Learning Algorithms for Perceptrons

Linear and Nonlinear

1

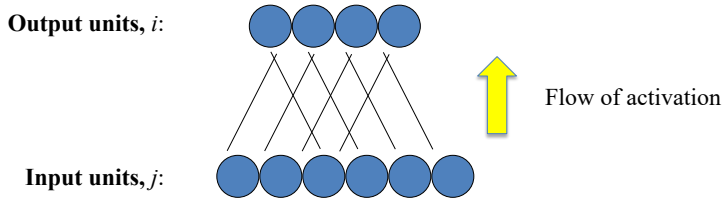
Hebbian Learning in a Linear Perceptron

2

A Simple (Linear) Perceptron

Linear: $a_i = n_i$

Hebbian Learning: $\Delta w_{ij} = a_i a_j$



3

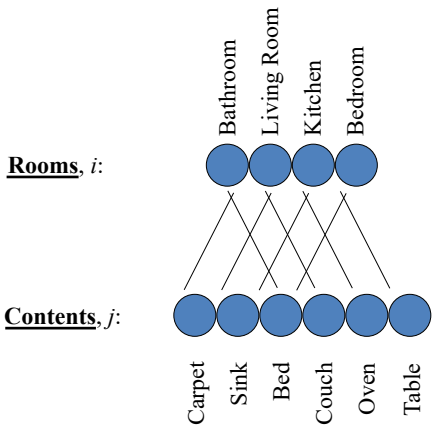
E.g., a neural net to learn to recognize rooms based on their contents

Contents:

Rooms:	Carpet	Sink	Bed	Couch	Oven	Table
Bathroom	0	0	1	0	0	0
Liv. Rm.	1	0	0	1	0	1
Kitchen	0	1	0	0	1	1
Bedroom	1	0	1	0	0	0

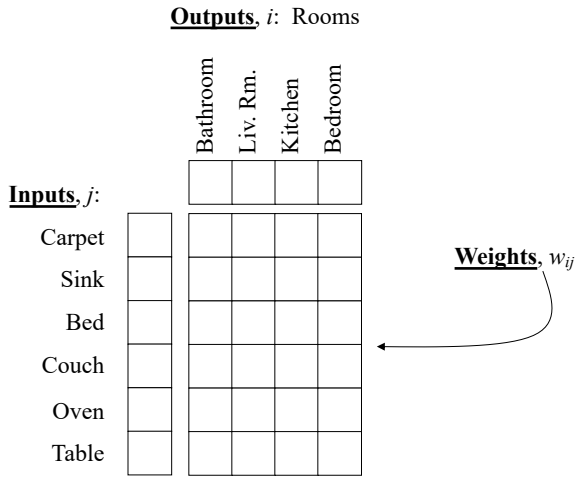
4

Network takes contents as inputs and activates rooms as outputs



5

As a matrix…



6

Train bathroom…

Outputs, *i*: Rooms

Inputs, *j*:		Bathroom	Liv. Rm.	Kitchen	Bedroom
		1	0	0	0
Carpet	0				
Sink	1	1			
Bed	0				
Couch	0				
Oven	0				
Table	0				

7

Test bathroom…

Outputs, *i*: Rooms

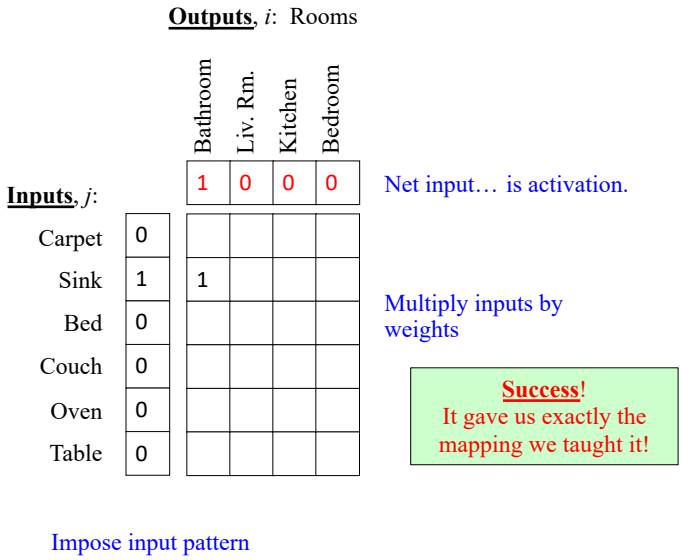
Inputs, *j*:		Bathroom	Liv. Rm.	Kitchen	Bedroom
Carpet	0				
Sink	1	1			
Bed	0				
Couch	0				
Oven	0				
Table	0				

Multiply inputs by weights

Impose input pattern

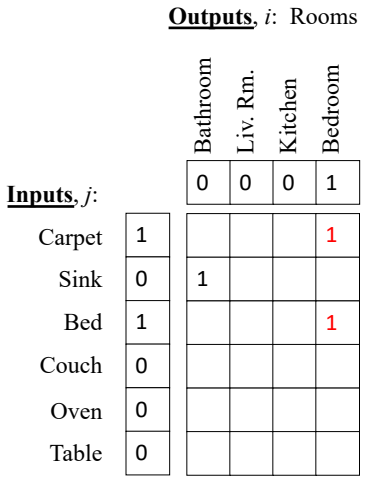
8

Test bathroom…



9

Train bedroom…



10

Test bedroom…

Outputs, *i*: Rooms

Inputs, *j*:		Bathroom	Liv. Rm.	Kitchen	Bedroom
		0	0	0	2
Carpet	1				1
Sink	0	1			
Bed	1				1
Couch	0				
Oven	0				
Table	0				

Hmmm…

11

Train living room…

Outputs, *i*: Rooms

Inputs, *j*:		Bathroom	Liv. Rm.	Kitchen	Bedroom
		0	1	0	0
Carpet	1		1		1
Sink	0	1			
Bed	0				1
Couch	1		1		
Oven	0				
Table	1		1		

12

Test living room…

Outputs, *i*: Rooms

Inputs, *j*:		Bathroom	Liv. Rm.	Kitchen	Bedroom
		0	3	0	1
Carpet	1		1		1
Sink	0	1			
Bed	0				1
Couch	1		1		
Oven	0				
Table	1		1		

Uh oh…

13

Train kitchen…

Outputs, *i*: Rooms

Inputs, *j*:		Bathroom	Liv. Rm.	Kitchen	Bedroom
		0	0	1	0
Carpet	0		1		1
Sink	1	1		1	
Bed	0				1
Couch	1		1		
Oven	1			1	
Table	1		1	1	

14

Test kitchen…

Outputs, *i*: Rooms

Inputs, *j*:		Bathroom	Liv. Rm.	Kitchen	Bedroom
		1	1	3	0
Carpet	0		1		1
Sink	1	1		1	
Bed	0				1
Couch	1		1		
Oven	1			1	
Table	1		1	1	

Pretty failish.

15

When will a linear perceptron trained with Hebbian learning *work*?

Where *work* means *give you as output the patterns you trained it to associate with its inputs*.

1) When input patterns that map to different output patterns are *orthogonal*.

Outputs, *i*: Rooms

Inputs, *j*:		Bathroom	Liv. Rm.	Kitchen	Bedroom
Carpet					1
Sink		1			
Bed					1
Couch					
Oven					
Table					

The weight matrix after training both Bathroom and Bedroom.

Note that

Bathroom: 0 1 0 0 0 0
and
Bedroom: 1 0 1 0 0 0

Are orthogonal

16

When will a linear perceptron trained with Hebbian learning *work*?

Where *work* means *give you as output the patterns you trained it to associate with its inputs*.

1) When input patterns that map to different output patterns are *orthogonal*.

Outputs, *i*: Rooms

Inputs, *j*:		Bathroom	Liv. Rm.	Kitchen	Bedroom
		1	0	0	0
Carpet	0				1
Sink	1	1			
Bed	0				1
Couch	0				
Oven	0				
Table	0				

The weight matrix after training both Bathroom and Bedroom.

Test with Bathroom

Output is exactly as trained: Perfect performance.

17

When will a linear perceptron trained with Hebbian learning *work*?

Where *work* means *give you as output the patterns you trained it to associate with its inputs*.

1) When input patterns that map to different output patterns are *orthogonal*.

2) When input patterns are of unit length (sqrt(sum of squared values) = 1).

Outputs, *i*: Rooms

Inputs, *j*:		Bathroom	Liv. Rm.	Kitchen	Bedroom
		1	0	0	0
Carpet	0				1
Sink	1	1			
Bed	0				1
Couch	0				
Oven	0				
Table	0				

Note that ||Bathroom|| = 1

Output is exactly as trained: Perfect performance.

18

When will a linear perceptron trained with Hebbian learning *work*?

Where *work* means *give you as output the patterns you trained it to associate with its inputs*.

1) When input patterns that map to different output patterns are *orthogonal*.

2) When input patterns are of unit length (sqrt(sum of squared values) = 1).

Outputs, *i*: Rooms

Inputs, *j*:		Bathroom	Liv. Rm.	Kitchen	Bedroom
		0	0	0	2
Carpet	0				1
Sink	1	1			
Bed	0				1
Couch	0				
Oven	0				
Table	0				

Test with Bedroom

Note that ||Bedroom|| = root(2)

Output is *not* exactly as trained:

Imperfect performance.

19

When will a linear perceptron trained with Hebbian learning *work*?

Where *work* means *give you as output the patterns you trained it to associate with its inputs*.

1) When input patterns that map to different output patterns are *orthogonal*.

2) When input patterns are of unit length (sqrt(sum of squared values) = 1).

Outputs, *i*: Rooms

Inputs, *j*:		Bathroom	Liv. Rm.	Kitchen	Bedroom
		1	1	3	0
Carpet	0		1		1
Sink	1	1		1	
Bed	0				1
Couch	1		1		
Oven	1			1	
Table	1		1	1	

And things only got worse as we trained more patterns.

20

When will a linear perceptron trained with Hebbian learning *work*?

Where *work* means *give you as output the patterns you trained it to associate with its inputs*.

1) When input patterns that map to different output patterns are *orthogonal*.
2) When input patterns are of unit length (sqrt(sum of squared values) = 1).

Freakishly stringent criteria.

Can we do better?

Variable learning rate? $\Delta w_{ij} = \varepsilon a_i a_j$ No help.

Nonlinear activation function combined with lateral inhibition between output units? *Now* you're talkin'. But that's not feed-forward. (Not a perceptron.)

First…

21

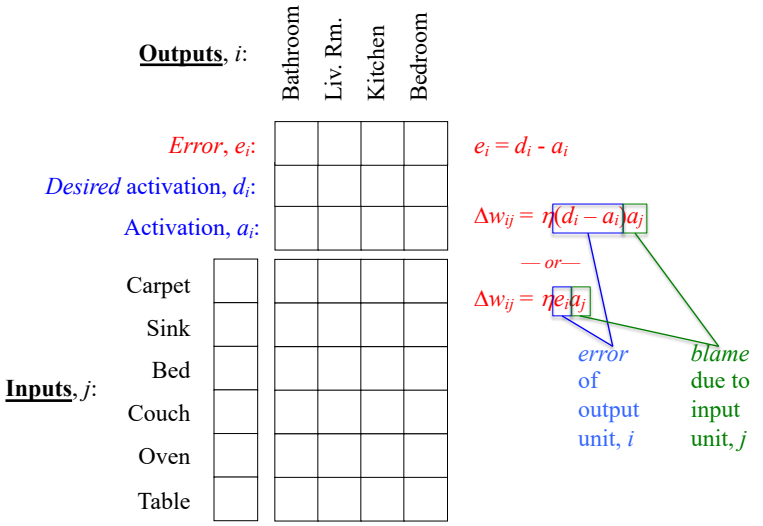
Error-Correction Learning in a Linear Perceptron using the Widrow-Hoff (1960) Algorithm

(Formally equivalent to Rescorla-Wagner learning algorithm. The conceptual basis of error back-propagation.)

22

Widrow-Hoff (1960) Error-Correction Learning

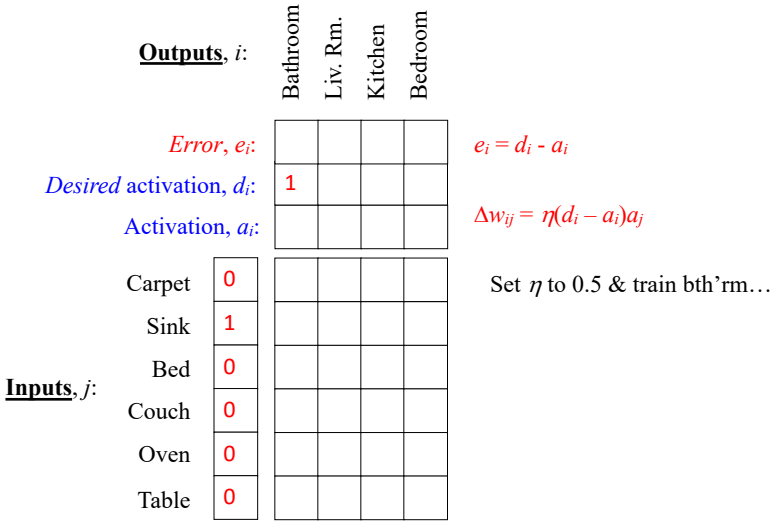
Basic Idea: Change in weights proportional, not to *coactivation*, but to *error*.



23

Widrow-Hoff (1960) Error-Correction Learning

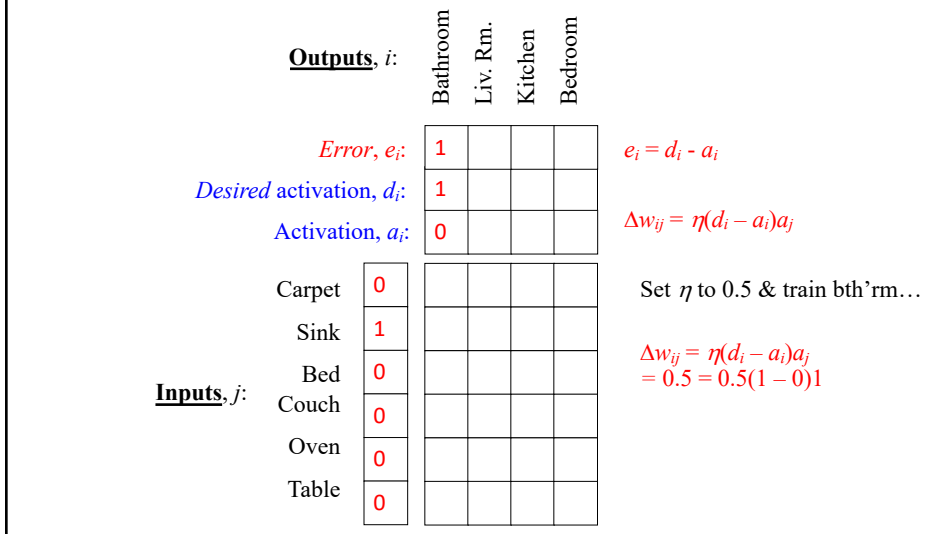
Basic Idea: Change in weights proportional, not to *coactivation*, but to *error*.



24

Widrow-Hoff (1960) Error-Correction Learning

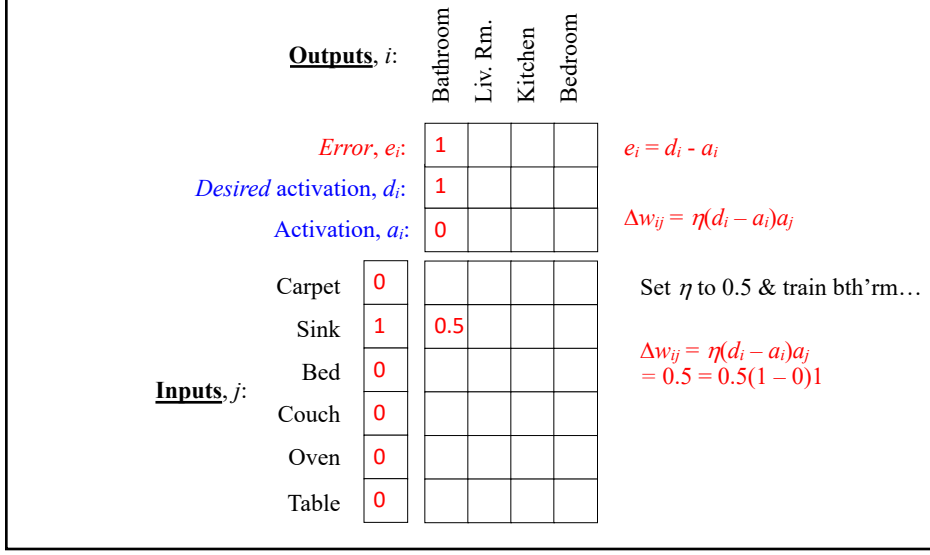
Basic Idea: Change in weights proportional, not to *coactivation*, but to *error*.



25

Widrow-Hoff (1960) Error-Correction Learning

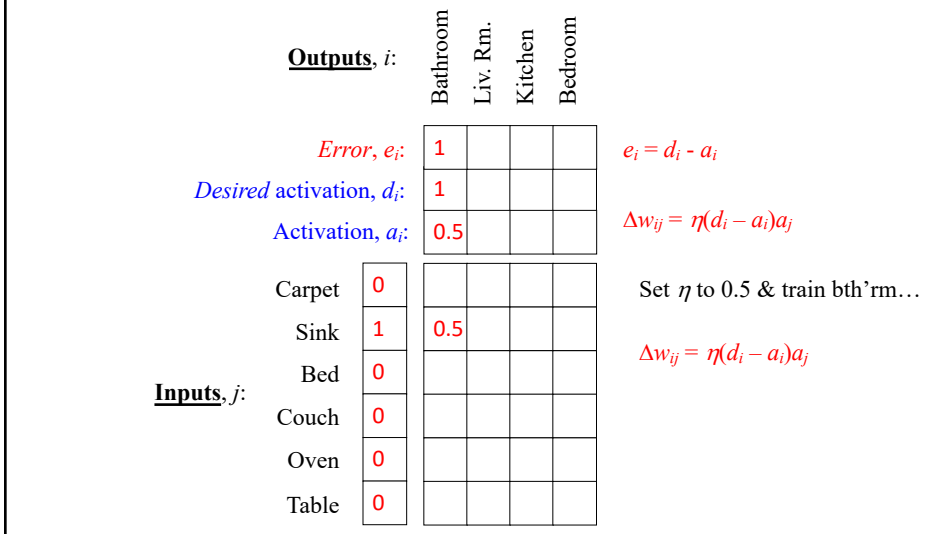
Basic Idea: Change in weights proportional, not to *coactivation*, but to *error*.



26

Widrow-Hoff (1960) Error-Correction Learning

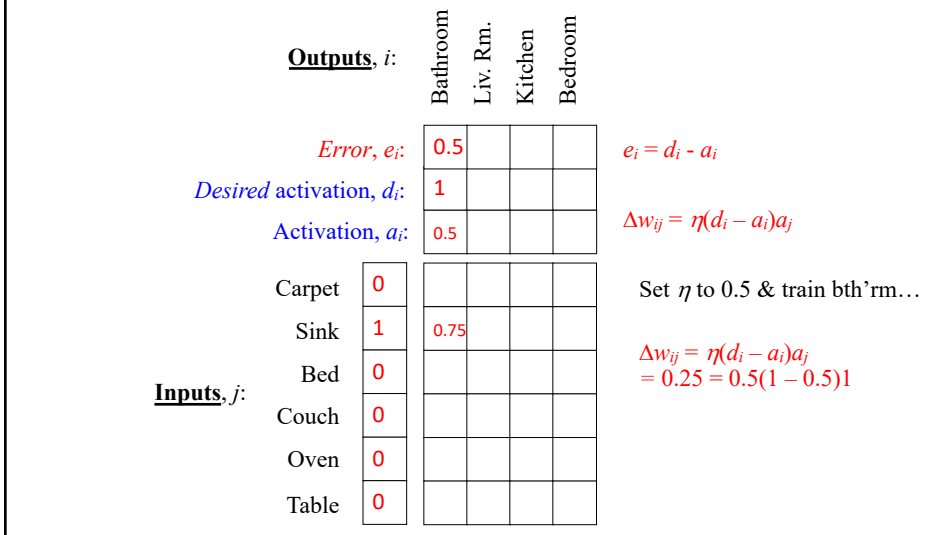
Basic Idea: Change in weights proportional, not to *coactivation*, but to *error*.



27

Widrow-Hoff (1960) Error-Correction Learning

Basic Idea: Change in weights proportional, not to *coactivation*, but to *error*.



28

Widrow-Hoff (1960) Error-Correction Learning

Basic Idea: Change in weights proportional, not to *coactivation*, but to *error*.

		Outputs, *i*: Bathroom	Liv. Rm.	Kitchen	Bedroom
	Error, e_i:	0			
	Desired activation, d_i:	1			
	Activation, a_i:	1.0			
Inputs, *j*: Carpet	0				
Sink	1	1.0			
Bed	0				
Couch	0				
Oven	0				
Table	0				

$e_i = d_i - a_i$

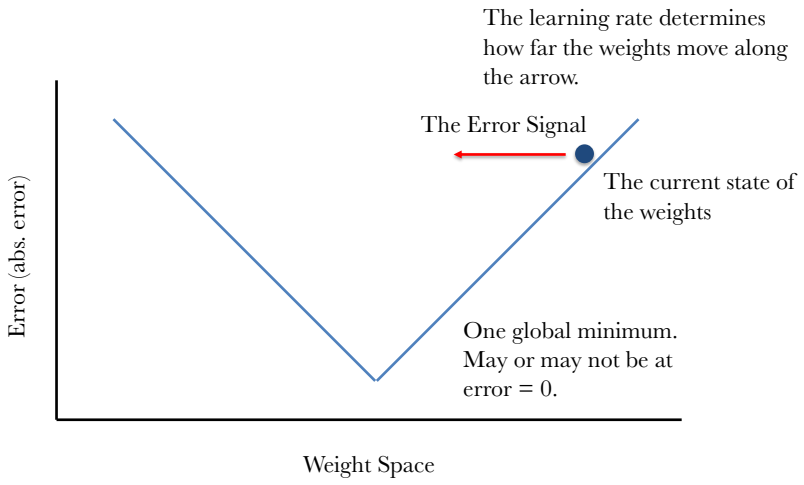
$\Delta w_{ij} = \eta(d_i - a_i)a_j$

Set η to 0.5 & train bathr'm…

At asymptote…

29

The Error Landscape of a Widrow Hoff Network (absolute rather than squared error)



30

Conclusion 1: Widrow-Hoff is Awesome!

- Yes, it's lovely, but um…
- It only works for *linear*, one-layer perceptrons.
- And any *n*-layer linear percepton is formally equivalent to some one-layer linear perceptron.
- In other words, no linear perceptron can be any more powerful than a one-layer linear perceptron.
- And we know that there are some mappings (like XOR, or the middle class) that cannot be computed by a one-layer perceptron.

31

Conclusion 2: Widrow-Hoff Sucks!

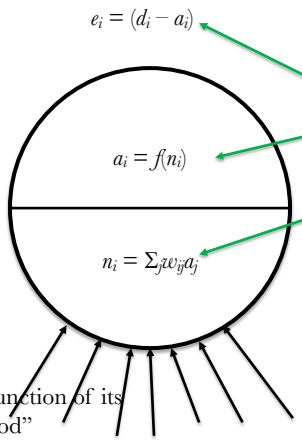
- Not so fast.
- What if we could generalize it to work with a nonlinear activation function in a multi-layer perceptron with hidden units?
- Well, that activation function would have to be differentiable, and the BTU is not!
- Plus, Widrow-Hoff is error-based.
  - Error is the difference between a unit's *desired* activation and its *actual* activation: $e_i = (d_i - a_i)$:
  - What's the "desired" activation of a hidden unit?
- Damn. You make some excellent points. Let me think about it for a quarter of a century or so…

32

The Activation Function Would Have to be Differentiable.

- Huh?
- Think about it.

We must therefore correct the error term by multiplying it by the derivative of the activation function.



Here's a neuron

Here's its net input

which comes over a bunch of weighted connections.

Here's its activation, which is some function of its input

Here's it's error, which is a function of its activation and the "will of god"

And here's the problem:

The change in the weights is based on error,

which is based on activation.

But changing the weights directly affects the *input*,

not the activation!

And the input differs from the activation because the activation function is nonlinear! (Oh shit!)

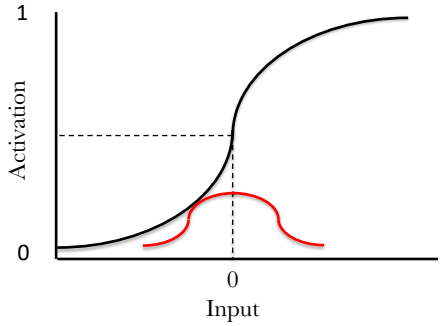
33

We must correct the error term by multiplying it by the derivative of the activation function.

- First, we need a differentiable nonlinear activation function.
- A convenient one (there are others) is the *logistic function*:

$$a_i = \frac{1}{1 + e^{-n_i}}$$

- whose derivative is simply $a_i(1 - a_i)$.

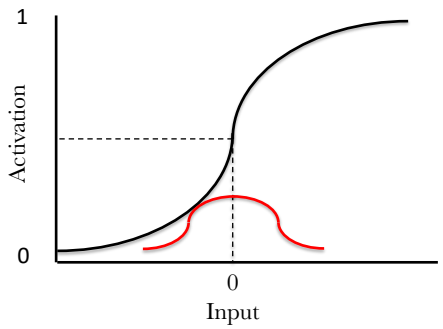


Please pardon the bad graphical approximation of the derivative.

34

We must correct the error term by multiplying it by the derivative of the activation function.

- What do we do with this knowledge?
- When computing error, multiply the raw error term, $e_i = (d_i - a_i)$…
- … by the derivative of the activation function, to get:
- $e_i = (d_i - a_i)a_i(1 - a_i)$
- *This* is the error term you will use to update the weights.



26 years of blood, sweat, tears, and suffering, all delivered to you in a few seconds' time.

You're welcome.

But we're not done.

35

We must correct the error term by multiplying it by the derivative of the activation function.

But care is required!

(you guys are *so* gonna forget this)

Let's talk about *kinds* of error:

1. There's the kind of error you use to change the connection weights:
   - This is $e_i = (d_i - a_i)a_i(1\text{-}a_i)$
   - This kind of error affects the inner workings of the network itself.
   - This measure of error is part of the learning algorithm.
   - This measure of error must include the derivative of the activation function.

2. And then there's the kind of error you use to *evaluate* the network:
   - This is $E_i = (d_i - a_i)^2$ for unit i and $E = \Sigma_i E_i/n$ for the network as a whole.
   - This error is an emergent phenomenon of the network as a whole.
   - This measure *describes* the network but does not affect its behavior.
   - This measure must *not* include the derivative of the activation function.

36

And don't forget our other problem:
What's the "desired" activation of a hidden unit?

- Oh shit.
- No kidding.
- The problem: You *don't know* what the hidden units should be doing, so how can you tell them what their error is?
- The answer: Propagate error "backward" from higher layers down to lower layers.
- The intuition: We can "blame" the error of any unit, *i*, in a higher layer on a unit, *j*, in the previous layer to the extent that *j* has a strong connection to *i*.
- The approach: Let the error of any "lower" unit, *j*, be the sum of all "higher" units', *i*, errors times the weights from *j* to *i*.
- To a first approximation, this means:

$$e_j = \sum_i w_{ij}\, e_i$$

- Watch the subscripts here. You *will* fuck this up. You have been warned.
- And don't forget to correct by the derivative of the activation function:

$$e_j = \sum_i w_{ij}\, e_i a_j(1-a_j)$$

- *Keep The Subscripts Straight.*

37

That's the hard part.

Armed with all that, everything else is easy and familiar.

Mostly:

- This algorithm is persnickety.
- The error landscape is a nightmare.
- Multiplying by the derivative makes training nets with more than a few layers functionally impossible.

- But the algorithm is *so sexy!*

- Surely there are ways to save it.

- There are tricks to make it work better.

- "You wanna get rich and famous?" Improve those algorithms.

- Challenge: Back propagation will never be able to explain how humans invented back propagation.

38

How to Run a Back Propagation Network

1. **Construct the network**
   - How many layers of units?
   - How many units in each layer?
   - Fully interconnect units in adjacent layers. *Randomize* the connection weights.
2. **Train the network**
   - Repeat until global error < threshold:
     - For each pattern in the training set:
       - Impose a pattern of activation in the input layer
       - Propagate activation forward
       - Calculate error at the output layer
       - Propagate error backward
       - Calculate weight changes based on error
     - Change the weights
3. **Test the network**
   - Test with trained patterns
   - Test with untrained patterns

39

Dark Magic

1. **Momentum**
   - Problem: Error landscape is highly irregular (unlike Widrow-Hoff):
     - Many local minima, ridges, valleys, etc.
   - Solution: *Momentum*
     - On each iteration of training, apply a fraction of the change you applied last iteration, in addition to applying the changes dictated by the error on this iteration
2. **Batch Updating of Weights**
   - Problem: Catastrophic forgetting
     - The last thing the network learns overwrites everything it learned before
   - Solution: Batch updating
     - Add up the weight changes due to each training pattern and apply them all at once, at the end. In this way, no training pattern comes "before" or "after" any other
3. **Bias Node**
   - Problem: Connection weights correspond to conditional probabilities, $p(i|j)$, bit nothing captures the base rate probability, $p(i)$, that unit i should be active
   - Solution: Bias node
     - A "node", $j = b$, in each layer (except the last) that is always active
     - The connection weight w_{ib}, representing $p(i|b)$, captures $p(i)$

Much more to come: Convolution nets, Recurrent nets, Deep nets…

40