The backpropagation algorithm

Andrew M. Saxe

Experimental Psychology
University of Oxford

Backpropagation from 30,000ft

- Learning algorithm for arbitrary, deep, complicated neural networks

- You've used it!
  - Google/Microsoft/Yahoo voice recognition, image search, machine translation, …

- The core idea behind many psychology & neuroscience models

Backpropagation from 30,000ft

- Can be written down in one line of math:

$$\Delta W = -\lambda \frac{\partial E}{\partial W}$$

- Sort of like Newton's laws
  - Three simple equations
  - But endless implications and consequences

- Need to understand it *intuitively* at *many levels*

Two meanings

- "Backprop" technically refers to a specific *algorithm*
- But often used as shorthand for a much broader *framework* with a galaxy of associated ideas and commitments
- Important not to confuse *backprop-the-algorithm* with *backprop-the-framework*—this has caused trouble in the past
- Your job is to learn backprop-the-framework

Levels of analysis

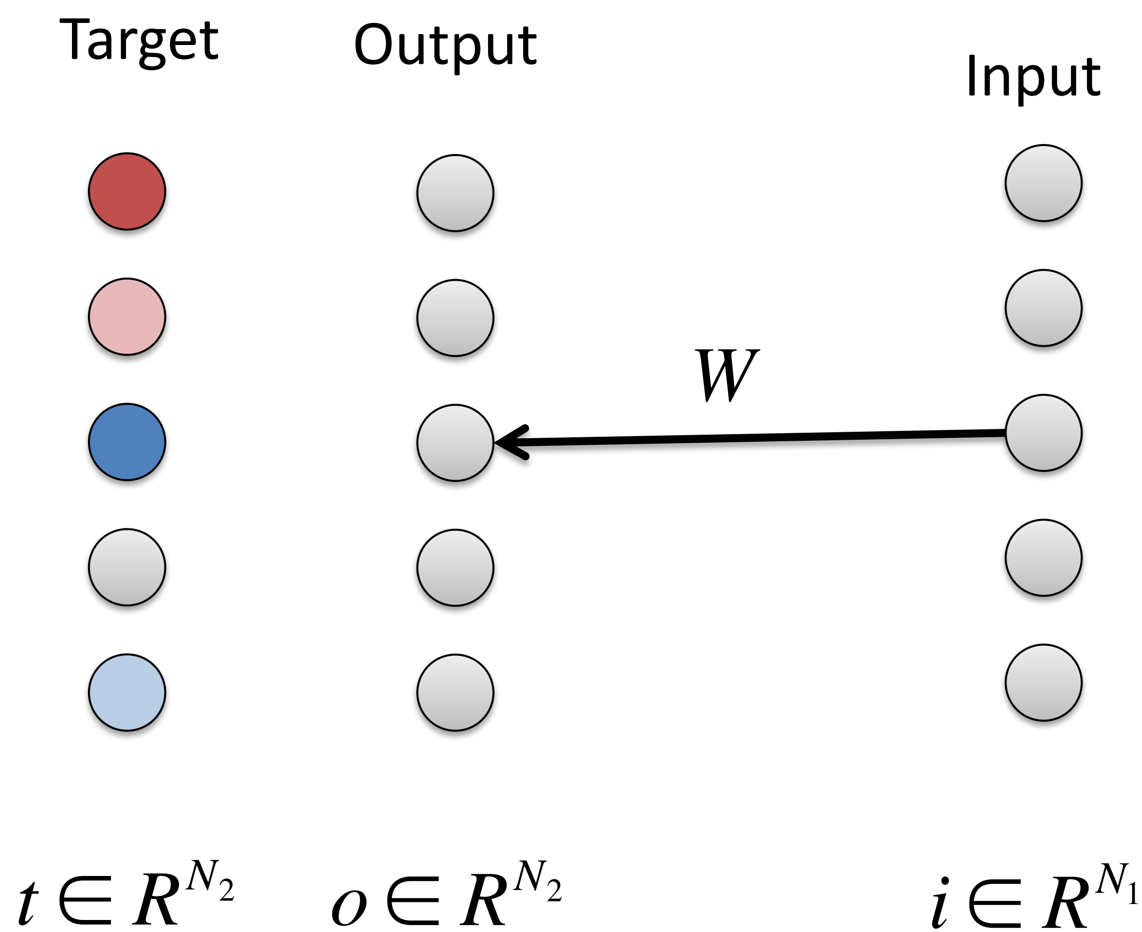
- Computational level
  - Learning as optimization
  - Gradient descent
- Algorithmic level
  - Backprop-the-algorithm
- Implementation level

Today

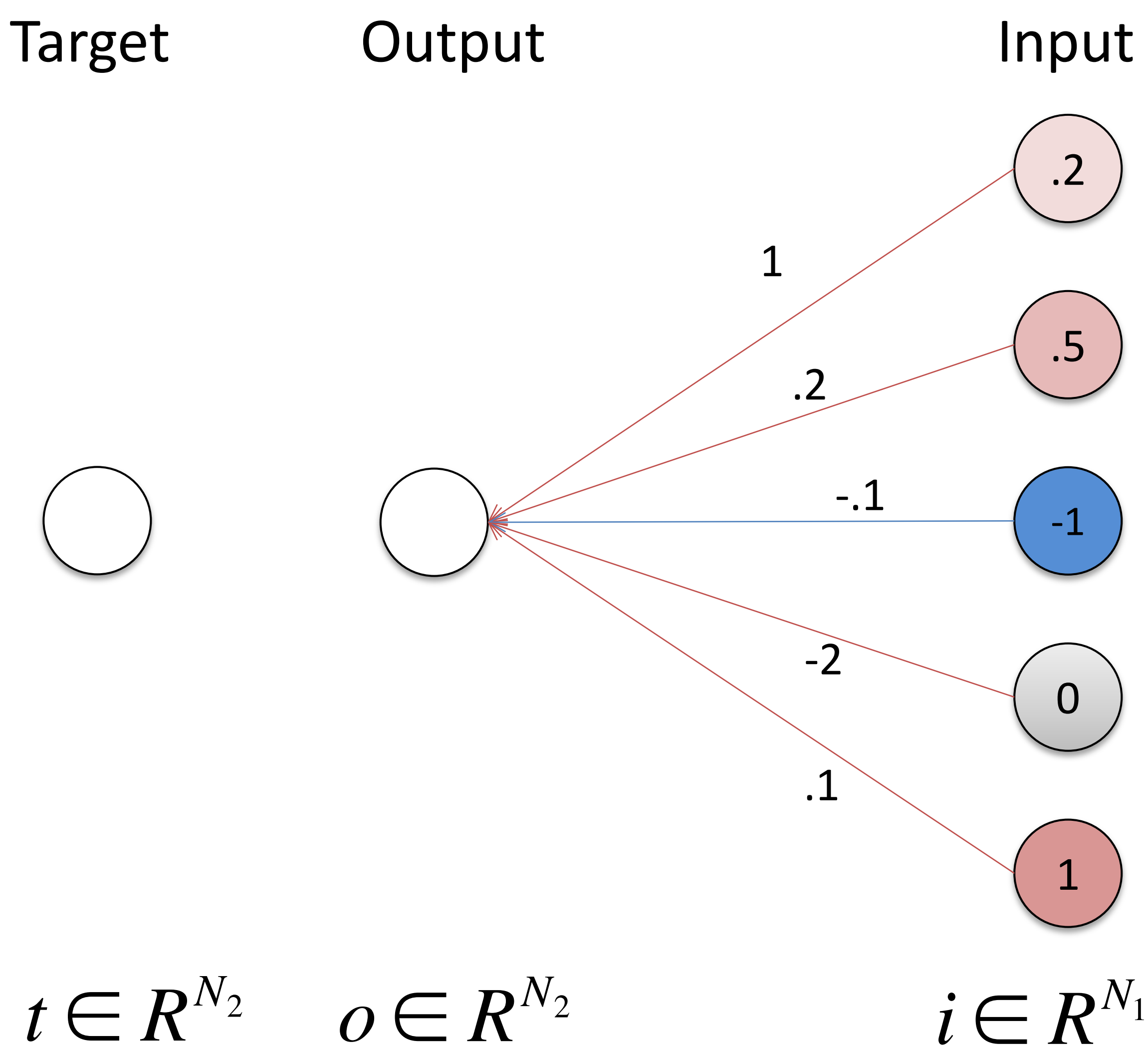
- Build up to learning in arbitrary deep network

1. One parallel layer
2. Many serial layers
3. Nonlinearities
4. The general case

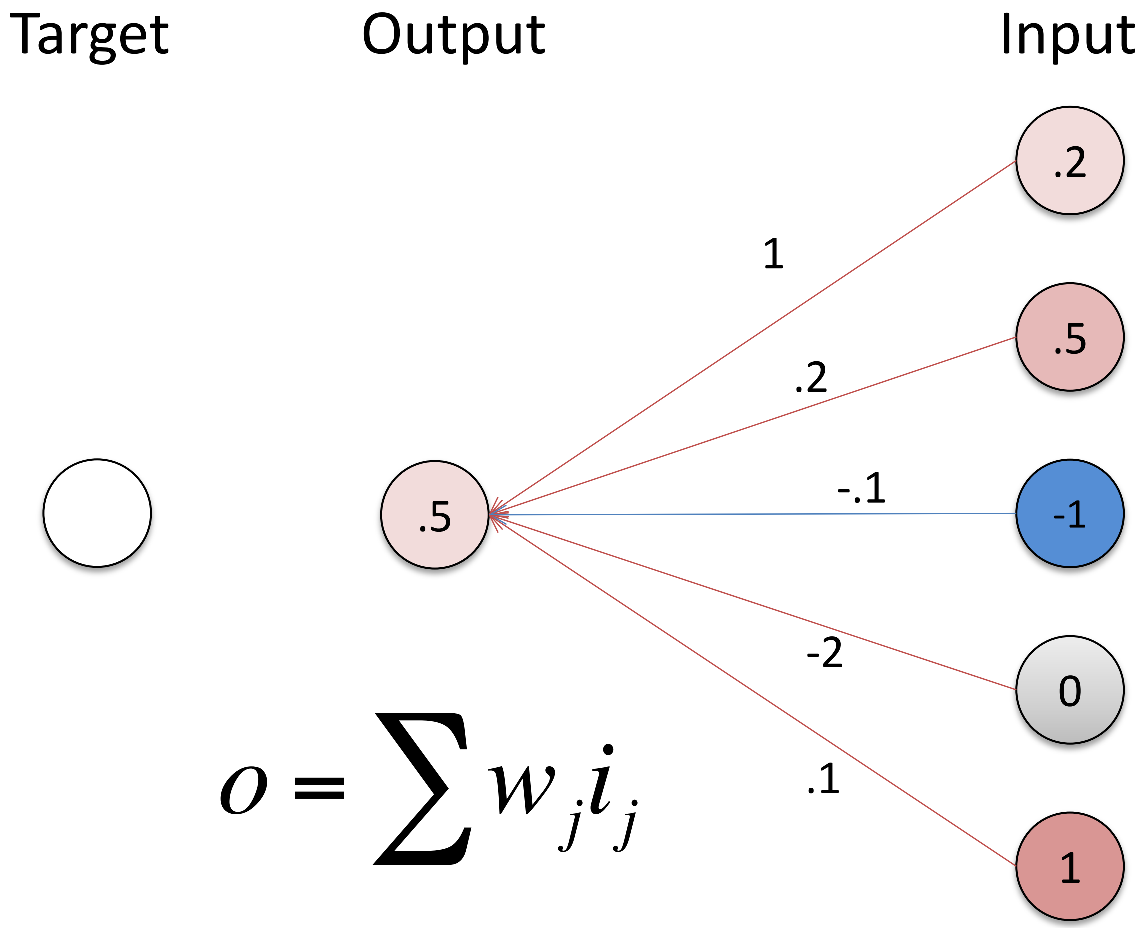
The pattern associator



One layer learning

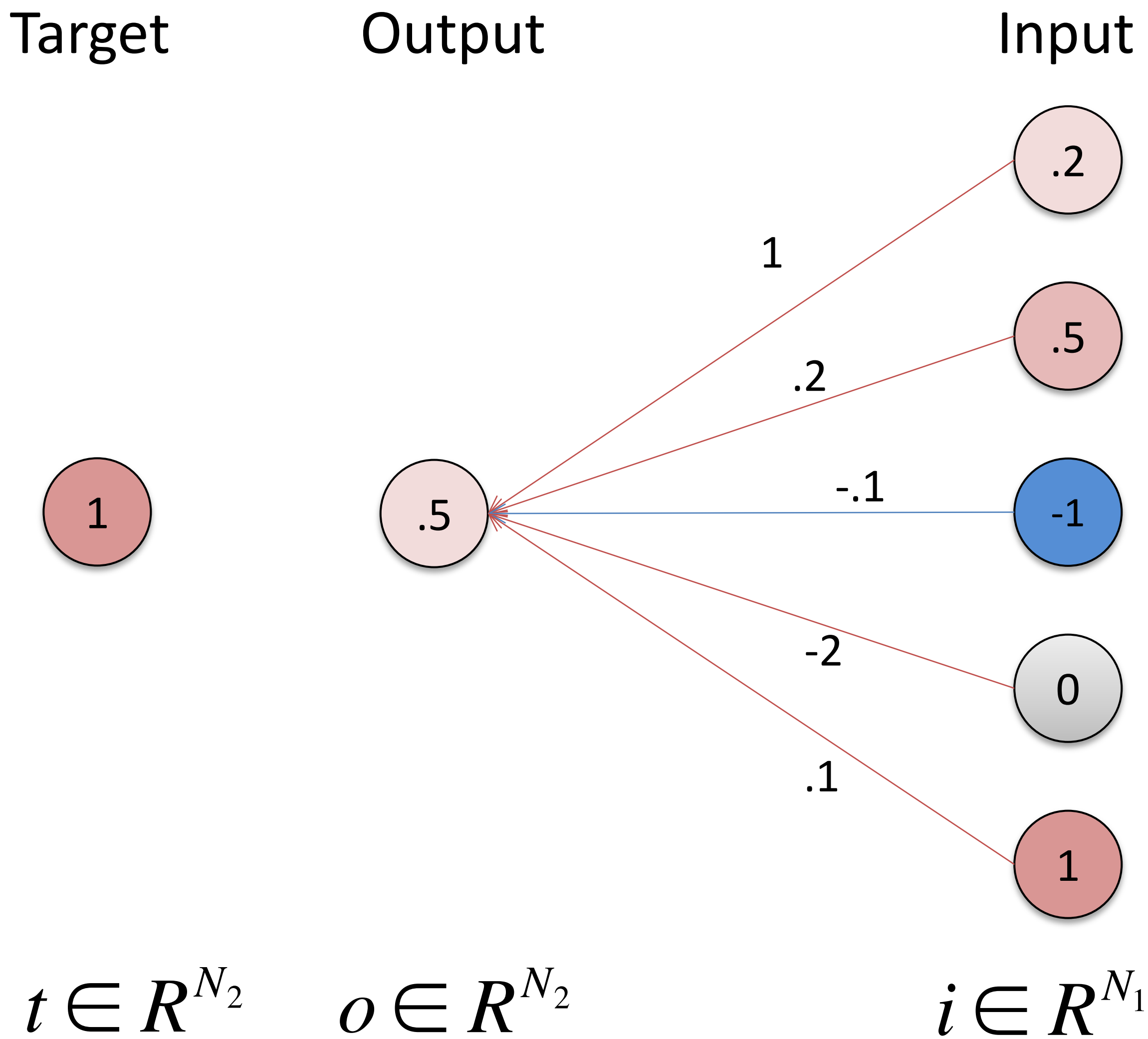


Forward propagation

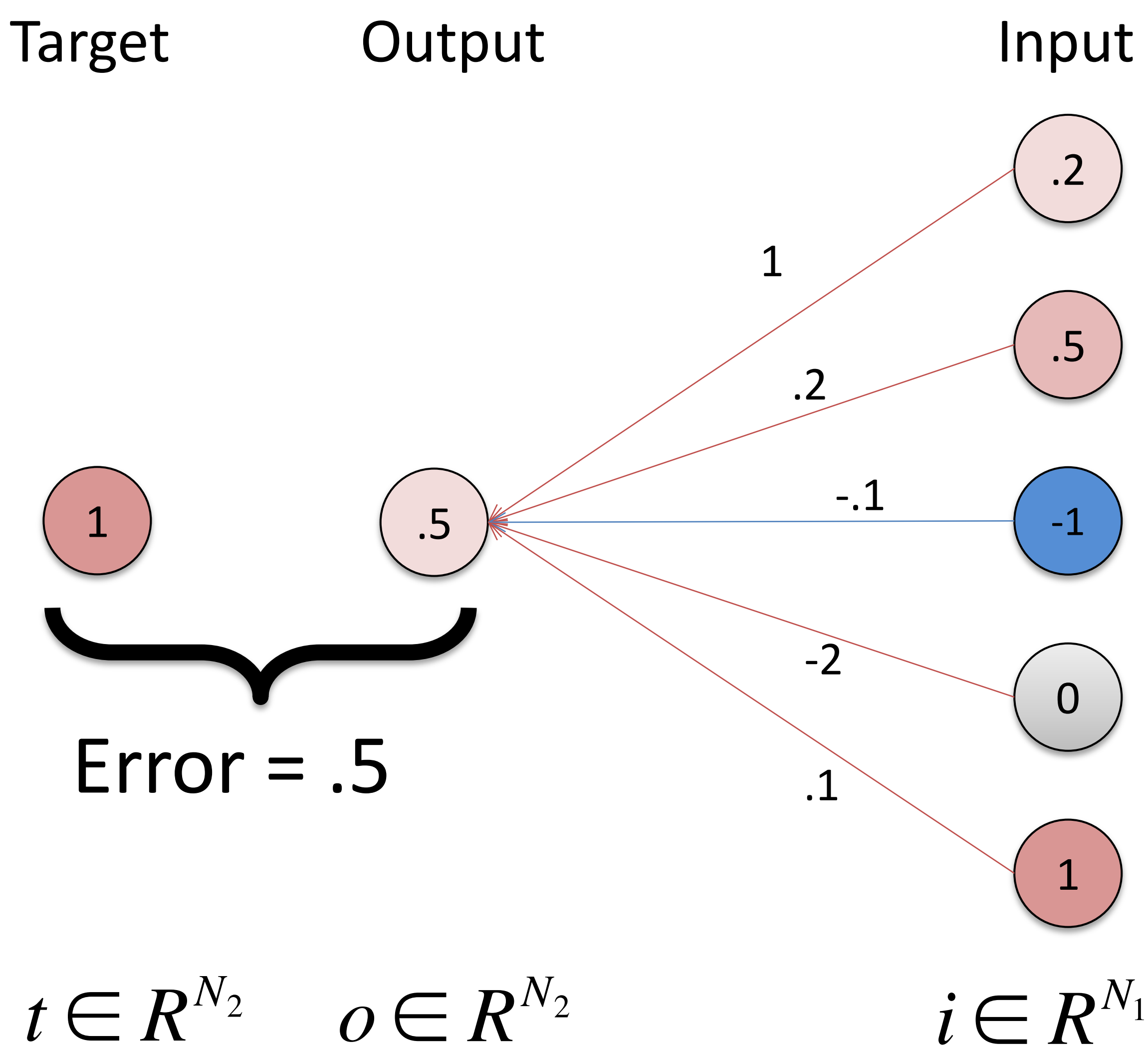


$$o = \sum w_j i_j$$

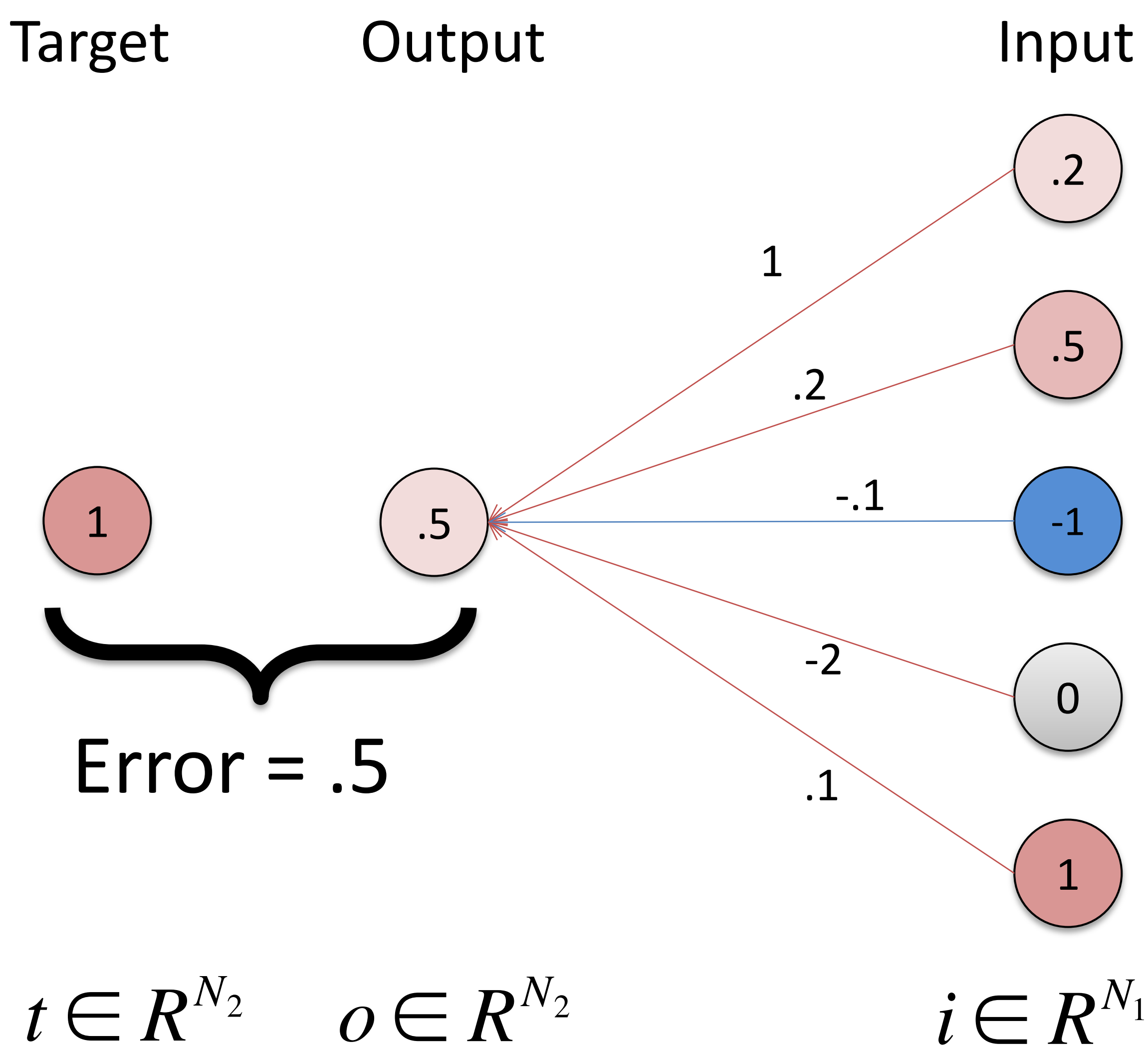
Target Feedback



Error signal

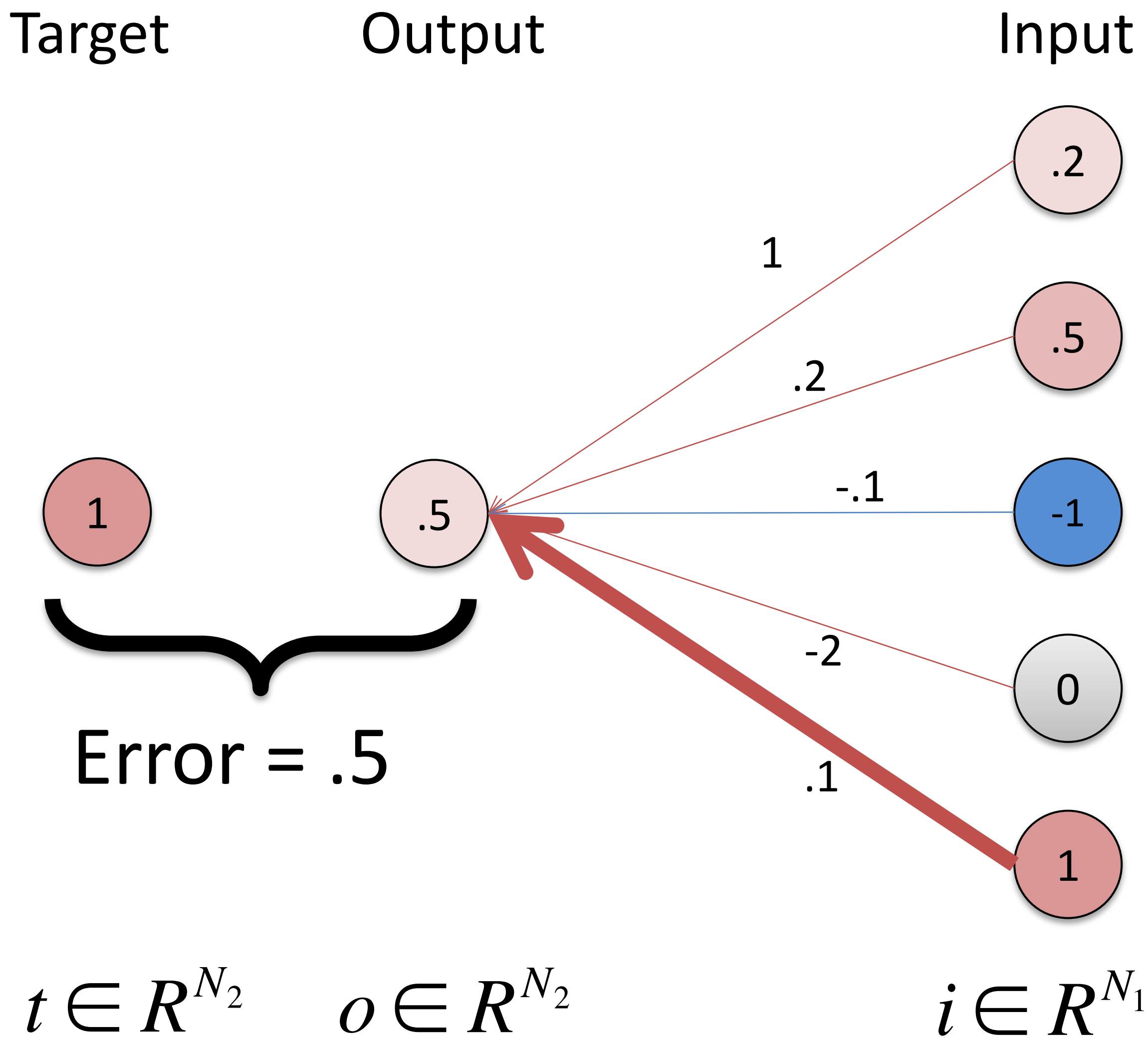


How to fix?



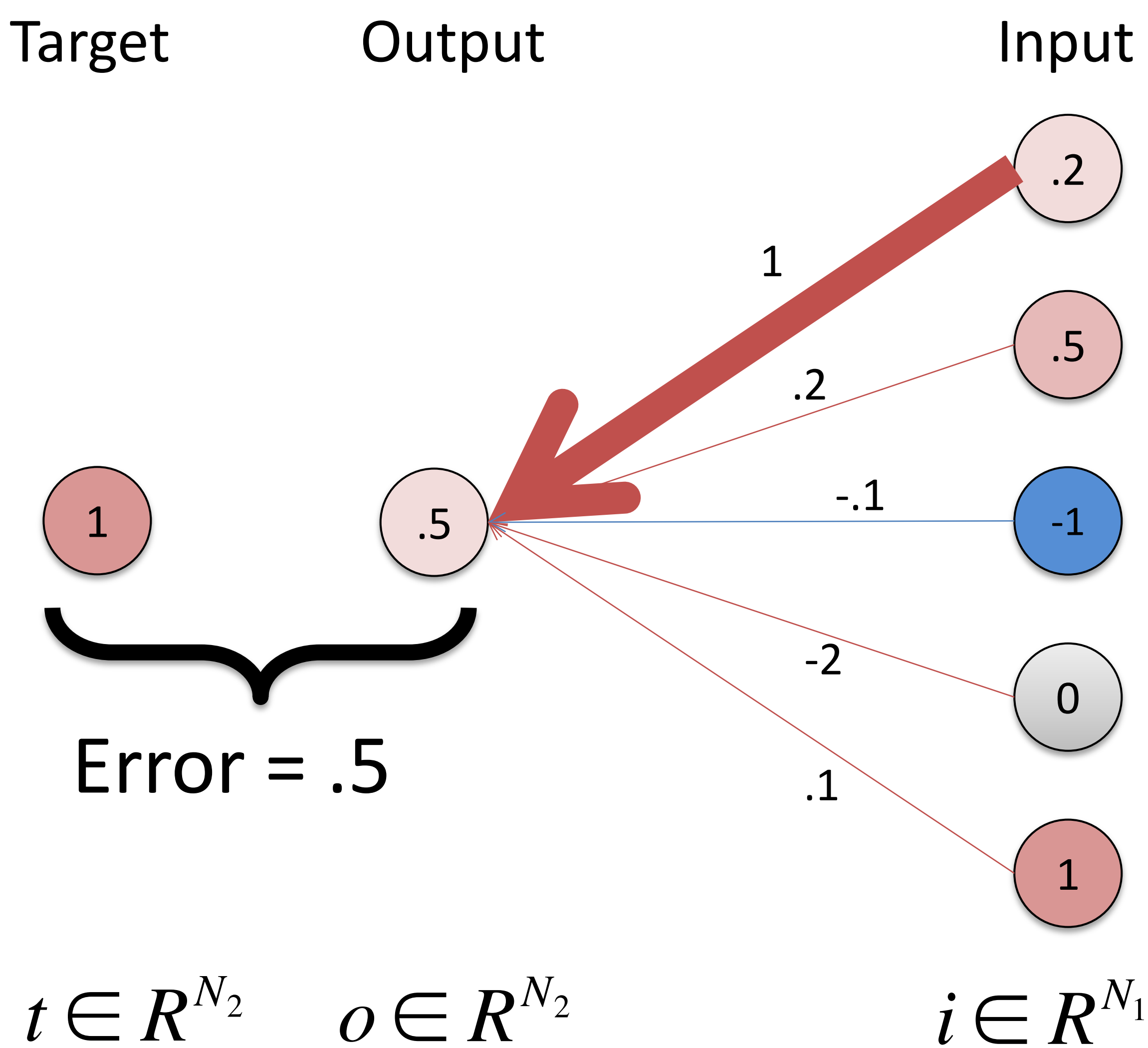
Target
Output
Input
1
.5
.2
.5
-1
0
1
1
.2
-.1
-2
.1
Error = .5
$t \in R^{N_2}$
$o \in R^{N_2}$
$i \in R^{N_1}$

How to fix?



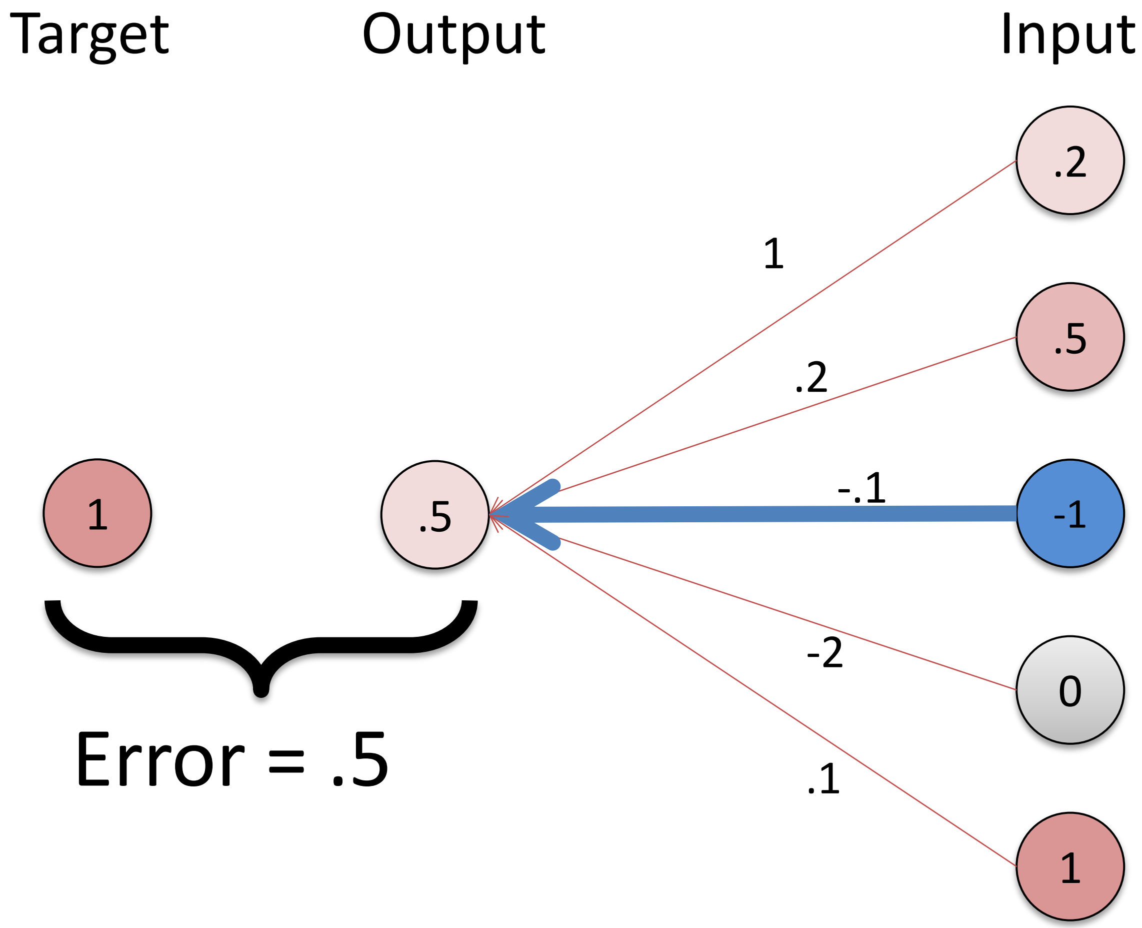
Target
Output
Input
1
.5
.2
.5
-1
0
1
1
.2
-.1
-2
.1
Error = .5
$t \in R^{N_2}$
$o \in R^{N_2}$
$i \in R^{N_1}$

How to fix?



Target
Output
Input
.2
1
.5
.2
-.1
1
.5
-1
-2
0
.1
1
Error = .5
$t \in R^{N_2}$
$o \in R^{N_2}$
$i \in R^{N_1}$

How to fix?



$t \in R^{N_2}$ $o \in R^{N_2}$ $i \in R^{N_1}$

Lots of choices

- How can we choose?

- Maybe: make small change that *most rapidly* decreases the error

Lots of choices

- How can we choose?
- Maybe: make small change that *most rapidly* decreases the error
- This is called Gradient Descent

Gradient Descent Learning

- Make small change in weights that most rapidly improves task performance

- Change each weight in proportion to the gradient of the error $\Delta W = -\lambda \frac{\partial E}{\partial W}$

Optimization view of learning

- The network and training data together define an *error function*, say, squared error

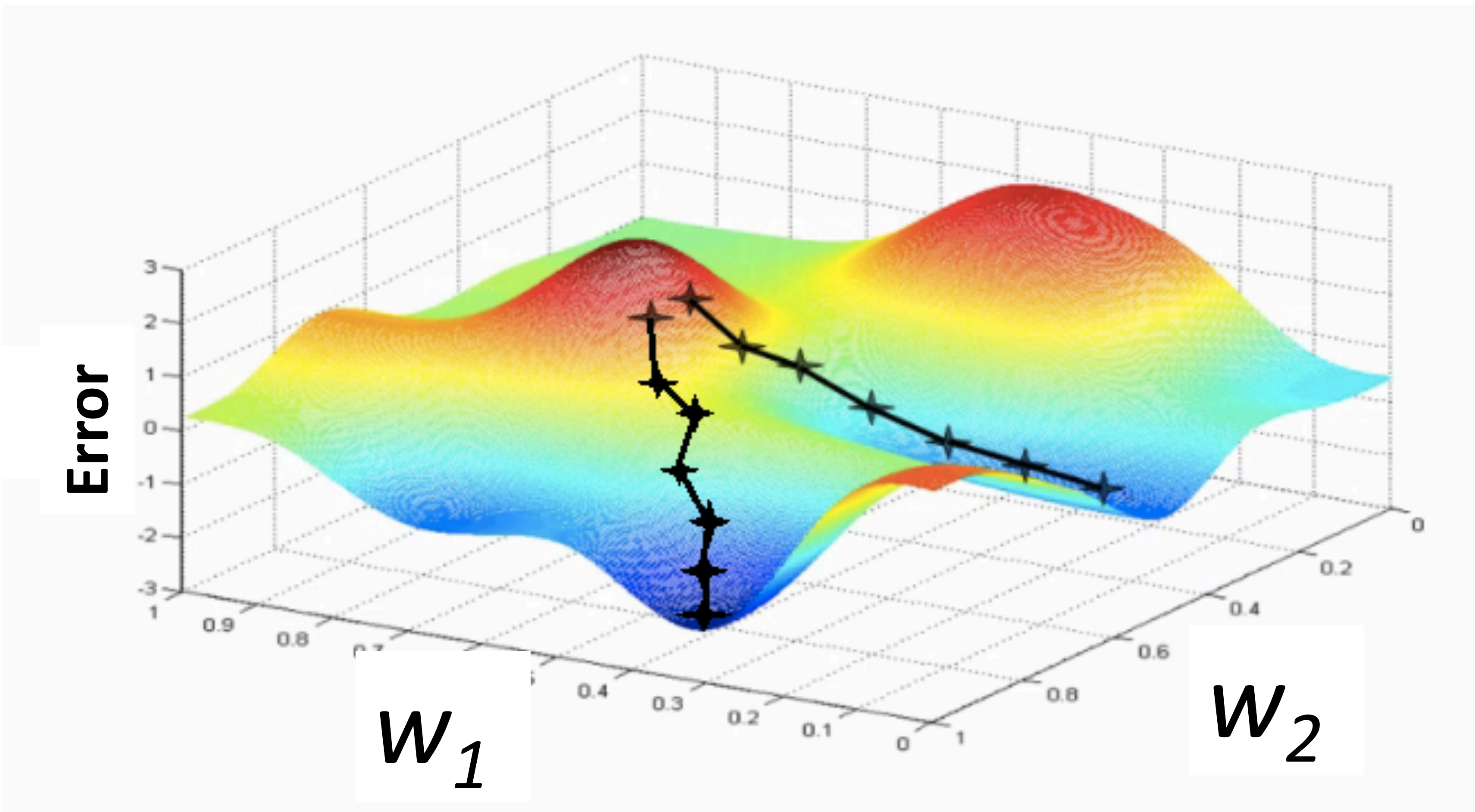
$$E(t,w,i) = (t-o)^2$$

- *Learning* is just minimizing this function w.r.t. w

$$w^* = \mathrm{argmin}_w\, E(t,w,i)$$

Gradient Descent

$$\Delta W = -\lambda \frac{\partial E}{\partial W}$$



http://blog.datumbox.com/wp-content/uploads/2013/10/gradient-descent.png

Delta rule derivation

$$E = (t - o)^2$$

$$= (t - \sum w_j i_j)^2$$

$$= (t - wi)^2$$

Row vector

Delta rule derivation

$$E = (t - o)^2$$

$$= (t - \sum w_j i_j)^2$$

$$= (t - wi)^2$$

$$\frac{\partial E}{\partial w_k} = -2(t - o)\frac{\partial E}{\partial w_k} o$$

$$= -2(t - o)\sum \frac{\partial E}{\partial w_k} w_j i_j$$

$$= -2(t - o) i_k$$

Delta rule derivation

$$E = (t - o)^2$$
$$= (t - \sum w_j i_j)^2$$
$$= (t - wi)^2$$

$$\frac{\partial E}{\partial w_k} = -2(t - o)\frac{\partial E}{\partial w_k} o$$
$$= -2(t - o)\sum \frac{\partial E}{\partial w_k} w_j i_j$$
$$= -2(t - o) i_k$$

Delta rule: $\Delta w = \lambda (t - o) i^T = \lambda e i^T$

What about *deep* networks?

- Delta rule only makes sense for one layer!

- How can we learn in deep, multilayer networks?

What about *deep* networks?

- Delta rule only makes sense for one layer!
- How can we learn in deep, multilayer networks?
- Why would we want deep, multilayer networks?

Why deep networks?

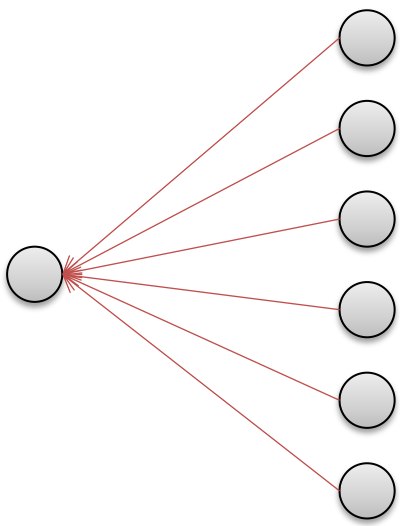
- Shallow networks can't represent any function, eg, XOR

- Anatomically and physiologically, the brain is deep

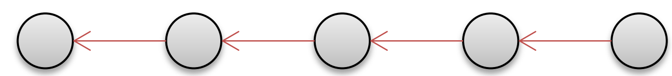
- Empirically, deep networks just *work better*

Two fundamental topologies

Parallel

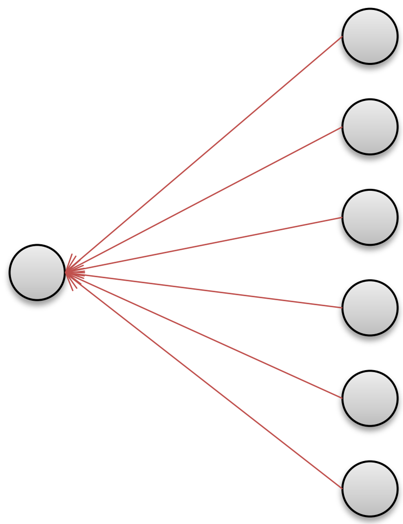


Serial

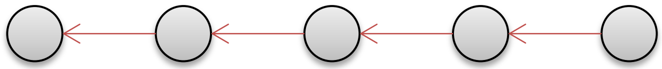


Two fundamental topologies

Parallel



Serial

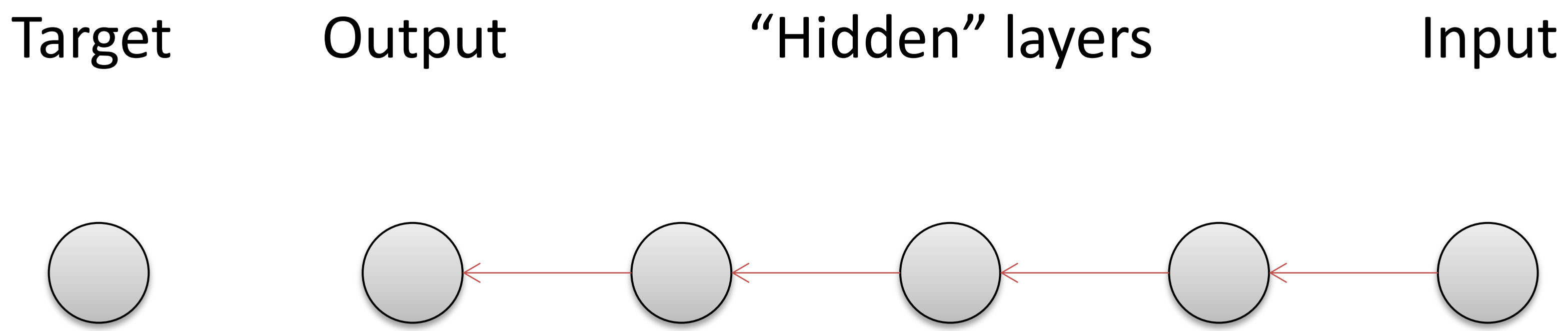


Gradient descent learning rule:

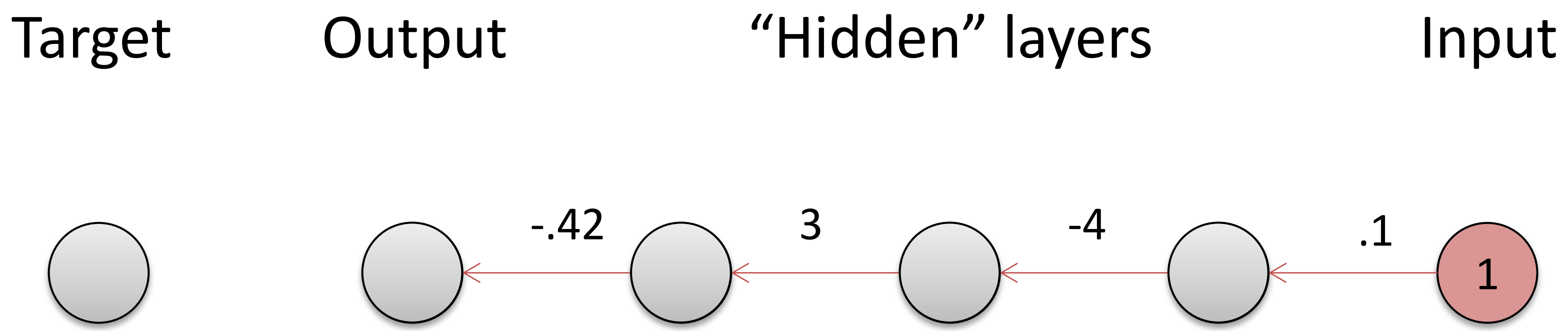
Delta rule

???

Many layer learning

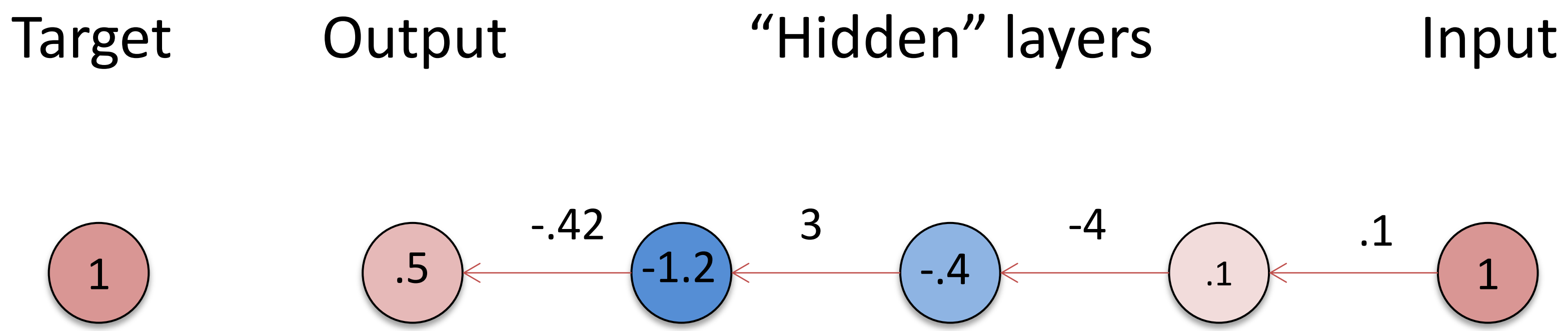


Forward propagation

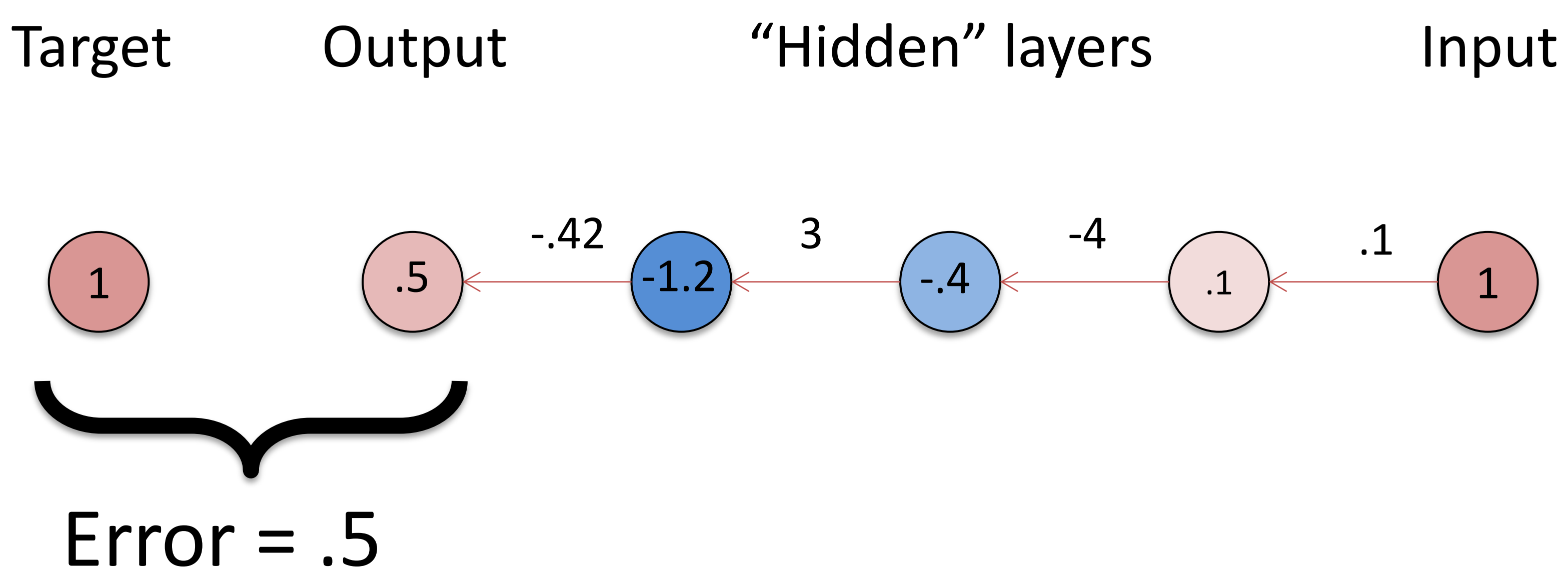


$$o = w_4 w_3 w_2 w_1 i = \left(\prod w_k\right) i$$

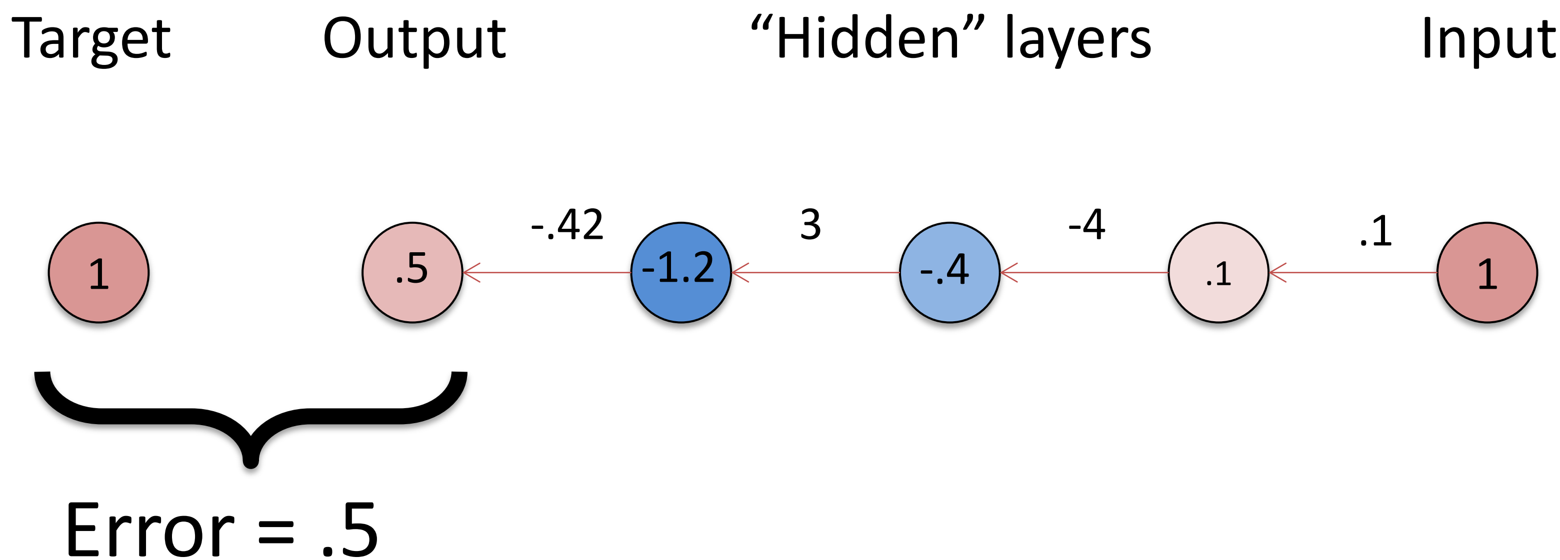
Target feedback



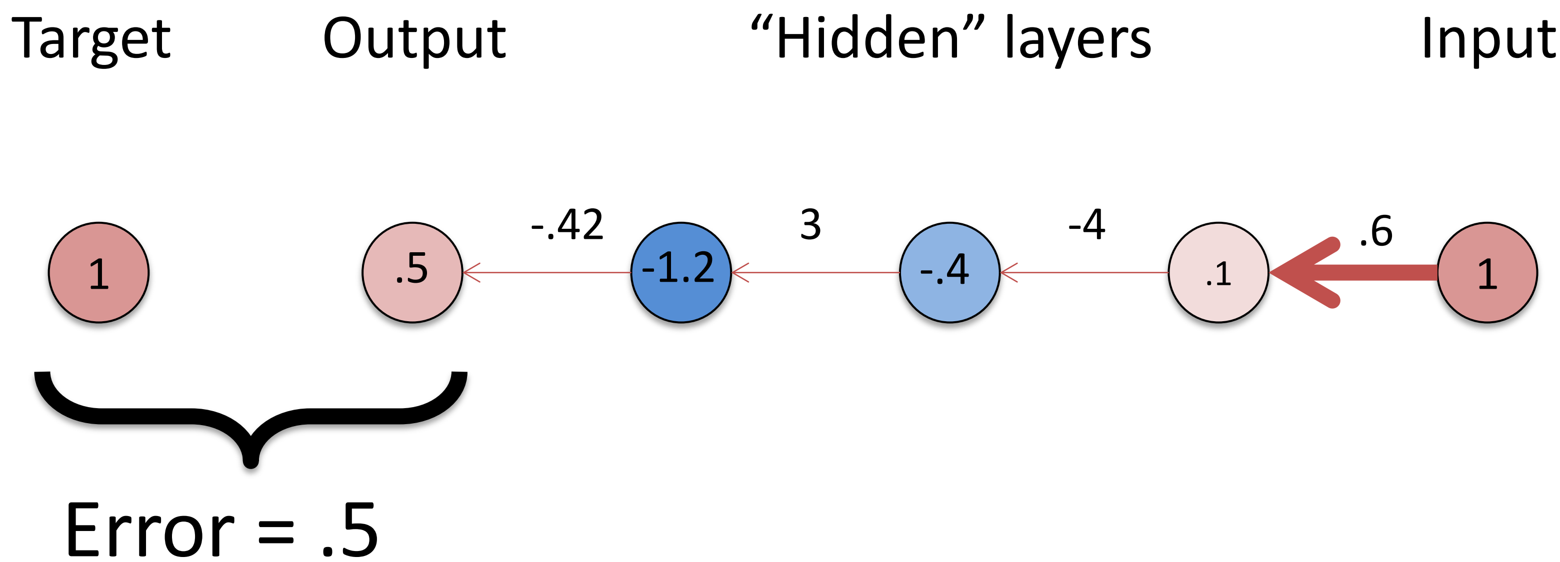
Error signal



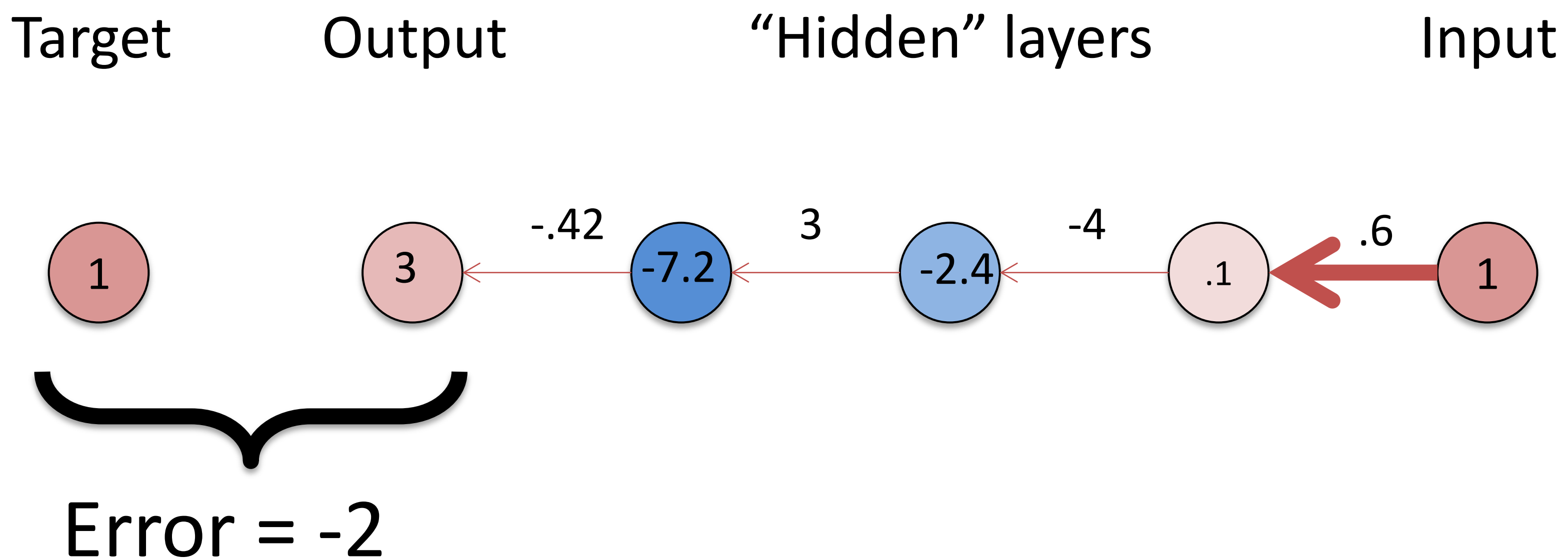
How to fix?



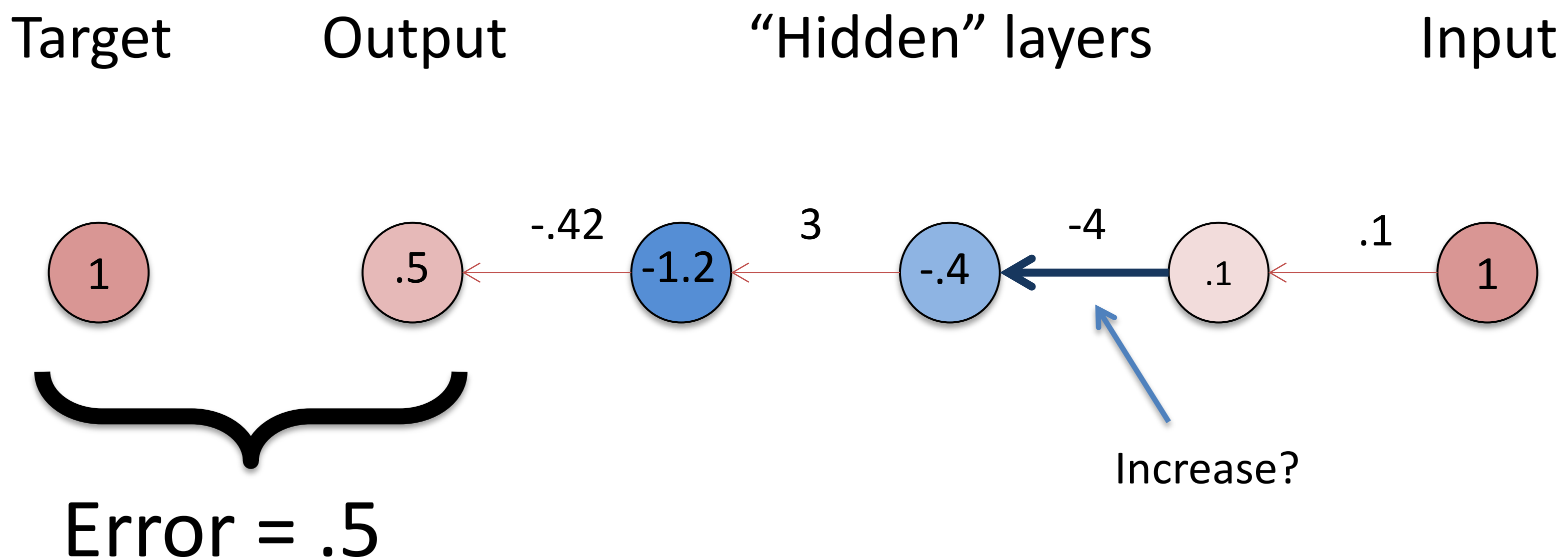
How to fix?



How to fix?

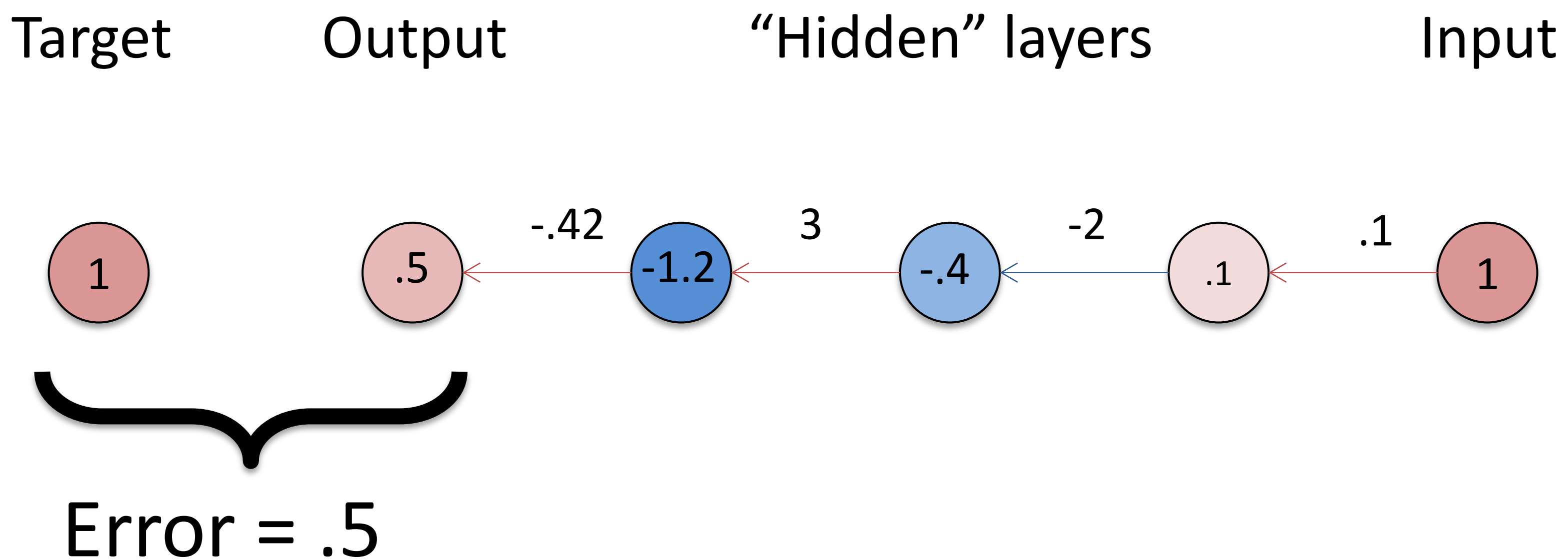


How to fix?

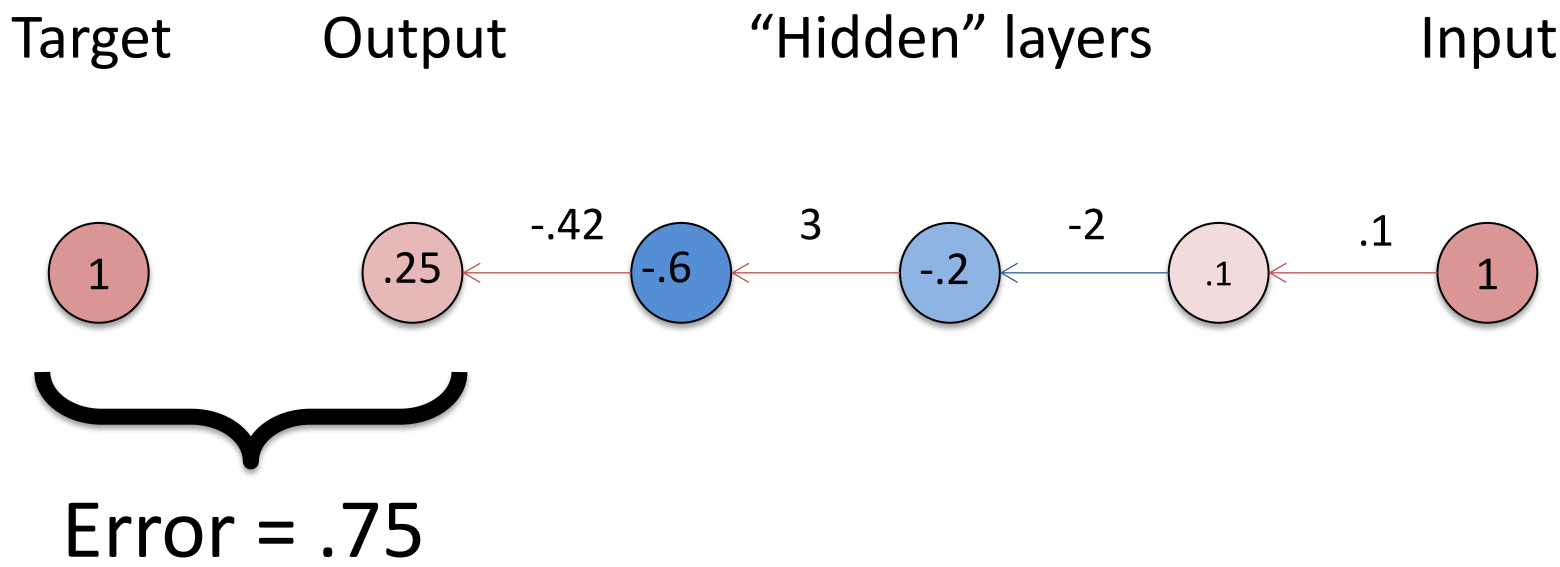


Target
Output
"Hidden" layers
Input
1
.5
-.42
-1.2
3
-.4
-4
.1
.1
1
Increase?
Error = .5

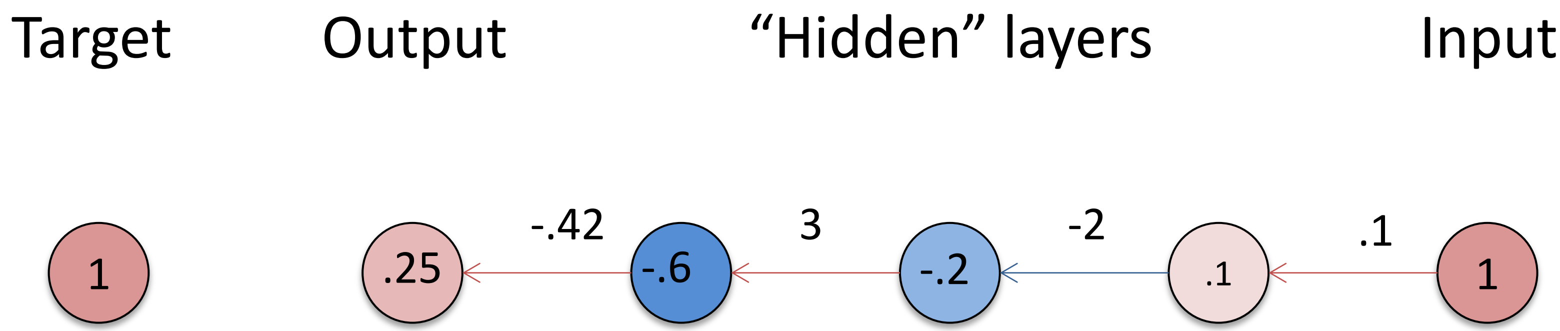
How to fix?



How to fix?



How to fix?

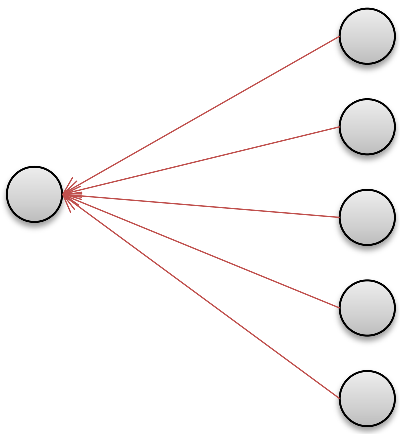


Must consider effect of all upstream weights to know how to change any given weight!

In a deep network, weights are *coupled*

Two fundamental topologies

Parallel

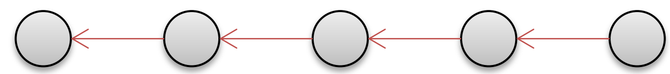


$$o = \sum w_j i_j$$

Sum of variables

Weights are **independent**

Serial



$$o = \left(\prod w_k\right) i$$

Product of variables

Weights are **coupled**

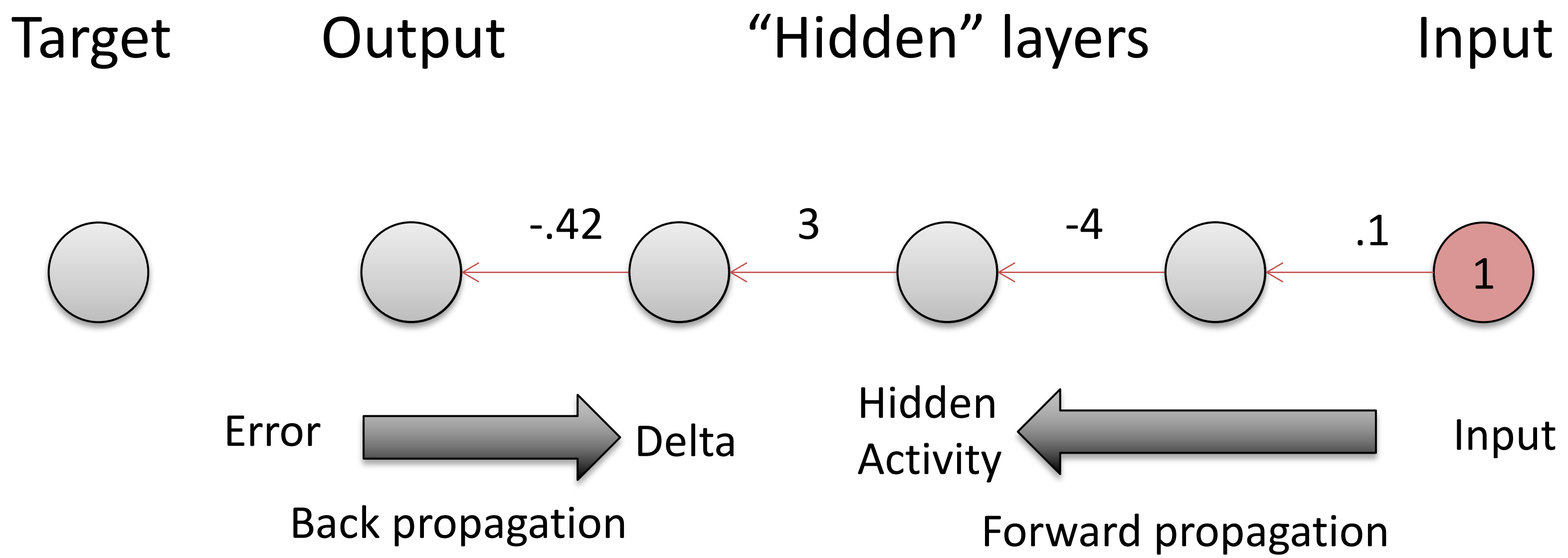
The solution: Backpropagation

- Send a signal from the output of the network back down towards the input

- This signal will encode how changing a weight will propagate to the output

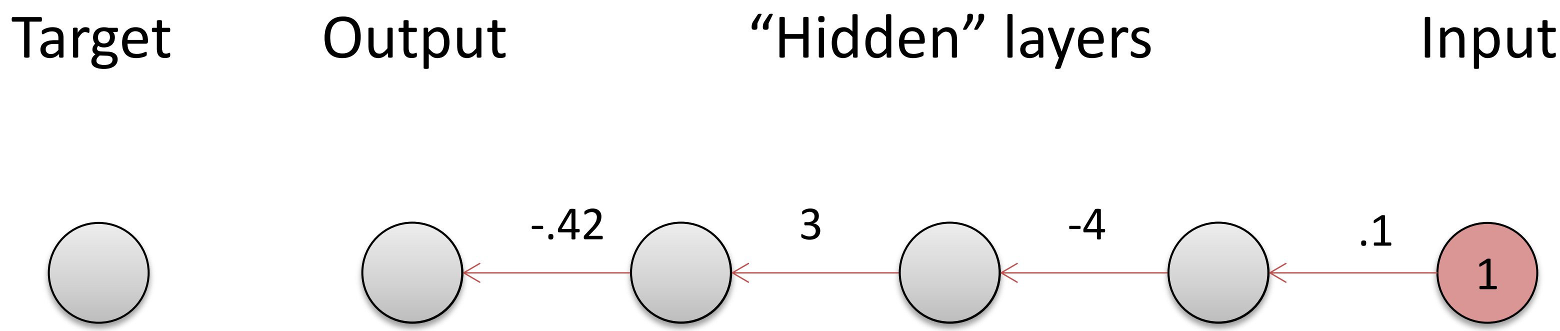
- Then use delta-like rule as usual

The solution: Backpropagation

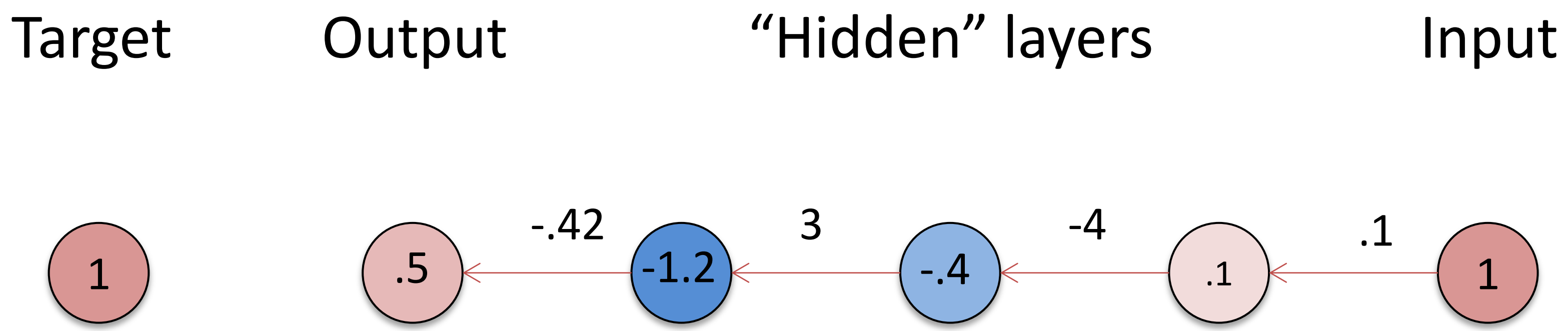


$$\Delta w_j = \lambda \delta_{j+1} h_j$$

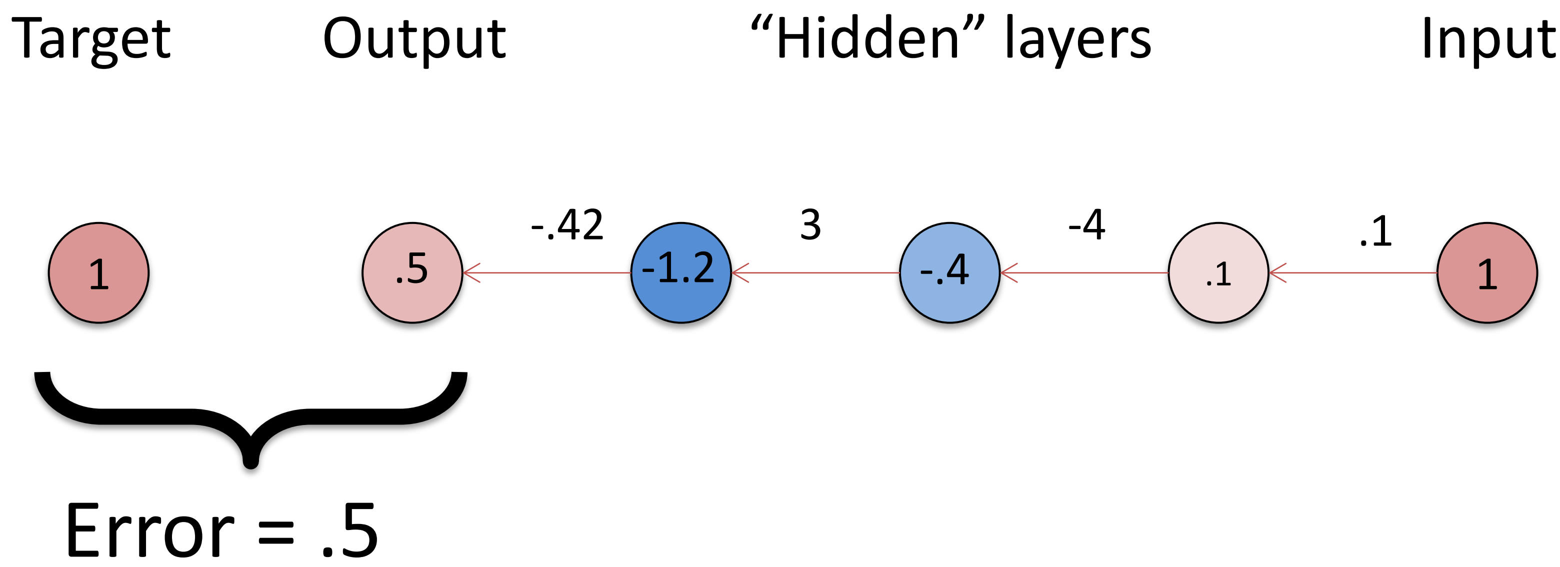
Forward propagation



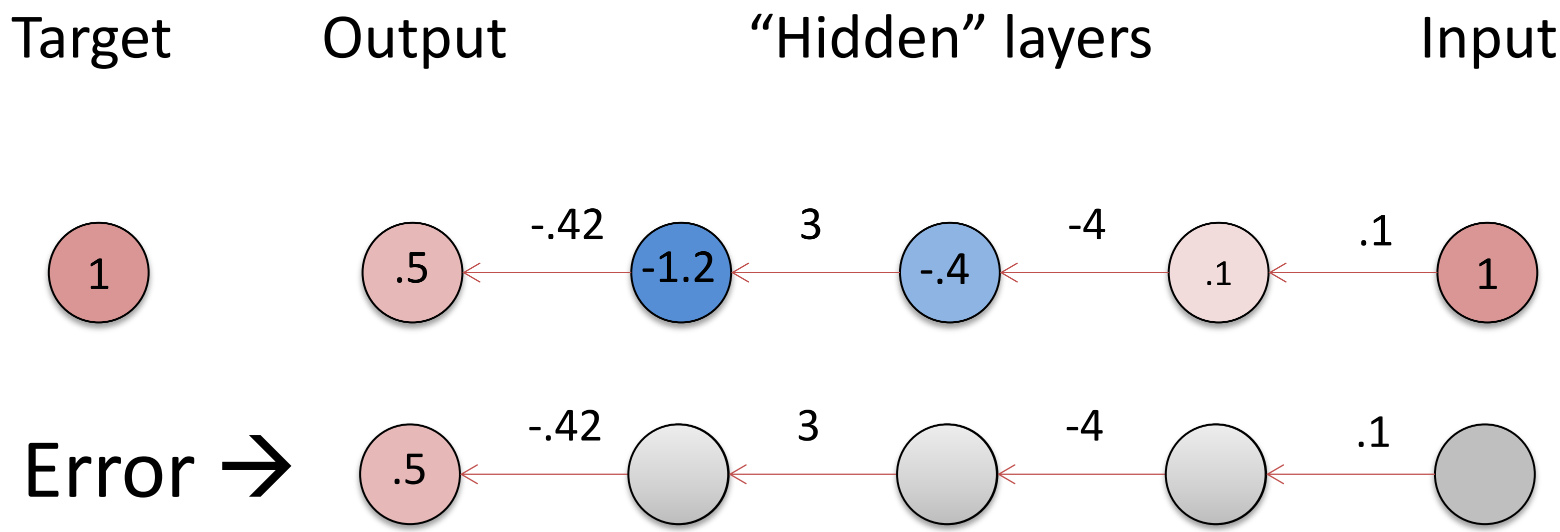
Target feedback



Error signal



Backpropagation



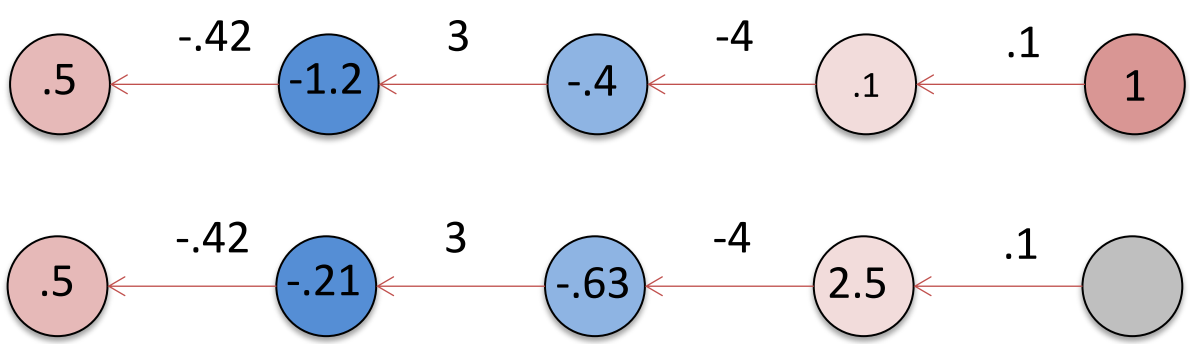
$$\delta_j = w_{j+1}\delta_{j+1}$$

Backpropagation

Output "Hidden" layers Input

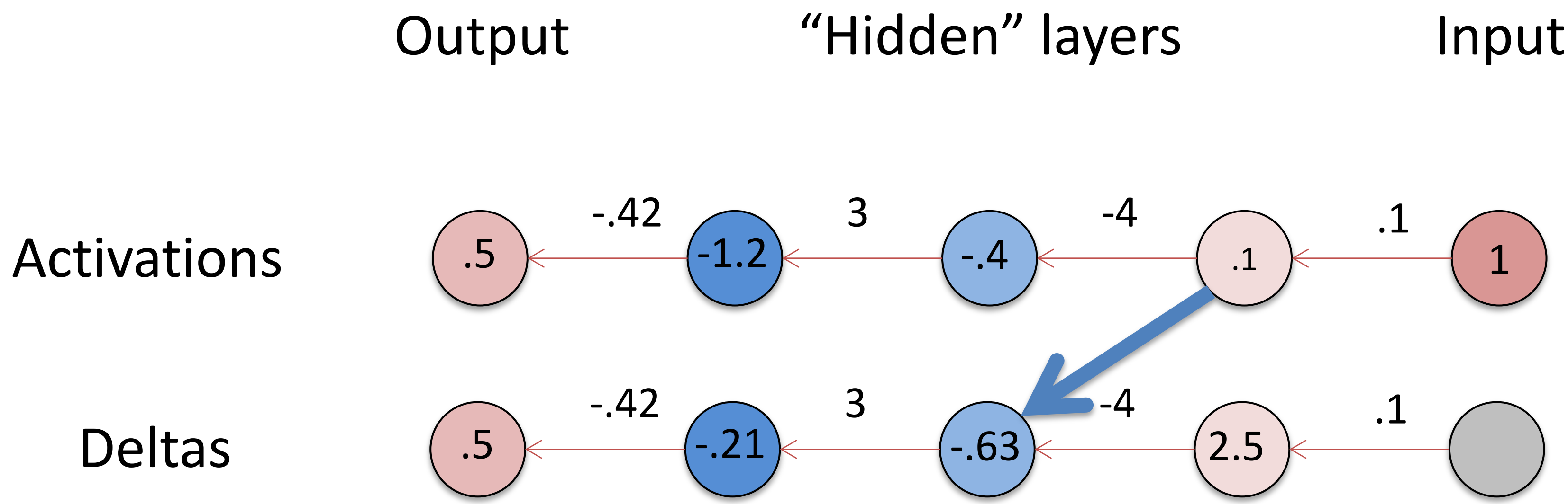
Activations

Deltas



Learning rule: $$\Delta w_j = \lambda \delta_{j+1} h_j$$

Backpropagation



Learning rule: $$\Delta w_j = \lambda \delta_{j+1} h_j$$

Derivation as gradient descent

$$\frac{\partial}{\partial w_2}(t-o)^2 = -2(t-o)\frac{\partial}{\partial w_2} w_4 w_3 w_2 w_1 i$$

$$= \underbrace{-2(t-o)w_4 w_3}_{\delta_3}\underbrace{w_1 i}_{h_2}$$

Learning rule: $\Delta w_j = \lambda \delta_{j+1} h_j$

Delta vs Backprop

Delta rule:

$$\Delta w = \lambda e\, i^T$$

- Requires actual error and input
- Restricted to one layer

Backprop:

$$\Delta w_j = \lambda \delta_{j+1} h_j$$

(where *j* indexes the layer)

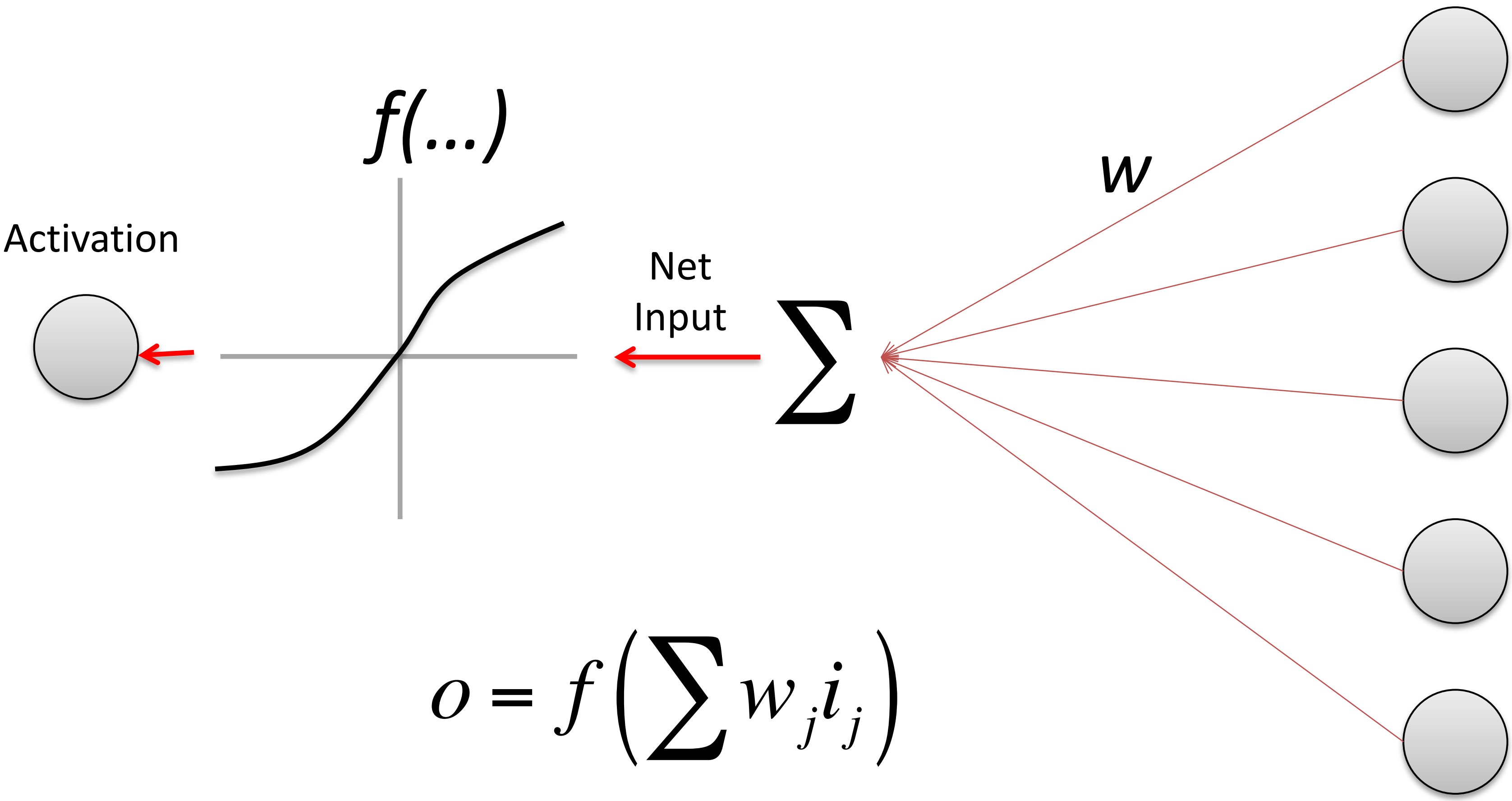
- Uses backpropagated error and hidden layer activity
- Works in deep network

- Both implement gradient descent

Nonlinearities

- So far have just looked at linear networks
- Linear networks cannot represent complicated functions (like XOR)
- Introduce neural nonlinearity or “activation function”

Activation function



$$o = f\left(\sum w_j i_j\right)$$

Gradient learning with nonlinearities

$$\frac{\partial E}{\partial w_k} = -2(t-o)\frac{\partial E}{\partial w_k} f\left(\sum w_j i_j\right)$$

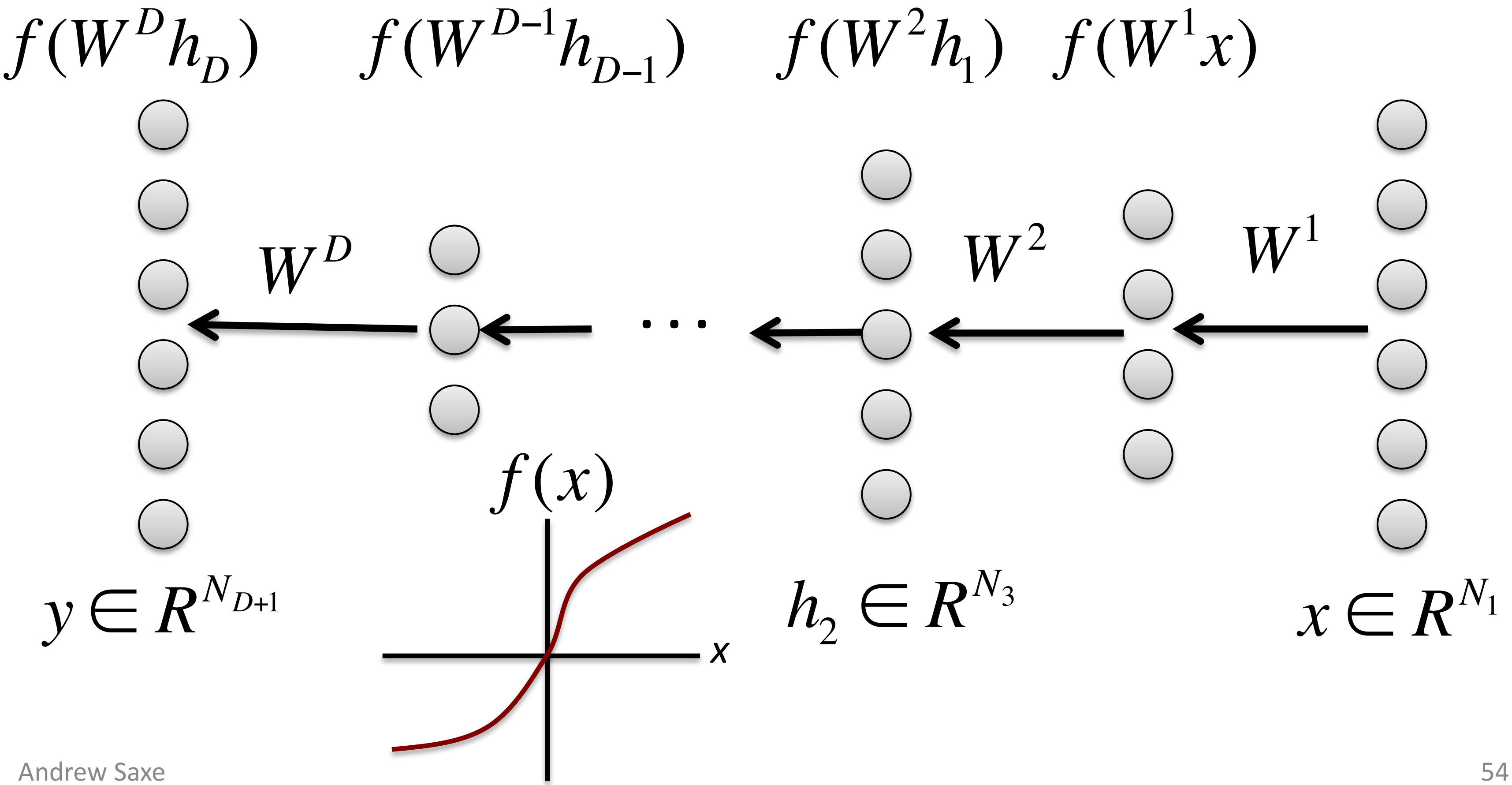
$$= -2(t-o)f'(n)\frac{\partial E}{\partial w_k}\sum w_j i_j$$

$$= -2(t-o)f'(n)i_k$$

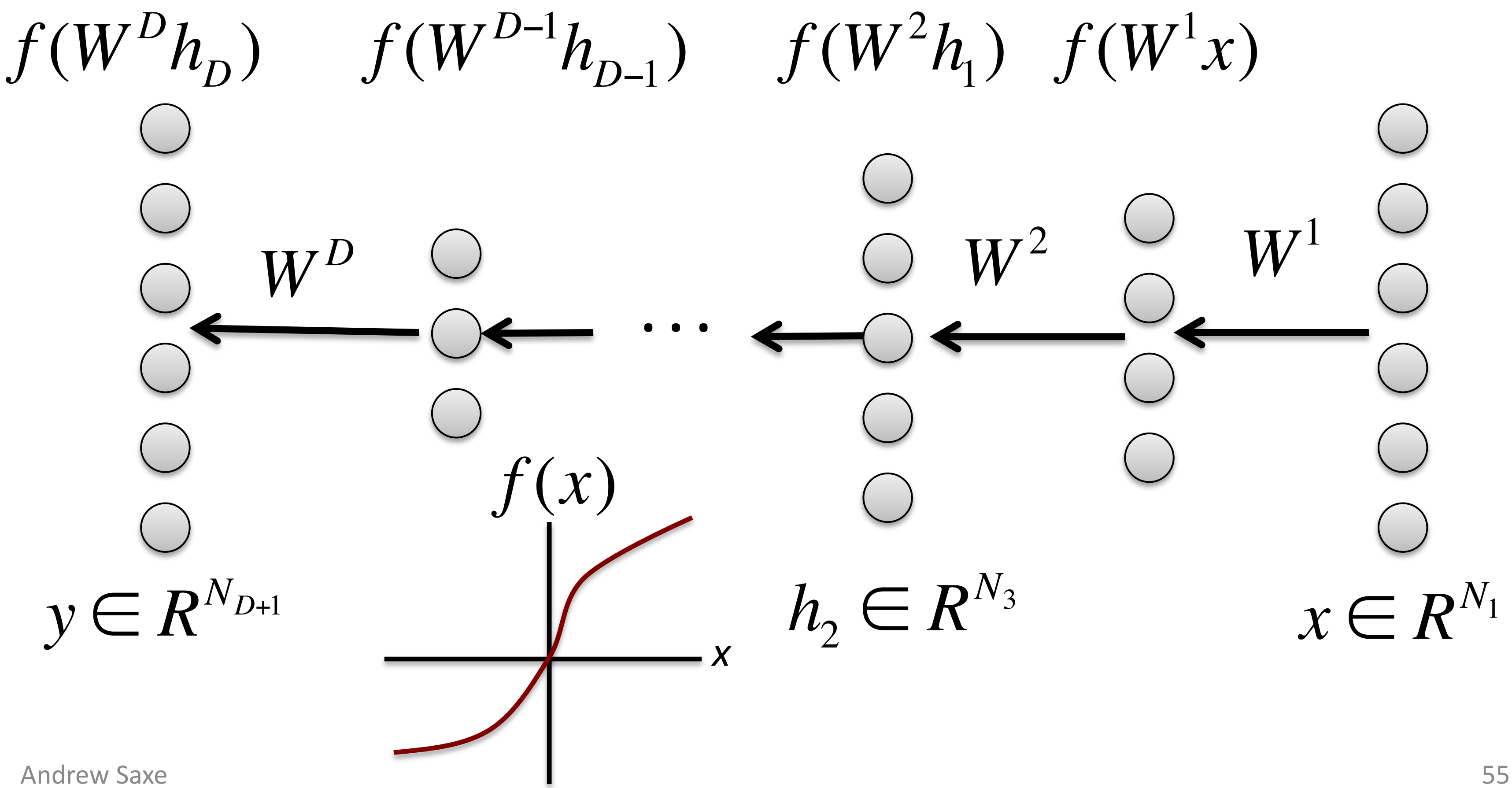
Scaled delta rule: $\Delta w = \lambda e f'(n) i^T$

Beyond the chain…

Now for the general case:
a mixture of serial and parallel structure with nonlinearities



Matrix notation



Matrix notation

$$o = f(W^D f(W^{D-1} \ldots f(W^2 f(W^1 i))\ldots))$$

Column vector

Column vector

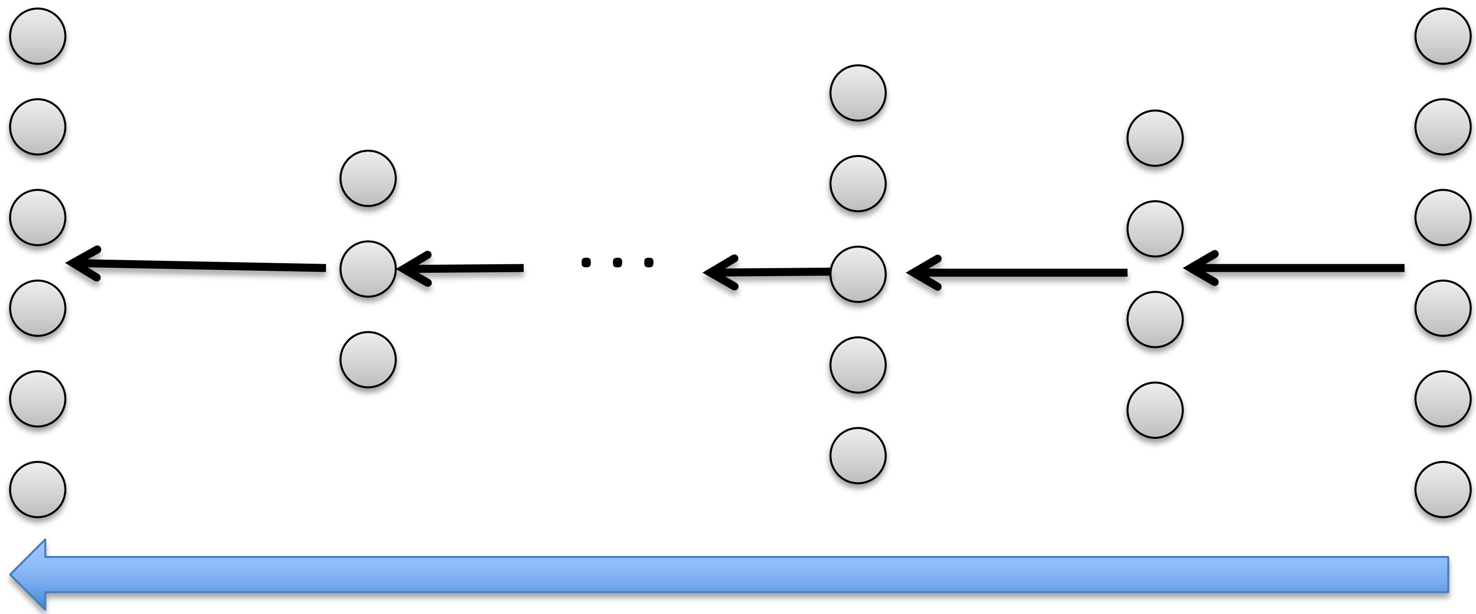
Matrix multiplication

Nonlinearity applied elementwise

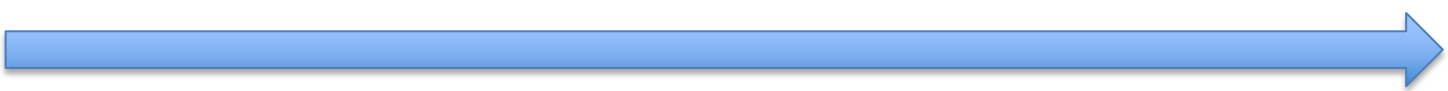
Putting the pieces together

- Single layer parallel: the delta rule
- Multilayer serial: backpropagation
- Nonlinearities: scale by derivative

Putting the pieces together



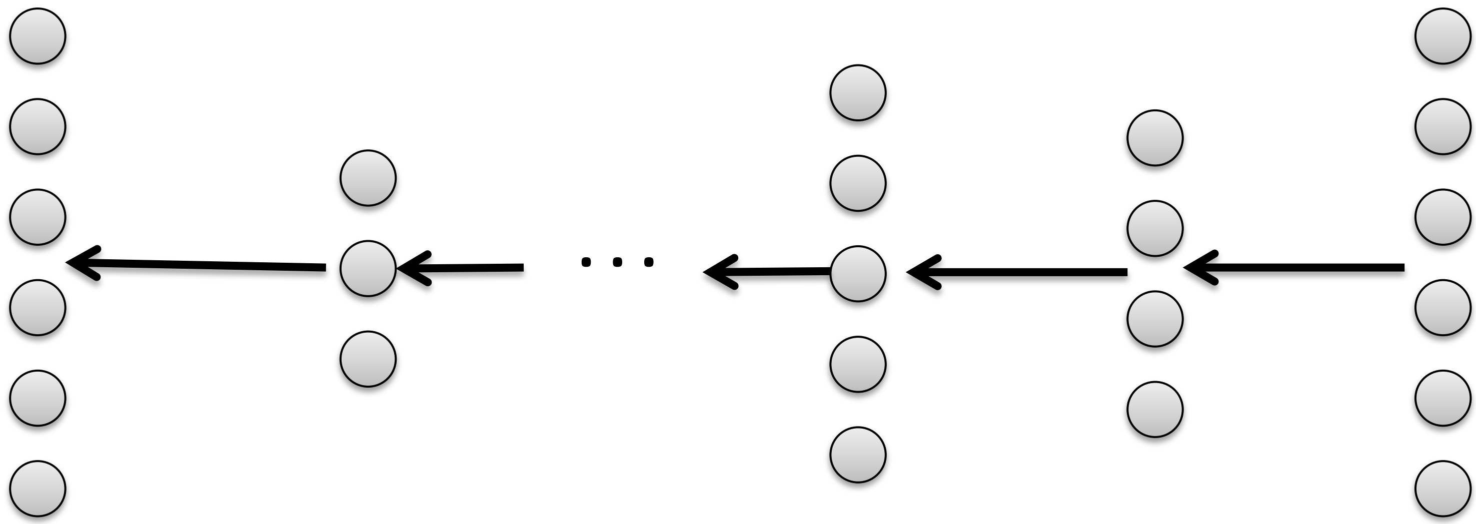
Forward prop input to compute *hidden activities, net inputs, output*



Back prop error to compute *deltas*

Learning rule: $$\Delta W^{j} = \lambda \delta_{j+1} h_j^T$$

Backpropagation



$$\delta_D = -(t-o) \bullet f'(n_D)$$

$$\delta_j = \left[\left(W^j \right)^T \delta_{j+1} \right] \bullet f'(n_j)$$

Learning rule: $\Delta W^j = \lambda \delta_{j+1} h_j^T$

Summary

- Computational level
  - Optimization view of learning
  - Gradient descent
- Algorithmic level
  - Backprop-as-algorithm
- Implementation level
  - Haven't touched on it
  - Not obvious how to implement in the brain, but some ideas exist (e.g. Lillicrap et al., 2016)