Neural Networks and Deep Learning

www.cs.wisc.edu/~dpage/cs760/

Goals for the lecture

you should understand the following concepts

- perceptrons
- the perceptron training rule
- linear separability
- hidden units
- multilayer neural networks
- gradient descent
- stochastic (online) gradient descent
- sigmoid function
- gradient descent with a linear output unit
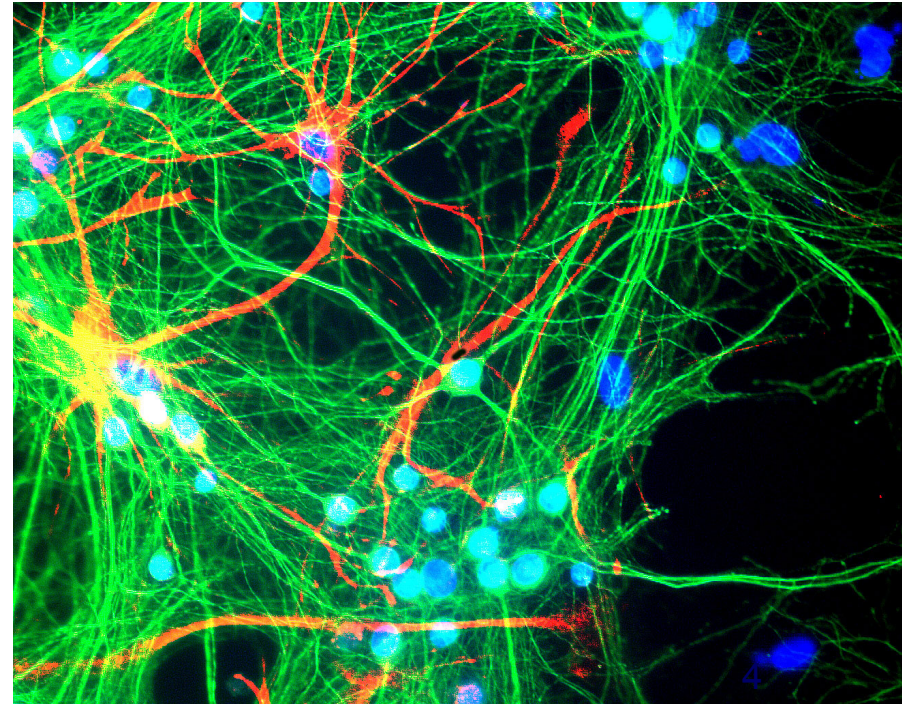- gradient descent with a sigmoid output unit
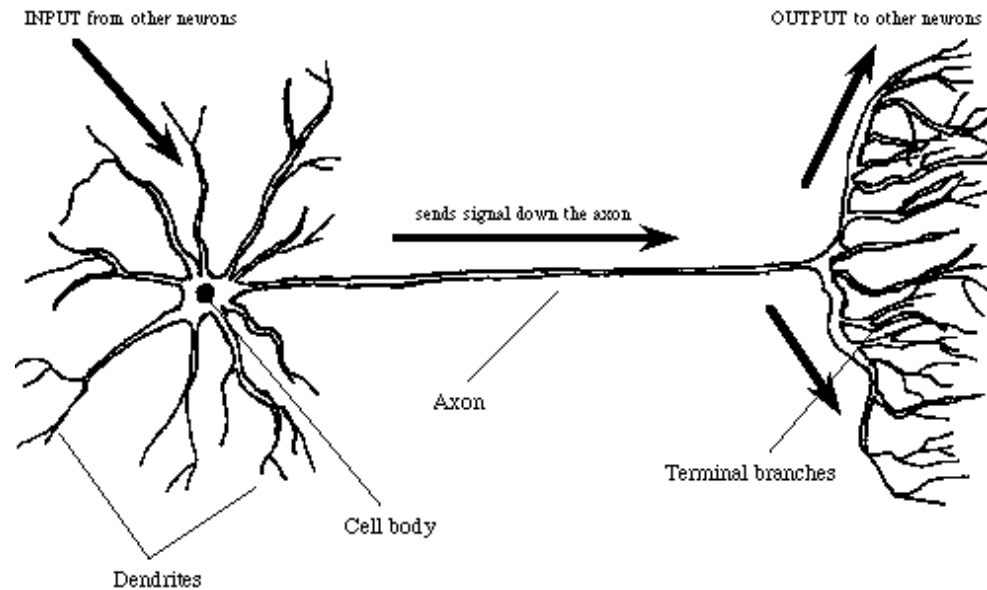- backpropagation

Goals for the lecture

you should understand the following concepts

- weight initialization
- early stopping
- the role of hidden units
- input encodings for neural networks
- output encodings
- recurrent neural networks
- autoencoders
- stacked autoencoders

Neural networks

- a.k.a. *artificial neural networks, connectionist models*
- inspired by interconnected neurons in biological systems
  - simple processing units
  - each unit receives a number of real-valued inputs
  - each unit produces a single real-valued output

INPUT from other neurons

OUTPUT to other neurons

sends signal down the axon

Axon

Terminal branches

Cell body

Dendrites

Perceptrons

[McCulloch & Pitts, 1943; Rosenblatt, 1959; Widrow & Hoff, 1960]

$$o = \begin{cases} 1 & \text{if } w_0 + \sum_{i=1}^{n} w_i x_i > 0 \\ 0 & \text{otherwise} \end{cases}$$
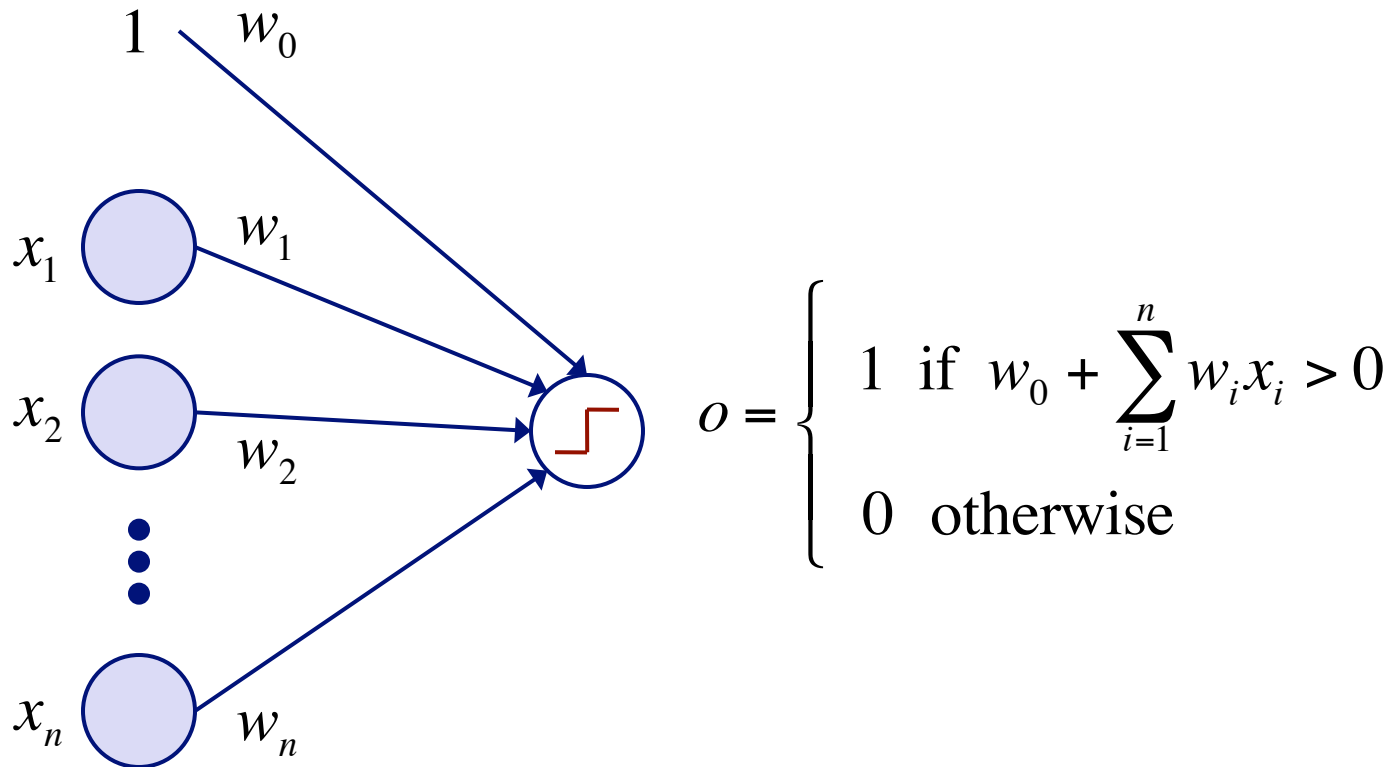
input units:
represent given x

output unit:
represents binary classification

Learning a perceptron: the perceptron training rule

1. randomly initialize weights

2. iterate through training instances until convergence

2a. calculate the output for the given instance

$$o = \begin{cases} 1 & \text{if } w_0 + \sum_{i=1}^{n} w_i x_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

2b. update each weight

$$\Delta w_i = \eta (y - o) x_i$$

η is *learning rate*; set to value << 1

$$w_i \leftarrow w_i + \Delta w_i$$

Representational power of perceptrons

perceptrons can represent only *linearly separable* concepts

$$o = \begin{cases} 1 & \text{if } w_0 + \sum_{i=1}^{n} w_i x_i > 0 \\ \\ 0 & \text{otherwise} \end{cases}$$
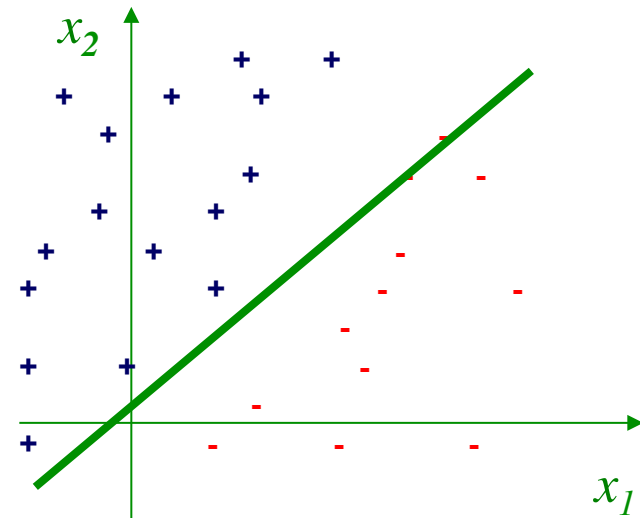
w

decision boundary given by:

$$1 \;\; \text{if} \;\; w_0 + w_1 x_1 + w_2 x_2 > 0$$
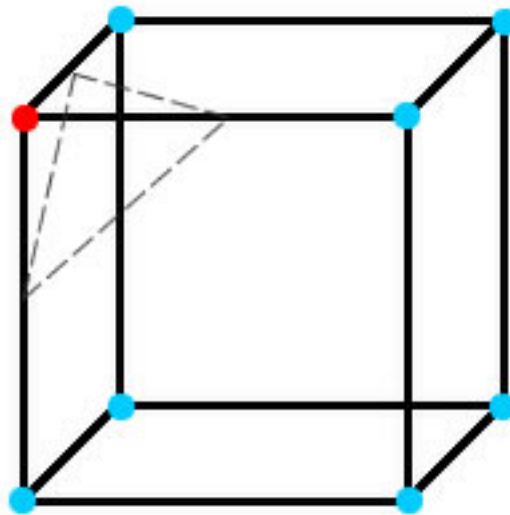
also write as: **wx** > 0

$$w_1 x_1 + w_2 x_2 = -w_0$$

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{w_0}{w_2}$$
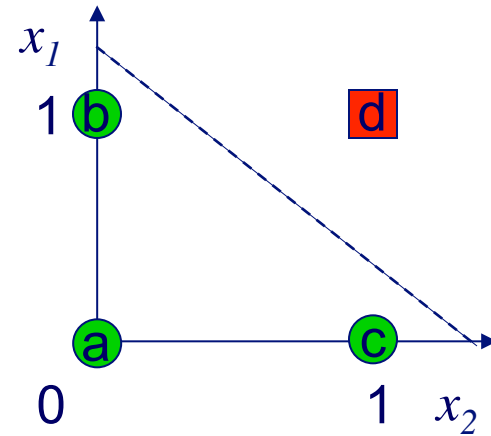
Representational power of perceptrons

- in previous example, feature space was 2D so decision boundary was a line
- in higher dimensions, decision boundary is a hyperplane
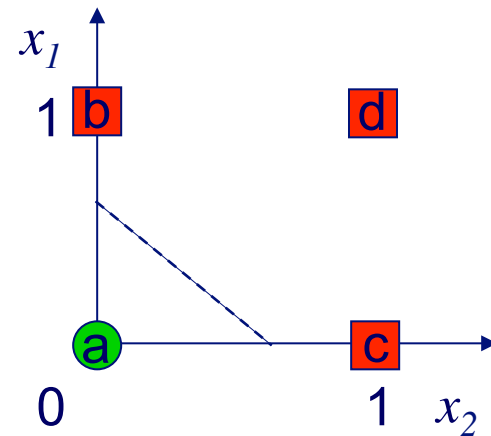
Some linearly separable functions

AND

	x_1	x_2	y
a	0	0	0
b	0	1	0
c	1	0	0
d	1	1	1

OR

	x_1	x_2	y
a	0	0	0
b	0	1	1
c	1	0	1
d	1	1	1

XOR is not linearly separable

	x_1	x_2	y
a	0	0	0
b	0	1	1
c	1	0	1
d	1	1	0
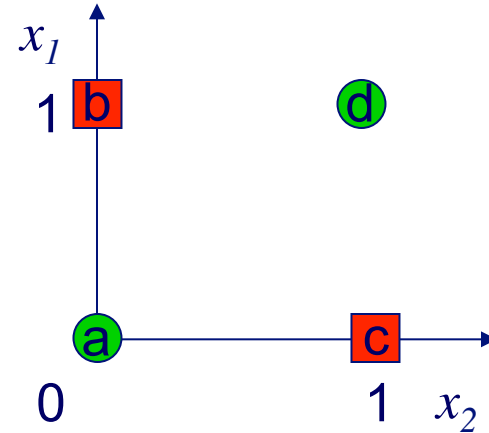
a multilayer perceptron can represent XOR
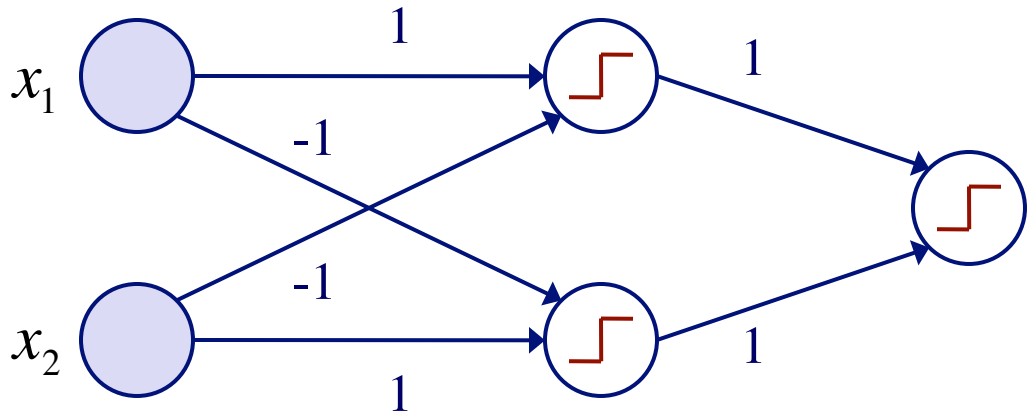
assume $w_0 = 0$ for all nodes

10

Example multilayer neural network

head   hid   ↑   who'd   hood
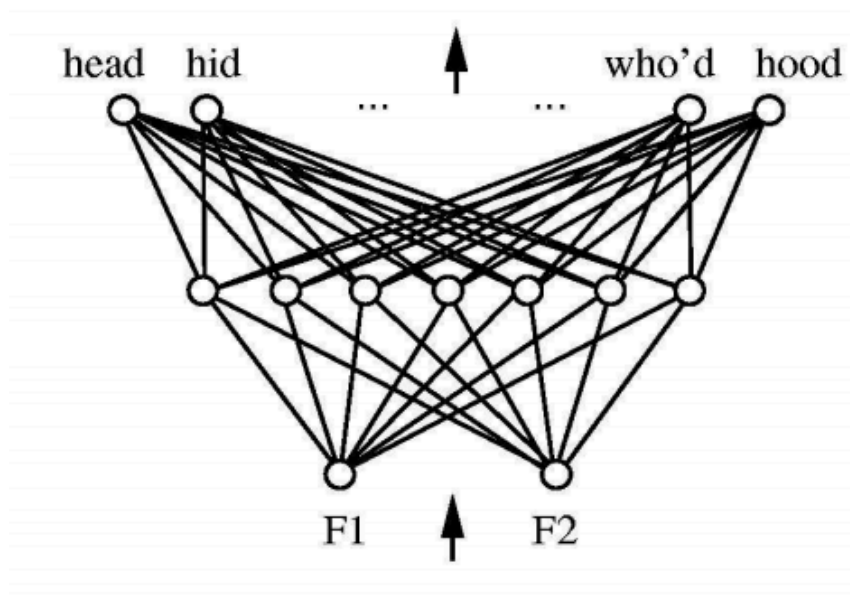
output units

hidden units

input units

F1   ↑   F2

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"

11

Decision regions of a multilayer neural network
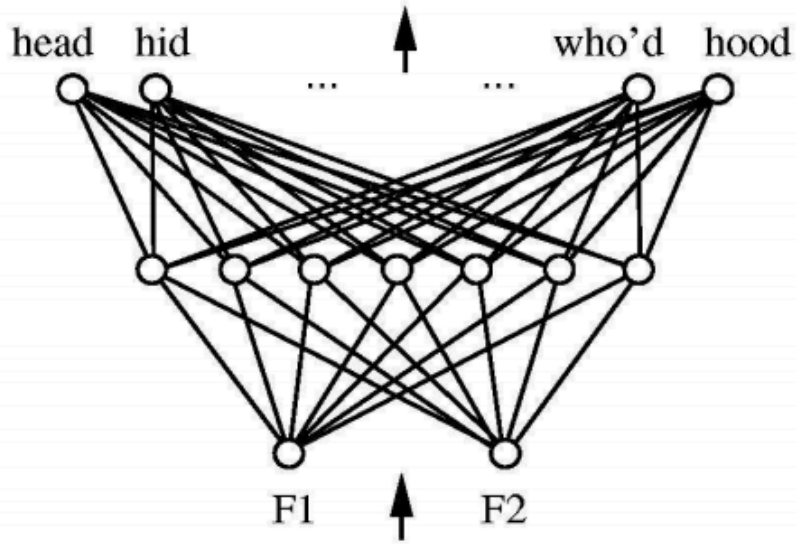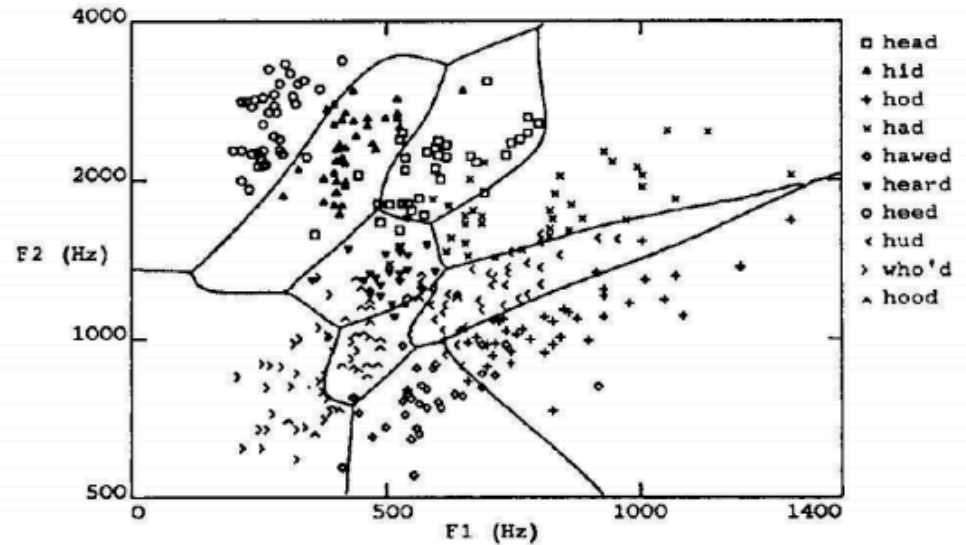

figure from Huang & Lippmann, *NIPS* 1988

input: two features from spectral analysis of a spoken sound

output: vowel sound occurring in the context "h__d"

12

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
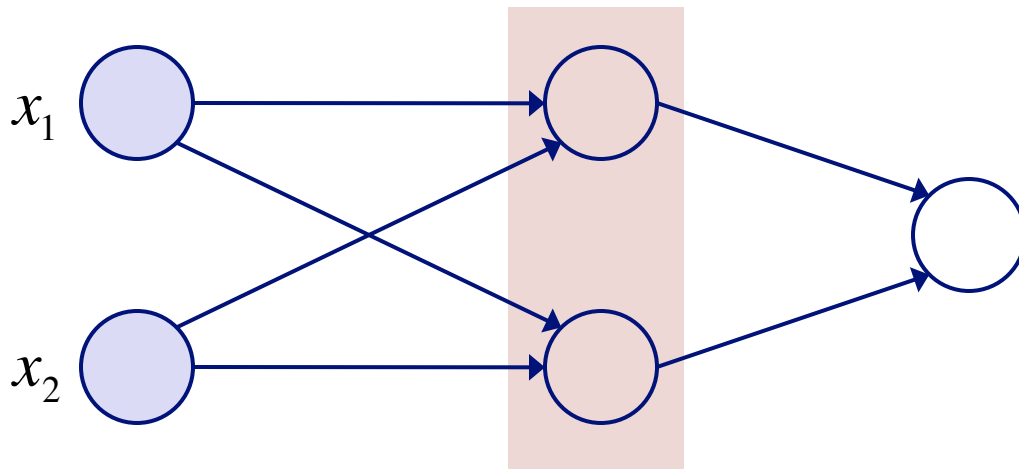
how to determine error signal for *hidden units?*

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

$$E(\boldsymbol{w}) = \frac{1}{2} \sum_{d \in D} \left(y^{(d)} - o^{(d)} \right)^2$$
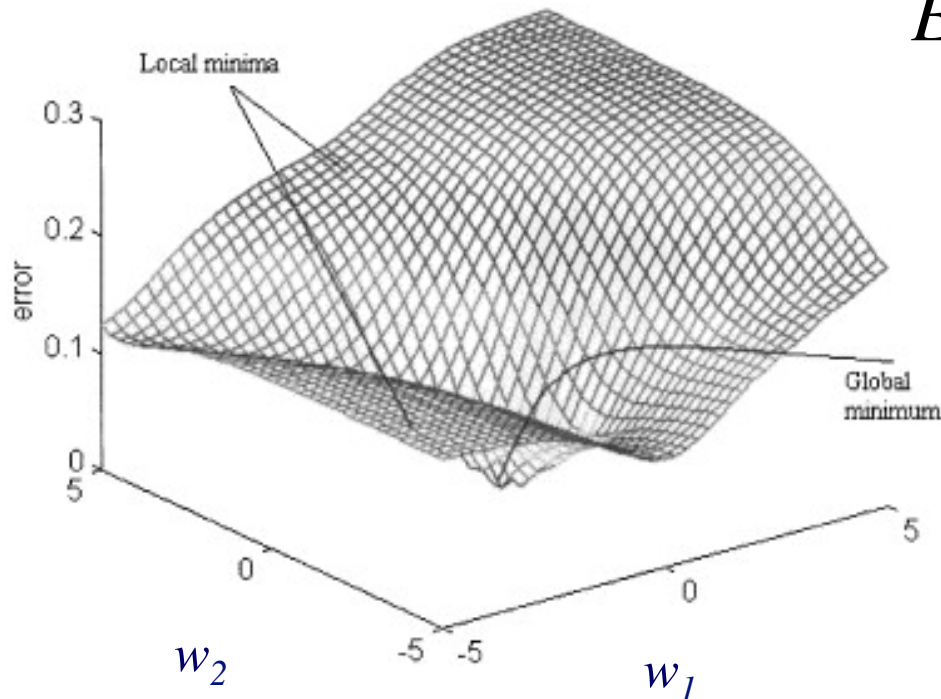
figure from Cho & Chow, *Neurocomputing* 1999

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

Error

w_1

w_2
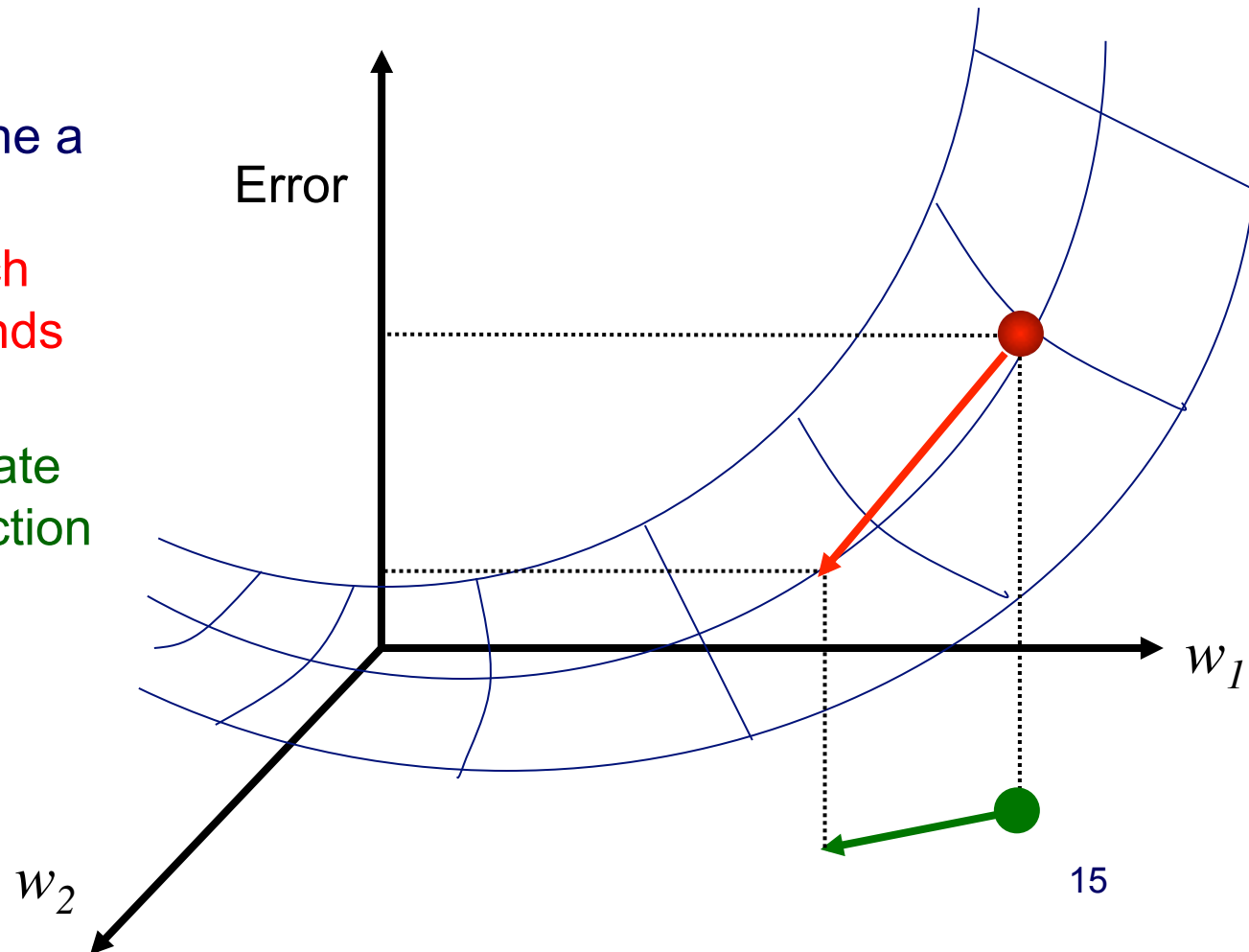
Gradient descent in weight space

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

take a step in the opposite direction

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

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

Error

w_1

w_2

$-\frac{\partial E}{\partial w_1}$

$-\frac{\partial E}{\partial w_2}$

16
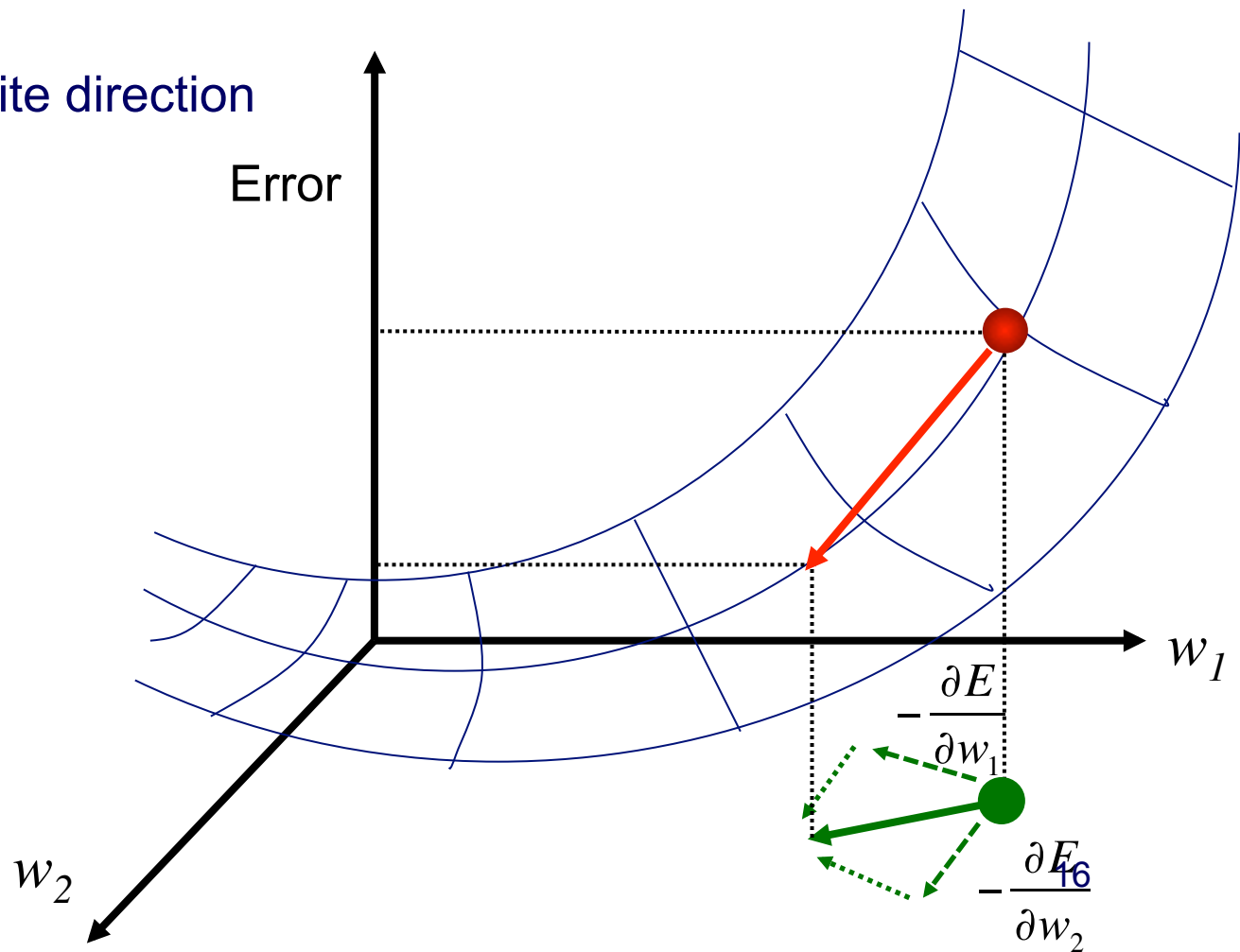
The sigmoid function

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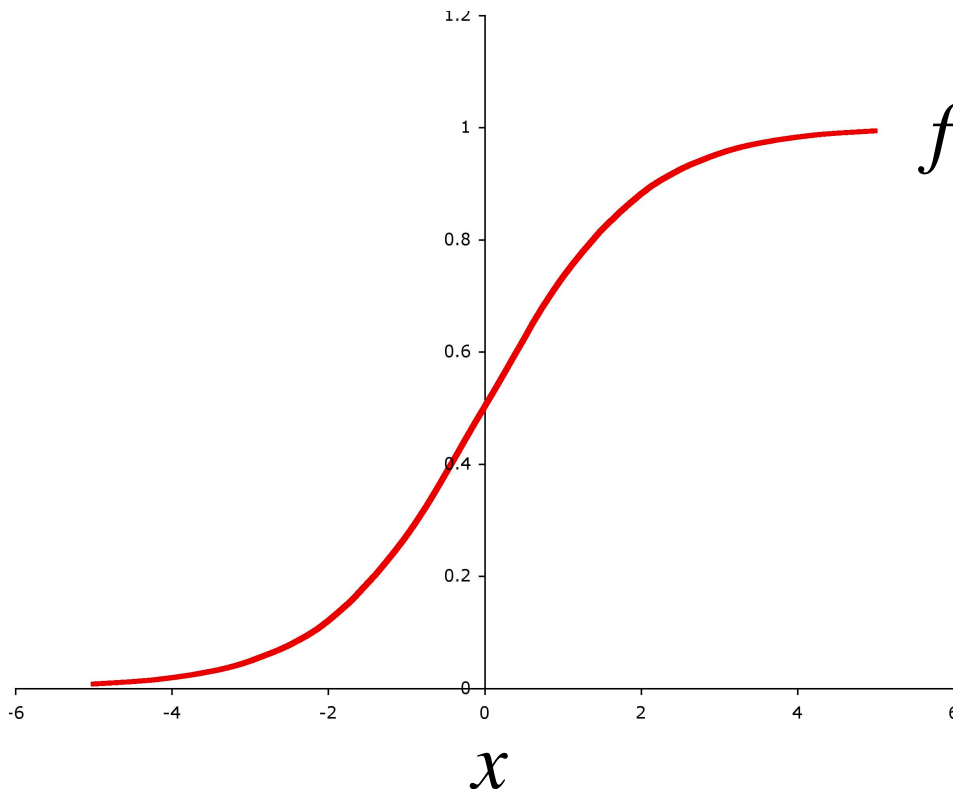
$$f(x) = \frac{1}{1 + e^{-x}}$$

x

The sigmoid function

for the case of a single-layer network
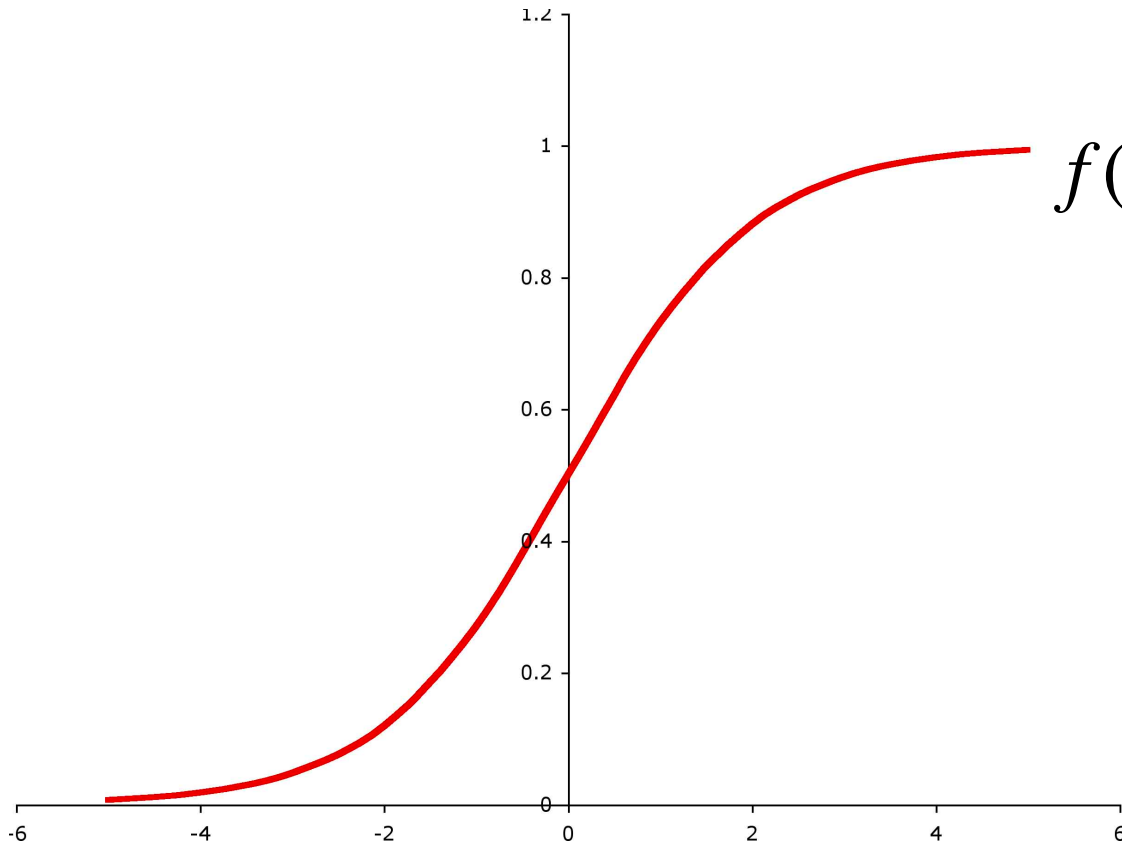
$$f(x) = \frac{1}{1 + e^{-\left(w_0 + \sum\limits_{i=1}^{n} w_i x_i\right)}}$$

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

Batch neural network training

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

initialize all weights in w to small random numbers

until stopping criteria met do

      initialize the error $E(w) = 0$

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

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

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

      calculate the gradient

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

      update the weights

$$\Delta w = -\eta \ \nabla E(w)$$

Online vs. batch training

- Standard gradient descent (batch training): calculates error gradient for the <u>entire training set</u>, 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 $E(\boldsymbol{w}) = \dfrac{1}{2} \left(y^{(d)} - o^{(d)} \right)^2$

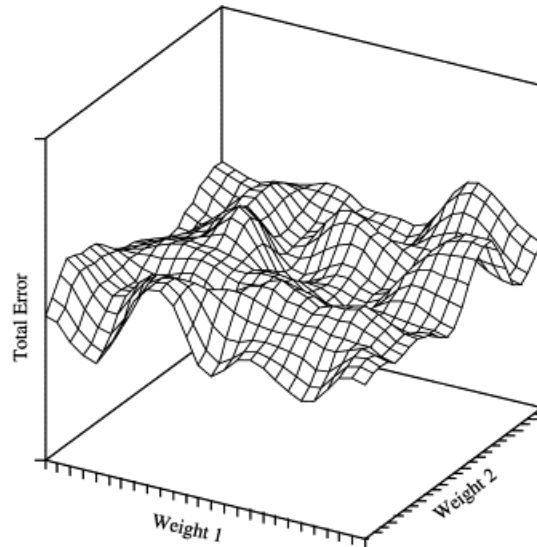   calculate the gradient

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

            update the weights

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

Convergence of gradient descent

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

- for a <u>multi-layer network</u>, this may be a *local minimum* (i.e. there may be a "better" solution elsewhere in weight space)

- for a <u>single-layer network</u>, 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

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

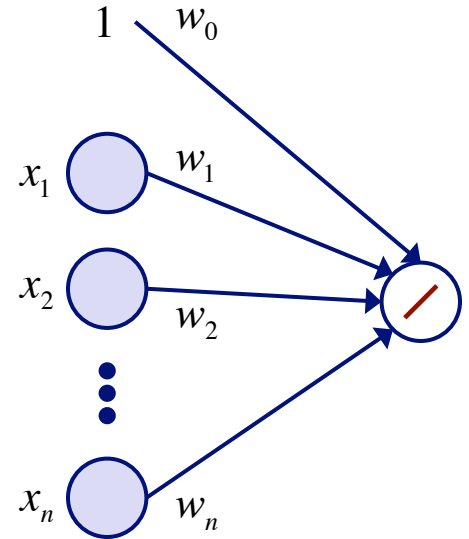$$net = w_0 + \sum_{i=1}^{n} w_i x_i$$

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

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

Stochastic gradient descent: simple case

let's focus on the online case (stochastic gradient descent):

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

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

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

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

$$= -\left(y^{(d)} - o^{(d)} \right) \left(x_i^{(d)} \right)$$
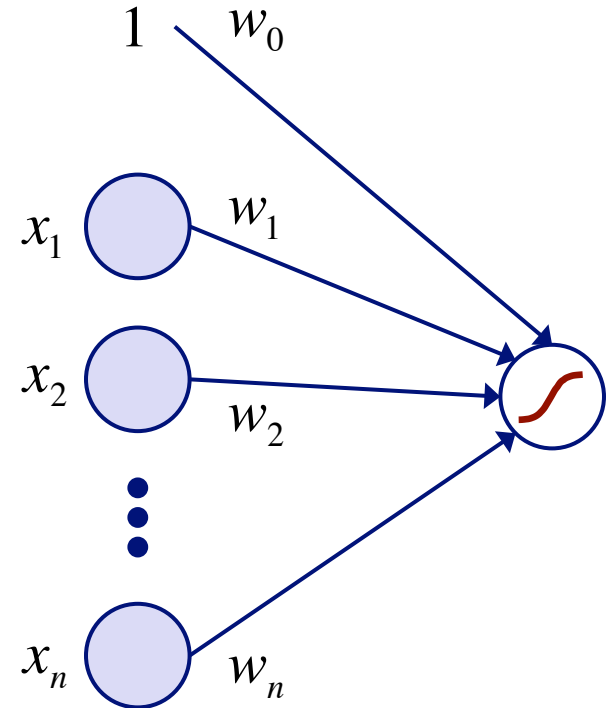
Gradient descent with a sigmoid

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:

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

1 w_0

x_1 w_1

x_2 w_2

x_n w_n

Stochastic GD with sigmoid output unit

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

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

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

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

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

$$= -\left(y^{(d)} - o^{(d)} \right) o^{(d)} (1 - o^{(d)}) x_i^{(d)}$$
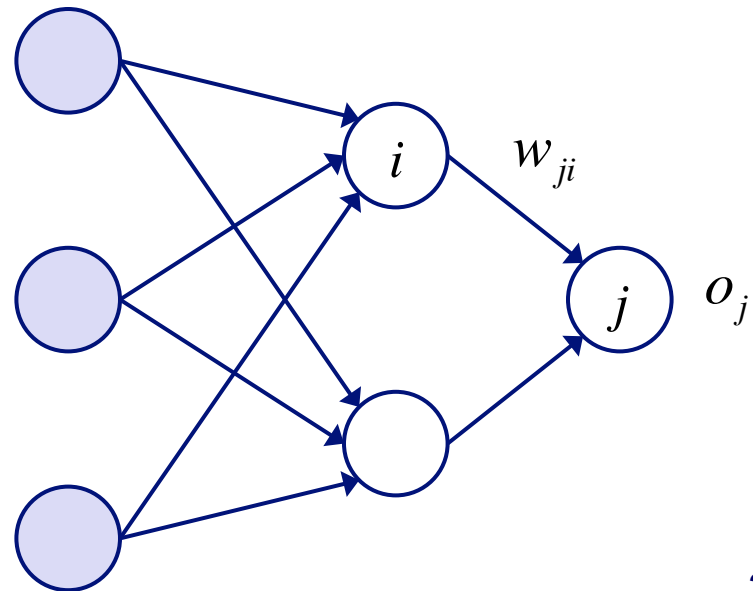
Backpropagation

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

  - linear output units

  - sigmoid output units

- how can we calculate $\dfrac{\partial E}{\partial w_i}$ 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 $\delta_j = -\dfrac{\partial E}{\partial net_j}$

Backpropagation

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

where $\quad \delta_j = -\dfrac{\partial E}{\partial net_j}$

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

same as single-layer net with sigmoid output

$$\delta_j = o_j(1 - o_j)\sum_k \delta_k w_{kj} \quad \text{if } j \text{ is a hidden unit}$$

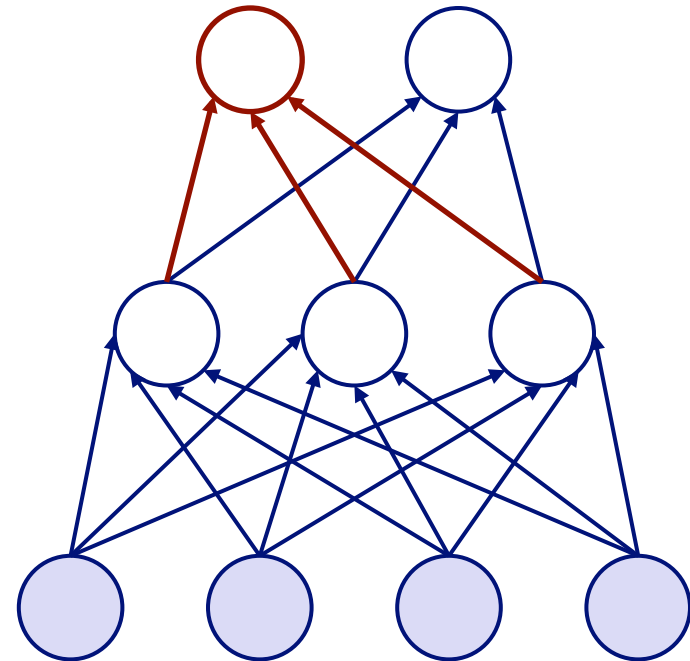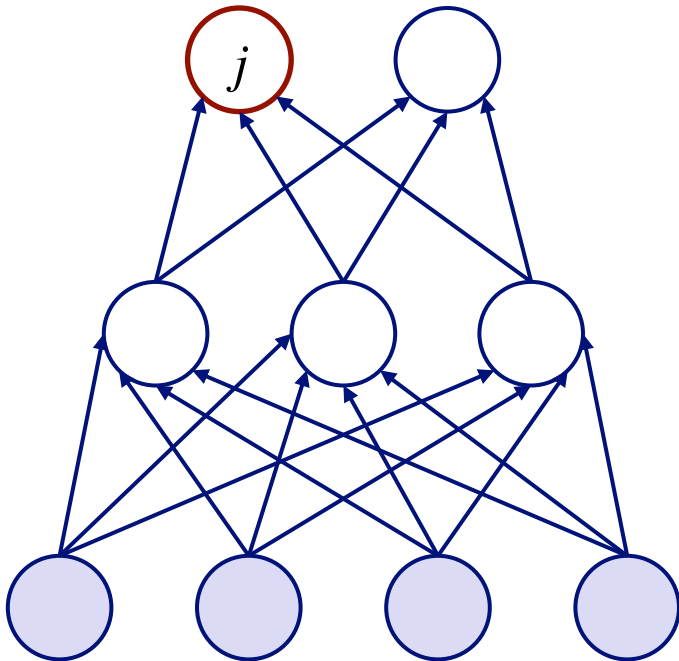sum of backpropagated contributions to error

Backpropagation illustrated

1. calculate error of output units

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

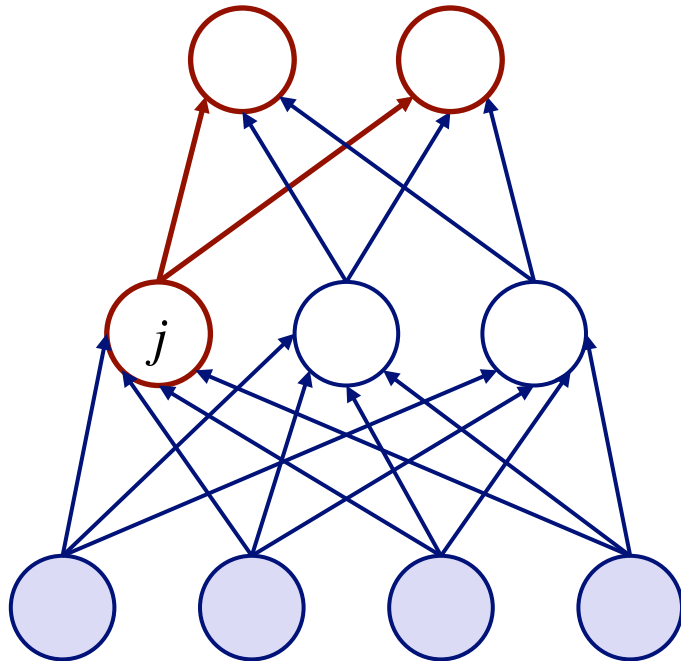2. determine updates for weights going to output units

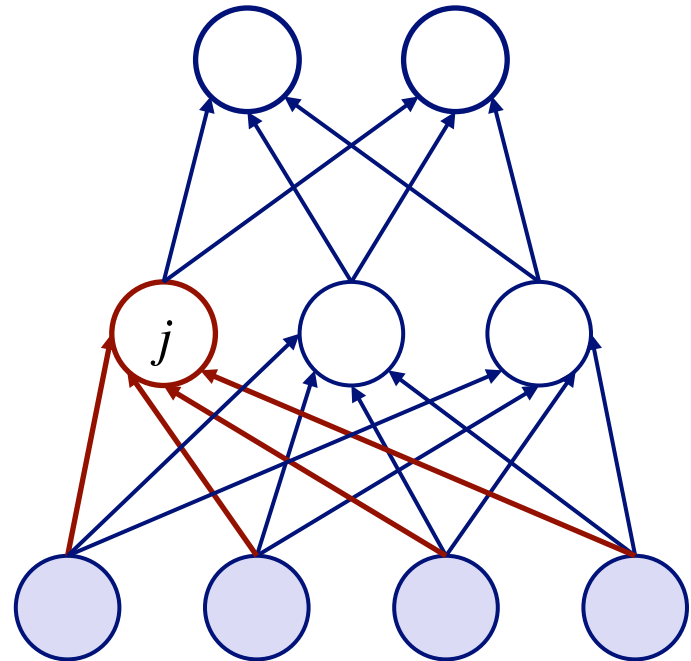$$\Delta w_{ji} = \eta \; \delta_j \; o_i$$

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

- *Minibatch*: in practice, randomly partition data into many parts (e.g., 10 examples each), and compute SGD gradient for one step with one of these parts (minibatches) instead of just one example

Initializing weights

- Weights should be initialized to
  - <u>small values</u> so that the sigmoid activations are in the range where the derivative is large (learning will be quicker)
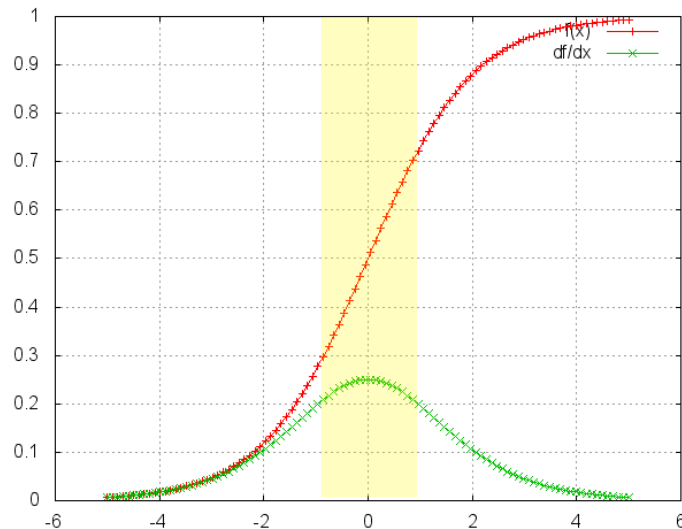
  - <u>random values</u> to ensure symmetry breaking (i.e. if all weights are the same, the hidden units will all represent the same thing)

- typical initial weight range [-0.01, 0.01]

Setting the learning rate

convergence depends on having an appropriate learning rate

$$-\frac{\partial E}{\partial w_{ij}}$$

w_{ij}

η too small (error goes down a little)

η too large (error goes up)
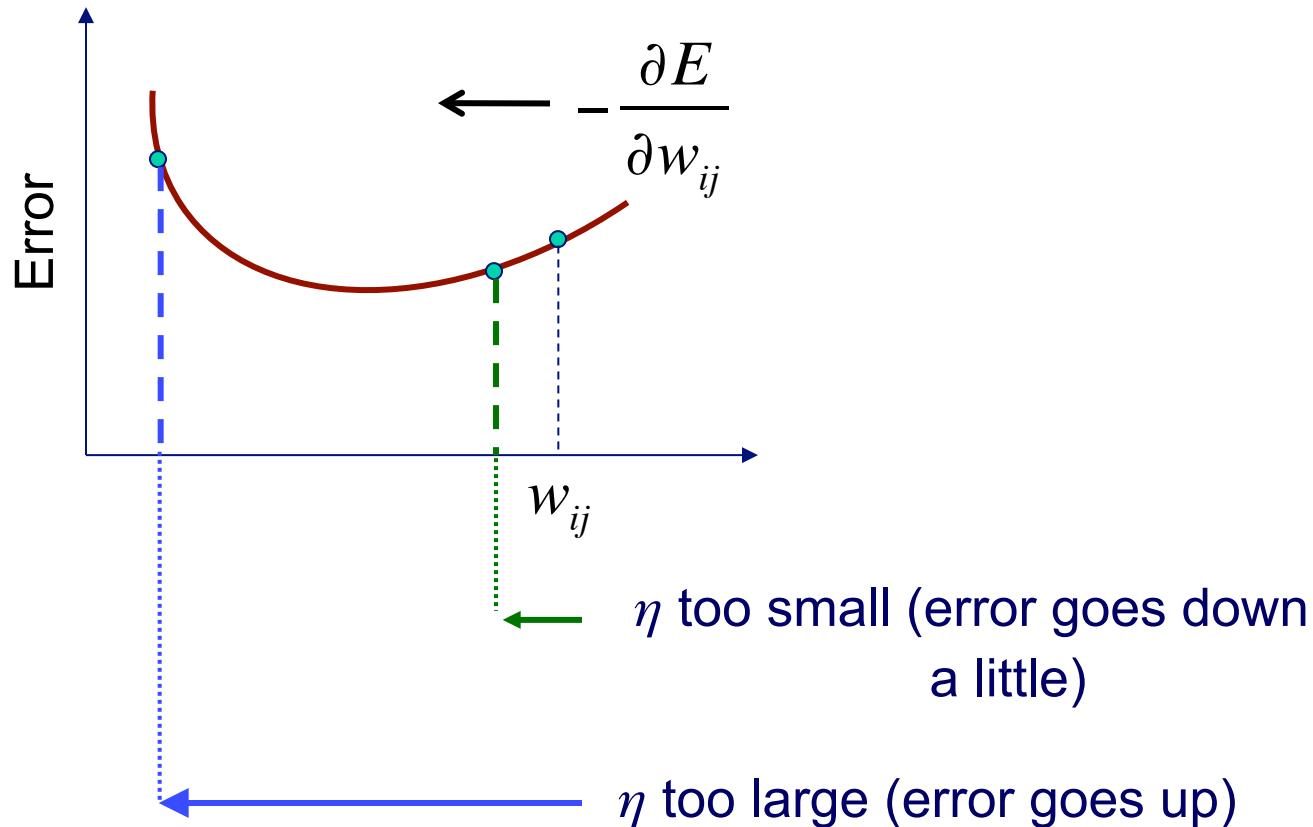
Stopping criteria

- 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

error

stop training here

training iterations
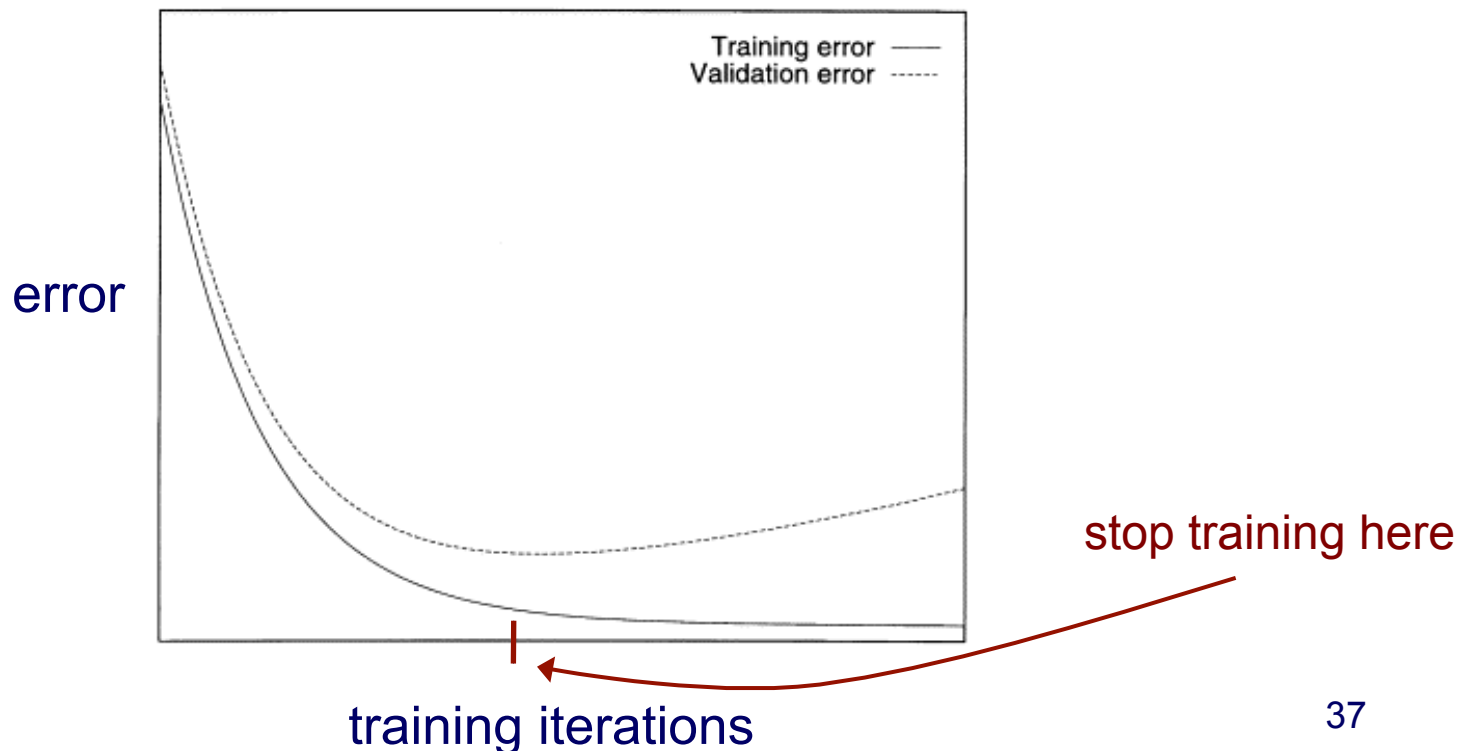
Input (feature) encoding for neural networks

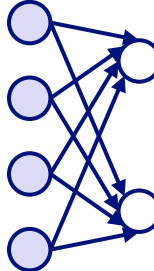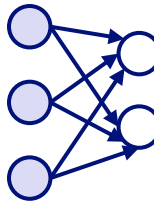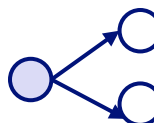nominal features are usually represented using a *1-of-k* encoding

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

ordinal features can be represented using a *thermometer* encoding

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

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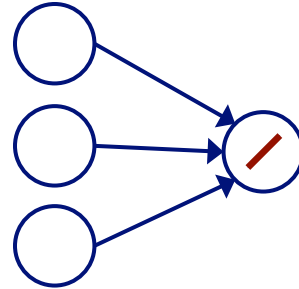
binary classification tasks usually use one sigmoid output unit
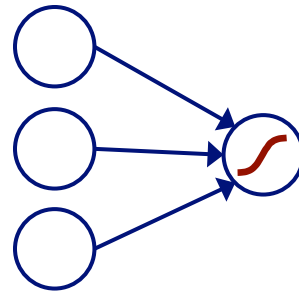
k-ary classification tasks usually use *k* sigmoid or *softmax* output units

$$o_i = \frac{e^{net_i}}{\displaystyle\sum_{j \in outputs} e^{net_j}}$$
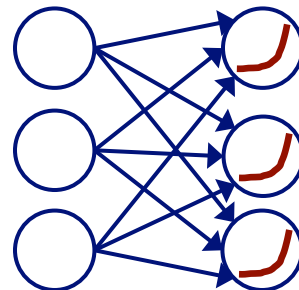
Recurrent neural networks

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
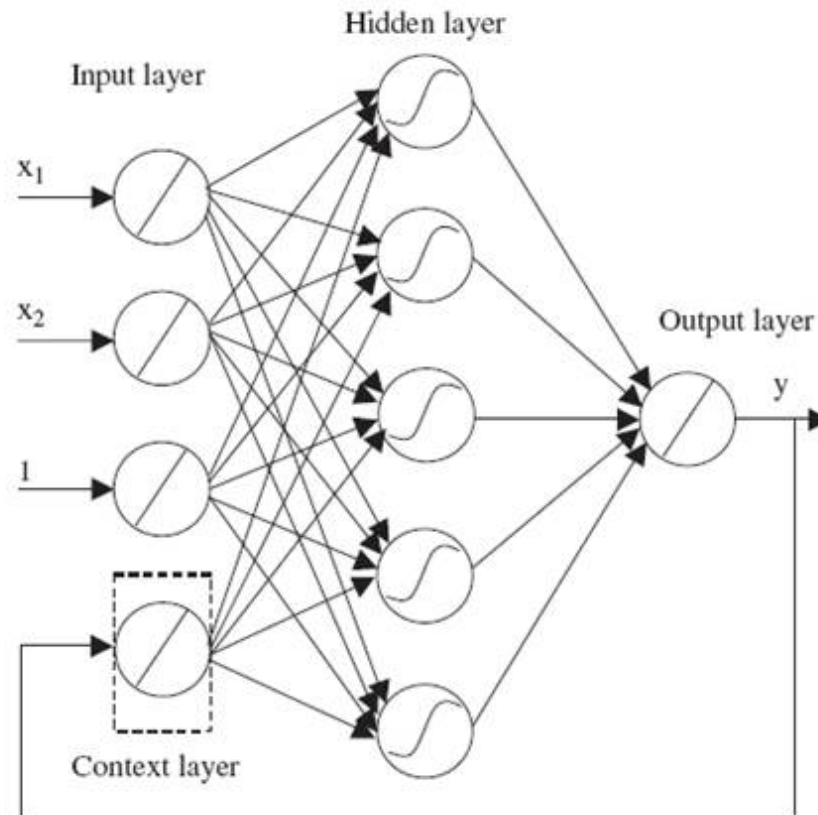
Alternative approach to training **deep networks**

- use <u>unsupervised learning</u> to to find useful hidden unit representations

NO UNSUPERVISED CHILDREN BEYOND THIS POINT

WARNING UNSUPERVISED CHILDREN WILL BE SOLD TO THE HIGHEST BIDDER

Unattended children will be given an espresso and a free puppy.

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	→	10000000
01000000	→	01000000
00100000	→	00100000
00010000	→	00010000
00001000	→	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	00000001
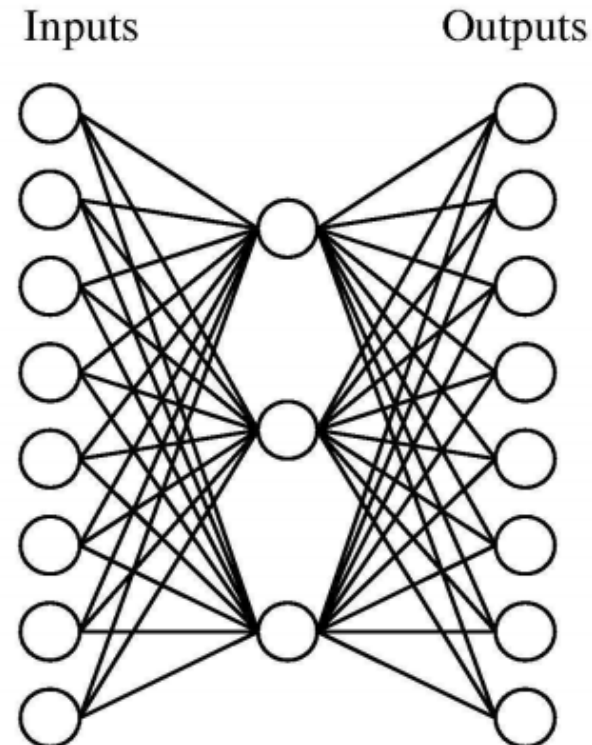
Inputs   Outputs

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
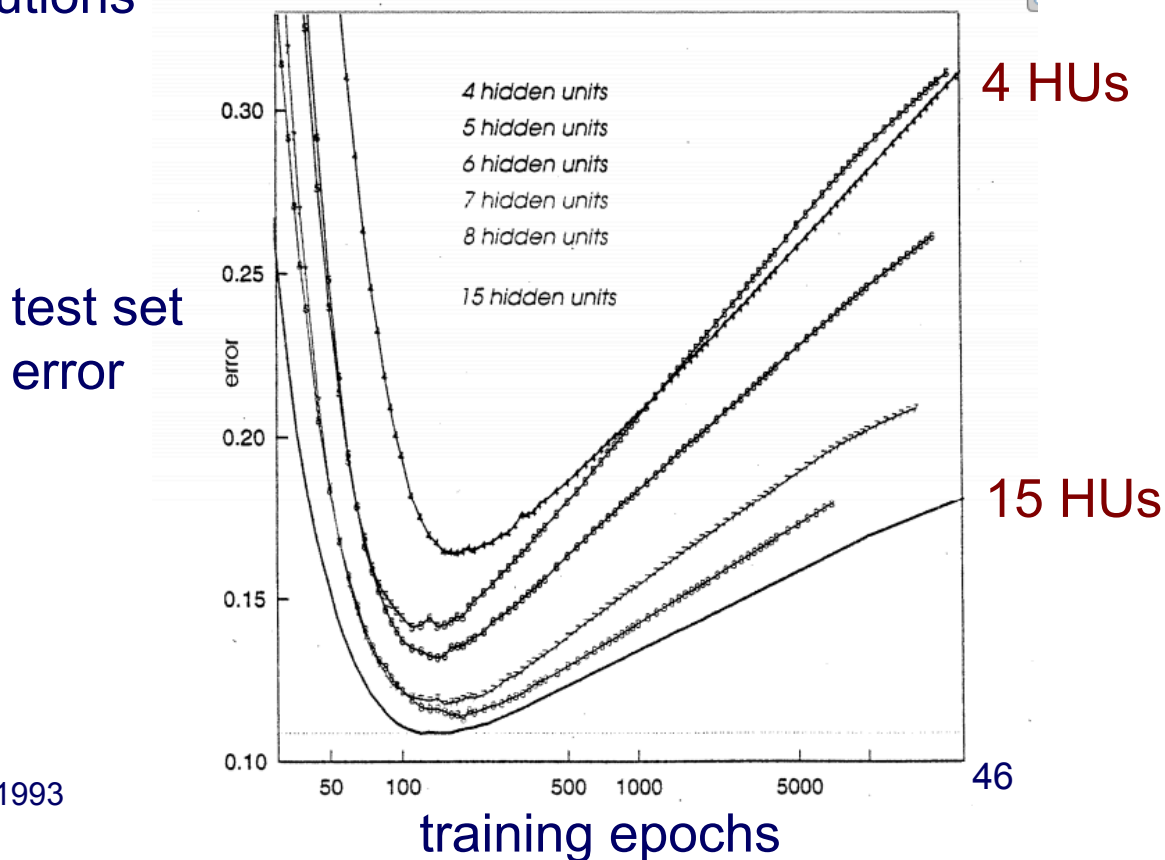
test set error

4 HUs

15 HUs

4 hidden units
5 hidden units
6 hidden units
7 hidden units
8 hidden units

15 hidden units

training epochs

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(**X**)||$\hat{s}$(**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

Autoencoders

- one approach: use *autoencoders* to learn hidden-unit representations
- in an autoencoder, the network is trained to reconstruct the inputs
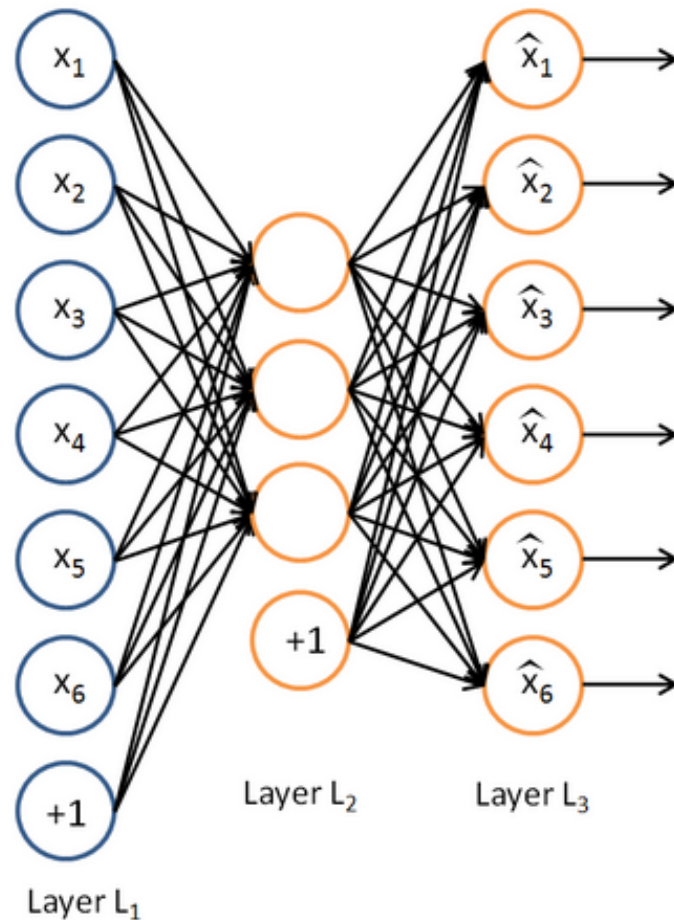
Autoencoder variants

- how to encourage the autoencoder to generalize

    - *bottleneck*: use fewer hidden units than inputs

    - *sparsity*: use a penalty function that encourages most hidden unit activations to be near 0
    [Goodfellow et al. 2009]

    - *denoising*: train to predict true input from corrupted input
    [Vincent et al. 2008]

    - *contractive*: force encoder to have small derivatives (of hidden unit output as input varies) [Rifai et al. 2011]

Stacking Autoencoders

- can be stacked to form highly nonlinear representations [Bengio et al. *NIPS* 2006]
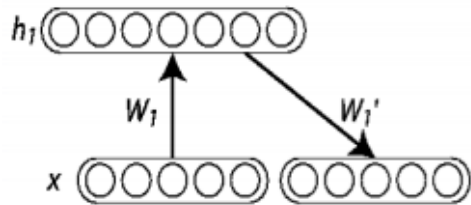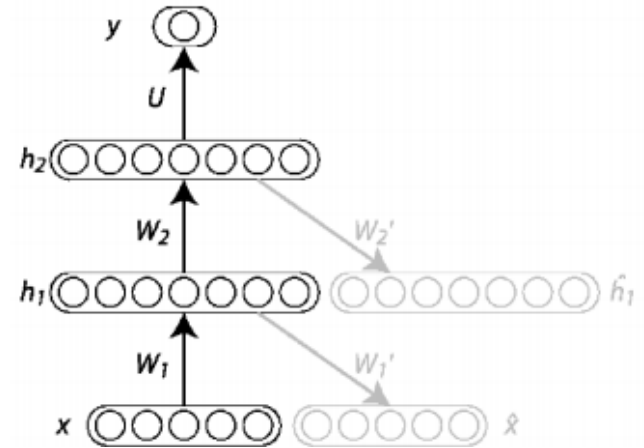

train autoencoder to represent x
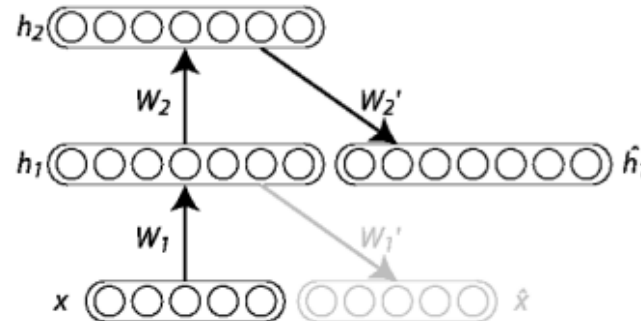
Discard output layer; train autoencoder to represent h_1

Repeat for k layers

discard output layer; train weights on last layer for supervised task

each W_i here represents the matrix of weights between layers

Fine-Tuning

- After completion, run backpropagation on the entire network to fine-tune weights for the supervised task

- Because this backpropagation starts with good structure and weights, its credit assignment is better and so its final results are better than if we just ran backpropagation initially

Why does the unsupervised training step work well?

- *regularization hypothesis*: representations that are good for $P(\boldsymbol{x})$ are good for $P(\mathrm{y} \mid \boldsymbol{x})$

- *optimization hypothesis*: unsupervised initializations start near better local minima of supervised training error

Deep learning not limited to neural networks

- First developed by Geoff Hinton and colleagues for *belief networks*, a kind of hybrid between neural nets and Bayes nets

- Hinton motivates the unsupervised deep learning training process by the credit assignment problem, which appears in belief nets, Bayes nets, neural nets, restricted Boltzmann machines, etc.
  - d-separation: the problem of evidence at a converging connection creating competing explanations
  - backpropagation: can't choose which neighbors get the blame for an error at this node

Room for Debate

- many now arguing that unsupervised *pre-training* phase not really needed…

- backprop is sufficient if done better
    - wider diversity in initial weights, try with many initial settings until you get learning
    - don't worry much about exact learning rate, but add *momentum:* if moving fast in a given direction, keep it up for awhile
    - *Need a lot of data for deep net backprop*

Problems with Backprop for Deep Neural Networks

- Overfits both training data and the particular starting point

- Converges too quickly to a suboptimal solution, even with SGD (gradient from one example or "minibatch" of examples at one time)

- Need more training data and/or fewer weights to estimate, or other regularizer

Trick 1: Data Augmentation

- Deep learning depends critically on "Big Data" – need many more training examples than features

- Turn one positive (negative) example into many positive (negative) examples

- Image data: rotate, re-scale, or shift image, or flip image about axis; image still contains the same objects, exhibits the same event or action

Trick 2: Parameter (Weight) Tying

- Normally all neurons at one layer are connected to next layer

- Instead, have only n features feed to one specific neuron at next level (e.g., 4 or 9 pixels of image go to one hidden unit summarizing this "super-pixel")

- Tie the 4 (or 9) input weights across all super-pixels… more data per weight

Weight Tying Example: Convolution

- Have a sliding window (e.g., square of 4 pixels, set of 5 consecutive items in a sequence, etc), and only the neurons for these inputs feed into one neuron, N1, at the next layer.

- Slide this window over by some amount and repeat, feeding into another neuron, N2, etc.

- Tie the input weights for N1, N2, etc., so they will all learn the same concept (e.g., diagonal edge).

- Repeat into new neurons N1', N2', etc., to learn other concepts.

Alternate Convolutional Layer with Pooling Layer

- Mean pooling: k nodes (e.g., corresponding to 4 pixels constituting a square in an image) are averaged to create one node (e.g., corresponding to one pixel) at the next layer.

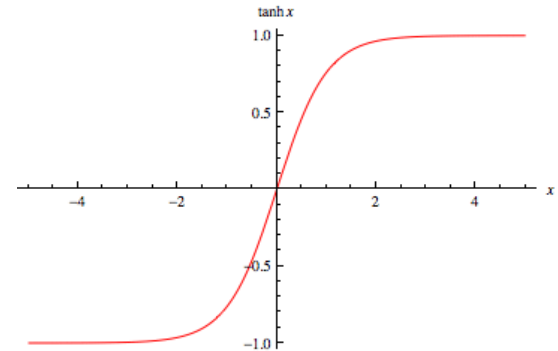- Max pooling: replace average with maximum

Used in Convolutional Neural Networks for Vision Applications
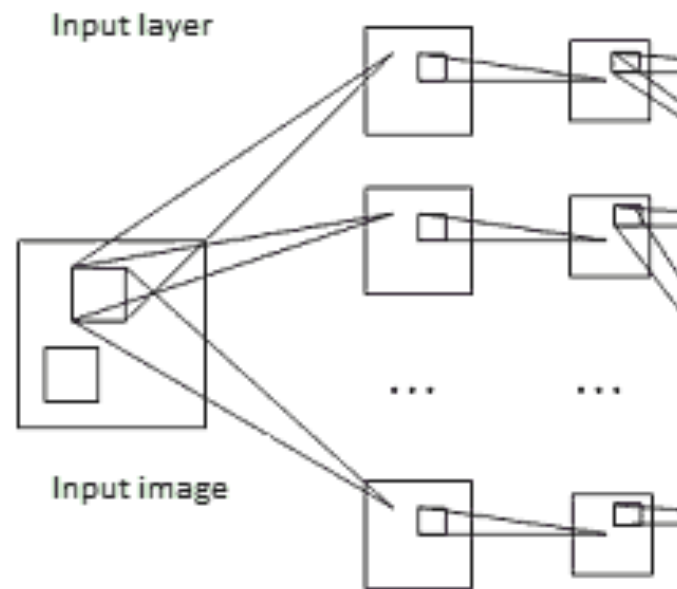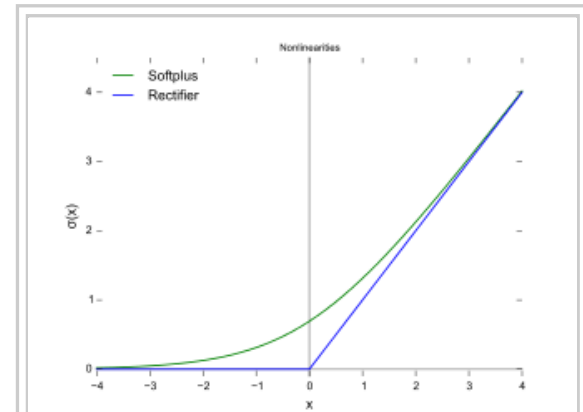
Trick 3: Alternative Activation

- Tanh: *(e^{2x}-1)/(e^{2x}+1)*   hyperbolic tangent

- ReLU: max(0,x) or *1/(1+e^{-x})*   rectified linear unit or softplus

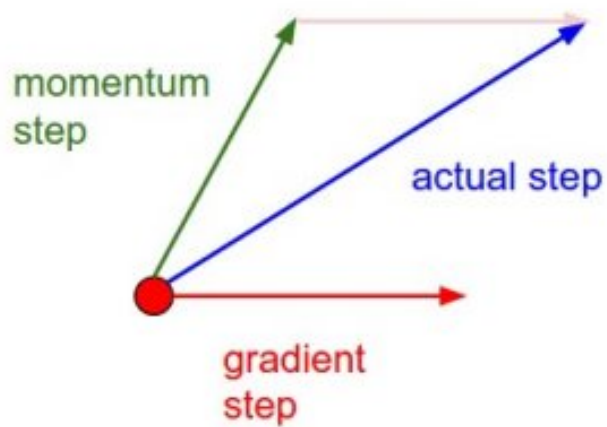Plot of the rectifier (blue) and softplus (green) functions near $x = 0$

Trick 4: Alternative Error Function

- Example: Cross-entropy

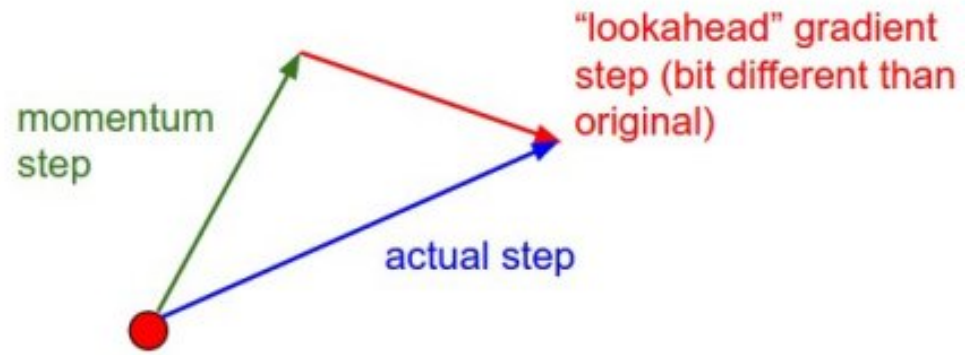$$C = -\frac{1}{n} \sum_{x} [y \ln o + (1 - y) \ln(1 - o)]$$

Trick 5: Momentum

Momentum update

momentum step

actual step

gradient step

Nesterov momentum update

momentum step

actual step

"lookahead" gradient step (bit different than original)
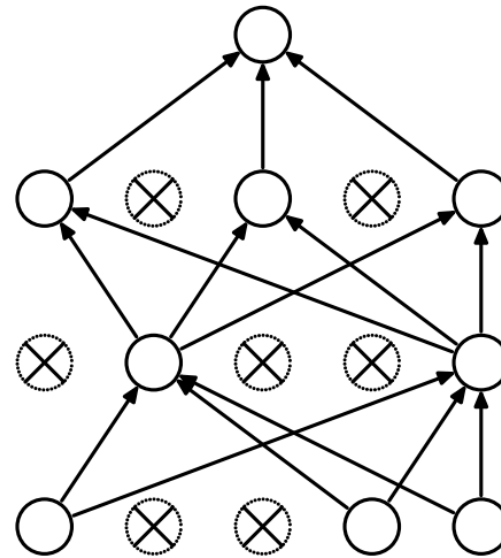
Trick 6: Dropout Training

- Build some redundancy into the hidden units

- Essentially create an "ensemble" of neural networks, but without high cost of training many deep networks

- *Dropout training*…

Dropout training

- On each training iteration, drop out (ignore) 50% of the units (or other 90%, or other) by forcing output to 0 during forward pass

- Ignore for forward & backprop (all training)
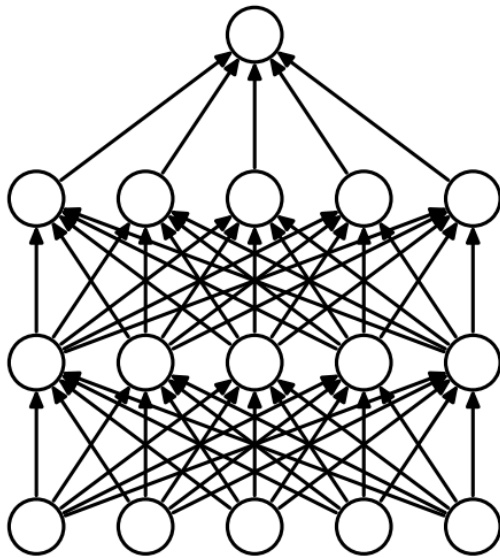
(a) Standard Neural Net

(b) After applying dropout.

Figures from Srivastava et al., *Journal of Machine Learning Research* 2014

At Test Time

- Final model uses all nodes
- Multiply each weight from a node by fraction of times node was used during training

w

Present with probability p

(a) At training time

$p\mathbf{w}$

Always present

(b) At test time
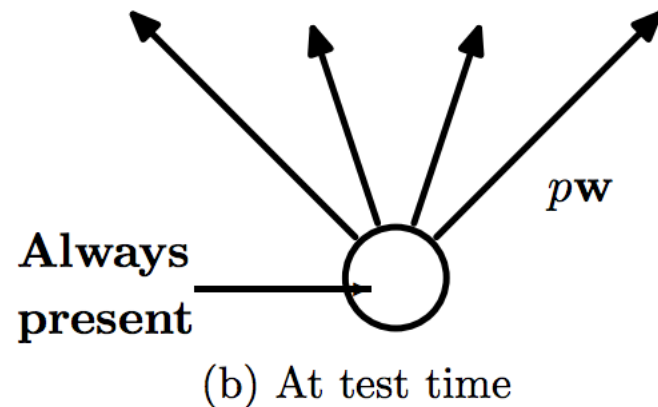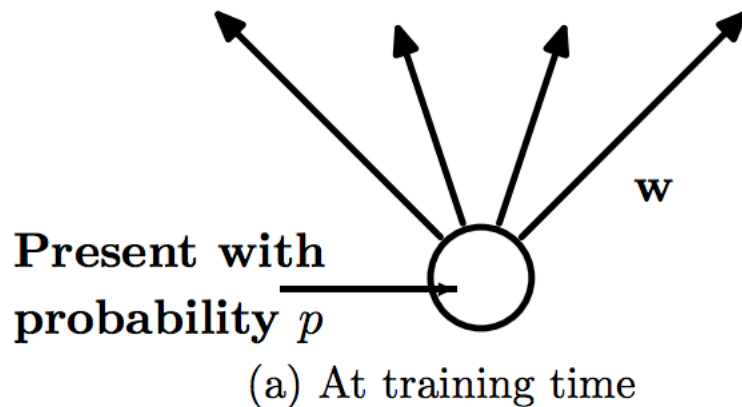
Figures from Srivastava et al., *Journal of Machine Learning Research* 2014

Trick 7: Batch Normalization

- If outputs of earlier layers are uniform or change greatly on one round for one mini-batch, then neurons at next levels can't keep up: they output all high (or all low) values

- Next layer doesn't have ability to change its outputs with learning-rate-sized changes to its input weights

- We say the layer has "saturated"

Another View of Problem

- In ML, we assume future data will be drawn from same probability distribution as training data

- For a hidden unit, after training, the earlier layers have new weights and hence generate input data for this hidden unit from a *new* distribution

- Want to reduce this *internal covariate shift* for the benefit of later layers

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;

Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Comments on Batch Normalization

- First three steps are just like standardization of input data, but with respect to only the data in mini-batch. Can take derivative and incorporate the learning of last step parameters into backpropagation.

- Note last step can completely un-do previous 3 steps

- But if so this un-doing is driven by the *later* layers, not the *earlier* layers; later layers get to "choose" whether they want standard normal inputs or not

Some Deep Learning Resources

- *Nature*, Jan 8, 2014: http://www.nature.com/news/computer-science-the-learning-machines-1.14481

- Ng Tutorial: http://deeplearning.stanford.edu/wiki/index.php/UFLDL_Tutorial

- Hinton Tutorial: http://videolectures.net/jul09_hinton_deeplearn/

- LeCun & Ranzato Tutorial: http://www.cs.nyu.edu/~yann/talks/lecun-ranzato-icml2013.pdf

Comments on neural networks

- stochastic gradient descent often works well for very large data sets

- backpropagation generalizes to
  - arbitrary numbers of output and hidden units
  - arbitrary layers of hidden units (in theory)
  - arbitrary connection patterns
  - other transfer (i.e. output) functions
  - other error measures

- backprop doesn't usually work well for networks with multiple layers of hidden units; recent work in *deep networks* addresses this limitation