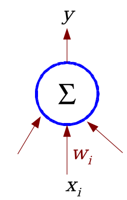
Backpropagation Learning

15-486/782: Artificial Neural Networks
David S. Touretzky

Fall 2006

1

LMS / Widrow-Hoff Rule



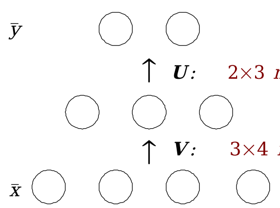
$$\Delta w_i = -\eta(y-d)x_i$$

Works fine for a single layer of trainable weights.

What about multi-layer networks?

2

With Linear Units, Multiple Layers Don't Add Anything



$$\bar{y} = \boldsymbol{U} \times (\boldsymbol{V}\bar{x}) = \underbrace{(\boldsymbol{U} \times \boldsymbol{V})}_{2\times 4}\bar{x}$$

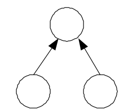
Linear operators are closed under composition.
Equivalent to a single layer of weights $\boldsymbol{W} = \boldsymbol{U} \times \boldsymbol{V}$

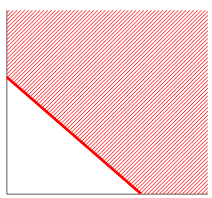
But with non-linear units, extra layers add computational power.

3

What Can be Done with Non-Linear (e.g., Threshold) Units?

1 layer of trainable weights

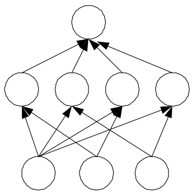


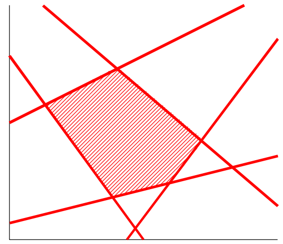


separating hyperplane

4

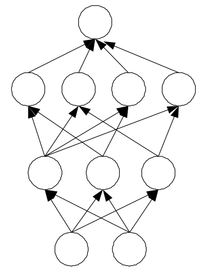
2 layers of
trainable
weights

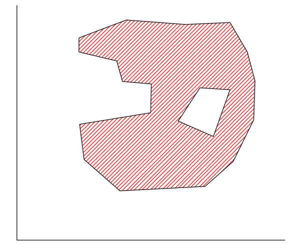




convex polygon region

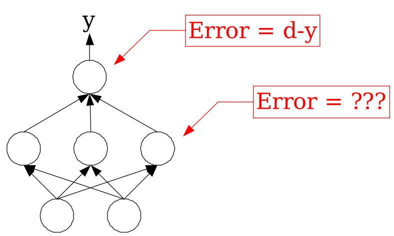
3 layers of
trainable
weights





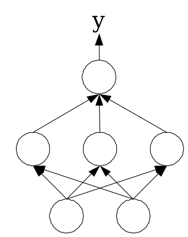
composition of polygons:
convex regions

How Do We Train A Multi-Layer Network?



Can't use perceptron training algorithm because
we don't know the 'correct' outputs for hidden units.

How Do We Train A Multi-Layer Network?



Define sum-squared error:

$$E = \frac{1}{2}\sum_{p} (d^p - y^p)^2$$

Use gradient descent error minimization:

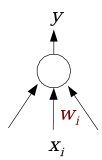
$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}}$$

Works if the nonlinear transfer function is differentiable.

Deriving the LMS or “Delta” Rule As Gradient Descent Learning

$$y = \sum_i w_i x_i$$

$$E = \frac{1}{2}\sum_p (d^p - y^p)^2 \qquad \frac{dE}{dy} = y - d$$



$$\frac{\partial E}{\partial w_i} = \frac{dE}{dy}\cdot\frac{\partial y}{\partial w_i} = (y-d)x_i$$

$$\Delta w_i = -\eta\frac{\partial E}{\partial w_i} = -\eta(y-d)x_i$$

How do we extend this to two layers?

Switch to Smooth Nonlinear Units

$$\text{net}_j = \sum_i w_{ij} y_i$$

$$y_j = \text{g}(\text{net}_j)$$ *g must be differentiable*

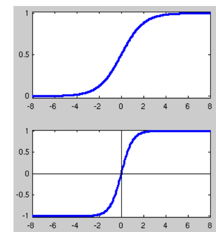
Common choices for g:

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

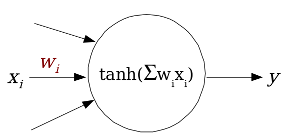
$$g'(x) = g(x)\cdot(1-g(x))$$

$$g(x) = \tanh(x)$$

$$g'(x) = 1/\cosh^2(x)$$



Gradient Descent with Nonlinear Units



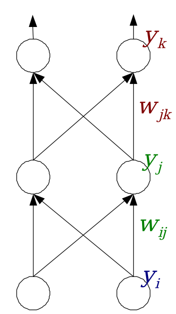
$$y = g(net) = \tanh\left(\sum_i w_i x_i\right)$$

$$\frac{dE}{dy} = (y-d), \qquad \frac{dy}{dnet} = 1/\cosh^2(net), \qquad \frac{\partial net}{\partial w_i} = x_i$$

$$\frac{\partial E}{\partial w_i} = \frac{dE}{dy}\cdot\frac{dy}{dnet}\cdot\frac{\partial net}{\partial w_i}$$

$$= (y-d)/\cosh^2\left(\sum_i w_i x_i\right)\cdot x_i$$

Now We Can Use The Chain Rule



$$\frac{\partial E}{\partial y_k} = (y_k - d_k)$$

$$\delta_k = \frac{\partial E}{\partial net_k} = (y_k - d_k)\cdot g'(net_k)$$

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial net_k}\cdot\frac{\partial net_k}{\partial w_{jk}} = \frac{\partial E}{\partial net_k}\cdot y_j$$

$$\frac{\partial E}{\partial y_j} = \sum_k\left(\frac{\partial E}{\partial net_k}\cdot\frac{\partial net_k}{\partial y_j}\right)$$

$$\delta_j = \frac{\partial E}{\partial net_j} = \frac{\partial E}{\partial y_j}\cdot g'(net_j)$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial net_j}\cdot y_i$$

Weight Updates

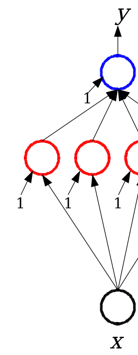
$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial net_k} \cdot \frac{\partial net_k}{\partial w_{jk}} = \delta_k \cdot y_j$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial net_j} \cdot \frac{\partial net_j}{\partial w_{ij}} = \delta_j \cdot y_i$$

$$\Delta w_{jk} = -\eta \cdot \frac{\partial E}{\partial w_{jk}} \qquad \Delta w_{ij} = -\eta \cdot \frac{\partial E}{\partial w_{ij}}$$

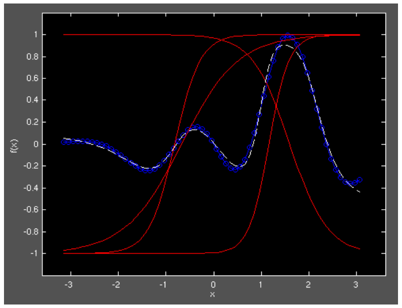
13

Function Approximation



Bumps from which we compose f(x)

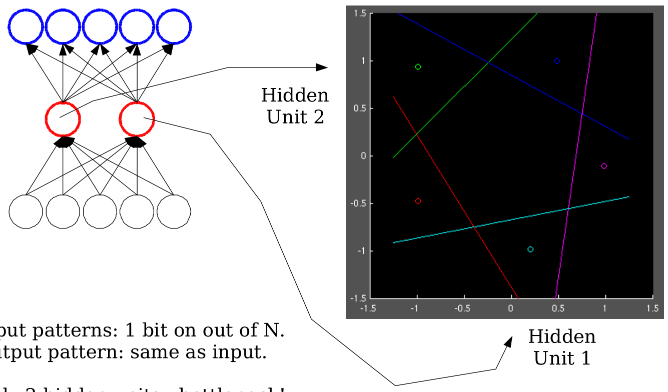
$\tanh(w_0 + w_1 x)$



3n+1 free parameters for *n* hidden units

14

Encoder Problem

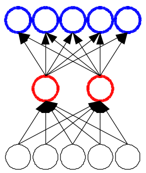


Input patterns: 1 bit on out of N.
Output pattern: same as input.

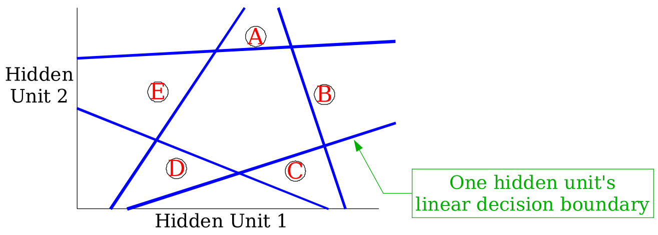
Only 2 hidden units: bottleneck!

15

5-2-5 Encoder Problem



Training patterns:						*Hidden code:*
A:	0	0	0	0	1	2, 0
B:	0	0	0	1	0	0, 2
C:	0	0	1	0	0	1, −1
D:	0	1	0	0	0	−1, 1
E:	1	0	0	0	0	−1, 0



16

Solving XOR

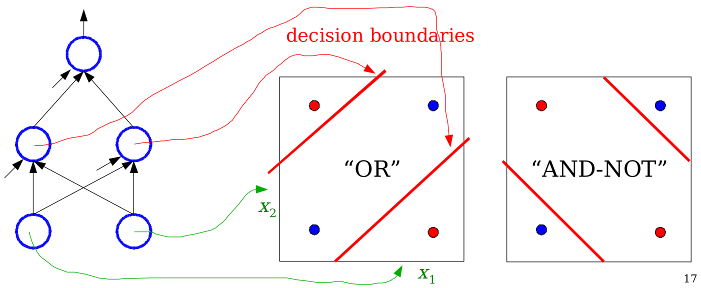
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Two solutions:

$$x_1 \overline{x}_2 \vee \overline{x}_1 x_2$$

$$(x_1 \vee x_2) \wedge \overline{x_1 \wedge x_2}$$

Try the bpxor demo. Which solution does it use?



Improving Backprop Performance

- Avoiding local minima
- Keep derivatives from going to zero
- For classifiers, use reachable targets
- Compensate for error attenuation in deep layers
- Compensate for fan-in effects
- Use momentum to speed learning
- Reduce learning rate when weights oscillate
- Use small initial random weights and small initial learning rate to avoid "herd effect"
- Cross-entropy error measure

Avoiding Local Minima

One problem with backprop is that the error surface is no longer bowl-shaped.

Gradient descent can get trapped in local minima.

In practice, this does not usually prevent learning.

"Noise" can get us out of local minima:

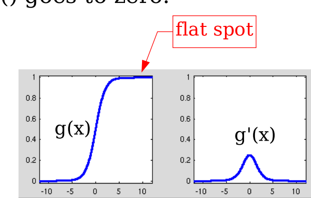
Stochastic update (one pattern at a time).

Add noise to training data, weights, or activations.

Large learning rates can be a source of noise due to overshooting.

Flat Spots

If weights become large, net_j becomes large, derivative of g() goes to zero.



Fahlman's trick: add a small constant to g'(x) to keep the derivative from going to zero. Typical value is 0.1.

Reachable Targets for Classifiers

Targets of 0 and 1 are unreachable by the logistic or tanh functions.

Weights get large as the algorithm tries to force each output unit to reach its asymptotic value.

Trying to get a “correct” output from 0.95 up to 1.0 wastes time and resources that should be concentrated elsewhere.

Solution: use “reachable targets” of 0.1 and 0.9 instead of 0/1. And don't penalize the network for overshooting these targets.

Error Signal Attenuation

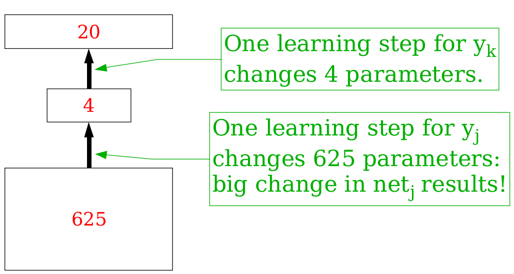
The error signal δ gets attenuated as it moves backward through multiple layers.

So different layers learn at different rates.

Input-to-hidden weights learn more slowly than hidden-to-output weights.

Solution: have different learning rates η for different layers.

Fan-In Affects Learning Rate



Solution: scale learning rate by fan-in.

Momentum

Learning is slow if the learning rate is set too low.

Gradient may be steep in some directions but shallow in others.

Solution: add a momentum term α.

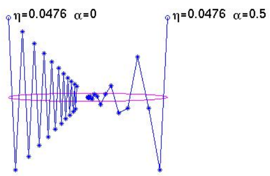
$$\Delta w_{ij}(t) = -\eta \frac{\partial E}{\partial w_{ij}(t)} + \alpha \cdot \Delta w_{ij}(t-1)$$

Typical value for α is 0.5.

If the direction of the gradient remains constant, the algorithm will take increasingly large steps.

Momentum Demo

Hertz, Krogh & Palmer figs. 5.10 and 6.3: gradient descent on a quadratic error surface E (no neural net) involved:



$$E = x^2 + 20y^2$$

$$\frac{\partial E}{\partial x} = 2\text{x}, \quad \frac{\partial E}{\partial y} = 40\text{y}$$

Initial $[x,y]=[-1,1]$ or $[1,1]$

Weights Can Oscillate If Learning Rate Set Too High

Solution: calculate the cosine of the angle between successive weight vectors.

$$\cos\theta = \frac{\vec{\Delta} w(t) \cdot \vec{\Delta} w(t-1)}{\|\vec{\Delta} w(t)\| \cdot \|\vec{\Delta} w(t-1)\|}$$

If cosine close to 1, things are going well.

If cosine < 0.95, reduce the learning rate.

If cosine < 0, we're oscillating: cancel the momentum.

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

The “Herd Effect” (Fahlman)

Hidden units all move in the same direction at once, instead of spreading out to divide and conquer.

Solution: use initial random weights, not too large (to avoid flat spots), to encourage units to diversify.

Use a small initial learning rate to give units time to sort out their “specialization” before taking large steps in weight space.

Add hidden units one at a time. (Cascor algorithm.)

Cross-Entropy Error Measure

- Alternative to sum-squared error for binary outputs; diverges when the network gets an output completely wrong.

$$E = \sum_p \left[d^p \log \frac{d^p}{y^p} + (1-d^p) \log \frac{1-d^p}{1-y^p} \right]$$

- Can produce faster learning for some types of problems.
- Can learn some problems where sum-squared error gets stuck in a local minimum, because it heavily penalizes “very wrong” outputs.

How Many Layers Do We Need?

Two layers of weights suffice to compute any “reasonable” function.

But it may require a lot of hidden units!

Why does it work out this way?

Lapedes & Farmer: any reasonable function can be approximated by a linear combination of localized “bumps” that are each nonzero over a small region.

These bumps can be constructed by a network with two layers of weights.

Early Application of Backprop: From DECtalk to NETtalk

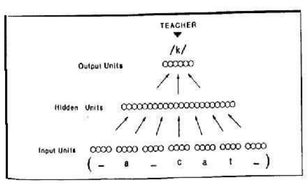
DECtalk was a text-to-speech program that drove a Votrax speech synthesizer board.

Contained 700 rules for English pronunciation, plus a large dictionary of exceptions.

Developed over several years by a team of linguists and programmers.

NETtalk Learns to Read

In 1987, Sejnowski & Rosenberg made national news when they used backprop to “teach” a neural network to “read aloud”.



Output: 23 phonetic feature units plus 3 for stress, syll. boundaries.

Hidden layer: 0-120 units.

Input: 7 letter window containing 7x29 = 206 units.

Training the network with 10,000 weights took 24 hours on a VAX-780 computer. (Today it would take a few minutes.)

Why Was NETtalk Interesting?

No explicit rules. No exception dictionary. Trained in less than a day. Programmers now obsolete!

NETtalk went through “developmental stages” as it learned to read. Analogous to child development?
- CV alternation: “babbling”
- word boundaries recognized: “pseudo-words”
- many words intelligible
- understandable text

(play audio)

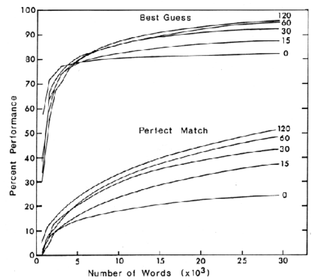
Graceful response to “damage” (some weights deleted, or noise added.) Rapid recovery with retraining. Analagous to human stroke patients?

Learning Curves for 0-120 Hidden Units

Training set was a 1000 word dictionary corpus; many irregular words.

No hiddens: 82% best guess.
120 hiddens: 98% best guess.

Errors in the no "hidden units" case were often inappropriate. Hidden units allow for more contextual influence by recognizing higher order features in the input.



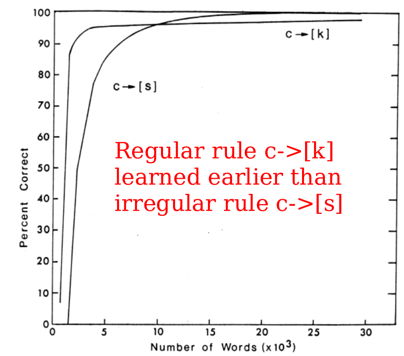
Test of Generalization Performance

Initial training: 1000 words, with 120 hidden units.

Testing set was a 20,012 word dictionary.

No additional training:
  77% best guess
  28% perfect match

After 5 training passes:
  90% best guess
  48% perfect match

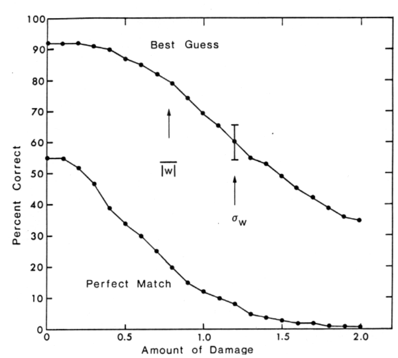


Effects of Damage

Std. dev. of the original, undamaged weights was 1.2

Random weight perturbations in [-.5,+.5] had little effect.

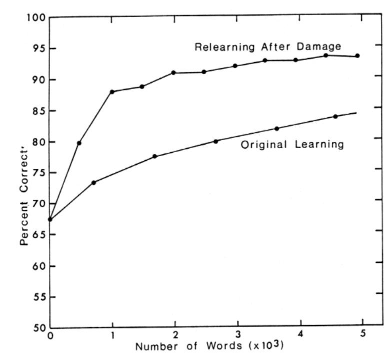
So each weight must convey only a few bits of information.



Relearning After Damage

Relearning was about 10 times faster to achieve similar performance.

Analogy to rapid recovery of language in stroke patients?



Was NETtalk Really Competitive?

Couldn't handle words with context-dependent pronunciations (“lead”) or stresses (“survey”).

Couldn't handle grammatical structure, e.g., questions vs. declarative sentences.

Lacked clever contextual tricks, such as:
- “he dove” vs. “the dove”
- “Dr. Smith” vs. “51 Rodeo Dr.”

But not bad for a seven letter window!