Neural Networks (Part 1)

CSE 4334 / 5334 Data Mining
Spring 2019

Won Hwa Kim

(Slides courtesy of Mark Craven and David Page Jr. at UW - Madison)

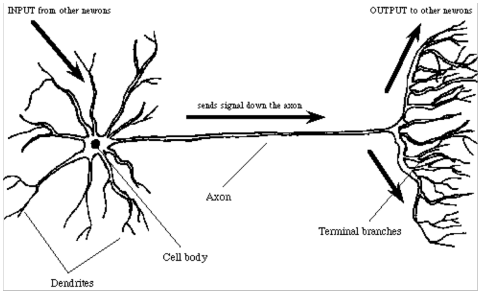
Goals for this lecture

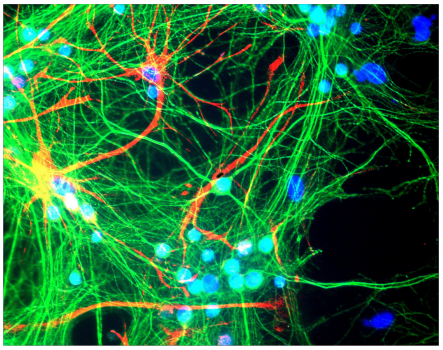
you should understand the following concepts

- perceptrons
- the perceptron training rule
- linear separability
- hidden units
- multilayer neural networks
- gradient descent
- stochastic (online) gradient descent
- activation functions
  - sigmoid, hyperbolic tangent, ReLU
- objective (error, loss) functions
  - squared error, cross entropy

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



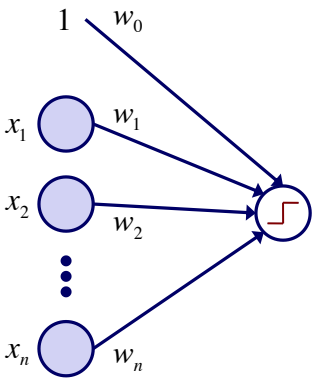


Perceptrons

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

Perceptrons

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

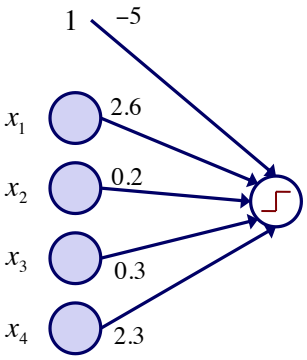


$$o = \begin{cases} 1 & \text{if } w_0 + \sum_{i=1}^{n} w_i x_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

input units:
represent given $\boldsymbol{x}$

output unit:
represents binary classification

Perceptron example



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

features, class labels are represented numerically

$$\mathbf{x} = \langle 1,0,0,1 \rangle \qquad w_0 + \sum_{i=1}^{n} w_i x_i = -0.1 \qquad o = 0$$

$$\mathbf{x} = \langle 1,0,1,1 \rangle \qquad w_0 + \sum_{i=1}^{n} w_i x_i = 0.2 \qquad o = 1$$

Training a perceptron

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

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

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

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

Representational power of perceptrons

perceptrons can represent only *linearly separable* concepts

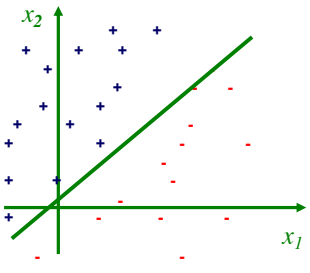
$$o = \begin{cases} 1 \text{ if } w_0 + \sum_{i=1}^{n} w_i x_i > 0 \\ 0 \text{ otherwise} \end{cases}$$

decision boundary given by:

$$1 \text{ if } w_0 + w_1 x_1 + w_2 x_2 > 0$$

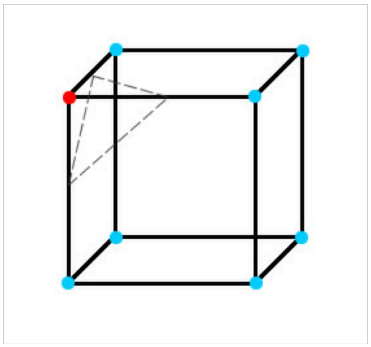
$$w_1 x_1 + w_2 x_2 = -w_0$$

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{w_0}{w_2}$$



Representational power of perceptrons

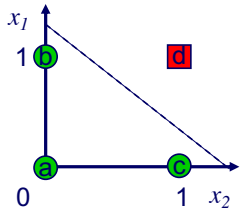
- in previous example, feature space was 2D so decision boundary was a line
- in higher dimensions, decision boundary is a hyperplane



Linearly separable functions

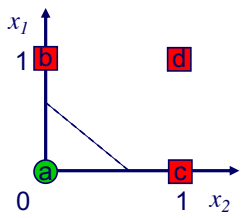
AND

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



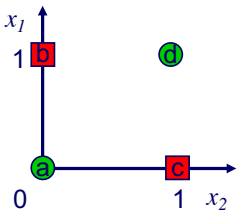
OR

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

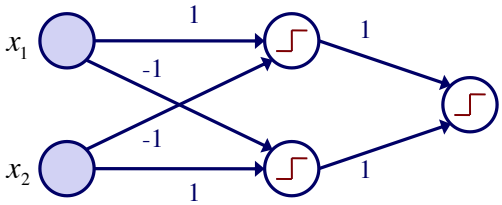


Is XOR linearly separable?

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

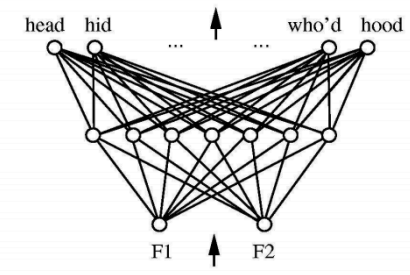


a multilayer perceptron can represent XOR



assume $w_0 = 0$ for all nodes

Example of multi-layer neural network



output units

hidden units

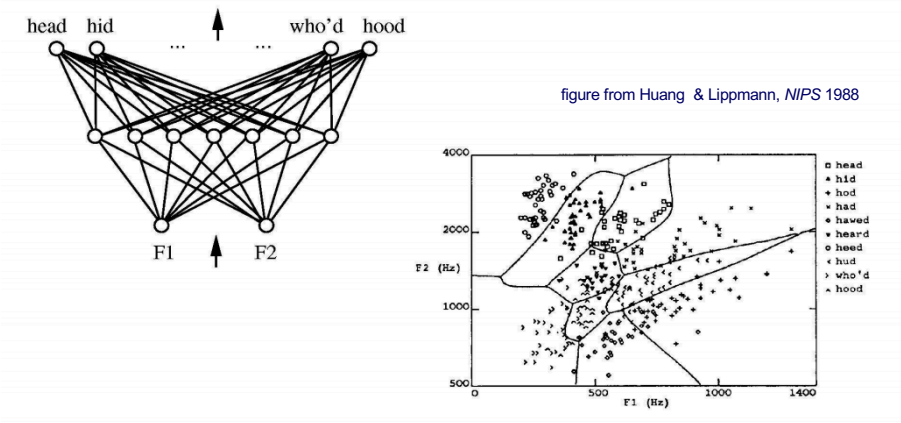
input units

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 multi-layer NN

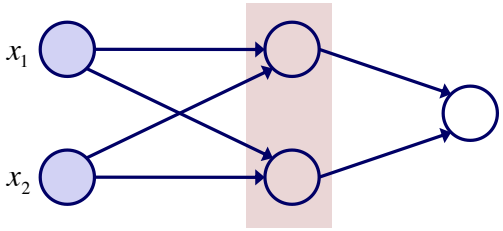


input: two features from spectral analysis of a spoken sound

output: vowel sound occurring in the context "h__d"

Learning in a multi-layer NN

- 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 $\boldsymbol{w}$

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

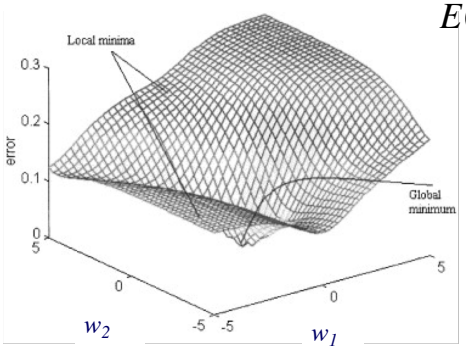


figure from Cho & Chow, *Neurocomputing* 1999

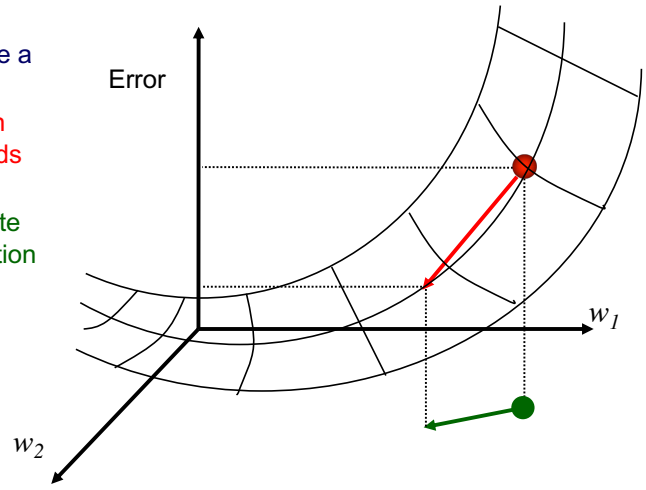
This *objective function* defines a surface over the model (i.e. weight) space

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



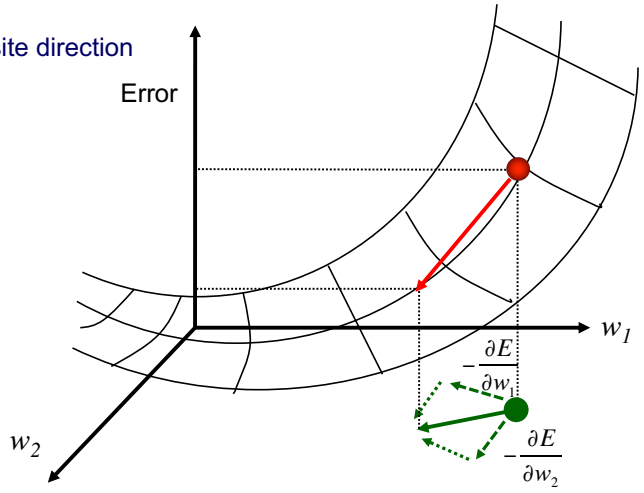
Gradient descent in weight space

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

take a step in the opposite direction

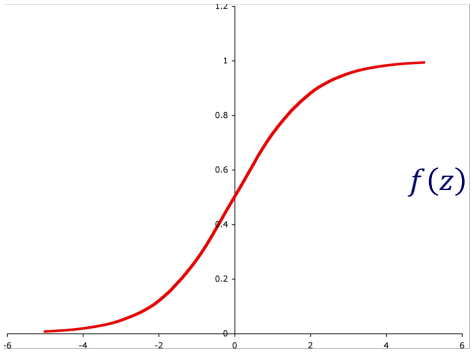
$$\Delta \boldsymbol{w} = -\eta \, \nabla E(\boldsymbol{w})$$

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



The sigmoid function

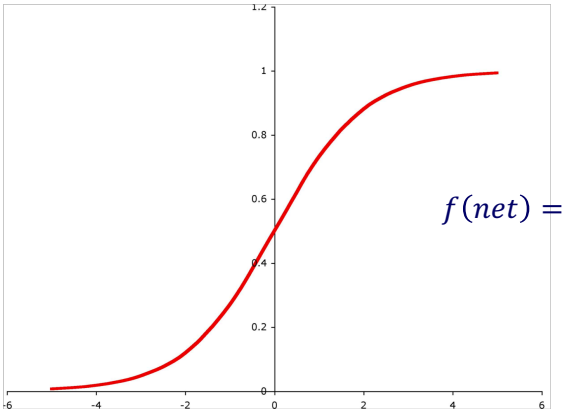
- to be able to differentiate E with respect to w_i, our network must represent a continuous function
- to do this, we can use *sigmoid functions* instead of threshold functions in our hidden and output units



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

The sigmoid function

for the case of a single-layer network



$$f(net) = \frac{1}{1 + e^{-(w_0 + \sum_i w_i x_i)}}$$

$$net = w_0 + \sum_i w_i \, x_i$$

Batch NN 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

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

Online vs. Batch learning

- 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 (or a small set of instances, a "mini batch"), then takes a step in weight space
  - much faster convergence
  - less susceptible to local minima

Online NN 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

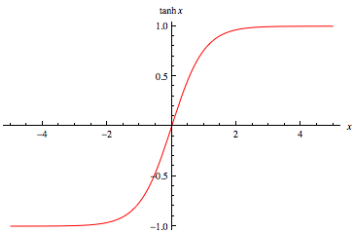
$$\nabla E(\boldsymbol{w}) = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n}\right]$$

update the weights

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

Other activation functions

- the sigmoid is just one choice for an *activation function*
- there are others we can use including



hyperbolic tangent

$$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$$

rectified linear (ReLU)

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

Other objective functions

- squared error is just one choice for an *objective function*
- there are others we can use including

cross entropy

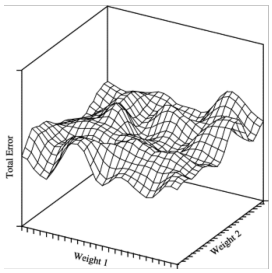
$$E(\boldsymbol{w}) = \sum_{d \in D} -y^{(d)} \ln\left(o^{(d)}\right) - \left(1 - y^{(d)}\right) \ln\left(1 - o^{(d)}\right)$$

multiclass cross entropy

$$E(\boldsymbol{w}) = -\sum_{d \in D} \sum_{i=1}^{\#\,classes} y_i^{(d)} ln\left(o_i^{(d)}\right)$$

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)
- Recent analysis suggests that local minima are probably rare in high dimensions; saddle points are more of a challenge [Dauphin et al., NIPS 2014]

Neural Networks (part 2)

CSE4334/5334 Data Mining

Won Hwa Kim

Many of the contents in this lecture are borrowed from Prof. Mark Craven / Prof. David Page Jr. at UW-Madison

Goals

you should understand the following concepts

- gradient descent with a linear output unit + squared error
- gradient descent with a sigmoid output unit + cross entropy
- backpropagation

Derivatives in NN

recall the chain rule from calculus

$$y = f(u)$$
$$u = g(x)$$

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

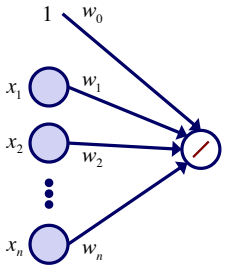
we'll make use of this as follows

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial o}\frac{\partial o}{\partial net}\frac{\partial net}{\partial w_i} \qquad net = w_0 + \sum_{i=1}^{n} w_i x_i$$

Gradient descent: simple case #1

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

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



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: simple case #1

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

$$\frac{\partial E^{(d)}}{\partial w_i} = \frac{\partial E^{(d)}}{\partial o^{(d)}}\frac{\partial o^{(d)}}{\partial net^{(d)}}\frac{\partial net^{(d)}}{\partial w_i}$$

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

$$\frac{\partial o^{(d)}}{\partial net^{(d)}} = 1 \qquad \text{(linear output unit)}$$

$$\frac{\partial net^{(d)}}{\partial w_i} = x_i^{(d)}$$

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

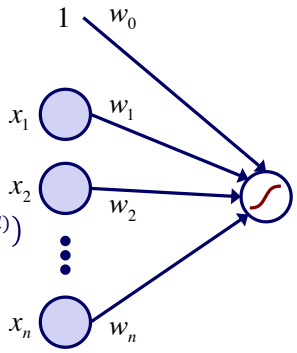
Gradient descent: simple case #2

Now let's consider the case in which we have a sigmoid output unit, no hidden units, and cross-entropy objective function:

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

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

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



useful property:

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

Gradient descent: simple case #2

$$\frac{\partial E^{(d)}}{\partial w_i} = \frac{\partial E^{(d)}}{\partial o^{(d)}} \frac{\partial o^{(d)}}{\partial net^{(d)}} \frac{\partial net^{(d)}}{\partial w_i}$$

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

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

$$\frac{\partial net^{(d)}}{\partial w_i} = x_i^{(d)}$$

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

Backpropagation

- now we've covered how to do gradient descent for single-layer networks
- how can we calculate $\frac{\partial E}{\partial w_i}$ for every weight in a multilayer network?

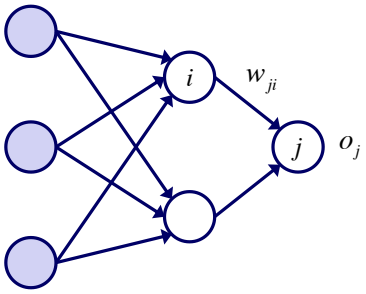
  ➔ backpropagate errors from the output units to the hidden units

Backpropagation

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

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

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

this term is 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 $\delta_j = -\frac{\partial E}{\partial net_j}$

suppose we're using sigmoids and cross-entropy

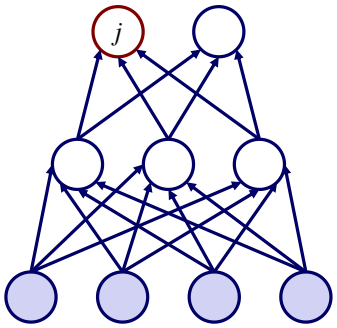
$\delta_j = y_j - o_j$ if j is an output unit } same as single-layer net

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

Backpropagation

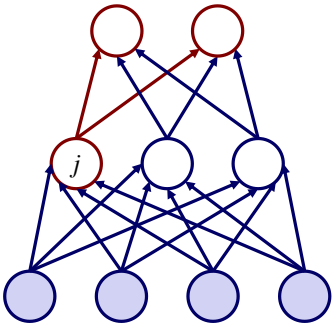
1. calculate error of output units

$$\delta_j = y_j - o_j$$



2. calculate error for hidden units

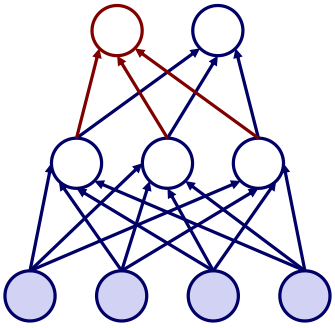
$$\delta_j = o_j(1 - o_j)\sum_k \delta_k w_{kj}$$



Backpropagation

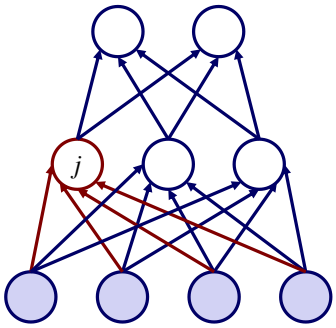
3. determine updates for weights going to output units

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



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

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



Backpropagation

- particular derivatives depend on **objective** and **activation** functions
- here we show derivatives for **squared error** and **sigmoid** functions
- gradient descent and backprop generalize to other cases in which these functions are differentiable

Neural Networks (part 3)

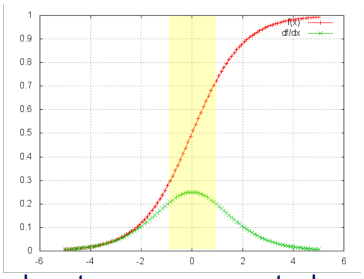
CSE4334/5334 Data Mining

Won Hwa Kim

Many of the contents in this lecture are borrowed from Prof. Mark Craven / Prof. David Page Jr. at UW-Madison

Initializing weights

- For sigmoid/tanh units, weights should be initialized to
  - small values so that the activations are in the range where the derivative is large (learning will be quicker)

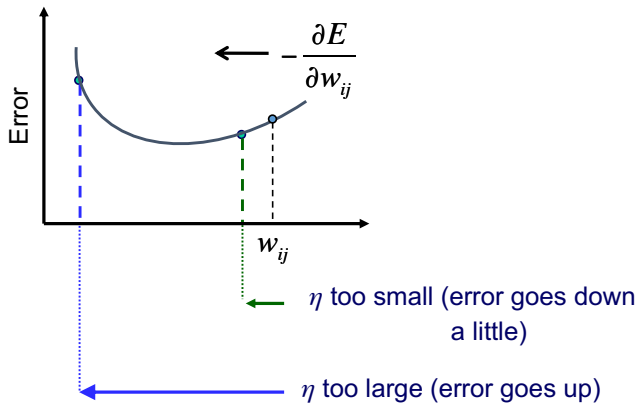


  - random values to ensure symmetry breaking (i.e. if all weights are the same, the hidden units will all represent the same thing)

- typical initial weight range [-0.01, 0.01]

Setting the learning rate

convergence depends on having an appropriate learning rate



Learning rate and momentum

- sometimes a *momentum* term is added

$$\Delta w_{ij}(t) = -\eta \frac{\partial E}{\partial w_{ij}} + \alpha \Delta w_{ij}(t-1)$$

weight update on iteration t | momentum | weight update on iteration t-1

- keeps weights moving in the same direction as the previous update
  - can help to avoid local minima
  - increases step size in flat regions, speeding convergence

Overfitting

- consider error of model h over
  - training data: $error_{\mathrm{D}}(h)$
  - entire distribution of data: $error_{\mathcal{D}}(h)$

- model $h \in H$ *overfits* the training data if there is an alternative model $h' \in H$ such that

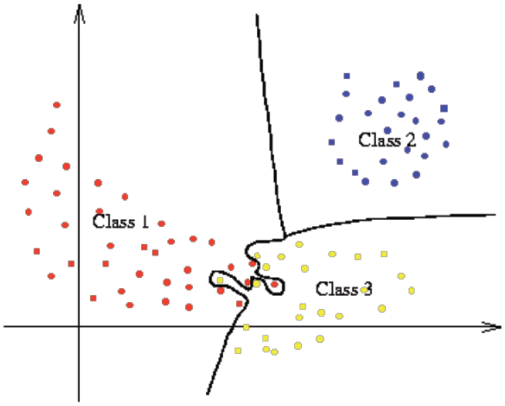
$$error_{\mathcal{D}}(h) > error_{\mathcal{D}}(h')$$

$$error_{\mathrm{D}}(h) < error_{\mathrm{D}}(h')$$

Overfitting

consider a problem with

- 2 continuous features
- 3 classes
- some noisy training instances

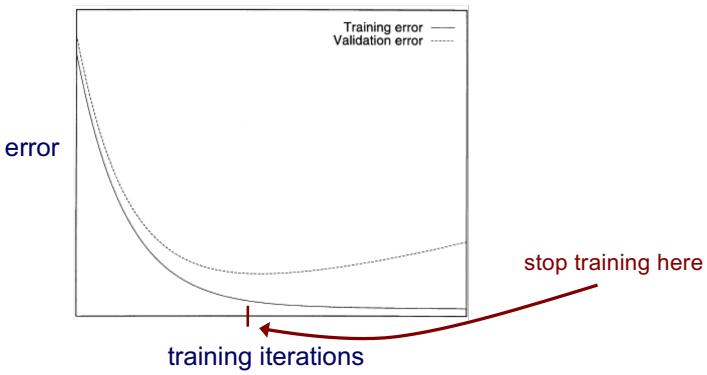


Regularization

- *Regularization* refers to an approach for avoiding overfitting by biasing the learning process away from completely fitting the training data
- E.g. decision-tree pruning is a regularization method
- some regularization methods for neural networks
  - early stopping
  - dropout
  - L1 or L2 penalty terms (we'll discuss these later in the semester)
  - expanding training data by creating new instances that are slightly perturbed versions of existing ones (e.g. rotating images)

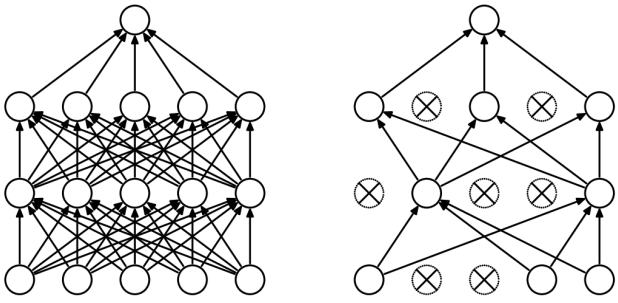
Stopping criteria

- 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



Dropout

On each training iteration
– randomly "drop out" a subset of the units and their weights
– do forward and backprop on remaining network



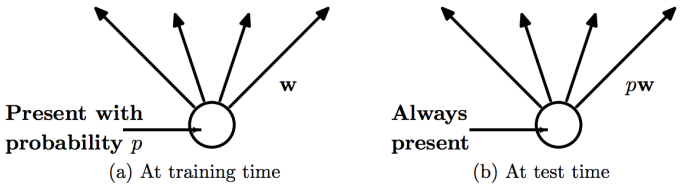
(a) Standard Neural Net (b) After applying dropout.

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

Dropout

At test time

- use all units and weights in the network
- adjust weights according to the probability that the source unit was dropped out



(a) At training time (b) At test time

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

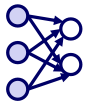
Input (feature) encoding for neural networks

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

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

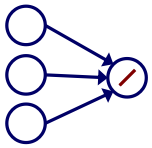


real-valued features can be represented using individual input units (we may want to scale/normalize them first though)

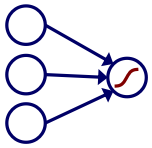
$$\text{precipitation} = [0.68]$$

Output encoding for neural networks

regression tasks usually use output units with linear activation functions

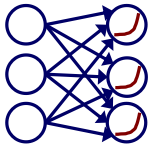


binary classification tasks usually use one sigmoid output unit



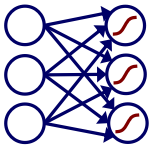
k-ary classification tasks usually use *k* sigmoid or *softmax* output units

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



Output encoding for neural networks

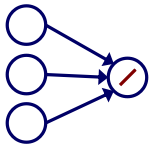
Multi-label classification tasks (i.e. multiple binary labels are predicted for each instance) usually use multiple sigmoid output units



Objective function for output codings

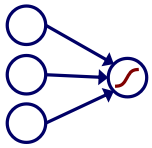
regression: squared error

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



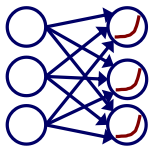
binary classification, multi-label classification: cross entropy

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



k-ary classification: multiclass cross entropy

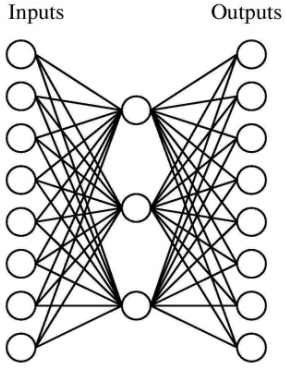
$$E(\boldsymbol{w}) = -\sum_{d \in D}\sum_{i=1}^{\#\,classes} y_i^{(d)} ln\left(o_i^{(d)}\right)$$



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



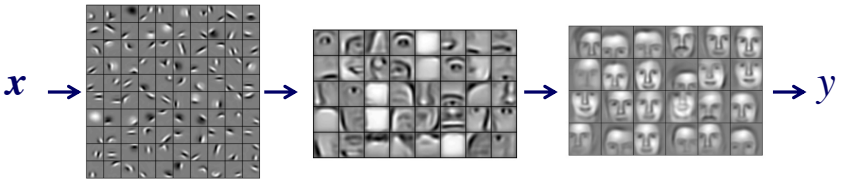
Hidden units

- in this task, hidden units learn a compressed numerical coding of the inputs/outputs **(autoencoder)**

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

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



[Figures from Lee et al., *ICML* 2009]

Backpropatagion with multiple hidden layers

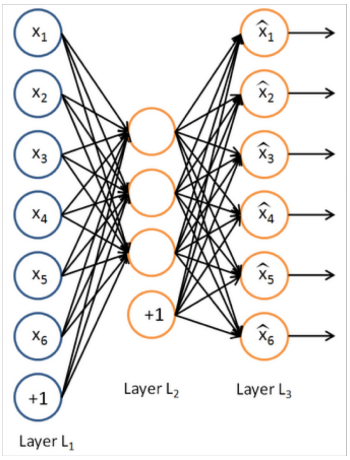
- in principle, backpropagation can be used to train arbitrarily deep networks (i.e. with multiple hidden layers)
- in practice, this doesn't usually work well with sigmoid units
  - diffusion of gradients leads to slow training in lower layers
- better ways of training deep networks (multiple hidden layers)
  1. *pretraining*: greedy layer-wise unsupervised learning
  2. backprop with *rectified linear units,* special architectures and other tricks

DN approach 1: Pretraining

1. Use unsupervised learning for greedy layer-wise training
   - allows abstractions to develop from one layer to the next
   - helps initialize network with good parameters
   - enables unlabeled data to be used for training!
2. Use supervised learning (gradient descent) to learn the last layer
   - … and often to refine the other layers

Pretraining: Autoencoders

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

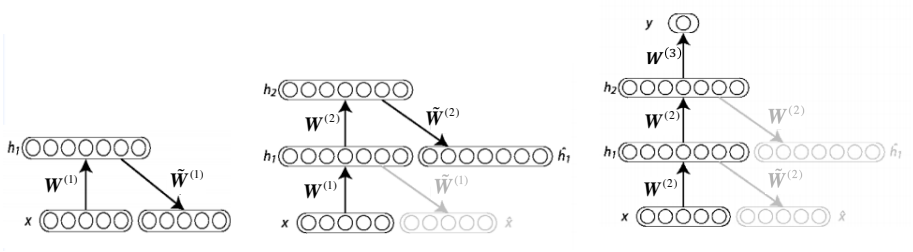


Pretraining: Autoencoder variants

- various approaches can be used 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 [Rifai et al. 2011]

Stacking autoencoders

- autoencoders can be stacked to form highly nonlinear representations



train autoencoder to represent $\boldsymbol{x}$

discard $\widetilde{\boldsymbol{W}}^{(1)}$
train autoencoder to represent h_1

discard $\widetilde{\boldsymbol{W}}^{(2)}$
train weights $\boldsymbol{W}^{(3)}$ for supervised task

each $\boldsymbol{W}^{(i)}$ here represents the matrix of weights between layers, and h_i represents the i^{th} layer of hidden units

Fine tuning

- after completion, can run backpropagation on the entire network to fine-tune weights for the supervised task
- because this backpropagation starts with good weights, its credit assignment is better and the learned model is likely to be better than if we just ran backpropagation initially

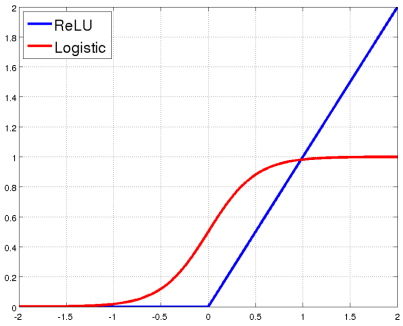
DN approach 2: Direct supervised training

- direct supervised training of deep networks commonly uses a few techniques to avoid slow training and overfitting
  - rectified linear units (ReLUs) instead of sigmoids
  - dropout
  - specialized architectures

Rectified linear units (ReLU)

- faster learning than sigmoids because gradients don't vanish as x increases
- more efficient computation because exponential function is not used

$$f(x) = \max(0, x)$$



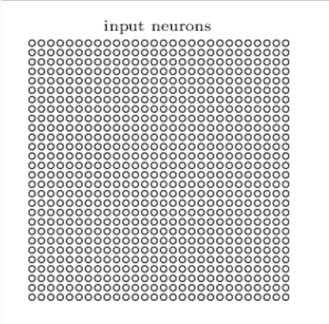
Convolutional Neural Network (CNN)

- well suited to tasks in which the input has spatial structure, such as images or sequences

- based on four key ideas
  - local receptive fields
  - weight sharing
  - pooling
  - multiple layers of hidden units

Convolutional Neural Network (CNN)

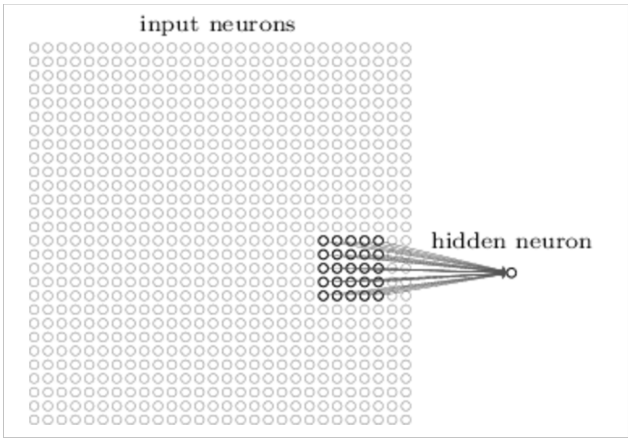
- suppose we have a task in which are instances are 28×28 pixel images
- we can represent each using 28×28 input units



[Figure from neuralnetworksanddeeplearning.com]

Convolutional Neural Network (CNN)

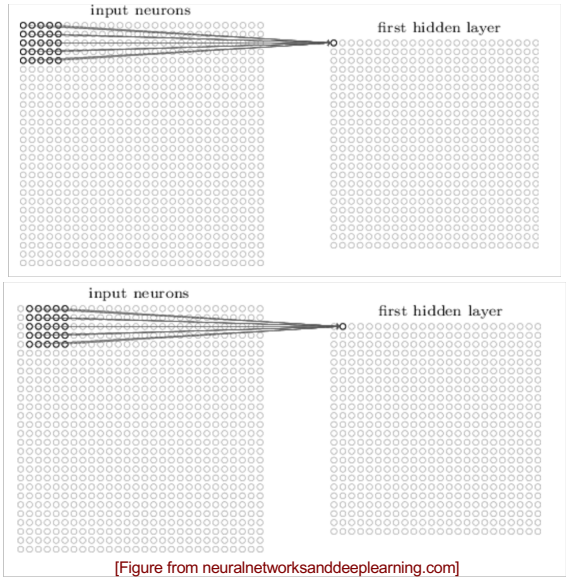
- we can connect hidden units so that each has a *local receptive field* (e.g. a 5 × 5 patch of the image).



[Figure from neuralnetworksanddeeplearning.com]

Convolutional Neural Network (CNN)

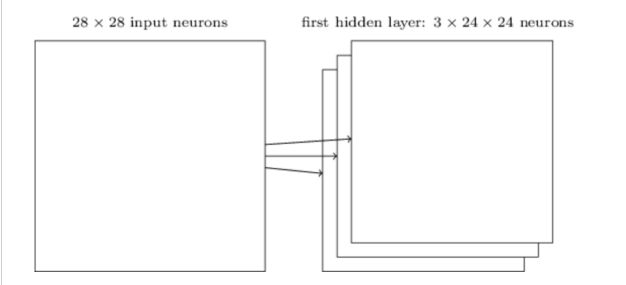
- we can have a set of these units that differ in their local receptive field
- all of the units share the same set of weights
- so the units detect same “feature” in the image, but at different locations



[Figure from neuralnetworksanddeeplearning.com]

Convolutional Neural Network (CNN)

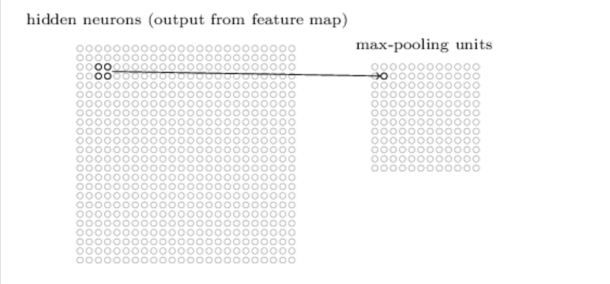
- a set of units that detect the same “feature” is called a *feature map*
- typically we’ll have multiple feature maps in each layer



[Figure from neuralnetworksanddeeplearning.com]

Convolutional Neural Network (CNN)

- feature-map layers are typically alternated with pooling layers
- each unit in a pooling layer outputs a max, or similar function, of a subset of the units in the previous layer



[Figure from neuralnetworksanddeeplearning.com]

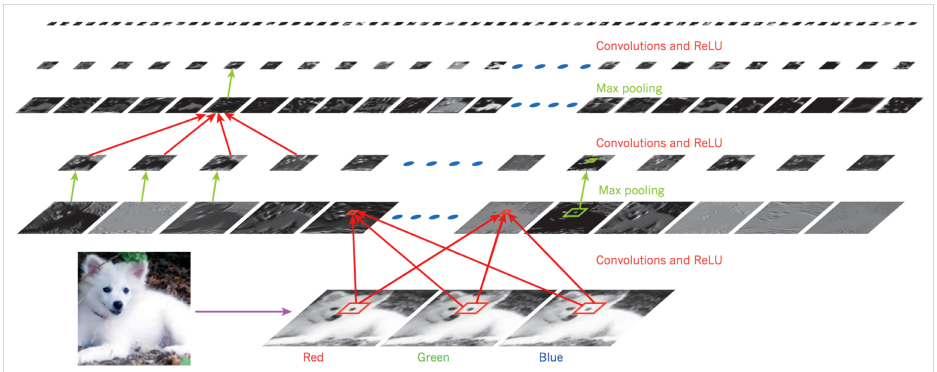
$$f(\boldsymbol{x}) = max(x_1 \dots x_i \dots)$$

$$f(\boldsymbol{x}) = log \sum_i e^{x_i}$$

$$f(\boldsymbol{x}) = \sqrt{\sum_i x_i^2}$$

Convolutional Neural Network (CNN)

- alternating layers of convolutional and pooling layers can be stacked

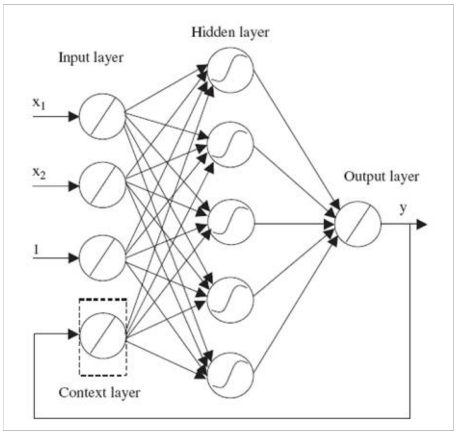


[Figure from LeCun et al., *Nature* 2015]

Recurrent Neural Network (RNN)

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



Comments…

- deep networks have had much recent success due to a combination of tricks and factors
  - rectified linear units to handle the diminishing gradients problem
  - dropout to avoid overfitting
  - sparsely connected architectures (e.g. convolutional networks) to incorporate task-specific bias
  - very large data sets and hardware to enable training with them

- stochastic gradient descent often works well for very large data sets even with simple models (i.e. no hidden units)
  - one pass (or a few passes) through the data set may be sufficient to learn a good model

- gradient descent/backpropagation generalizes to
  - arbitrary numbers of output and hidden units
  - arbitrary layers of hidden units
  - arbitrary connection patterns
  - multiple activation functions
  - multiple objective functions