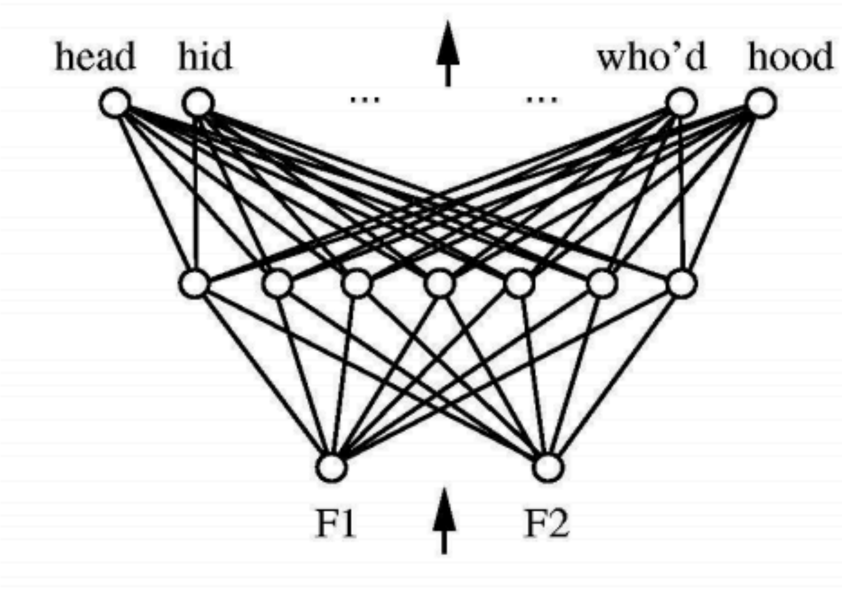
# Example multilayer neural network




output units

hidden units

input units

figure from Huang & Lippmann, *NIPS* 1988

input: two features from spectral analysis of a spoken sound

output: vowel sound occurring in the context "h__d"

# Decision regions of a multilayer neural network

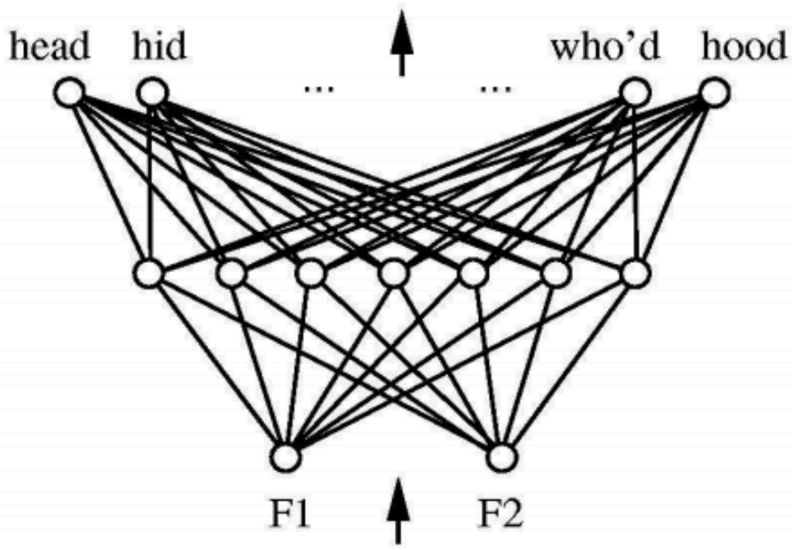
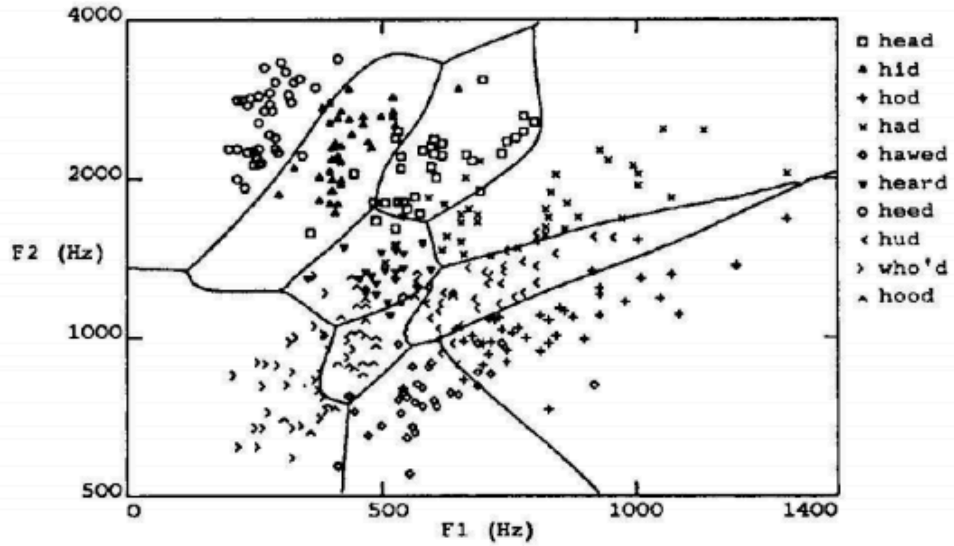



figure from Huang & Lippmann, *NIPS* 1988

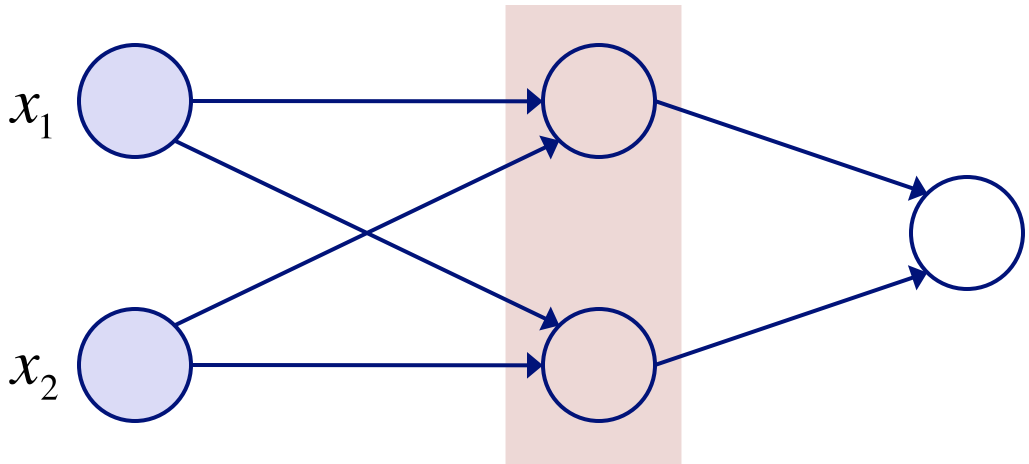



input: two features from spectral analysis of a spoken sound

output: vowel sound occurring in the context "h__d"

# Learning in multilayer networks

- work on neural nets fizzled in the 1960's
  - single layer networks had representational limitations (linear separability)
  - no effective methods for training multilayer networks




how to determine error signal for *hidden units?*

- revived again with the invention of *backpropagation* method [Rumelhart & McClelland, 1986; also Werbos, 1975]
  - key insight: require neural network to be differentiable; use *gradient descent*

# Gradient descent in weight space

Given a training set $D = \left\{(\boldsymbol{x}^{(1)}, y^{(1)})\ldots(\boldsymbol{x}^{(m)}, y^{(m)})\right\}$ we can specify an error measure that is a function of our weight vector $\boldsymbol{w}$

$$E(\boldsymbol{w}) = \frac{1}{2}\sum_{d \in D}\left(y^{(d)} - o^{(d)}\right)^2$$

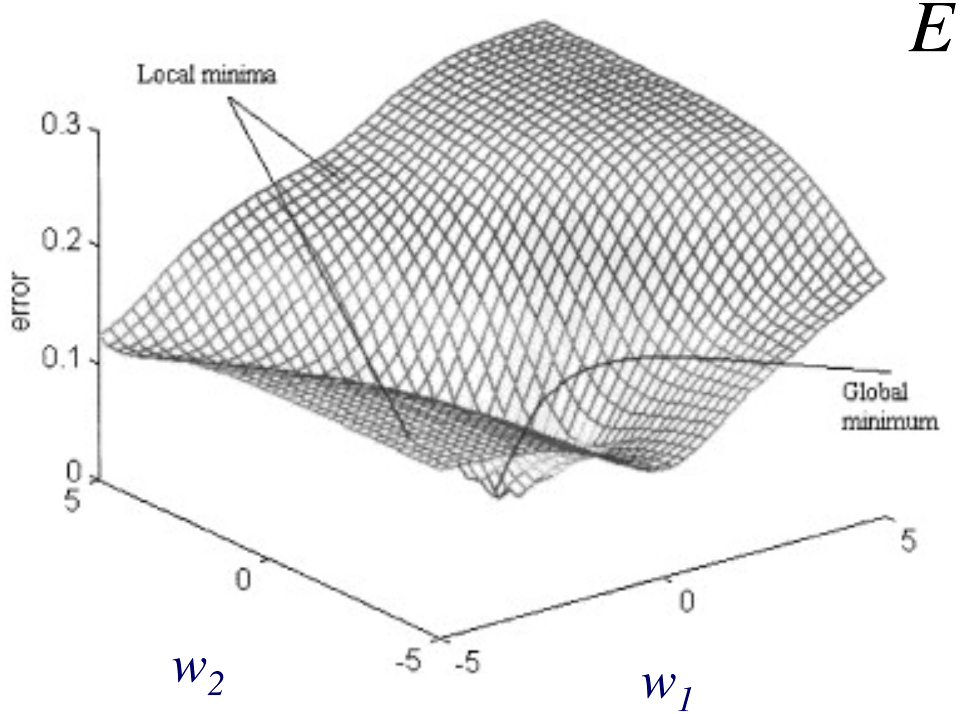



figure from Cho & Chow, *Neurocomputing* 1999

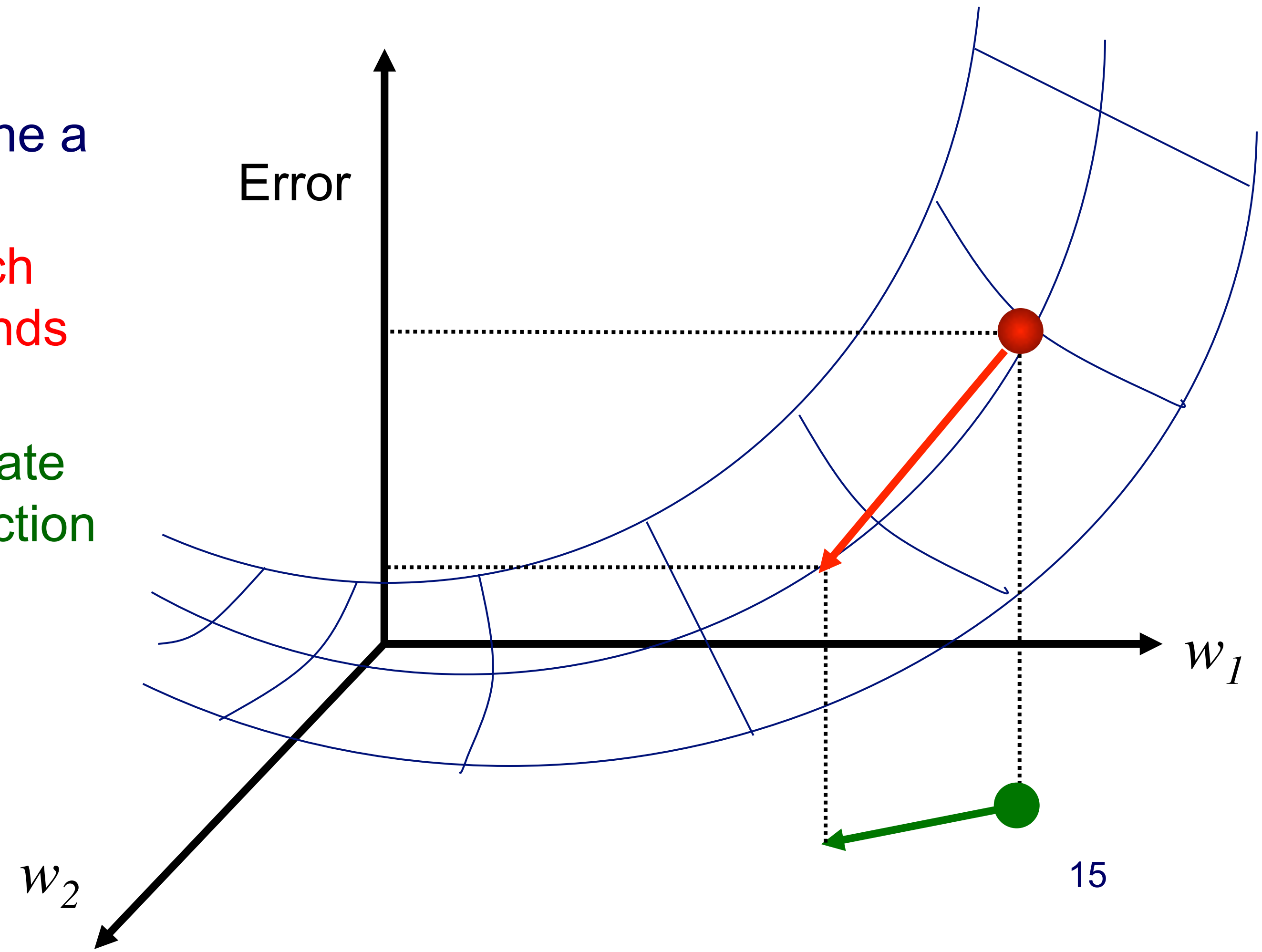
This error measure defines a surface over the hypothesis (i.e. weight) space

# Gradient descent in weight space

gradient descent is an iterative process aimed at finding a minimum in the error surface

on each iteration

- current weights define a point in this space
- find direction in which error surface descends most steeply
- take a step (i.e. update weights) in that direction



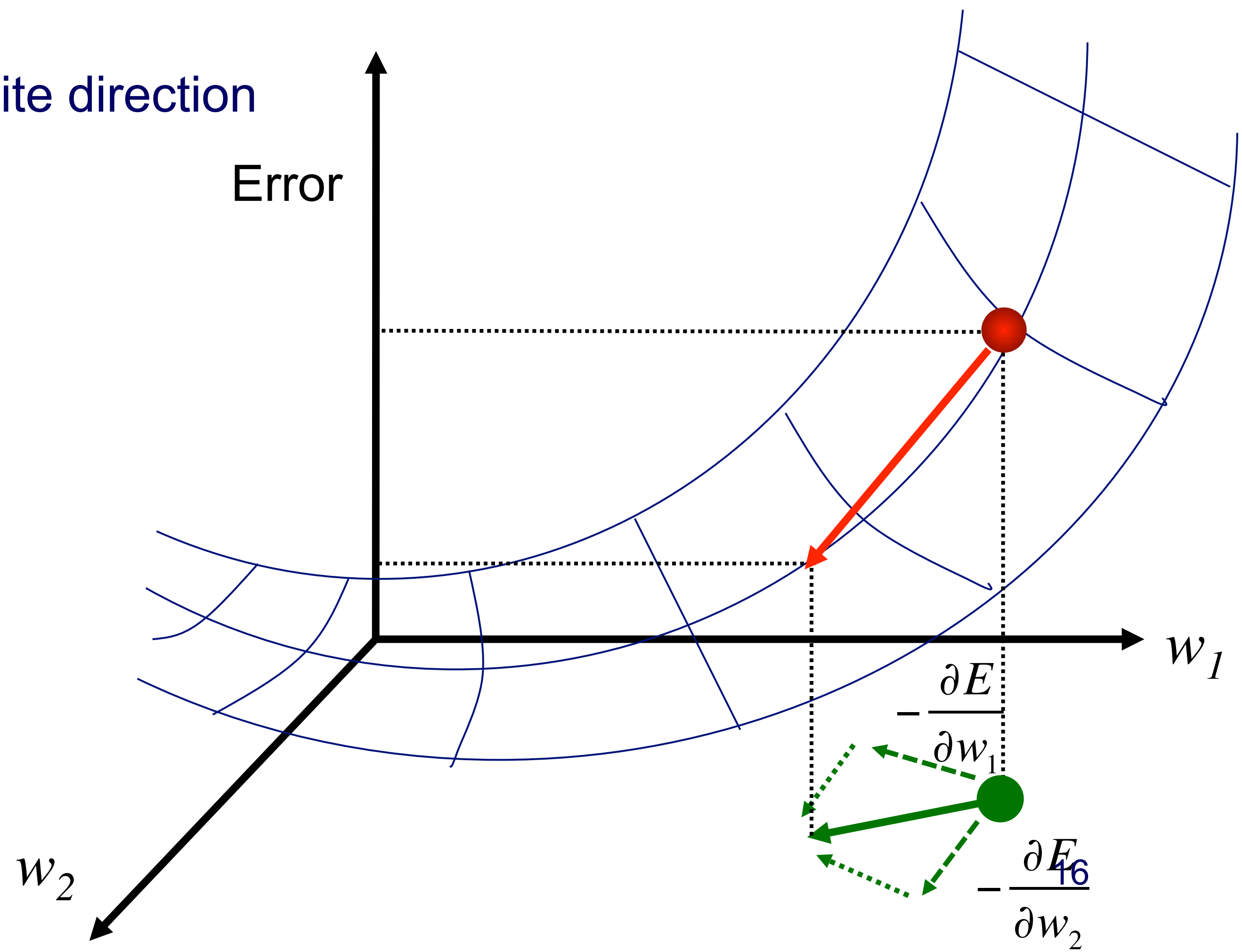
# Gradient descent in weight space

calculate the gradient of $E$: $\nabla E(\mathbf{w}) = \left[ \frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n} \right]$

take a step in the opposite direction

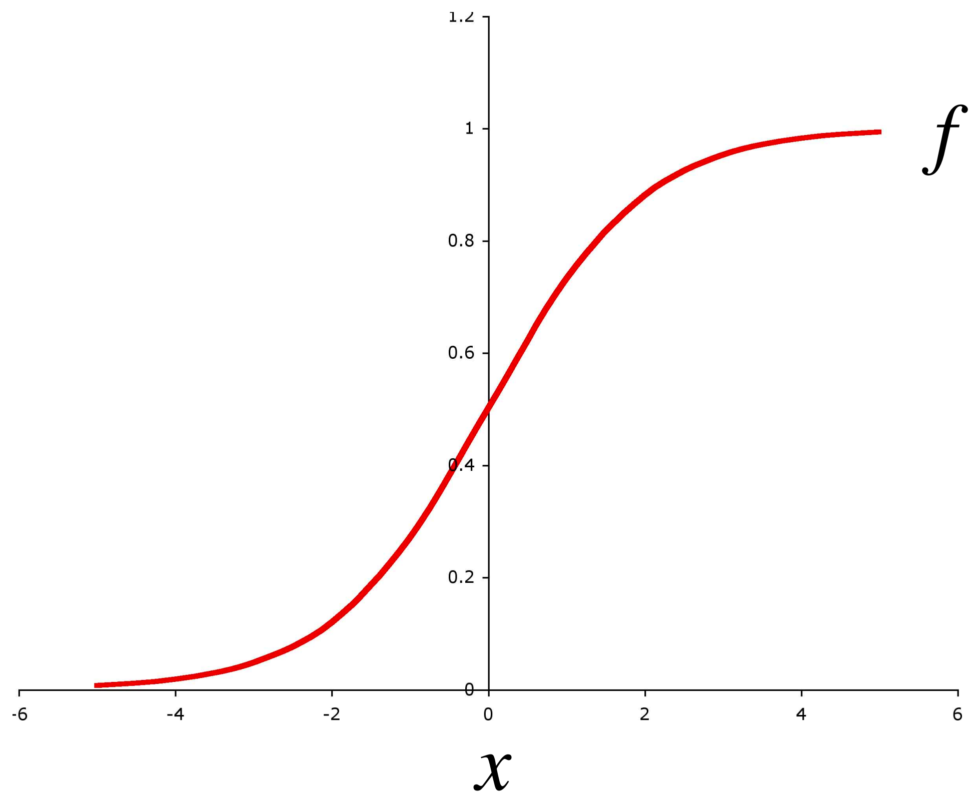
$$\Delta \mathbf{w} = -\eta \, \nabla E(\mathbf{w})$$

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



# The sigmoid function

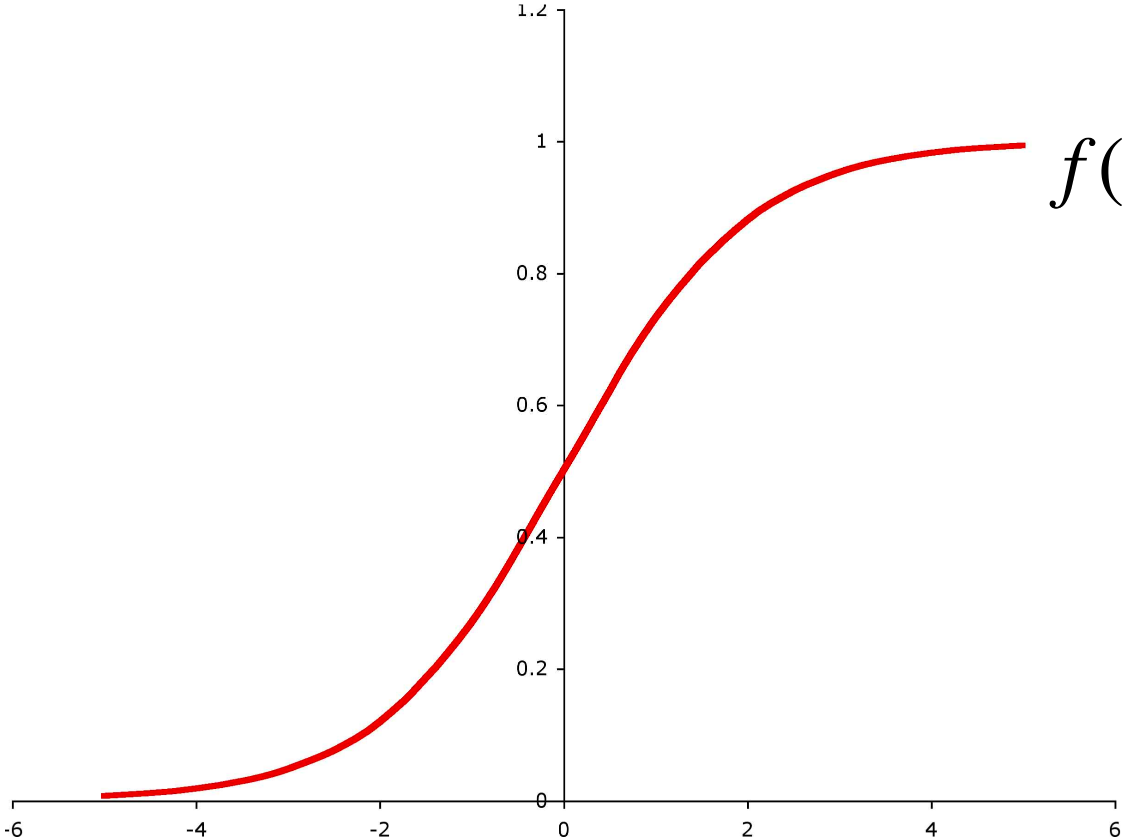
- to be able to differentiate $E$ with respect to $w_i$ , our network must represent a continuous function
- to do this, we use *sigmoid functions* instead of threshold functions in our hidden and output units




$$f(x) = \frac{1}{1 + e^{-x}}$$

# The sigmoid function

for the case of a single-layer network




$$f(x) = \frac{1}{1 + e^{-\left(w_0 + \sum_{i=1}^{n} w_i x_i\right)}}$$

$$w_0 + \sum_{i=1}^{n} w_i x_i$$

# *Batch* neural network training

**given**: network structure and a training set $D = \left\{ (\boldsymbol{x}^{(1)}, y^{(1)}) \ldots (\boldsymbol{x}^{(m)}, y^{(m)}) \right\}$

initialize all weights in $\boldsymbol{w}$ to small random numbers

until stopping criteria met do

initialize the error $E(\boldsymbol{w}) = 0$

for each $(\boldsymbol{x}^{(d)}, y^{(d)})$ in the training set

input $\boldsymbol{x}^{(d)}$ to the network and compute output $o^{(d)}$

increment the error $E(\boldsymbol{w}) = E(\boldsymbol{w}) + \frac{1}{2}\left(y^{(d)} - o^{(d)}\right)^2$

calculate the gradient

$$\nabla E(\boldsymbol{w}) = \left[ \frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n} \right]$$

update the weights

$$\Delta \boldsymbol{w} = -\eta \, \nabla E\left(\boldsymbol{w}\right)$$

# Online vs. batch training

- Standard gradient descent (batch training): calculates error gradient for the entire training set, before taking a step in weight space

- *Stochastic gradient descent* (online training): calculates error gradient for a single instance, then takes a step in weight space
  - much faster convergence
  - less susceptible to local minima

# *Online* neural network training (stochastic gradient descent)

**given**: network structure and a training set $D = \left\{ (\boldsymbol{x}^{(1)}, y^{(1)}) \ldots (\boldsymbol{x}^{(m)}, y^{(m)}) \right\}$

initialize all weights in $\boldsymbol{w}$ to small random numbers

until stopping criteria met do

for each $(\boldsymbol{x}^{(d)}, y^{(d)})$ in the training set

input $\boldsymbol{x}^{(d)}$ to the network and compute output $o^{(d)}$

calculate the error $E(\boldsymbol{w}) = \frac{1}{2}\left(y^{(d)} - o^{(d)}\right)^2$

calculate the gradient

$$\nabla E(\boldsymbol{w}) = \left[ \frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n} \right]$$

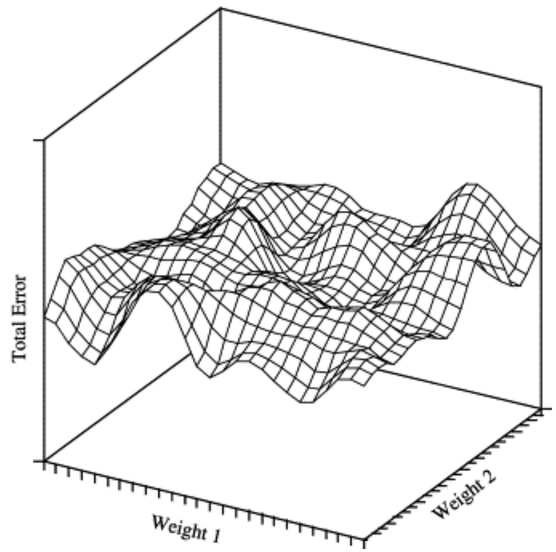
update the weights

$$\Delta \boldsymbol{w} = -\eta \, \nabla E\left(\boldsymbol{w}\right)$$

# Convergence of gradient descent

- gradient descent will converge to a minimum in the error function

- for a multi-layer network, this may be a *local minimum* (i.e. there may be a “better” solution elsewhere in weight space)




- for a single-layer network, this will be a global minimum (i.e. gradient descent will find the “best” solution)

# Taking derivatives in neural nets

recall the chain rule from calculus

$$y = f(u)$$

$$u = g(x)$$

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u}\frac{\partial u}{\partial x}$$

we'll make use of this as follows

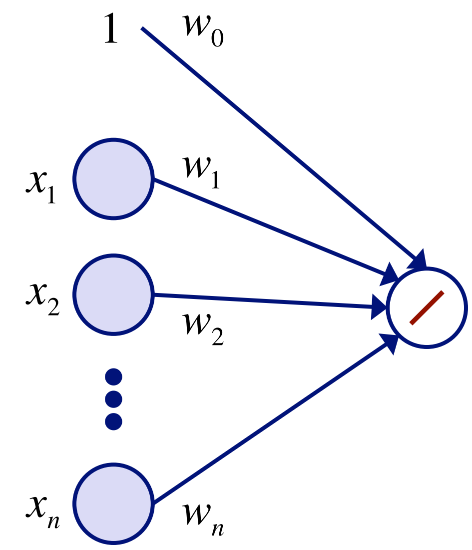
$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial o}\frac{\partial o}{\partial net}\frac{\partial net}{\partial w_i}$$

$$net = w_0 + \sum_{i=1}^{n} w_i x_i$$

# Gradient descent: simple case

Consider a simple case of a network with one linear output unit and no hidden units:

$$o^{(d)} = w_0 + \sum_{i=1}^{n} w_i x_i^{(d)}$$




let's learn $w_i$'s that minimize squared error

$$E(\boldsymbol{w}) = \frac{1}{2} \sum_{d \in D} \left( y^{(d)} - o^{(d)} \right)^2$$

batch case

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

online case

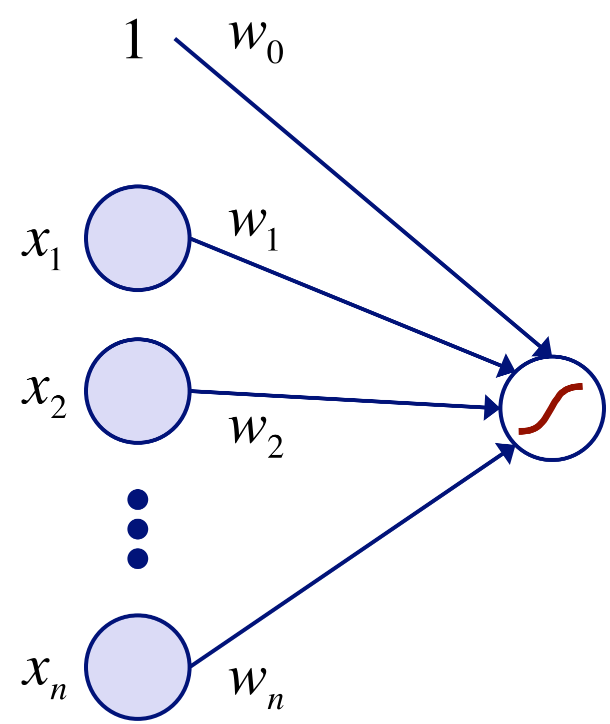
$$\frac{\partial E^{(d)}}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \left( y^{(d)} - o^{(d)} \right)^2$$

# Gradient descent with a sigmoid

Now let's consider the case in which we have a sigmoid output unit and no hidden units:

$$net^{(d)} = w_0 + \sum_{i=1}^{n} w_i x_i^{(d)}$$

$$o^{(d)} = \frac{1}{1 + e^{-net^{(d)}}}$$




useful property:

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

# Backpropagation

- now we've covered how to do gradient descent for single-layer networks with
  - linear output units
  - sigmoid output units

- how can we calculate $\frac{\partial E}{\partial w_i}$ for every weight in a multilayer network?

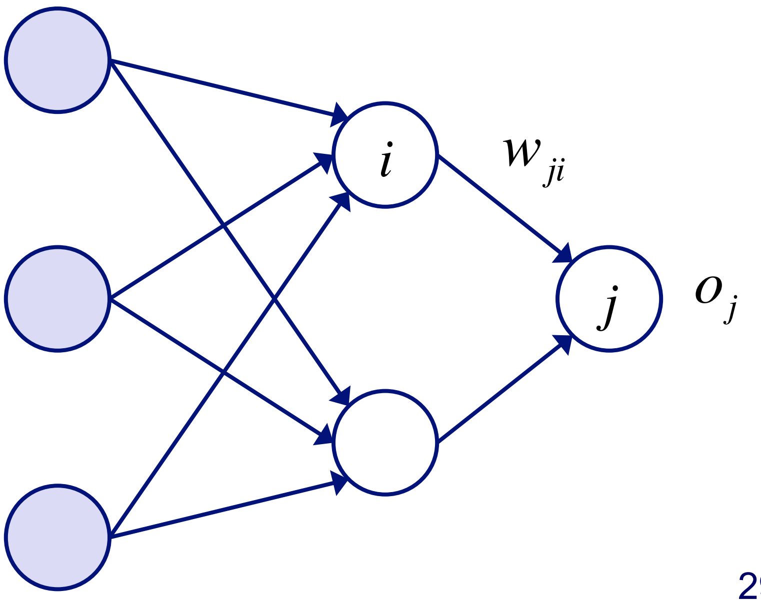
  ➔ backpropagate errors from the output units to the hidden units

# Backpropagation notation

let's consider the online case, but drop the $^{(d)}$ superscripts for simplicity

we'll use

- subscripts on $y, o, net$ to indicate which unit they refer to
- subscripts to indicate the unit a weight emanates from and goes to



# Backpropagation

each weight is changed by $$\Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}}$$

$$= -\eta \frac{\partial E}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}}$$

$$= \eta \ \delta_j \ o_i$$

$x_i$ if $i$ is an input unit

where $$\delta_j = -\frac{\partial E}{\partial net_j}$$

# Backpropagation

each weight is changed by $$\Delta w_{ji} = \eta \; \delta_j \; o_i$$

where $$\delta_j = -\frac{\partial E}{\partial net_j}$$

$$\delta_j = o_j(1 - o_j)(y_j - o_j)$$ if $j$ is an output unit

same as single-layer net with sigmoid output

$$\delta_j = o_j(1 - o_j)\underbrace{\sum_k \delta_k w_{kj}}_{\text{sum of backpropagated contributions to error}}$$ if $j$ is a hidden unit

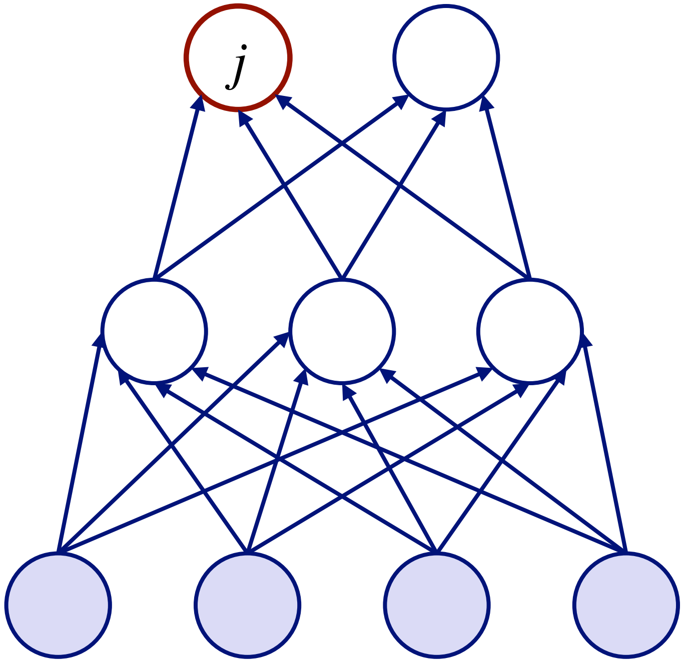
# Backpropagation illustrated

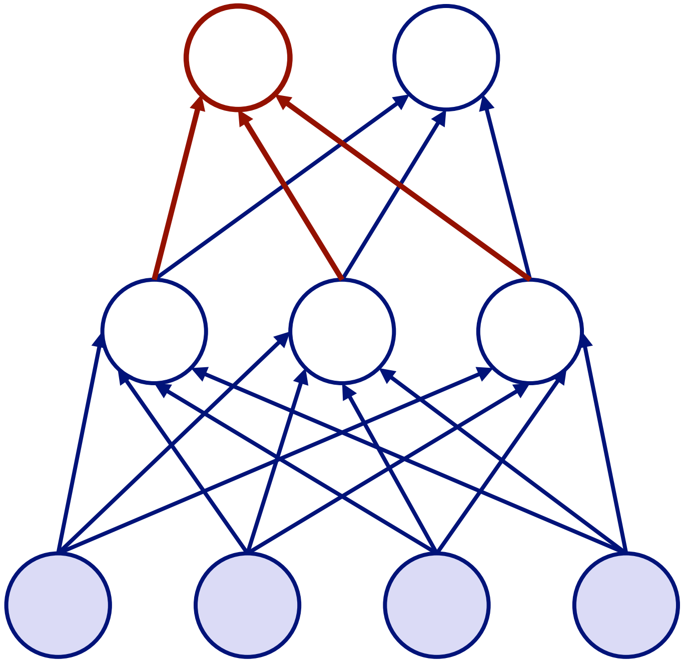
1. calculate error of output units

$$\delta_j = o_j(1 - o_j)(y_j - o_j)$$

2. determine updates for weights going to output units

$$\Delta w_{ji} = \eta \; \delta_j \; o_i$$

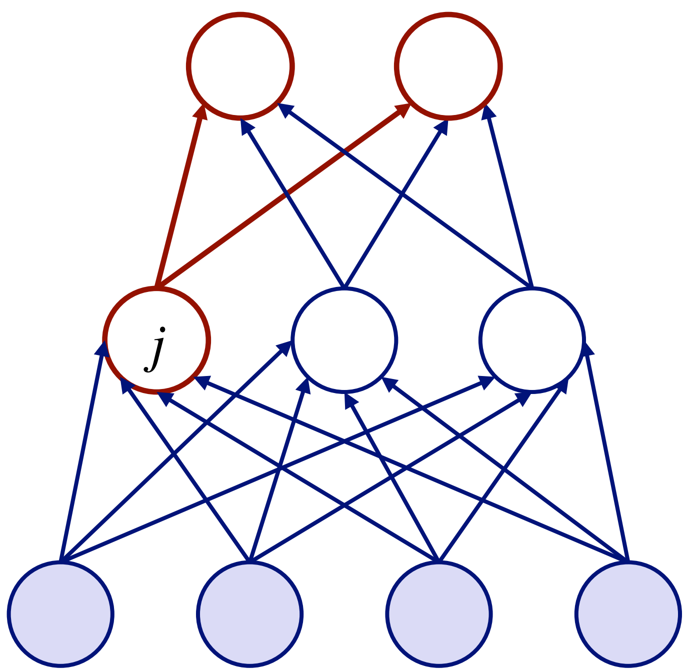


# Backpropagation illustrated

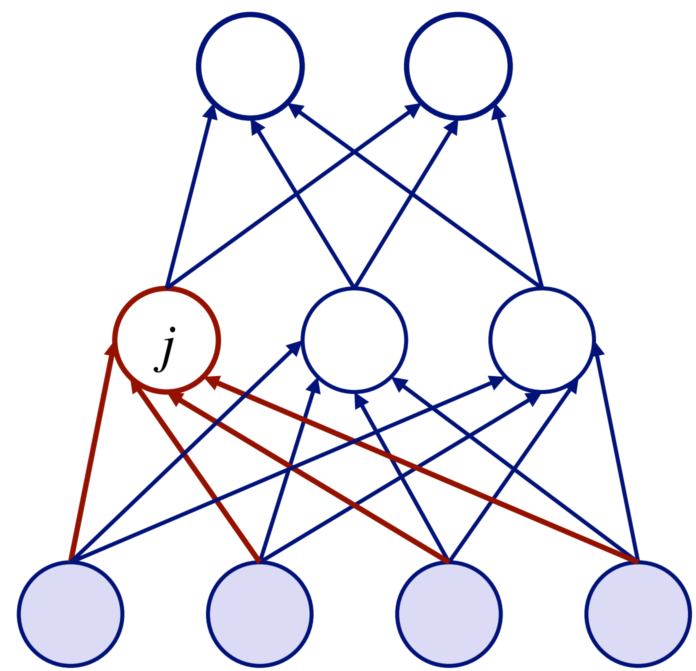
3. calculate error for hidden units

$$\delta_j = o_j(1 - o_j)\sum_k \delta_k w_{kj}$$




4. determine updates for weights to hidden units using hidden-unit errors

$$\Delta w_{ji} = \eta\ \delta_j\ o_i$$

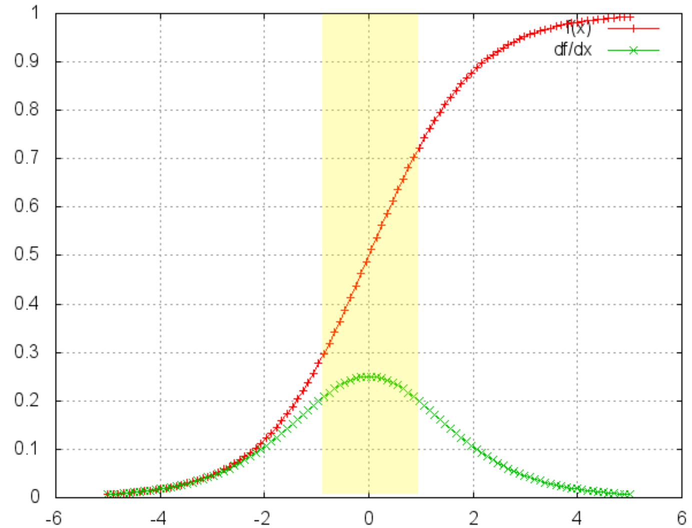


# Neural network jargon

- *activation*: the output value of a hidden or output unit
- *epoch*: one pass through the training instances during gradient descent
- *transfer function*: the function used to compute the output of a hidden/ output unit from the net input

# Initializing weights

- Weights should be initialized to
  - small values so that the sigmoid activations are in the range where the derivative is large (learning will be quicker)

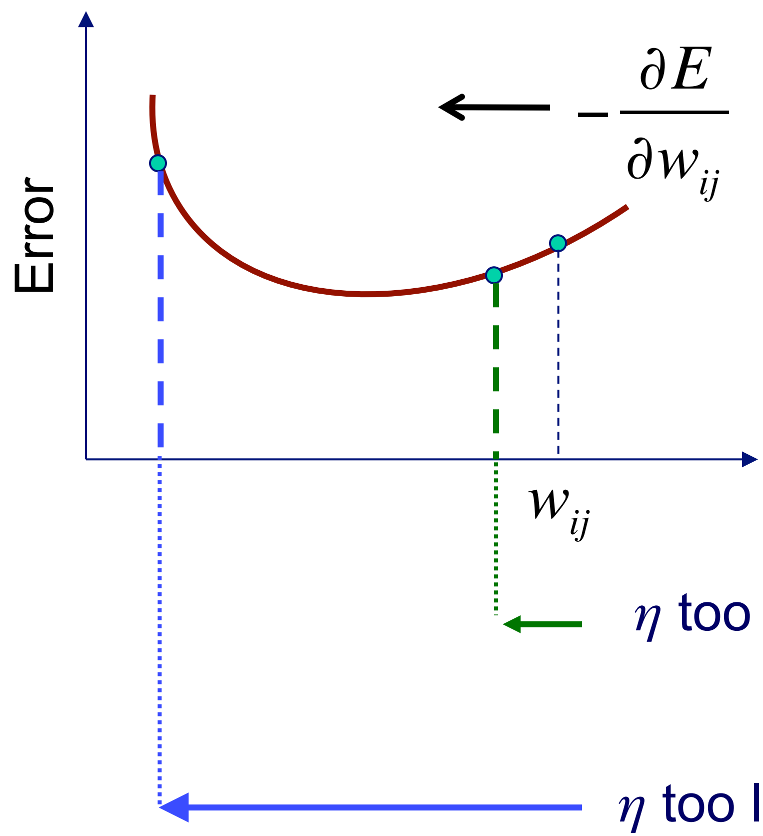



  - random values to ensure symmetry breaking (i.e. if all weights are the same, the hidden units will all represent the same thing)

- typical initial weight range [-0.01, 0.01]

# Setting the learning rate

convergence depends on having an appropriate learning rate



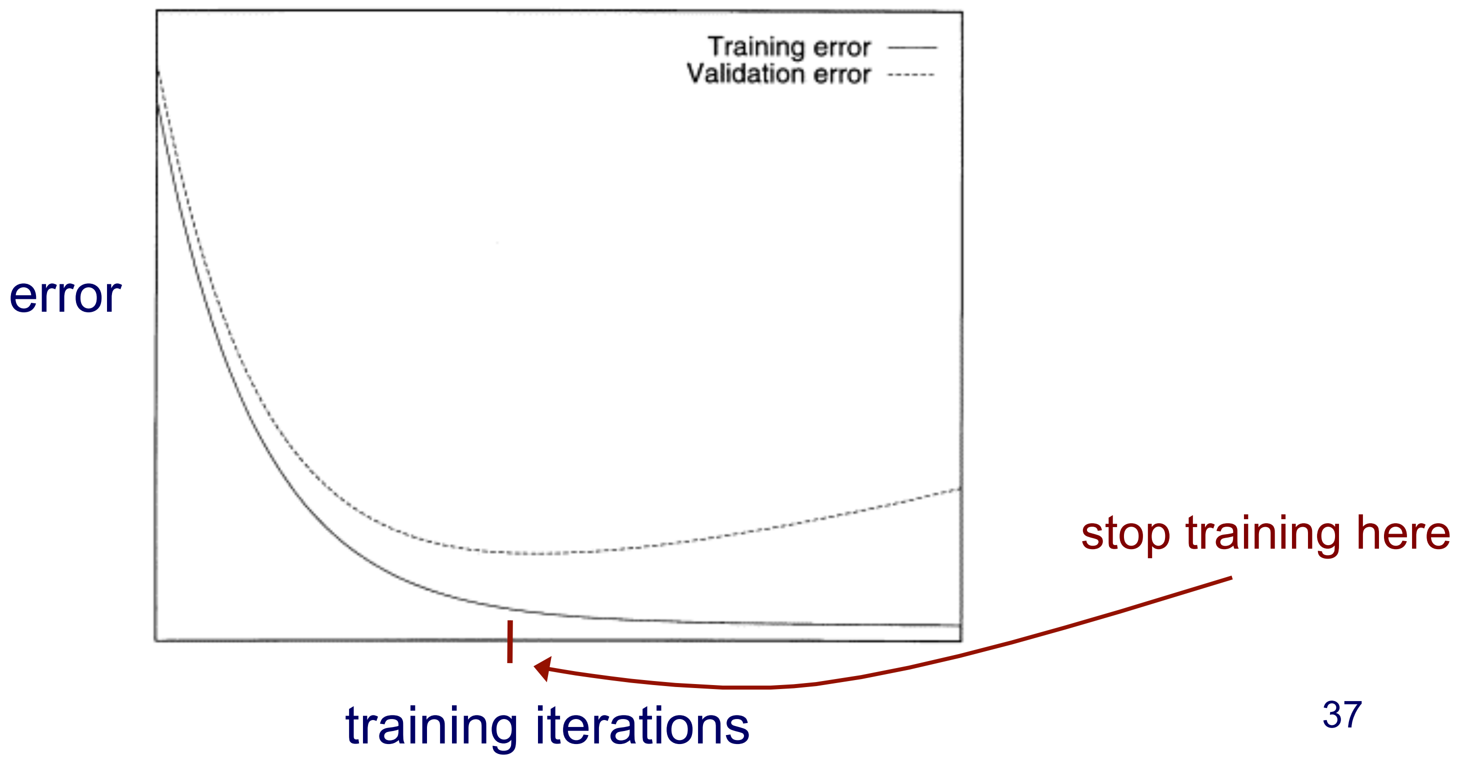

$\eta$ too small (error goes down a little)

$\eta$ too large (error goes up)

# Stopping criteria

- conventional gradient descent: train until local minimum reached

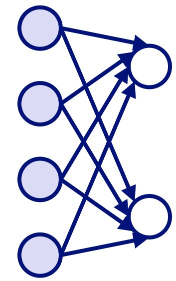
- empirically better approach: *early stopping*
  - use a validation set to monitor accuracy during training iterations
  - return the weights that result in minimum validation-set error



# Input (feature) encoding for neural networks

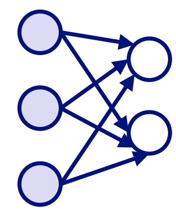
nominal features are usually represented using a *1-of-k* encoding

$$A = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad C = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad G = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad T = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$



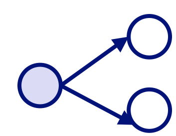
ordinal features can be represented using a *thermometer* encoding

$$\text{small} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad \text{medium} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \quad \text{large} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$



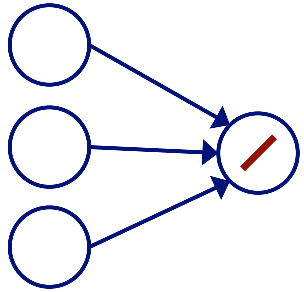
real-valued features can be represented using individual input units (we may want to scale/normalize them first though)

$$\text{precipitation} = \begin{bmatrix} 0.68 \end{bmatrix}$$

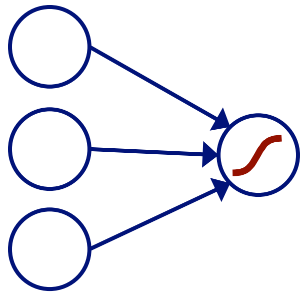


# Output encoding for neural networks

regression tasks usually use output units with linear transfer functions

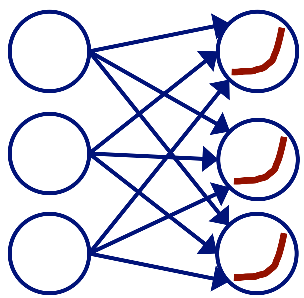


binary classification tasks usually use one sigmoid output unit



*k*-ary classification tasks usually use *k* sigmoid or *softmax* output units

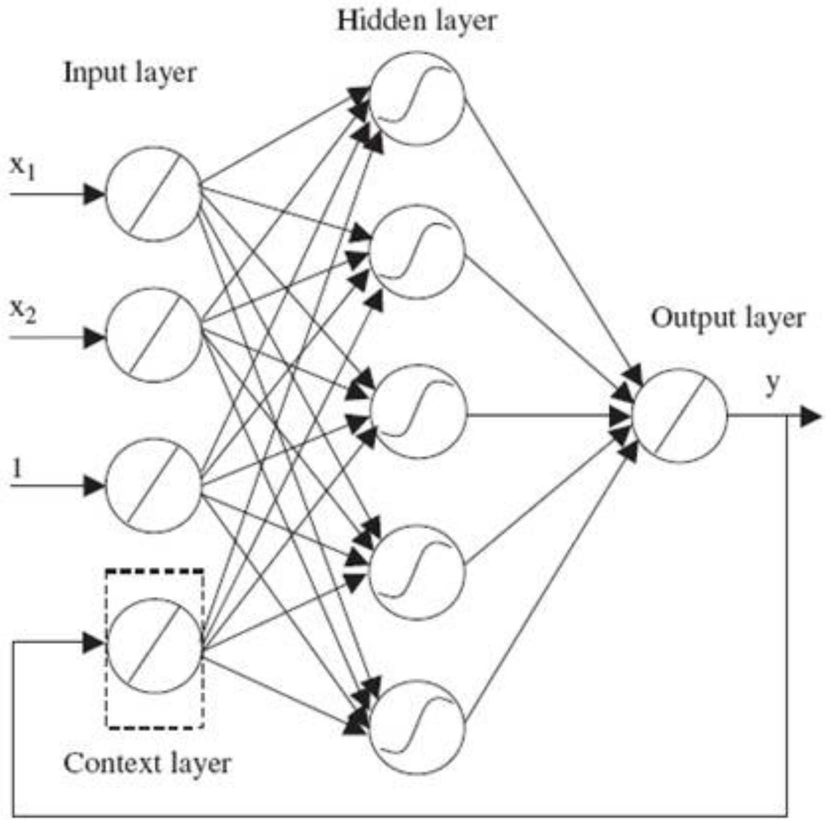
$$o_i = \frac{e^{net_i}}{\sum_{j \in outputs} e^{net_j}}$$



# Recurrent neural networks

recurrent networks are sometimes used for tasks that involve making sequences of predictions

- Elman networks: recurrent connections go from hidden units to inputs
- Jordan networks: recurrent connections go from output units to inputs



# Alternative approach to training **deep networks**

- use unsupervised learning to to find useful hidden unit representations



# Learning representations

- the feature representation provided is often the most significant factor in how well a learning system works

- an appealing aspect of multilayer neural networks is that they are able to change the feature representation

- can think of the nodes in the hidden layer as new features constructed from the original features in the input layer

- consider having more levels of constructed features, e.g., pixels -> edges -> shapes -> faces or other objects

# Competing intuitions

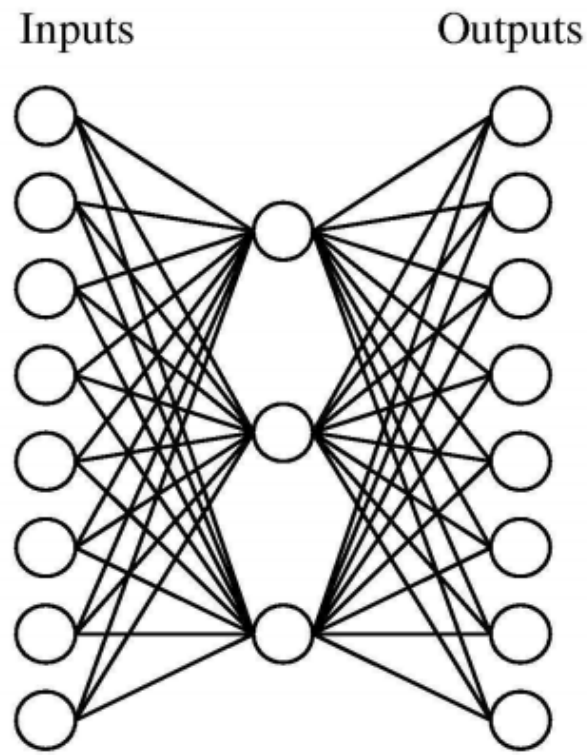
- Only need a 2-layer network (input, hidden layer, output)
  - Representation Theorem (1989): Using sigmoid activation functions (more recently generalized to others as well), can represent any continuous function with a single hidden layer
  - Empirically, adding more hidden layers does not improve accuracy, and it often degrades accuracy, when training by standard backpropagation

- Deeper networks are better
  - More efficient representationally, e.g., can represent *n*-variable parity function with polynomially many (in *n*) nodes using multiple hidden layers, but need exponentially many (in *n*) nodes when limited to a single hidden layer
  - More structure, should be able to construct more interesting derived features

# The role of hidden units

- Hidden units transform the input space into a new space where perceptrons suffice
- They numerically represent “constructed” features
- Consider learning the target function using the network structure below:

| 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 |



# The role of hidden units

- In this task, hidden units learn a compressed numerical coding of the inputs/outputs

| 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 |

# How many hidden units should be used?

- conventional wisdom in the early days of neural nets: prefer small networks because fewer parameters (i.e. weights & biases) will be less likely to overfit
- somewhat more recent wisdom: if early stopping is used, larger networks often behave as if they have fewer "effective" hidden units, and find better solutions

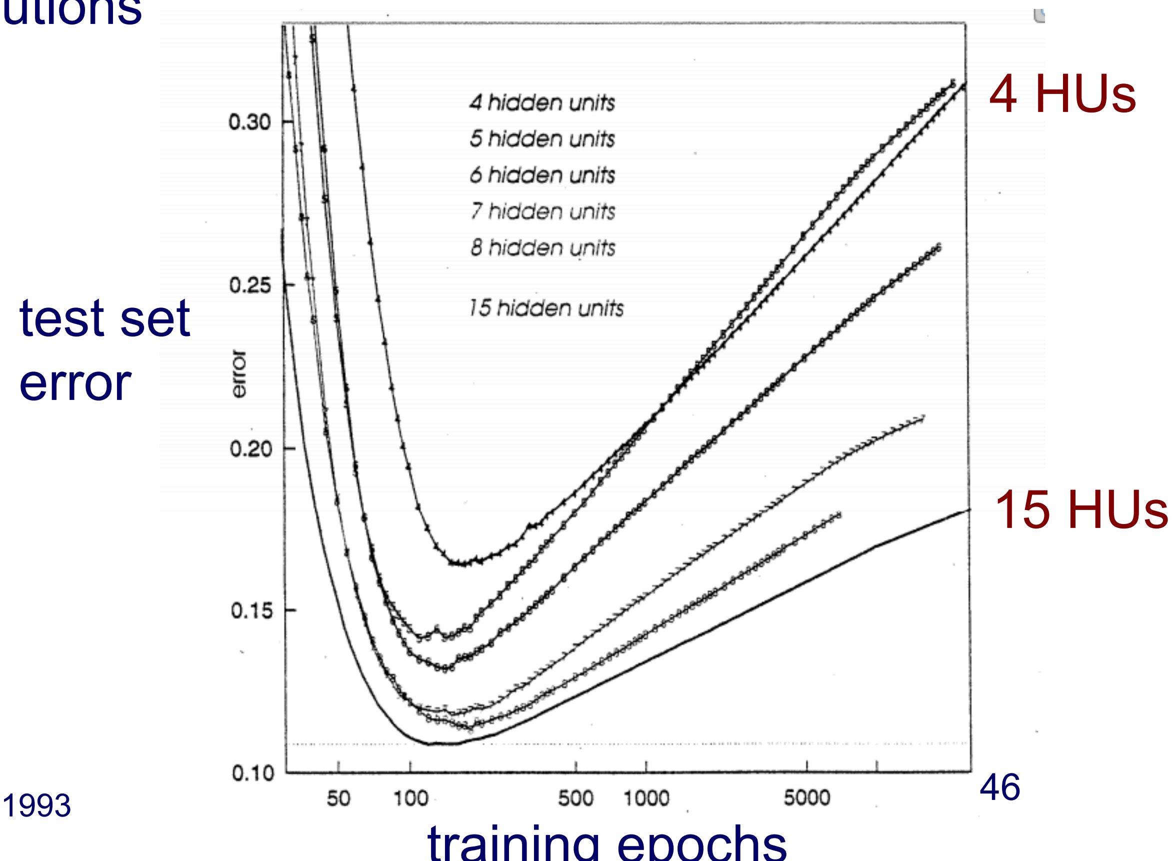



Figure from Weigend, *Proc. of the CMSS* 1993

# Another way to avoid overfitting

- Allow many hidden units but force each hidden unit to output mostly zeroes: tend to meaningful concepts

- Gradient descent solves an optimization problem—add a "regularizing" term to the objective function

- Let **X** be vector of random variables, one for each hidden unit, giving average output of unit over data set. Let target distribution *s* have variables independent with low probability of outputting one (say 0.1), and let $\hat{s}$ be empirical distribution in the data set. Add to the backpropagation target function (that minimizes δ's) a penalty of $KL(s(\mathbf{X}) || \hat{s}(\mathbf{X}))$

# Backpropagation with multiple hidden layers

- in principle, backpropagation can be used to train arbitrarily deep networks (i.e. with multiple hidden layers)
- in practice, this doesn't usually work well
  - there are likely to be lots of local minima
  - diffusion of gradients leads to slow training in lower layers
    - gradients are smaller, less pronounced at deeper levels
    - errors in credit assignment propagate as you go back