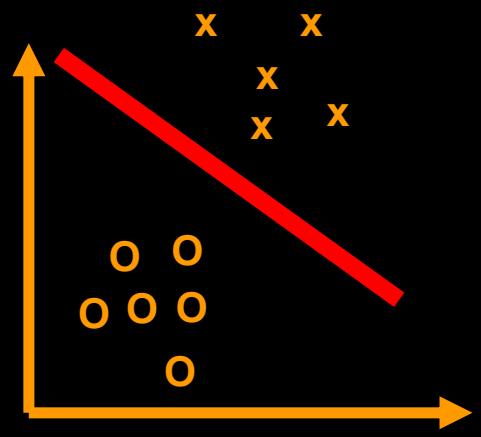
Supervised learning

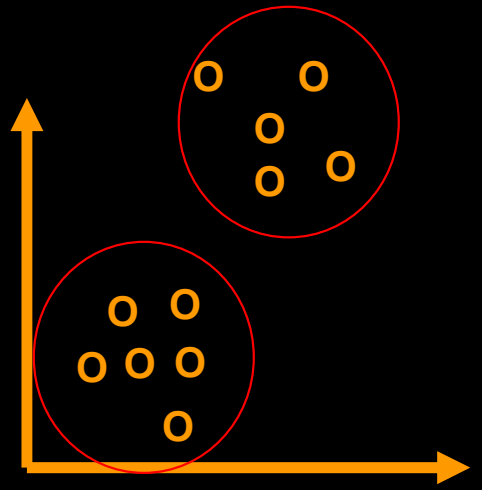
COGS-Q355

Supervised vs. unsupervised learning

- Supervised learning: You know the desired values of the output units, so you compute the error term and update the network to better produce the desired outputs



- Unsupervised learning: You don't know what the desired outputs are nor do you know the category labels. So you have to try to figure out what is the structure of the data, typically by grouping similar data points into category groupings



A brief history of error-driven learning

- The Perceptron
- The XOR problem
- Backpropagation
- Solving the XOR problem
- Deep learning

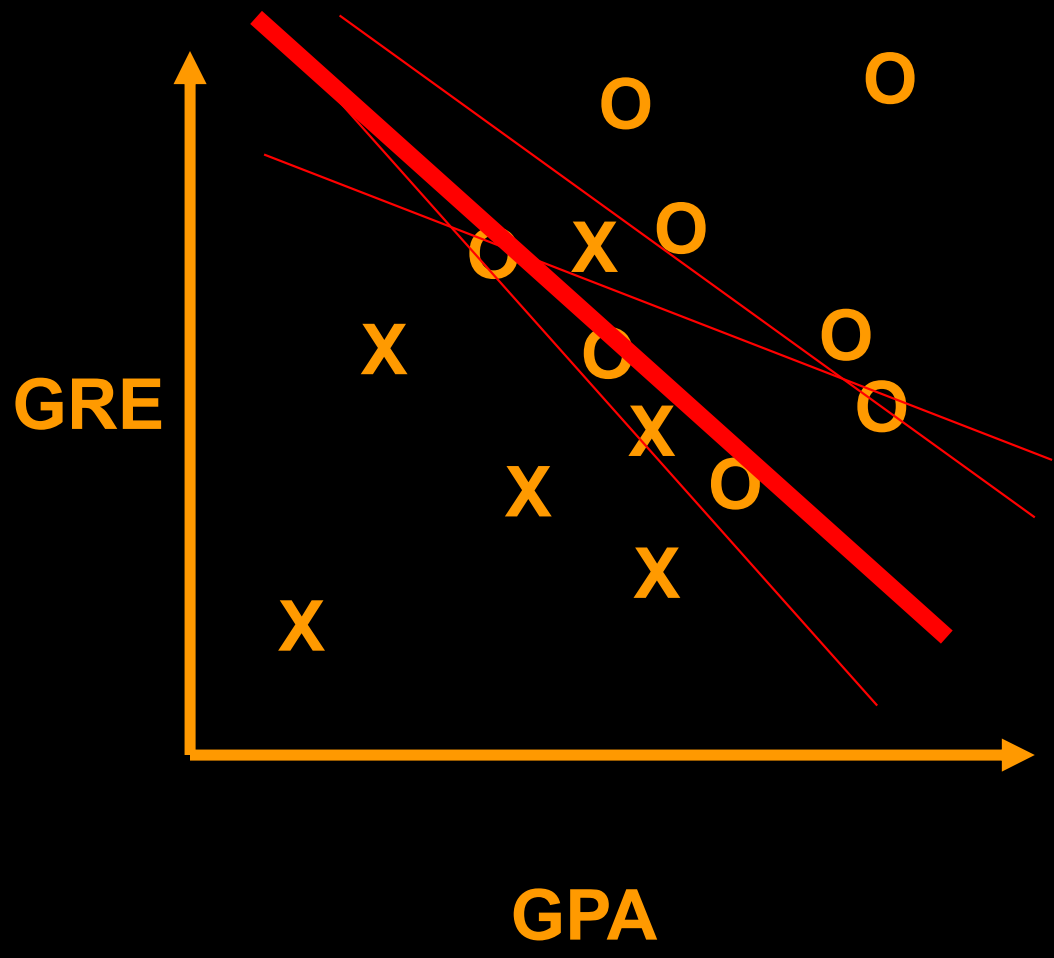
Example problem: linear separation

- Grad school admission criteria
- X=not admitted
- O=admitted

- How do you distinguish the classes?

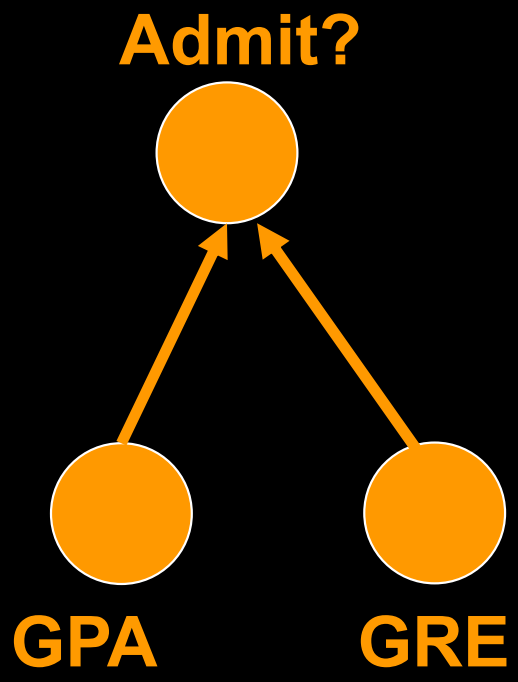
- E.g. Linear separation:

- A*GPA + B*GRE = C
- What are A and B?
- In general, classes might be separated by a hyperplane



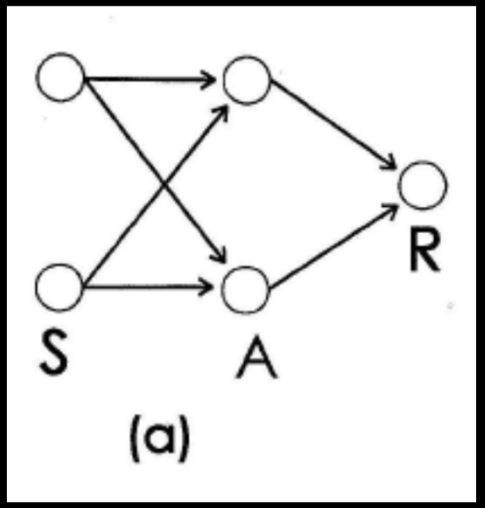
McCullough-Pitts

- Neurons fire whenever the sum of excitation reaches a threshold, and there is no inhibition
- Didn't address learning
- All-or-nothing inputs
- "weights" are essentially uniform, e.g. 1



Perceptrons (Rosenblatt, 1962)

- Introduced connection weights, so postsynaptic response is a weighted (algebraic) sum of the inputs



DEFINITION 22: A *simple perceptron* is any perceptron satisfying the following five conditions:

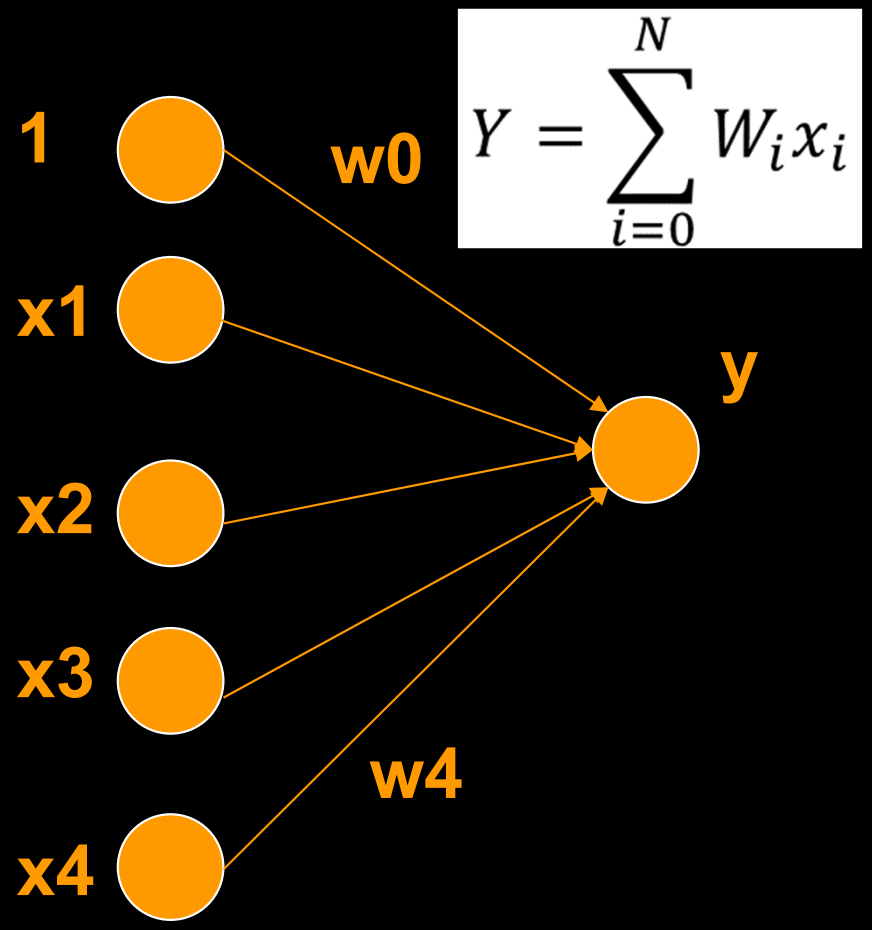
1. There is only one *R*-unit, with a connection from every *A*-unit.
2. The perceptron is series-coupled, with connections only from *S*-units to *A*-units, and from *A*-units to the *R*-unit.
3. The values of all sensory to *A*-unit connections are fixed (do not change with time).
4. The transmission time of every connection is either zero or equal to a fixed constant, τ.
5. All signal generating functions of *S*, *A*, and *R* units are of the form $u_i^*(t) = f(\alpha_i(t))$, where $\alpha_i(t)$ is the algebraic sum of all input signals arriving simultaneously at the unit u_i.

DEFINITION 23: An *elementary perceptron* is a simple perceptron with simple *R*- and *A*-units, and with transmission functions of the form $c_{ij}^*(t) = u_i^*(t - \tau)\, v_{ij}(t)$.

Perceptron Learning

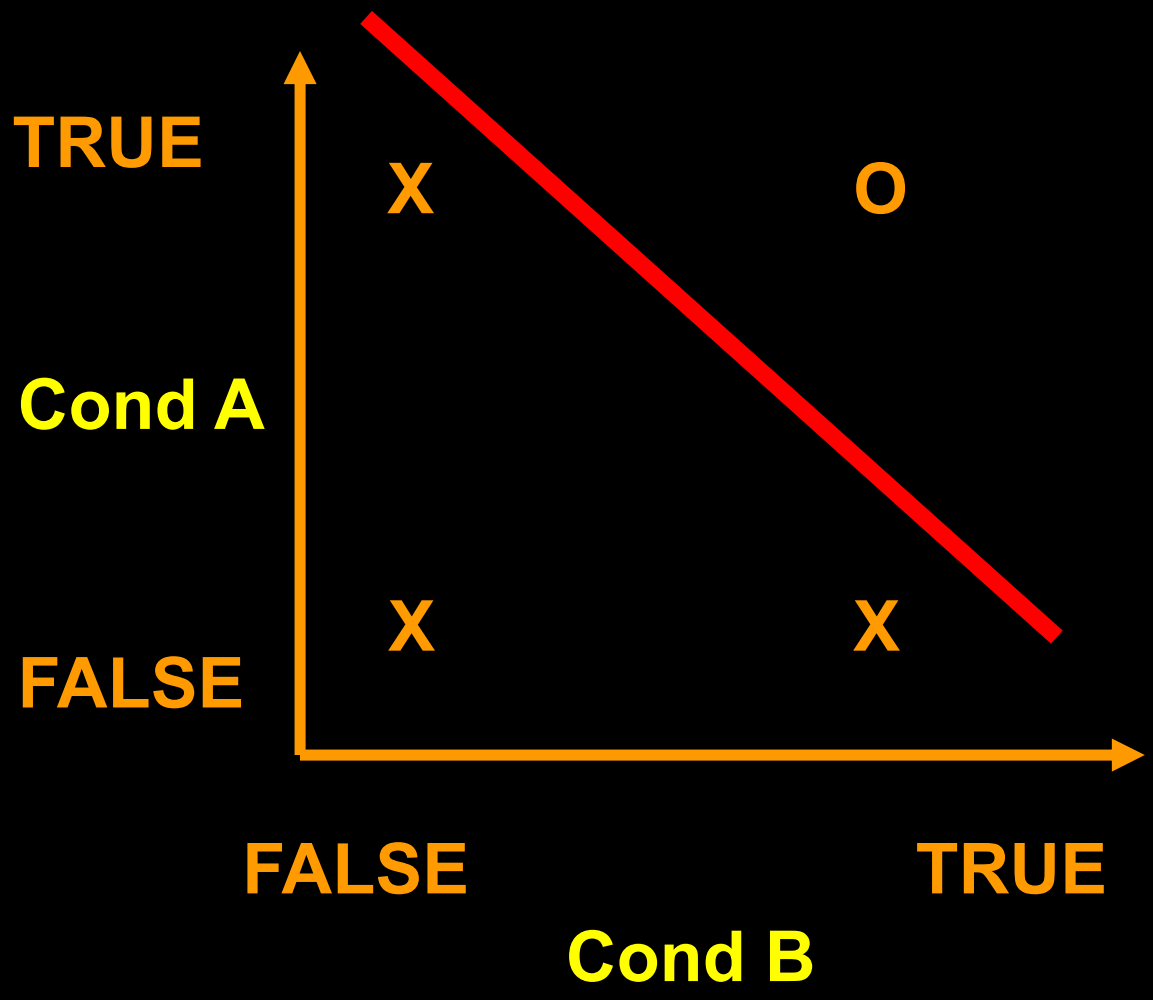
- Introduced connection weights, so postsynaptic response is a weighted (algebraic) sum of the inputs (i.e. dot product)
- Initialize weights to small random values
- Compute y (output)
- $W_i(t+1)=W_i(t)+alpha*(d-y)*x_i$
- Alpha = learning rate
- d=desired output
- Y=actual output
- So (d-y)=error term

- The Delta Rule (Widrow-Hoff)

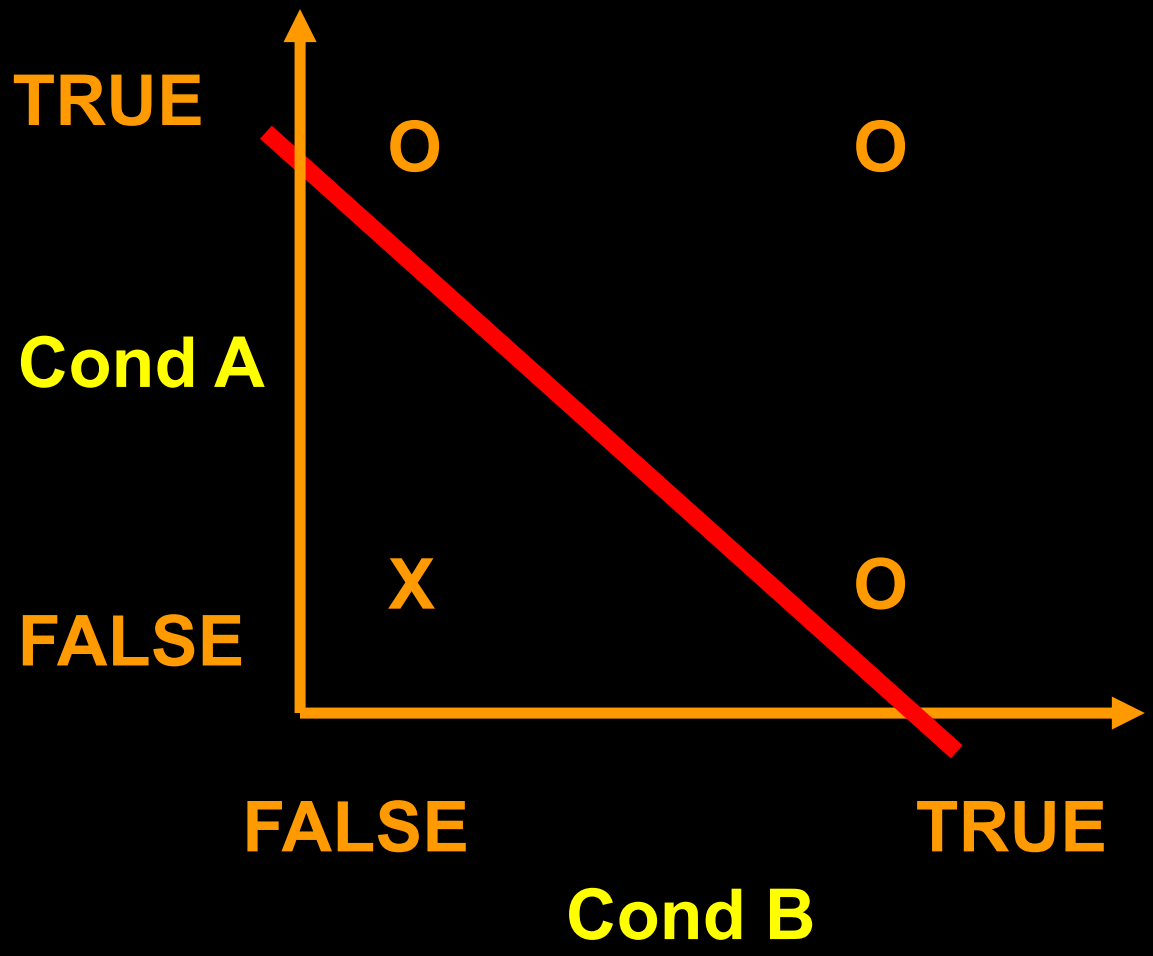


Simple classifications of logic -- AND

- How could a perceptron learn to compute the logic functions?
  - OR
  - AND
  - XOR

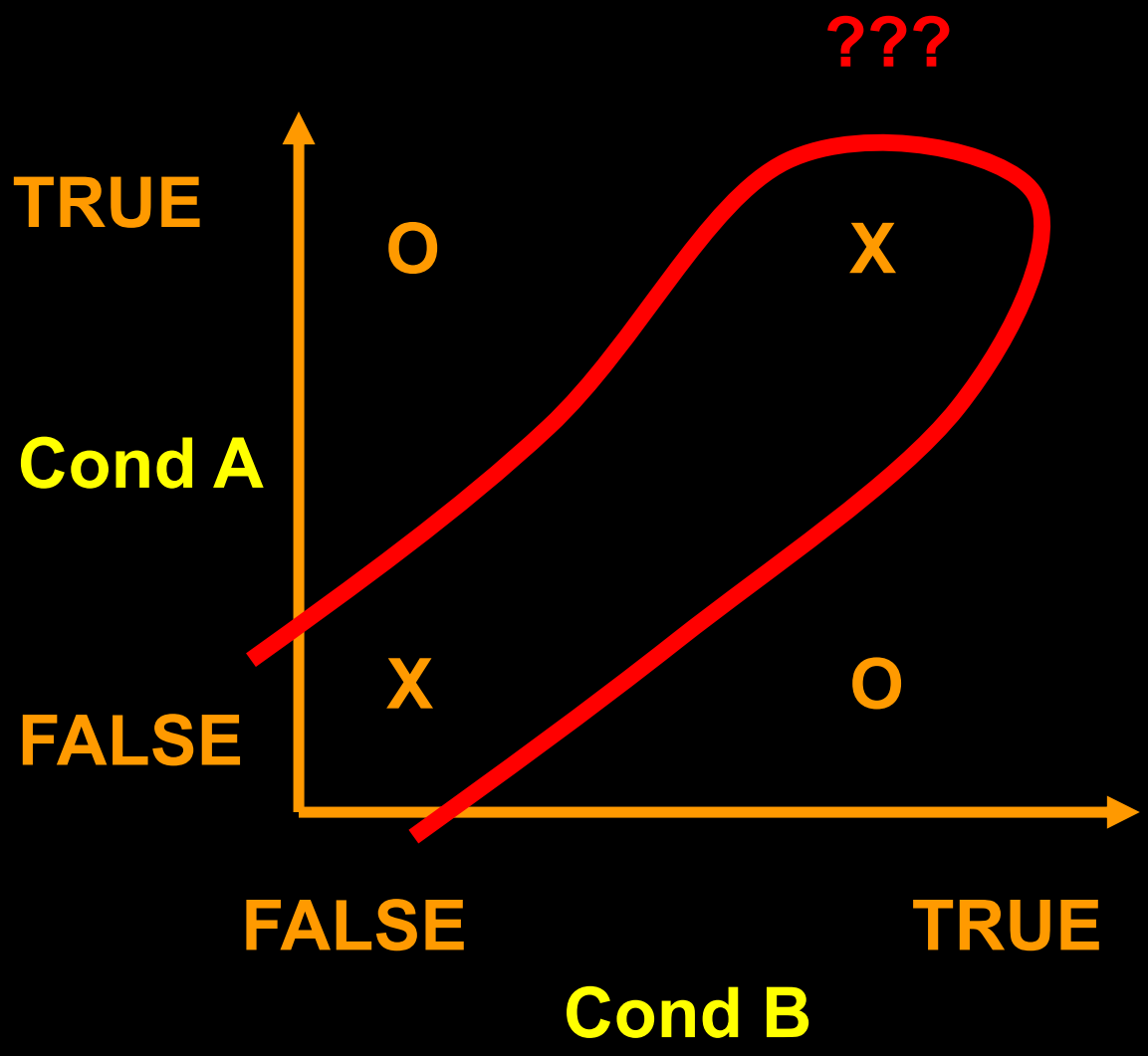


Simple classifications of logic -- OR



Simple classifications of logic - - XOR

- THERE IS NO LINEAR CLASSIFIER THAT CAN SOLVE THE XOR PROBLEM
- Minsky & Papert (1969) pointed this out, and that stifled neural network research for the next decade plus



Review of calculus: chain rule

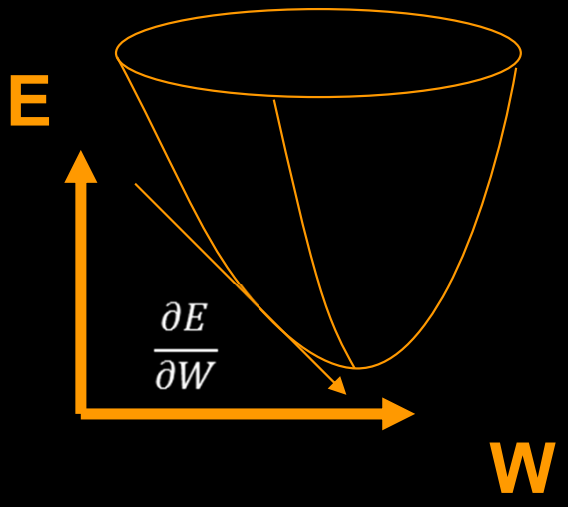
- Consider a set of functions f(u) and u(x)
- Suppose g(x) = f(u(x))
- What is $\frac{d}{dx}$g(x)?
- Chain rule: $\frac{d}{dx}$g(x) = $\frac{df}{du}\frac{du}{dx}$
- E.g. g(x)=sin(x^2). Then f(u)=sin(u), and u=x^2.
- $\frac{d}{dx}$g(x) = $\frac{df}{du}\frac{du}{dx}$=cos($x^2$)*2x

Widrow-Hoff Delta Rule

- Suppose the error at the network output is error =(d-y). Then the objective function is:
- E= $\sum_{i=1}^{N}(d_i - y_i)^2$, which is the sum-squared error (SSE). This is minimized when the weights are optimal.

$y=Wx$

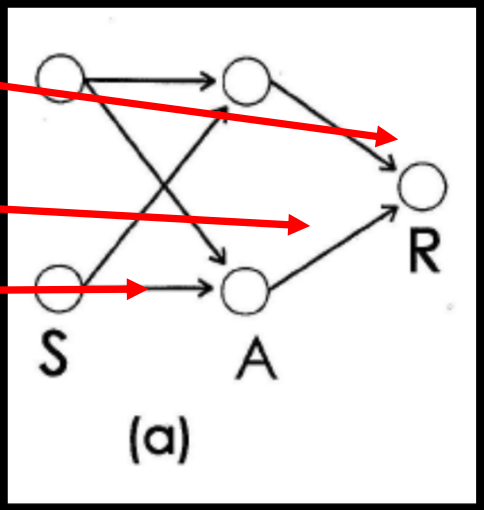
- So W(new) = W(old) -$\alpha \frac{\partial E}{\partial W}$
- $\frac{\partial E}{\partial W}=\frac{\partial E}{\partial y}*\frac{\partial y}{\partial W}$=2(d-y)x. Ignore the 2 as it gets lumped into alpha
- So W(new) = W(old) α(d-y)x. (d-y) is the error term
- The learning rule is the delta rule and implements Error driven learning



Multi-Layer Perceptrons

- It turns out that if you add a HIDDEN LAYER, then perceptrons can solve the XOR problem.
- That leads to another problem though, which is how do you solve the credit assignment problem?

- You know the error here:
- So you can calculate the weight update here
- But how do you calculate the weight update here??
- I.e. how do you assign credit (or blame)?



SP22 questions

- Slightly related to our topic, I recently have come across the term "Connectionism" in various Cognitive Science areas. From what I have found, this is the theory that the mind operates similarly to a multilayer perception, as opposed to manipulating symbols. Most of what I have read so far on this is coming from Stanford Encyclopedia of Philosophy, but if you knew of any other resources or had any insight I would definitely be interested. Thank you!

- In regards to the Widrow-Hoff rule, I found it very similar to this video from 3Blue1Brown (https://www.youtube.com/watch?v=IHZwWFHWa-w) , which is about gradient descent in order to better define the weights in a preceptor. If you're looking for more information, I highly suggest you watch this.

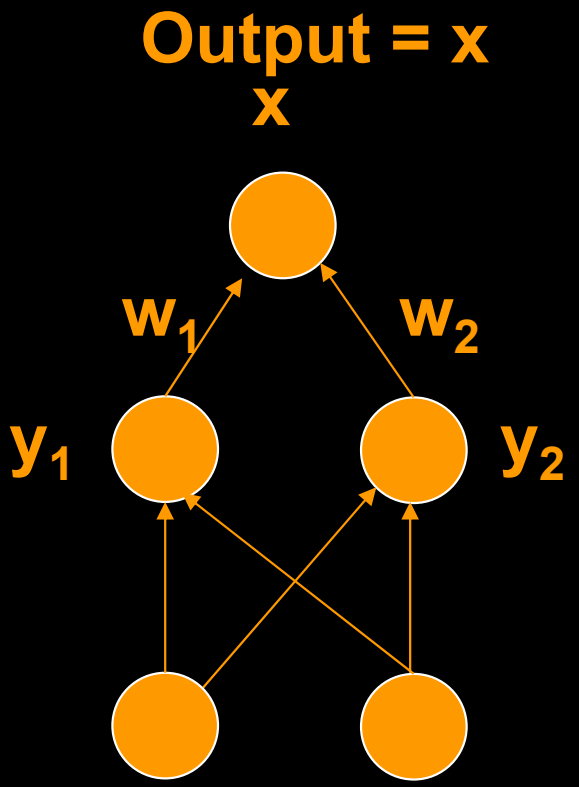
- How reliable are the final weights of the network once the system has converged on the desired output? As in, are the final weights always the same after sufficient training? I would imagine that if you initialize the network with small random weights and the learning rate is based on the overall output, couldn't the weight for a node that started with a randomly higher weight increase more than other nodes that are equally responsive to a given input? Could you make slightly different variants of the same network or do they mostly stabilize into the same configuration?

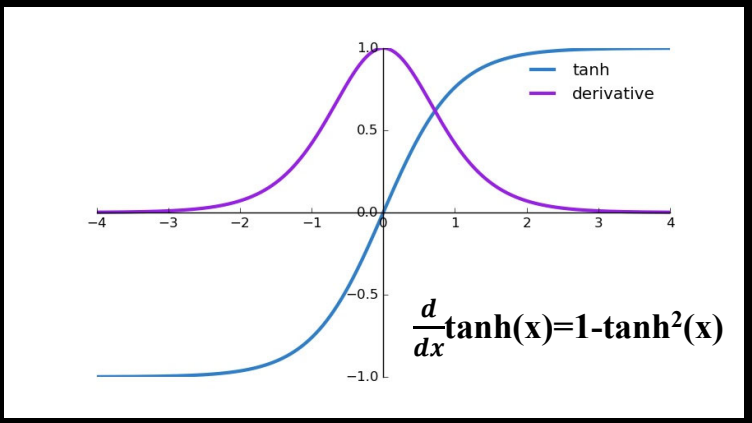
- I've heard that two-layer neural networks are universal function approximators, but I haven't really heard about the specific benefits of having more than one hidden layer (besides in a sorta hand-wavey "yup we added a bunch more layers and it's better" kinda way). I'd be interested in hearing a more general idea of what sort of preprocessing the additional layers could be doing, and how exactly they would (I'm assuming) make it easier for the network to converge to the correct set of weights. I'm assuming they must be somehow encouraging the network to find a more general solution to the problem, i.e. one that doesn't suffer from overfitting as much?

Multi-Layer Perceptrons (Rumelhart '86 Nature)

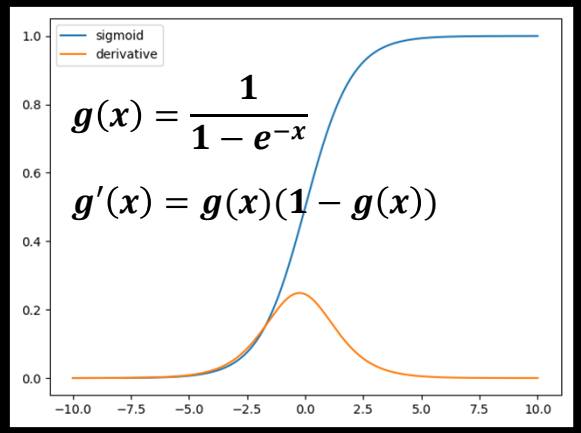
- Each unit has a nonlinear signal function, e.g. the logistic function $f(x)=1/(1-e^{-x})$ – Otherwise it would all be linear, and there would be no point in having hidden layers! (the product of two matrices is simply one matrix!)

- When there are more than 2 layers, you have to first run the activity FORWARD through the network, to compute all the activations at each node.

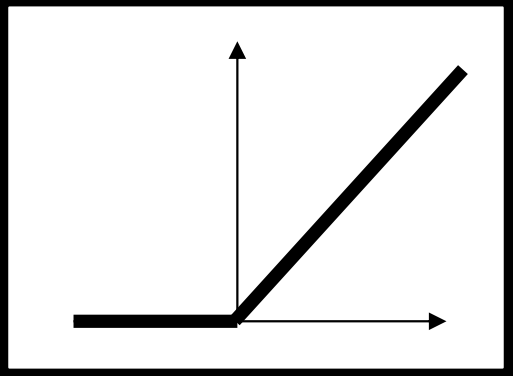




Tanh()



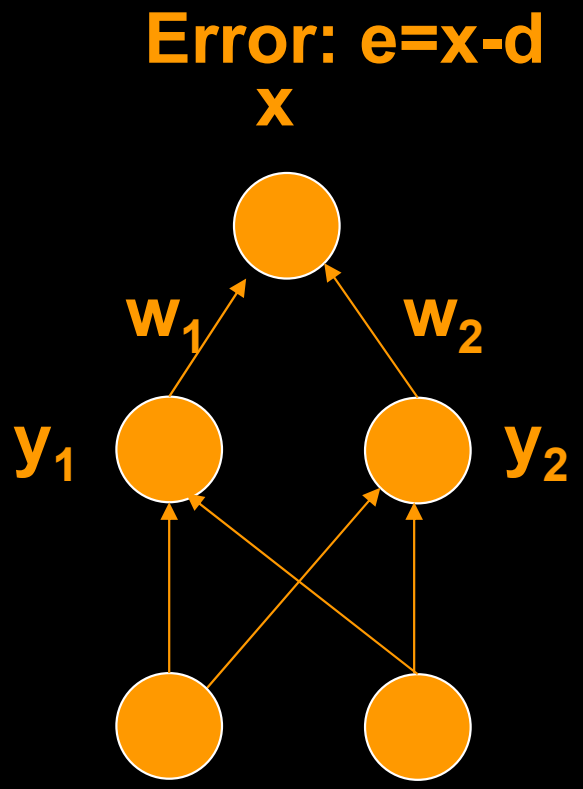
Sigmoid (logistic)



ReLU (rectified linear)

Multi-Layer Perceptrons (Rumelhart '86 Nature)

- The output of the network is a prediction (x) of the desired output (d)
- The short version: Once you have the error (e) computed, then you have to take the final output error (e = x-d) and propagate it BACKWARD through the network (multiplying by the same connection weights!), so you know what the error term is for each node including the hidden layer nodes
- Finally, you have to update the weights at each node using the delta rule.
- The long version: How can we update the set of weights w so as to MINIMIZE the error (i.e. have the network produce the desired output given the input?



Multi-Layer Perceptrons (Rumelhart '86 Nature)

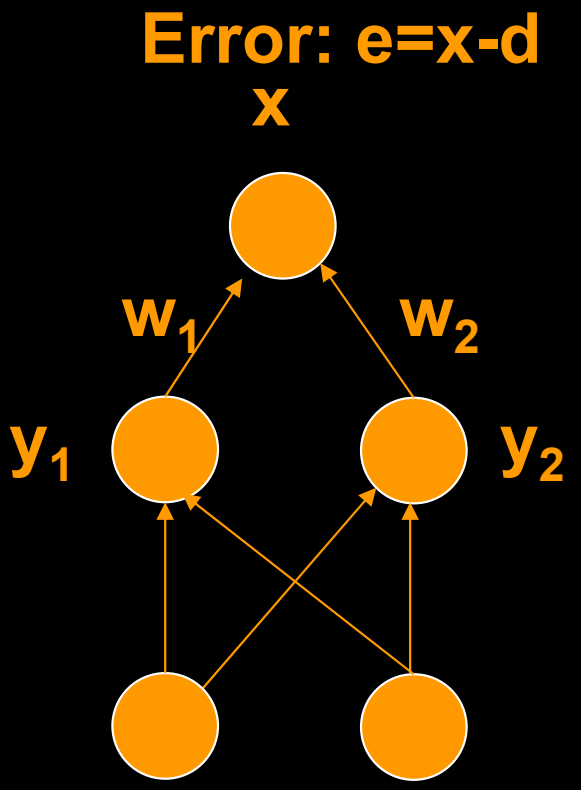
- The long version: How can we update the set of weights w so as to MINIMIZE the error (i.e. have the network produce the desired output given the input?

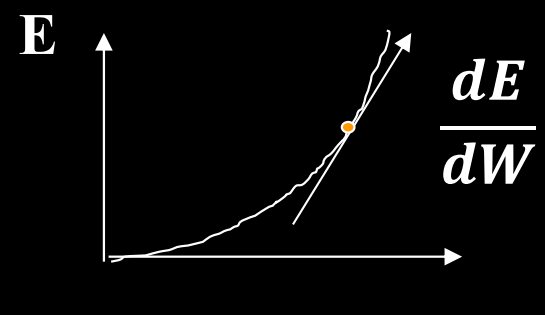
- One simple way is to calculate the SUM SQUARED ERROR, i.e.

- $E = \sum_{i=1}^{N} e_i^2$

- Note that E is smallest when the output at each node is the same as the desired output.

- Our goal of learning is to adjust W so as to minimize E
- But how? Basically compute dE/dw, then subtract it off of W, which will reduce E. (This is gradient descent)

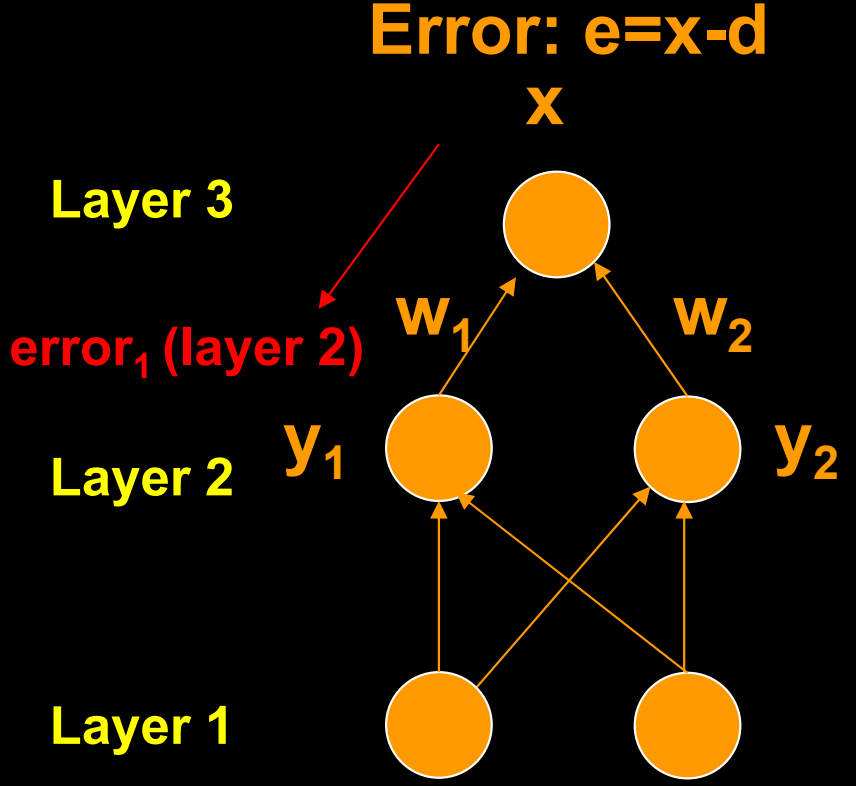




Multi-Layer Perceptrons (Rumelhart '86 Nature)

- Step 1: Compute all the net inputs to each unit
  - $Net_i = sum_j (y_j * w_{ij})$
  - X=f(net), where f(net)=1/(1-e^{-net})
- Step 2: Propagate the error backward
  - For output units, W(new)=W(old)-$\alpha \frac{\partial E}{\partial w}$
  - $\frac{\partial E}{\partial w} = \frac{\partial E}{\partial x} * \frac{\partial x}{\partial net} * \frac{\partial net}{\partial w}$ (chain rule)
  - =(x-d)*f'(net)*y (error * f' *output)
  - (Notice that f(net) must be differentiable!)
  - Gives an error "activation" signal at each node
  - For earlier layers, error is simply propagated backwards, e.g. :
    - $error_1$ (layer 2)= $e*w_1*f'(y_1)$

BACKPROPAGATION!



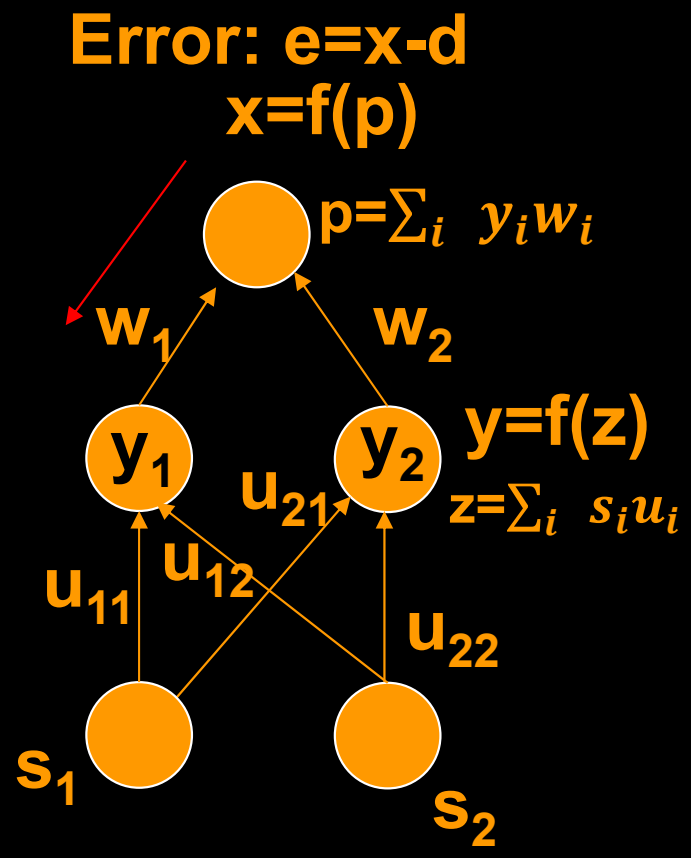
Overall error:

$$E = \frac{1}{2} \sum_c \sum_j (y_{j,c} - d_{j,c})^2$$

$$\frac{\partial E}{\partial x} = 2e \qquad \frac{\partial x}{\partial net} = f'(net) \qquad \frac{\partial net}{\partial w} = y$$

Error backpropagation derivation

- Suppose lower level weights u_{ij} and lower inputs s_k, and signal y=f(z)
- How do we change u_{ij}??
- Answer: Chain rule.
- start with:



$$\frac{\partial E}{\partial u} = \frac{\partial E}{\partial x}\frac{\partial x}{\partial p}\frac{\partial p}{\partial y}*\frac{\partial y}{\partial z}\frac{\partial z}{\partial u}$$

$$e*f'(p)*w\ *f'(z)*s$$

Backpropagate error (ε) → $\frac{\partial E}{\partial u} = \varepsilon*f'(z)*s$

Delta rule at layer 2! (with derivative, and just add learning rate)

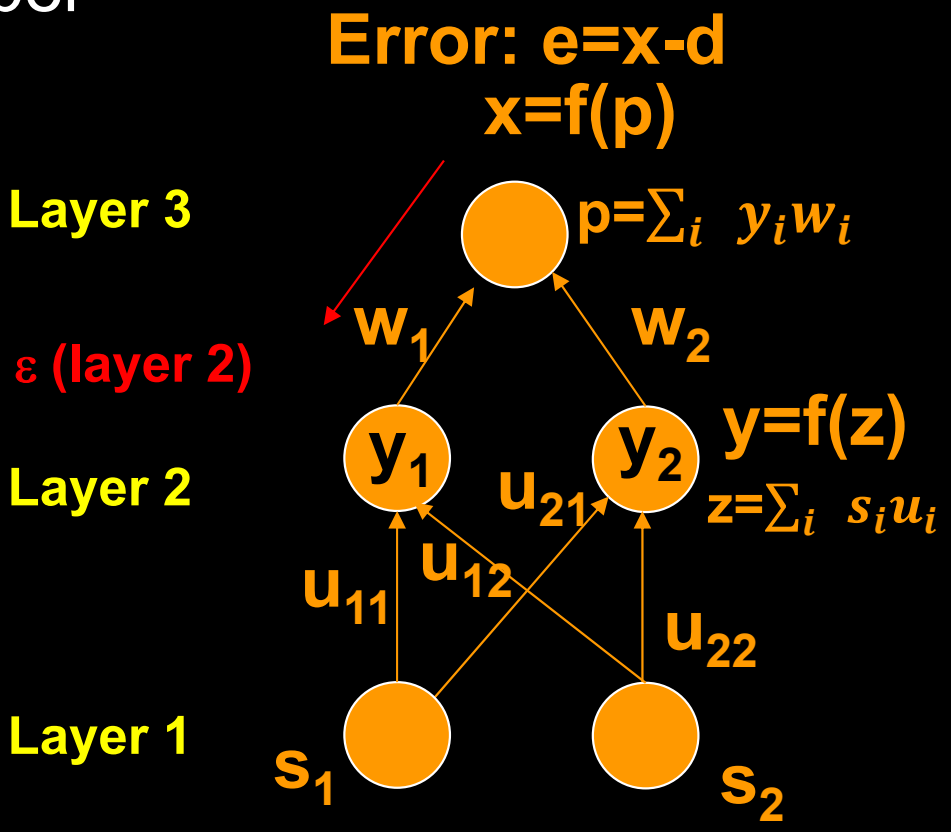
Error backpropagation derivation

- But there is more than one unit per layer!
- $\frac{\partial E}{\partial u} = \frac{\partial E}{\partial x}\frac{\partial x}{\partial p}\frac{\partial p}{\partial y}*\frac{\partial y}{\partial z}\frac{\partial z}{\partial u}$

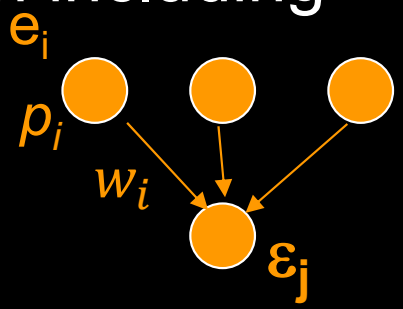
$$e*f'(p)*w \; *f'(z)*s$$

Backpropagate error (ε)

→ $\frac{\partial E}{\partial u} = \varepsilon*f'(z)*s$



- With a little more math including multiple units:
- $\varepsilon_j = \sum_i e_i w_i f'(p_i)$



ε_j is the lower layer's "error"
→Just apply delta rule to lower layer!

Backpropagation

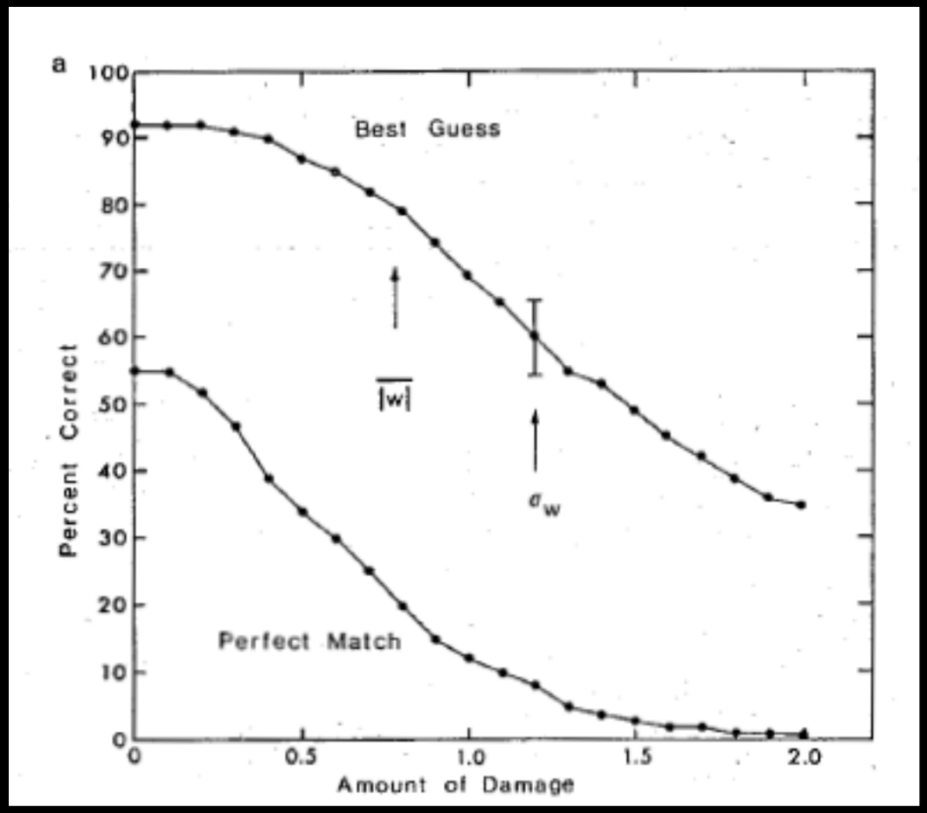
- Overview of backprop: https://www.youtube.com/watch?v=Ilg3gGewQ5U

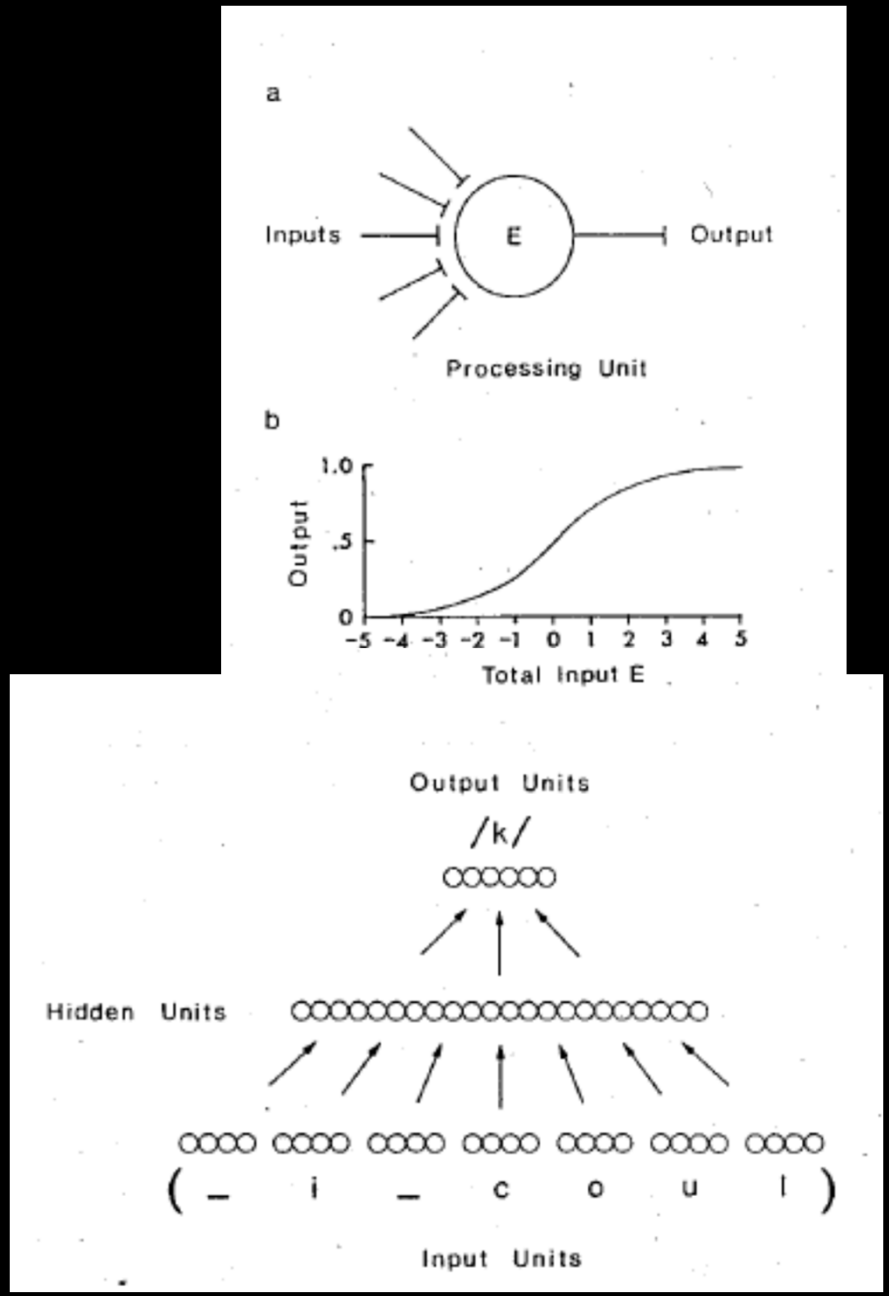
- With at least one hidden layer, could solve the XOR problem
- Made a huge splash from 1986 onward (PDP book cited 18000+ times!)
- Could learn to map arbitrary input patterns to arbitrary output patterns
- Nowadays, use about 6 layers instead of 3, and call it “deep learning”. Next thing you’ll be making $$ working for google!

- Important: must divide data into “training set” vs. “test set”
  - Train the weights on the training set
  - Then test for performance on the test set

Backpropagation - NetTalk

- Nettalk (Sejnowski & Rosenberg, 1986) – use backpropagation to train a text-to-speech mapper.
- Graceful degradation when damaged, due to parallel distributed processing (PDP)





Backpropagation - NetTalk

- Give the output a “child’s voice”

- Nettalk demo
- https://www.youtube.com/watch?v=gakJIr3GecE

- Claim that backpropagation is a model of children learning to talk:

> During the early stages of learning in NETtalk, the sounds produced by the network are uncannily similar to early speech sounds in children (26). However,

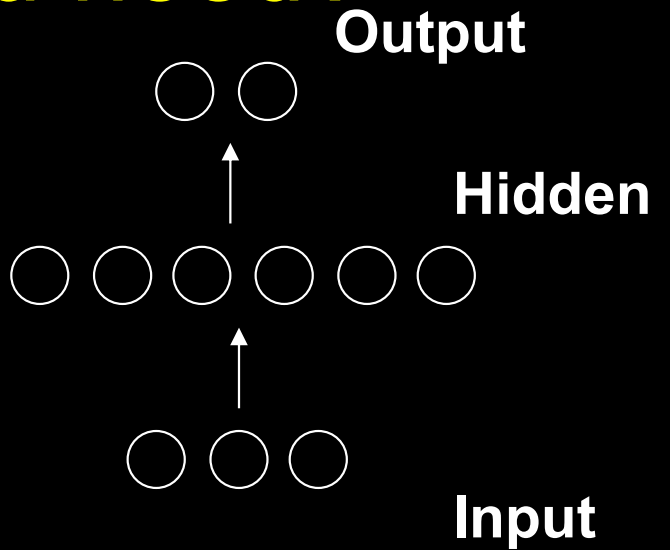
- Example of a totemic model, in which the totem is believed to (magically) take on the properties of the system being modeled.

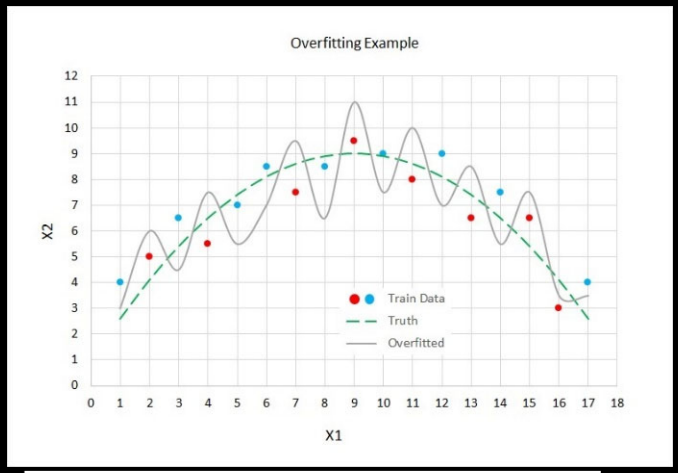
Python backprop exercise

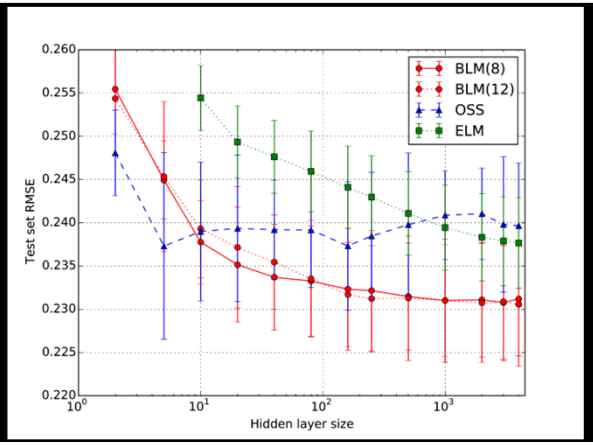
- Siraj simple backprop exercise, relevant to Assignment 5:
- https://www.youtube.com/watch?v=h3l4qz76JhQ

How many hidden units do you need?

- More hidden units means the network can learn more complicated mappings more accurately
- BUT more hidden units means more training examples are required, and there is a risk of overfitting
- Overfitting means learning the training data very well but WITHOUT being able to generalize to the test set, so that test set performance is not good
- Bottom line: tradeoff between capacity and overfitting. Find just the right number of hidden units; more is not better





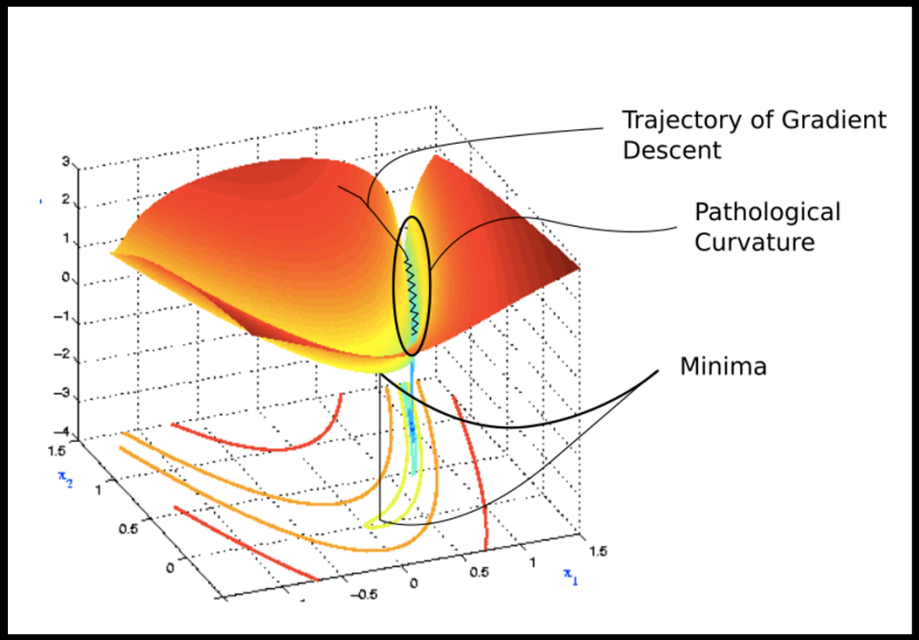


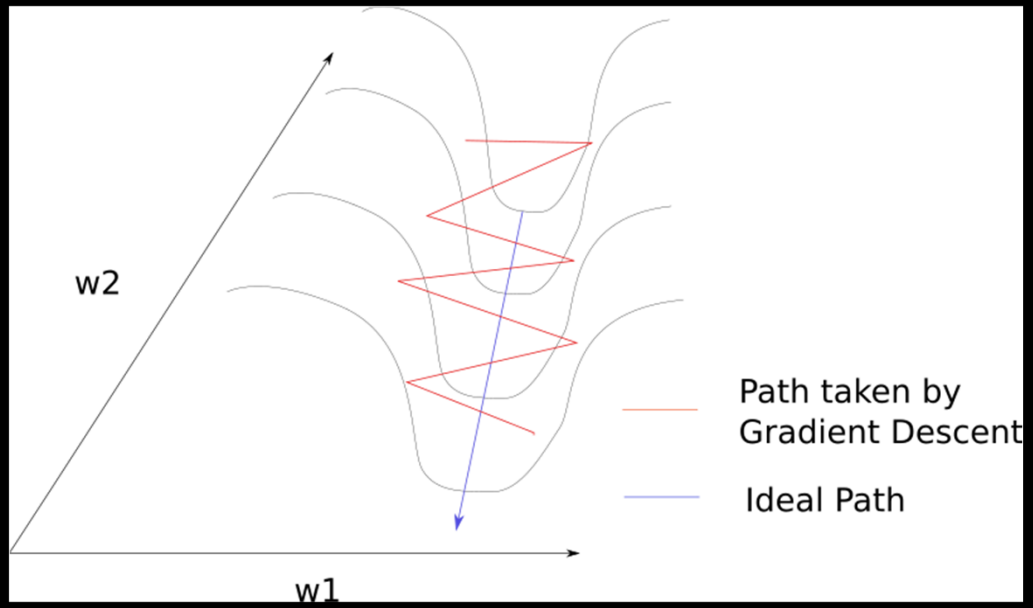
Better cost functions?

- Usually the sum squared error is used as a cost function, but can also use cross-entropy
- Cross entropy:
- $C = -\frac{1}{N}\sum_j \quad \sum_N [d \ln x + (1-d)\ln(1-x)]$
- Where d is the desired output, and x is the actual output, and j indexes the output neurons (i.e. how many units in the output layer), and N is the number of training items
- The weight update rule for cross entropy then derives from $\frac{\partial C}{\partial w} = \frac{1}{N}\sum_x x_j(\sigma(x) - y)$
  - Note: No derivative of the sigmoid here, so learning doesn't slow down when the output is very large or very negative
- Learning is: W(new)=W(old) - $\alpha \frac{\partial C}{\partial W}$

Learning modifications

- What happens when the Error function (E) has a narrow ridge ("pathological curvature")?
- Problem: Weight updates are inefficient





https://blog.paperspace.com/intro-to-optimization-momentum-rmsprop-adam/

Learning modifications

- One solution: Momentum
- For each weight element, keep a moving average of the updates. This tends to cancel out zig-zag oscillations. η (eta) typically 0.5→0.9

Repeat Until Convergence {

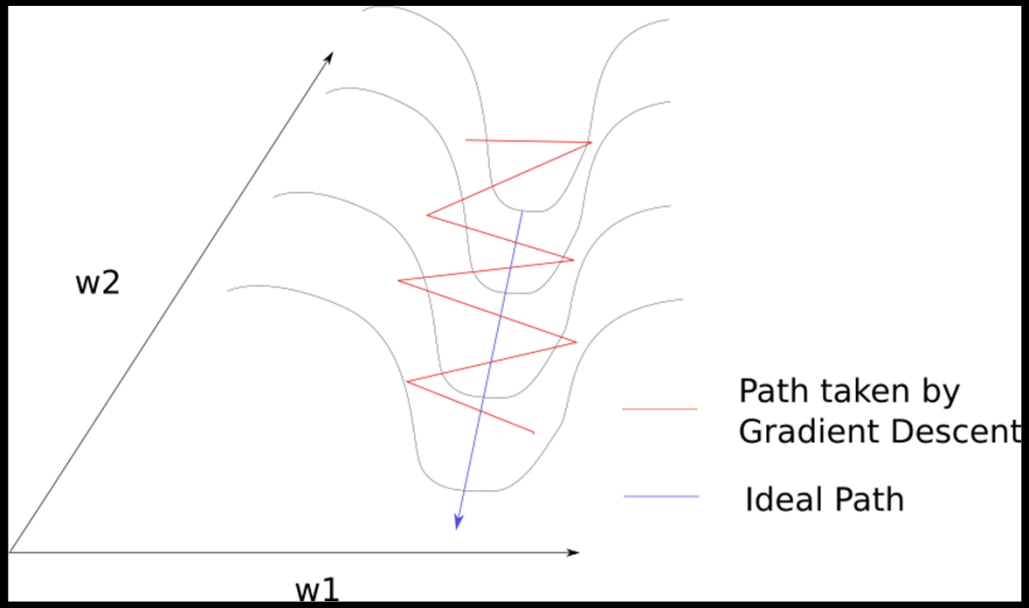
$$\nu_j \leftarrow \eta * \nu_j - \alpha * \nabla_w \sum_1^m L_m(w)$$

$$\omega_j \leftarrow \nu_j + \omega_j$$

}

$\eta * \nu_j$

Coefficient of Momentum — *Retained Gradient*



https://blog.paperspace.com/intro-to-optimization-momentum-rmsprop-adam/

Learning modifications, part 2

- Another solution: Slow down updates with big gradients (RMSprop)
- Estimate running average *square* of gradient
- Divide weight vector gradient by its root-mean-square
- Update the weight

For each Parameter w^j

(*j subscript dropped for clarity*)

$$\nu_t = \rho\nu_{t-1} + (1-\rho) * g_t^2$$

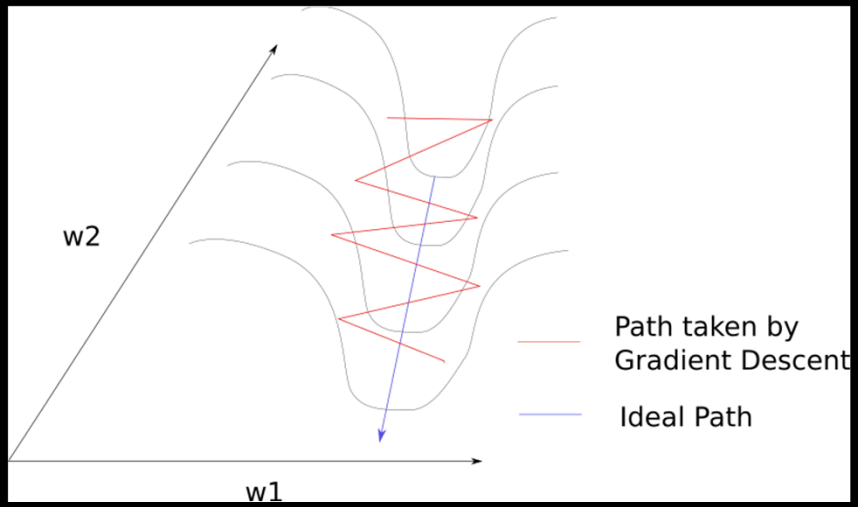
$$\Delta\omega_t = -\frac{\eta}{\sqrt{\nu_t + \epsilon}} * g_t$$

$$\omega_{t+1} = \omega_t + \Delta\omega_t$$

η : *Initial Learning rate*

ν_t : *Exponential Average of squares of gradients*

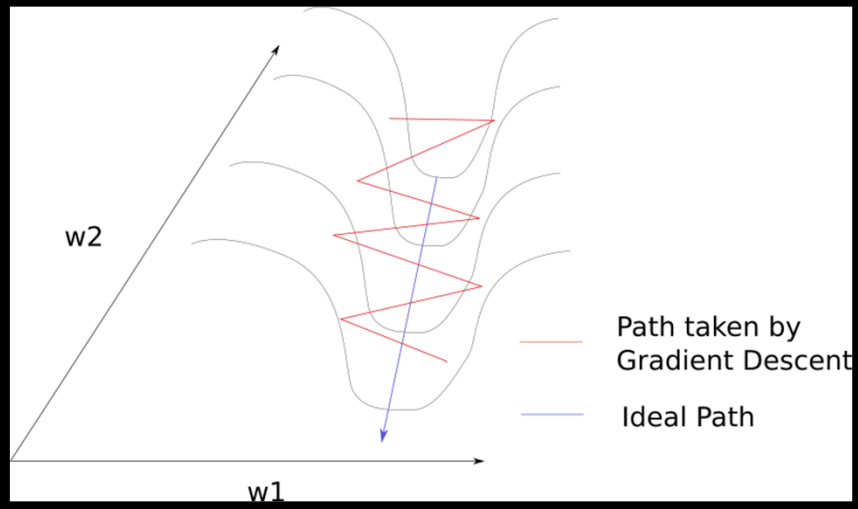
g_t : *Gradient at time t along* ω^j



https://blog.paperspace.com/intro-to-optimization-momentum-rmsprop-adam/

Learning modifications, part 3

- Better solution: combine momentum and RMSprop = Adam (adaptive moment optimization)
- Adam is currently a commonly used training method
- Momentum
- RMS
- Combine both for Δw



For each Parameter w^j

(*j subscript dropped for clarity*)

$$\nu_t = \beta_1 * \nu_{t-1} - (1 - \beta_1) * g_t$$

$$s_t = \beta_2 * s_{t-1} - (1 - \beta_2) * g_t^2$$

$$\Delta\omega_t = -\eta \frac{\nu_t}{\sqrt{s_t + \epsilon}} * g_t$$

$$\omega_{t+1} = \omega_t + \Delta\omega_t$$

η : *Initial Learning rate*

g_t : *Gradient at time t along* ω^j

ν_t : *Exponential Average of gradients along* ω_j

s_t : *Exponential Average of squares of gradients along* ω_j

β_1, β_2 : *Hyperparameters*

https://blog.paperspace.com/intro-to-optimization-momentum-rmsprop-adam/