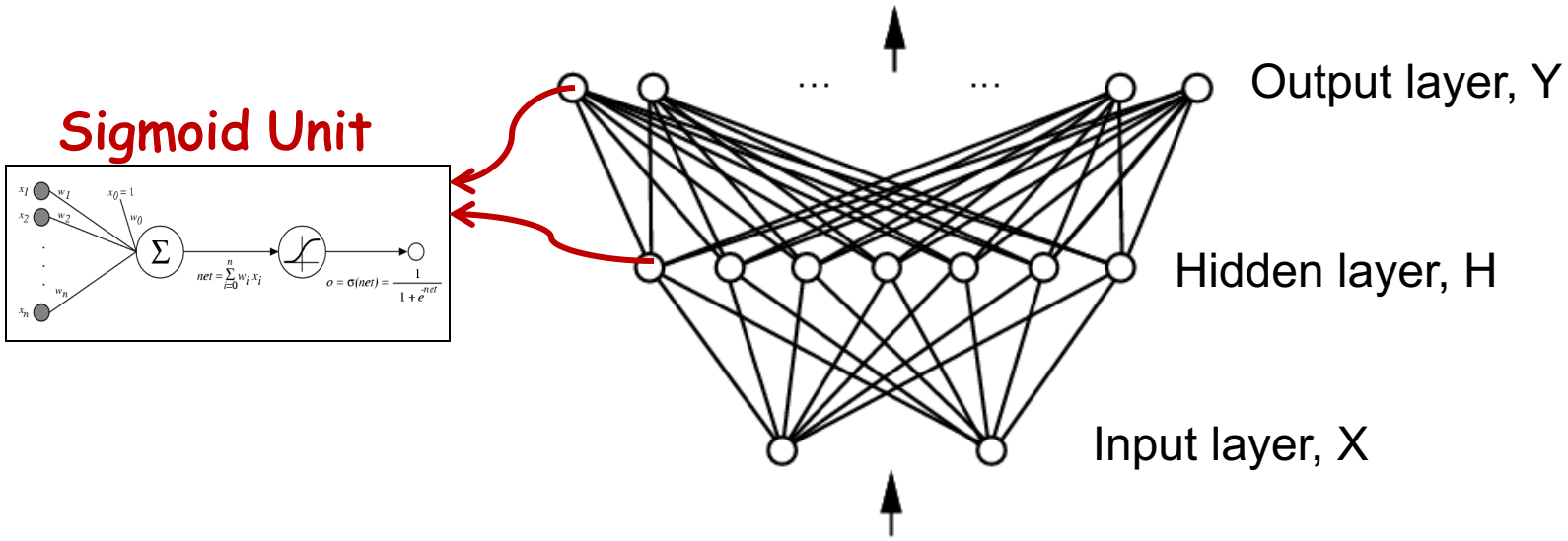
$$\text{Output, } o(\mathbf{x}) = \sigma(w_0 + \sum_i w_i X_i) = \frac{1}{1 + \exp(-(w_0 + \sum_i w_i X_i))}$$



Neural Networks to learn f: X → Y

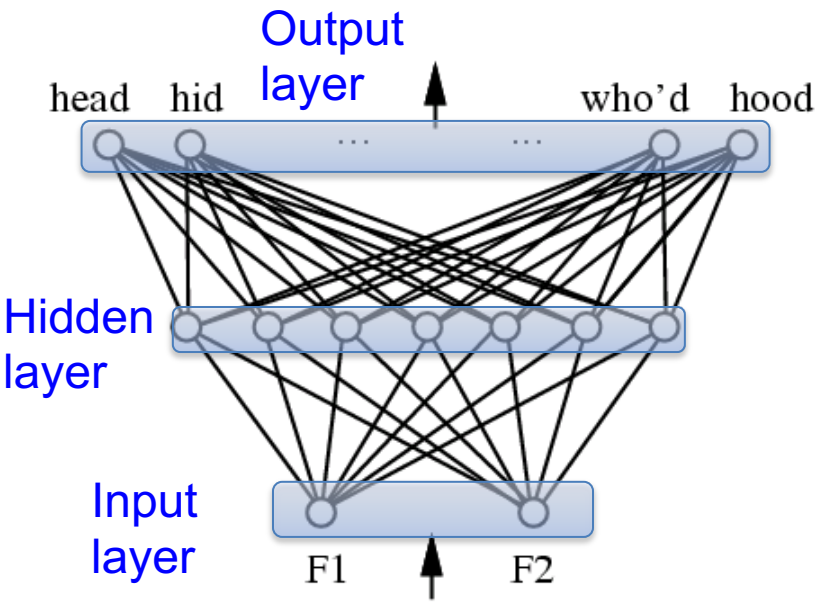
- f can be a **non-linear** function
- X (vector of) continuous and/or discrete variables
- Y (**vector** of) continuous and/or discrete variables

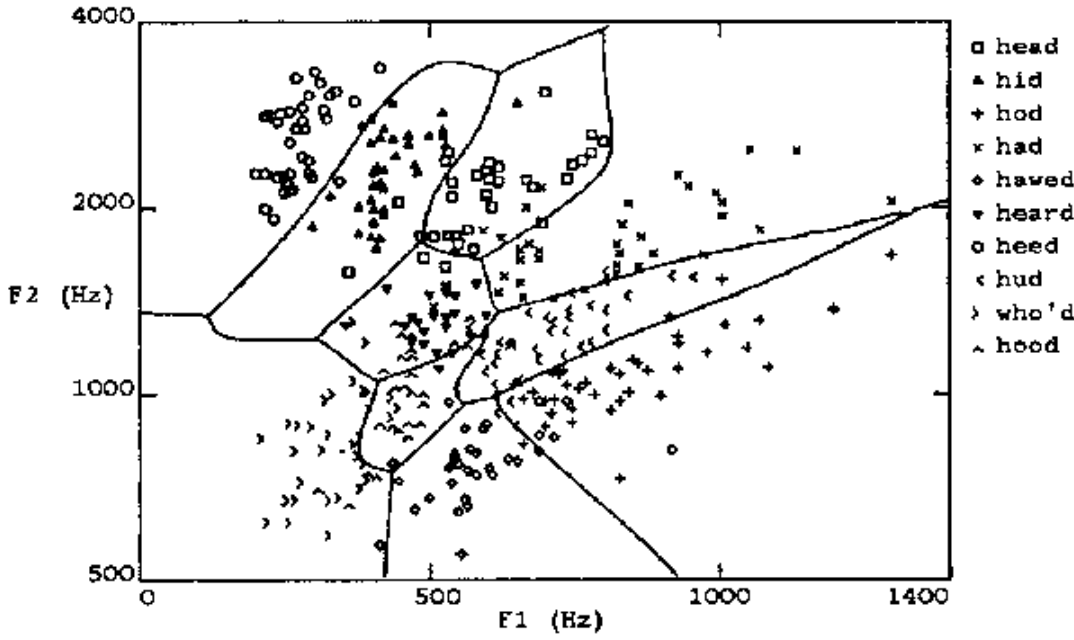
- Neural networks - Represent f by *network* of sigmoid (more recently ReLU – next lecture) units :



Multilayer Networks of Sigmoid Units

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

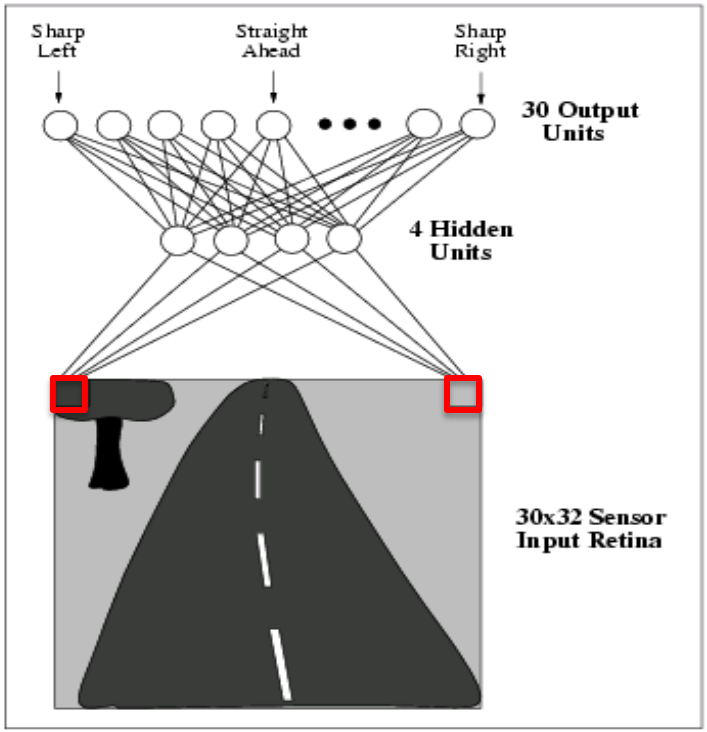




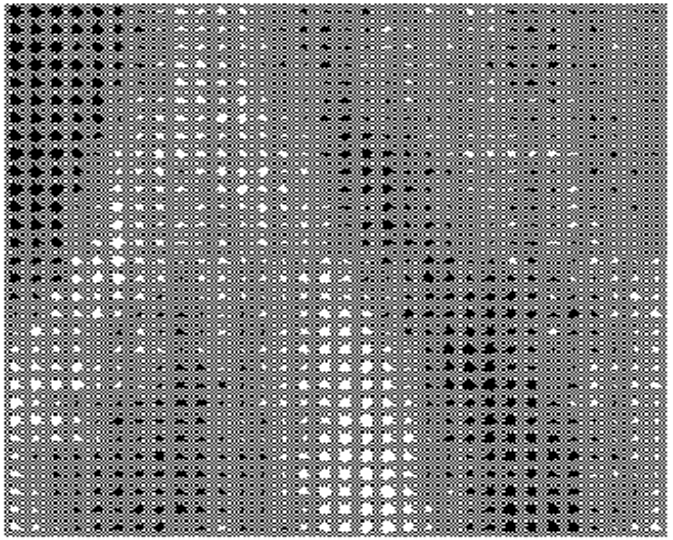
Two layers of logistic units

Highly non-linear decision surface

Neural Network trained to drive a car!



Weights to output units from 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

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].

Expressive Capabilities of NNs

1 hidden layer NN demo

https://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html

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:

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

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 – l2 loss

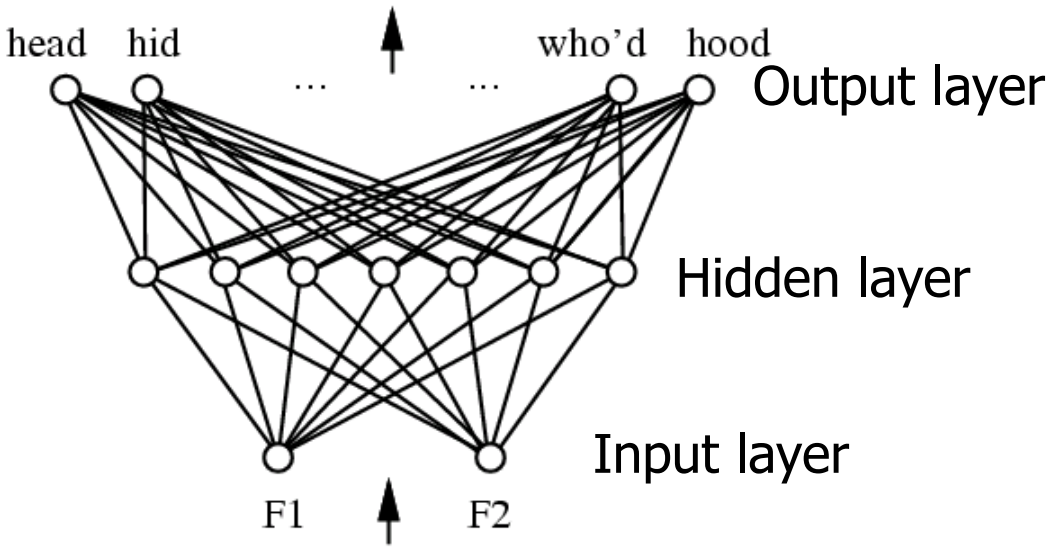
Train weights of all units to minimize sum of squared errors of predicted network outputs

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

Output of learned neural network

E[*W*]

- Objective E[*W*] is no longer convex in *W*.
- Still use Gradient descent to minimize E[*W*].
- Training is slow with lot of data and lot of weights!



Stochastic gradient descent

Stochastic gradient descent (SGD): Simplify computation by using a single data point at each iteration (instead of sum over all data points)

$$W \leftarrow \arg\min_{W} \underbrace{\sum_{l} (y^l - \hat{f}(x^l))^2}_{\text{E}[W]}$$

At each iteration of gradient descent

- Approximate E[W] $\approx (y^l - \hat{f}(x^l))^2$
- Stochastic Gradient =

Cycle through all points, then restart OR choose random data point at each iteration

Gradient descent for training NNs

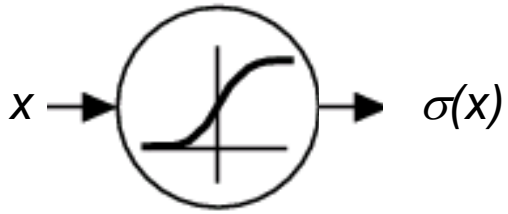
Gradient descent via **Chain rule** for computing gradients

Gradient of loss with respect to one weight w_i

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

Stochastic gradient:

Derivative of sigmoid



$\sigma(x)$ is the sigmoid function

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

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

Differentiable

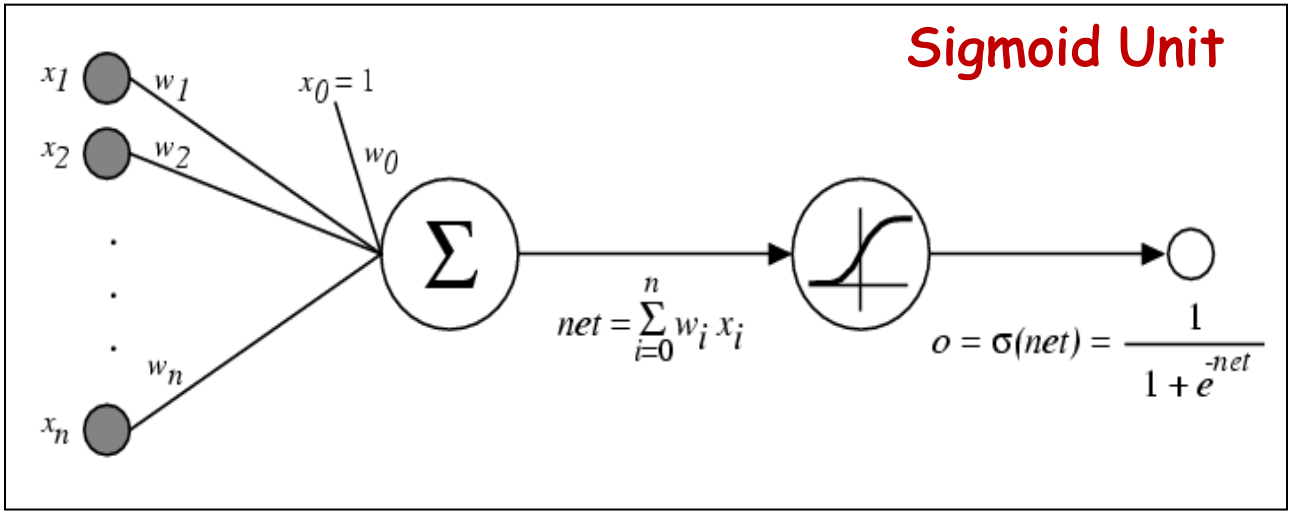
A. σ(x)(1- σ(x))

B. σ(x) σ(-x)

C. - σ(x)

D. $\sigma(x)^2$

Gradient Descent for 1 node



$$\frac{\partial o}{\partial w_i} = \frac{\partial o}{\partial net} \cdot \frac{\partial net}{\partial w_i} = o(1-o)x_i$$

Gradient descent step:

Backpropagation

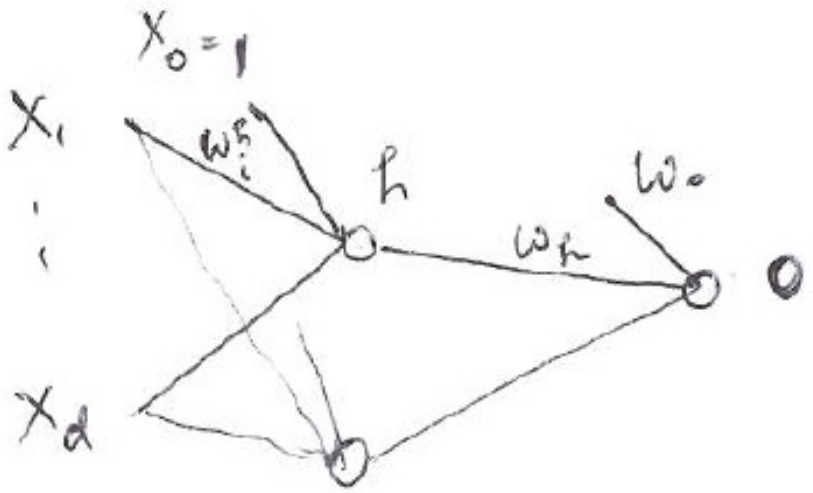
Backpropagation: Efficient implementation of (Stochastic) Gradient descent for Neural networks with multiple layers

chain rule for gradients

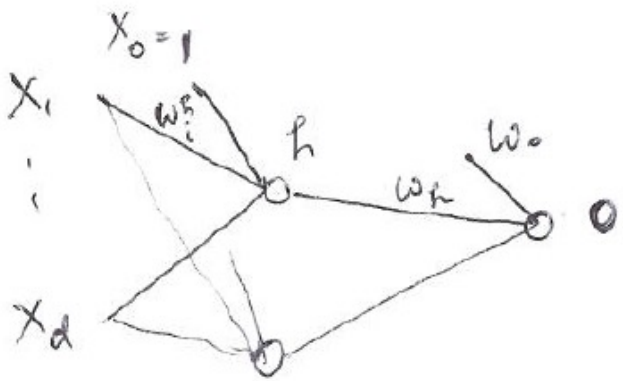
+

layer-wise computation

(going backward from output to input)



Gradient Descent for 1 hidden layer 1 output NN



$$o = \sigma\left(w_0 + \sum_h w_h o_h\right) \equiv \sigma\left(\sum_h w_h o_h\right)$$

$$o_h = \sigma\left(w_0^h + \sum_i w_i^h x_i\right) \equiv \sigma\left(\sum_i w_i^h x_i\right)$$

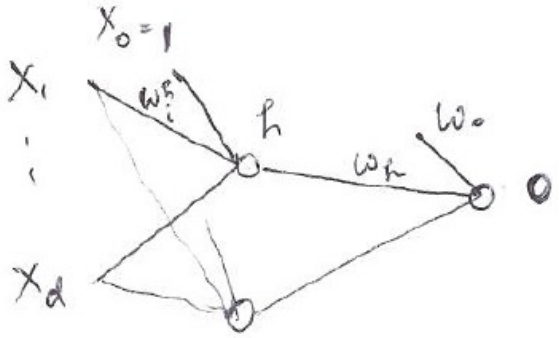
Gradient of the output with respect to one **final** layer weight w_h

$$\frac{\partial o}{\partial w_h} = o(1-o)o_h$$

Backpropagation Algorithm

using Stochastic gradient descent

1 final output unit



Initialize all weights to small random numbers.
Until satisfied, Do

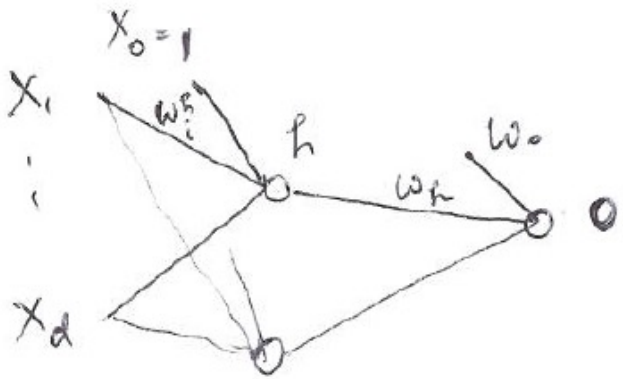
- For each training example, Do
  1. Input the training example to the network and compute the network outputs → Using Forward propagation
  2. $$\delta \leftarrow o(1-o)(y-o)$$
  3.
  4. Update each network weight w_h

$$w_h \leftarrow w_h + \Delta w_h$$

where

$$\Delta w_h = \eta \delta o_h$$

Gradient Descent for 1 hidden layer 1 output NN



$$o = \sigma\left(w_0 + \sum_h w_h o_h\right) \equiv \sigma\left(\sum_h w_h o_h\right)$$

$$o_h = \sigma\left(w_0^h + \sum_i w_i^h x_i\right) \equiv \sigma\left(\sum_i w_i^h x_i\right)$$

Gradient of the output with respect to one **hidden** layer weight w_i^h

$$\frac{\partial o}{\partial w_i^h} = \frac{\partial o}{\partial o_h} \cdot \frac{\partial o_h}{\partial w_i^h}$$

$\frac{\partial o_h}{\partial w_i^h} =$

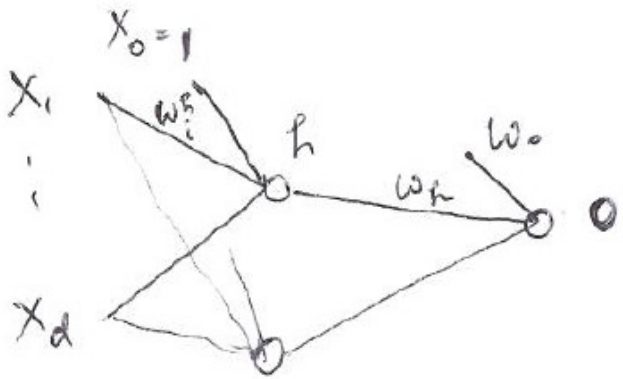
A. $o_h(1-o_h)$

B. $o_h(1-o_h)x_i$

C. $o(1-o)\,x_i$

D. $o(1-o)$

Gradient Descent for 1 hidden layer 1 output NN



$$o = \sigma\left(w_0 + \sum_h w_h o_h\right) \equiv \sigma\left(\sum_h w_h o_h\right)$$

$$o_h = \sigma\left(w_0^h + \sum_i w_i^h x_i\right) \equiv \sigma\left(\sum_i w_i^h x_i\right)$$

Gradient of the output with respect to one **hidden** layer weight w_i^h

$$\frac{\partial o}{\partial w_i^h} = \frac{\partial o}{\partial o_h} \cdot \frac{\partial o_h}{\partial w_i^h}$$

$\frac{\partial o}{\partial o_h} =$

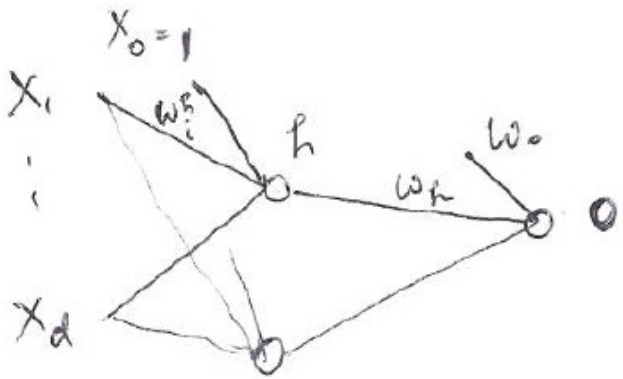
A. $o(1-o)x_i$

B. $o(1-o)o_h$

C. $o(1-o)\, w_h$

D. $o(1-o)$

Gradient Descent for 1 hidden layer 1 output NN



$$o = \sigma\left(w_0 + \sum_h w_h o_h\right) \equiv \sigma\left(\sum_h w_h o_h\right)$$

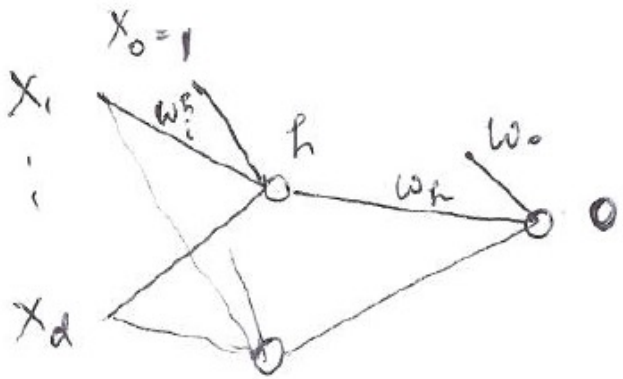
$$o_h = \sigma\left(w_0^h + \sum_i w_i^h x_i\right) \equiv \sigma\left(\sum_i w_i^h x_i\right)$$

Gradient of the output with respect to one **hidden** layer weight w_i^h

$$\frac{\partial o}{\partial w_i^h} = \frac{\partial o}{\partial o_h} \cdot \frac{\partial o_h}{\partial w_i^h} = o(1-o)w_h \cdot o_h(1-o_h)x_i$$

$$\frac{\partial o_h}{\partial w_{i'}^h} = \frac{\partial o}{\partial o_h} \cdot \frac{\partial o_h}{\partial w_{i'}^h} = o(1-o)w_h \cdot o_h(1-o_h)x_{i'}$$

Gradient Descent for 1 hidden layer 1 output NN



$$o = \sigma\left(w_0 + \sum_h w_h o_h\right) \equiv \sigma\left(\sum_h w_h o_h\right)$$

$$o_h = \sigma\left(w_0^h + \sum_i w_i^h x_i\right) \equiv \sigma\left(\sum_i w_i^h x_i\right)$$

Gradient of the output with respect to one **final** layer weight w_h

$$\frac{\partial o}{\partial w_h} = o(1-o)o_h$$

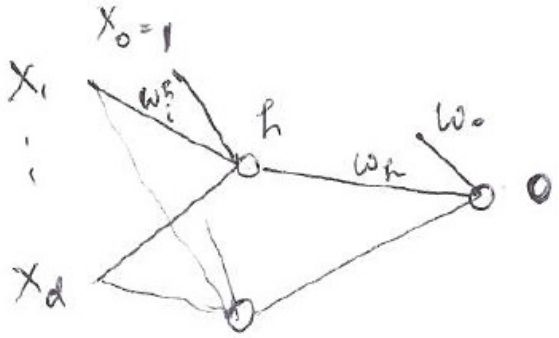
Gradient of the output with respect to one **hidden** layer weight w_i^h

$$\frac{\partial o}{\partial w_i^h} = \frac{\partial o}{\partial o_h} \cdot \frac{\partial o_h}{\partial w_i^h} = o(1-o)w_h \cdot o_h(1-o_h)x_i$$

Backpropagation Algorithm

using Stochastic gradient descent

1 final output unit



Initialize all weights to small random numbers.
Until satisfied, Do

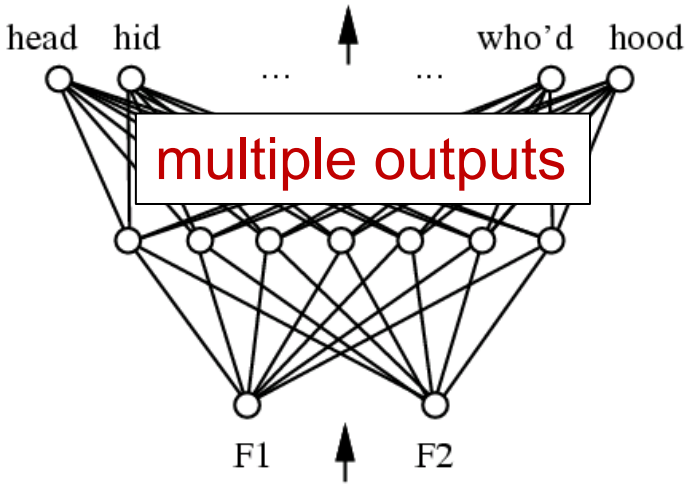
- For each training example, Do
  1. Input the training example to the network and compute the network outputs → Using Forward propagation
  2. $$\delta \leftarrow o(1-o)(y-o)$$
  3. For each hidden unit h
  $$\delta_h \leftarrow o_h(1-o_h)w_h\delta$$
  4. Update each network weight $w_{i,j}$
  $$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$
  where
  $$\Delta w_{i,j} = \eta \delta_j o_i$$

w_{ij} = wt from i to j

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

Backpropagation Algorithm

using Stochastic gradient descent



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

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

  3. For each hidden unit h

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

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

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

where

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

y_k = label of current training example for output unit k

o_k or o_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)

Objective/Error no longer convex in weights

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

HW2

- Cross-entropy error metric for multi-class classification

$$-\sum_k y_k \, log \, \widehat{y_k}$$ loss for single data point

One-hot encoding – encode label as a vector [y_1, y_2, … y_K]

where y_k = 1 if label is k and 0 otherwise

Interpret vector as probability distribution

HW2

- Cross-entropy error metric for multi-class classification

$$-\sum_k y_k \, log \, \widehat{y_k}$$

Entropy of a random variable X:

$E_{X\sim p}[-\log p(X)]$ small p(X) => more information

$-\log_2 p(X)$ = number of bits needed to encode an outcome X when we know true distribution p

Cross-entropy = expected number of bits needed to encode a random draw of X when using distribution q

$E_{X\sim p}[-\log q(X)]$ Minimized when q=p

HW2

- Cross-entropy error metric for multi-class classification

$$-\sum_k y_k \, log \, \widehat{y_k}$$

Cross-entropy = expected number of bits needed to encode a random draw of X when using distribution q

$$E_{X\sim p}[-\log q(X)]$$

Interpret one-hot-encoding *y* and $\hat{y}$ as distributions.