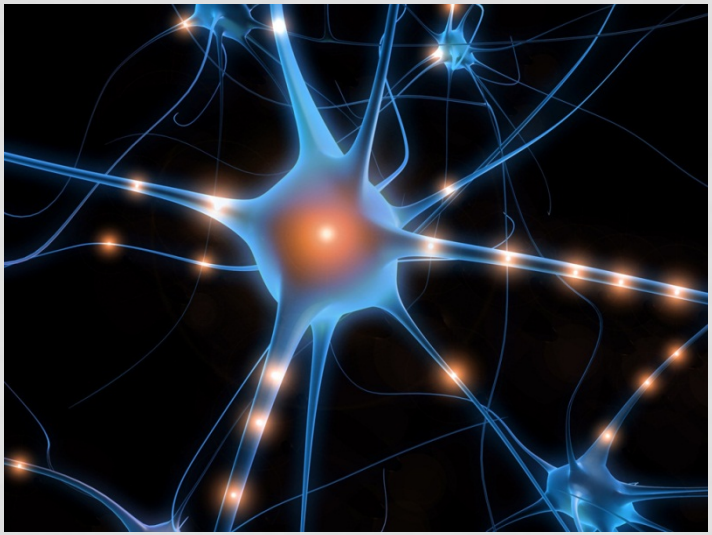
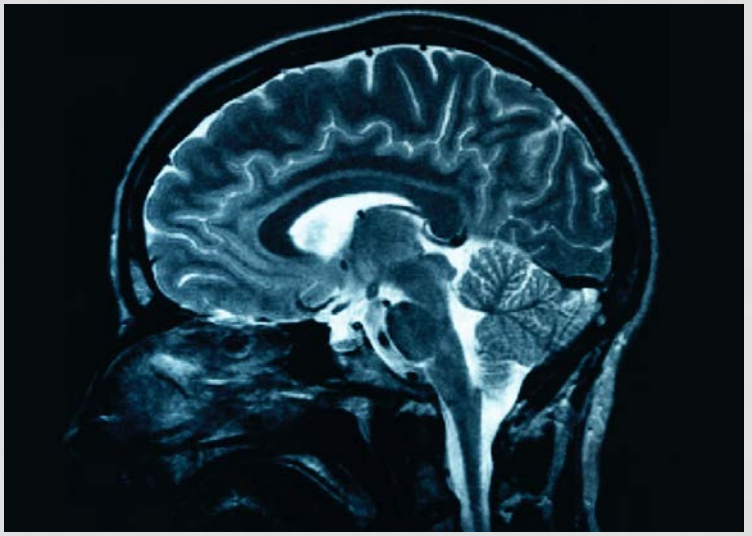




NEURAL NETWORKS

REFERENCES:

- "ARTIFICIAL INTELLIGENCE FOR GAMES"
- "ARTIFICIAL INTELLIGENCE: A NEW SYNTHESIS
- HTTPS://MATTMAZUR.COM/2015/03/17/A-STEP-BY-STEP-BACKPROPAGATION-EXAMPLE/ "

HISTORY

- In the 70's vs today
- variants
  - **(Multi-layer) Feed-forward (perceptron)**
  - Hebbian
  - Recurrent
  - ...

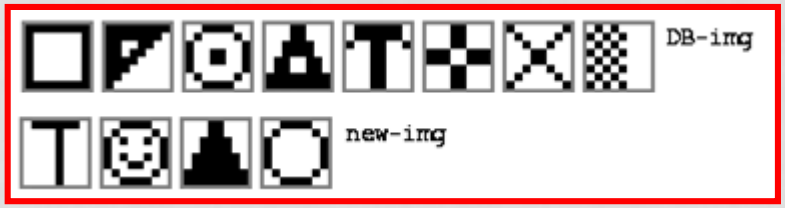
OVERVIEW

- Givens:
- Process:
- Using the Neural Network.
- Basically: a **classifier**.

EXAMPLE APPLICATION #1

- Modeling a enemy AI.
  - **Supervised training**
  - **Unsupervised training**

EXAMPLE APPLICATION #2 (I.E. THE BORING LAB)



PERCEPTRONS

- Modeled after a single neuron.
- Components:
  - Dendrites (Input)
  - Axon (Output)
  - Soma (Activation Function)

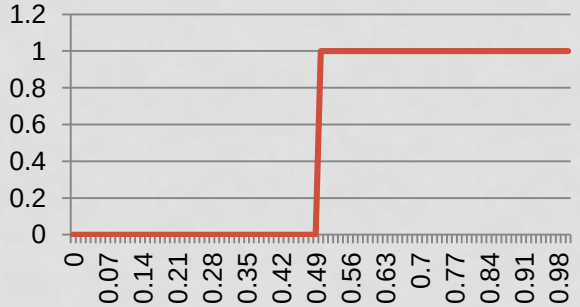
PERCEPTRON ACTIVATION FUNCTIONS

- Notation:
  - $\vec{I}$ the vector of all input values. For us, input values are 0 or 1.
  - $\vec{W}$ the vector of weight values (same size as I)
    - Positive, or negative, no limit.
    - I usually initialize to -0.5 to +0.5.
  - $\sum = \vec{I} \bullet \vec{W}$ the total (weighted) input to the perceptron.

PERCEPTRON ACTIVATION FUNCTIONS, CONT.

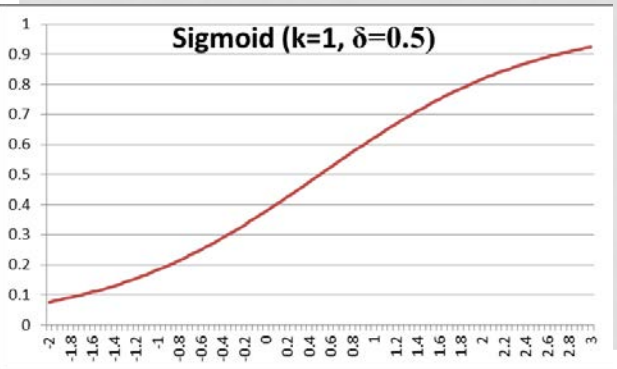
- Square function:

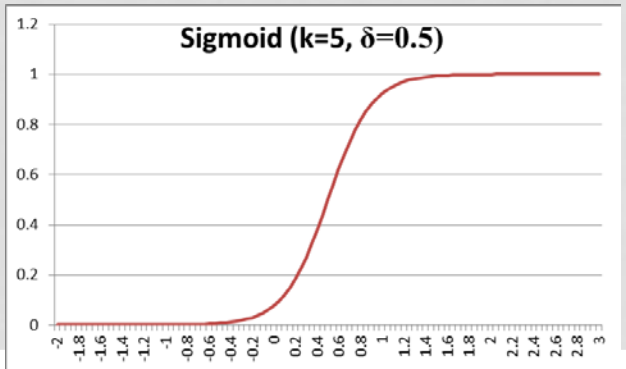
$$f(\Sigma) = \begin{cases} 1 \; if \; \Sigma \geq \delta \\ 0 \; if \; \Sigma < \delta \end{cases}$$

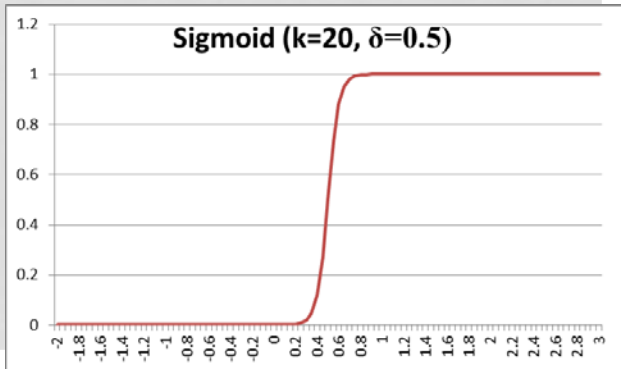


- Sigmoid [logistic] function (the one we'll use):

$$f(\Sigma) = \frac{1}{1 + e^{-k(\Sigma - \delta)}}$$







EXAMPLES

- Two inputs, one output, k=5.0
- AND

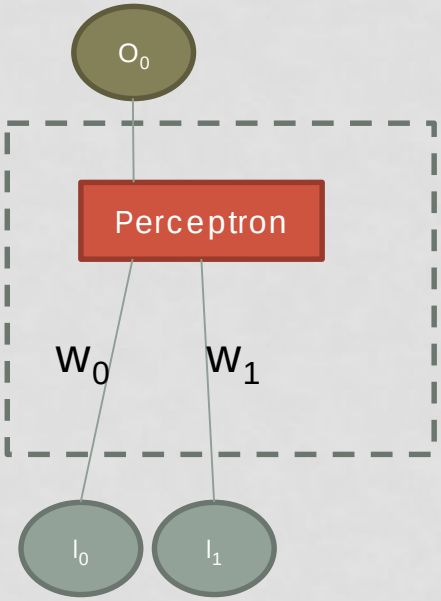
$\vec{W} = [0.5 \quad 0.5]$ δ=0.75

Input	Σ	f(Σ)
[0 0]	0.0	0.023
[0 1]	0.5	0.223
[1 0]	0.5	0.223
[1 1]	1.0	**0.7773**

- OR

$\vec{W} = [0.5 \quad 0.5]$ δ=0.75

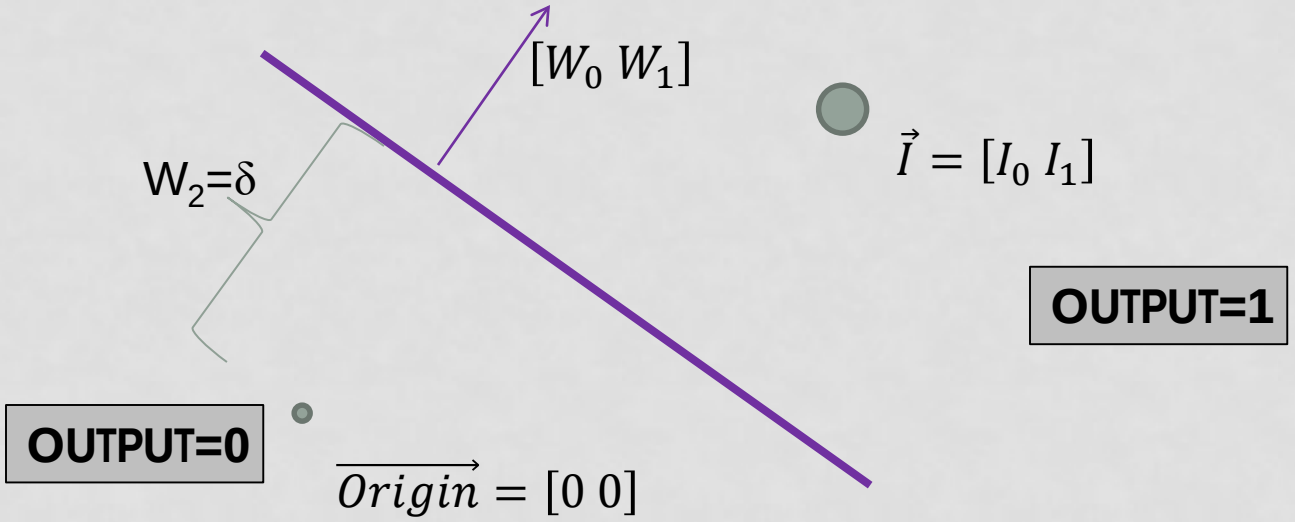
Input	Σ	f(Σ)
[0 0]	0.0	0.119
[0 1]	0.5	**0.622**
[1 0]	0.5	**0.622**
[1 1]	1.0	**0.953**



- XOR
  - Problem!
  - Can't be done with a single perceptron

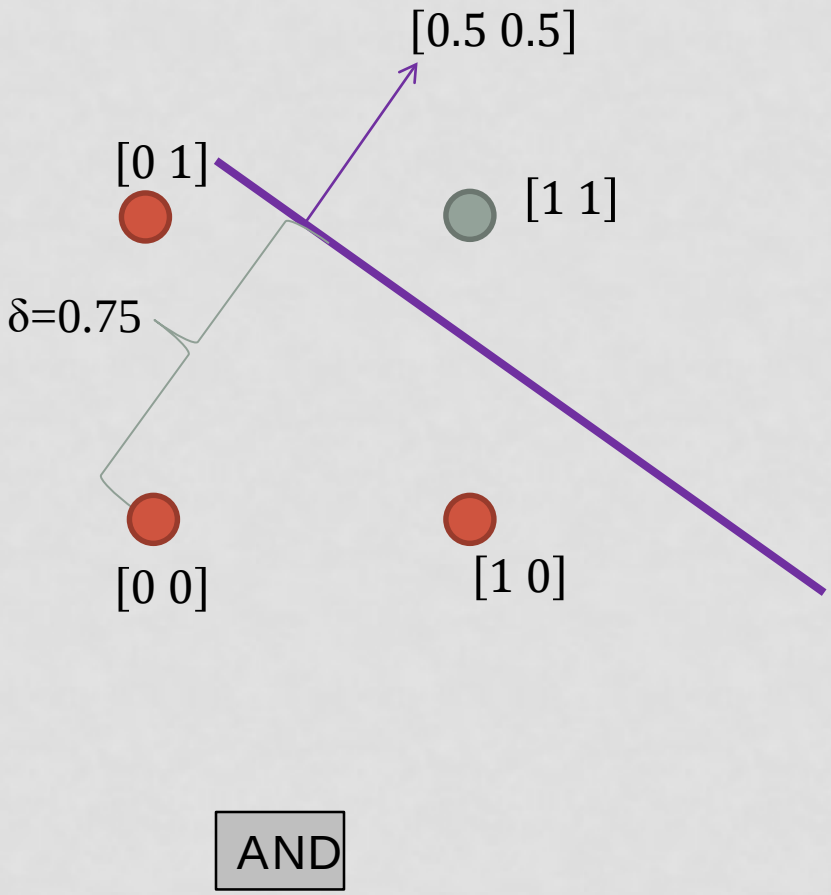
WHY NOT XOR?

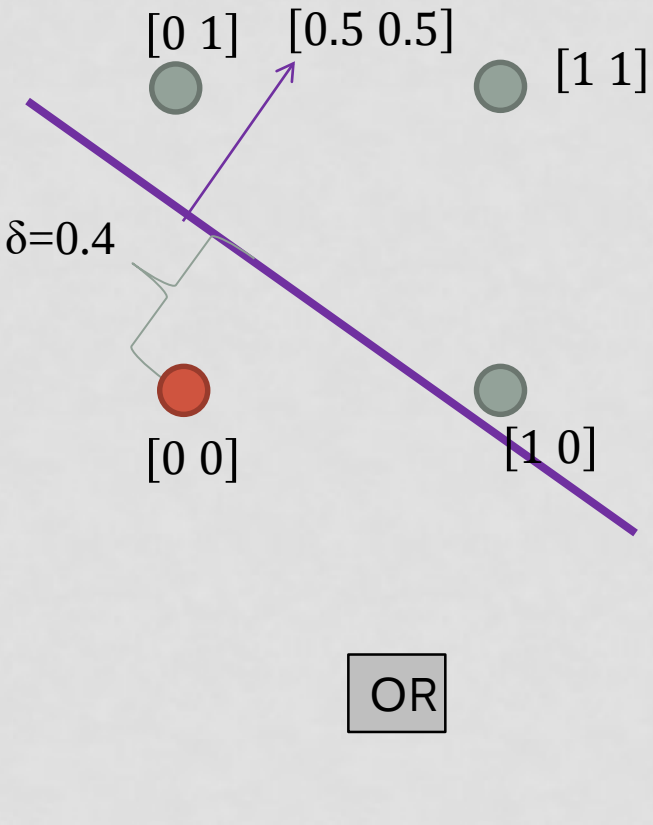
- The first two examples are **linearly separable**



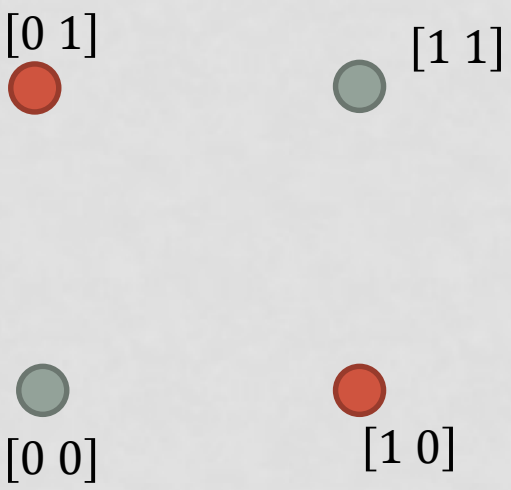
In higher dimensions, the "dividing line" is a **hyper-plane**, not a plane

AND AND OR



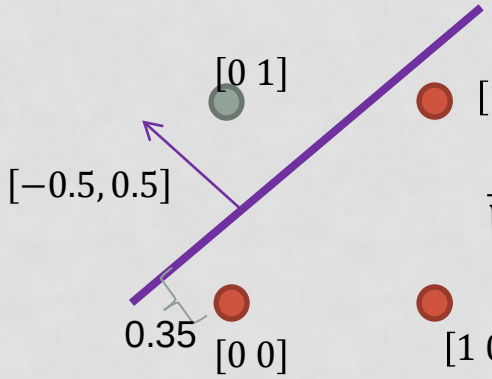


XOR



You can't draw a line to separate the "True"'s from the "False"'s

MULTI-LAYER PERCEPTRON NETWORKS

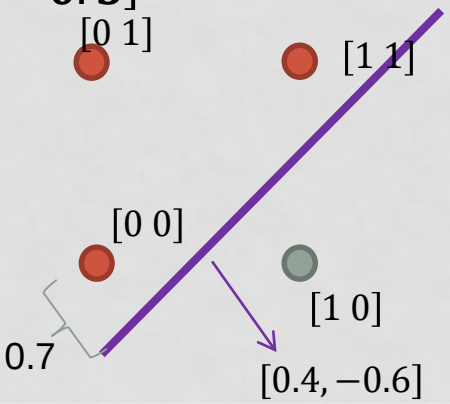


$\overrightarrow{W_0} = [-0.5 \quad 0.4]$

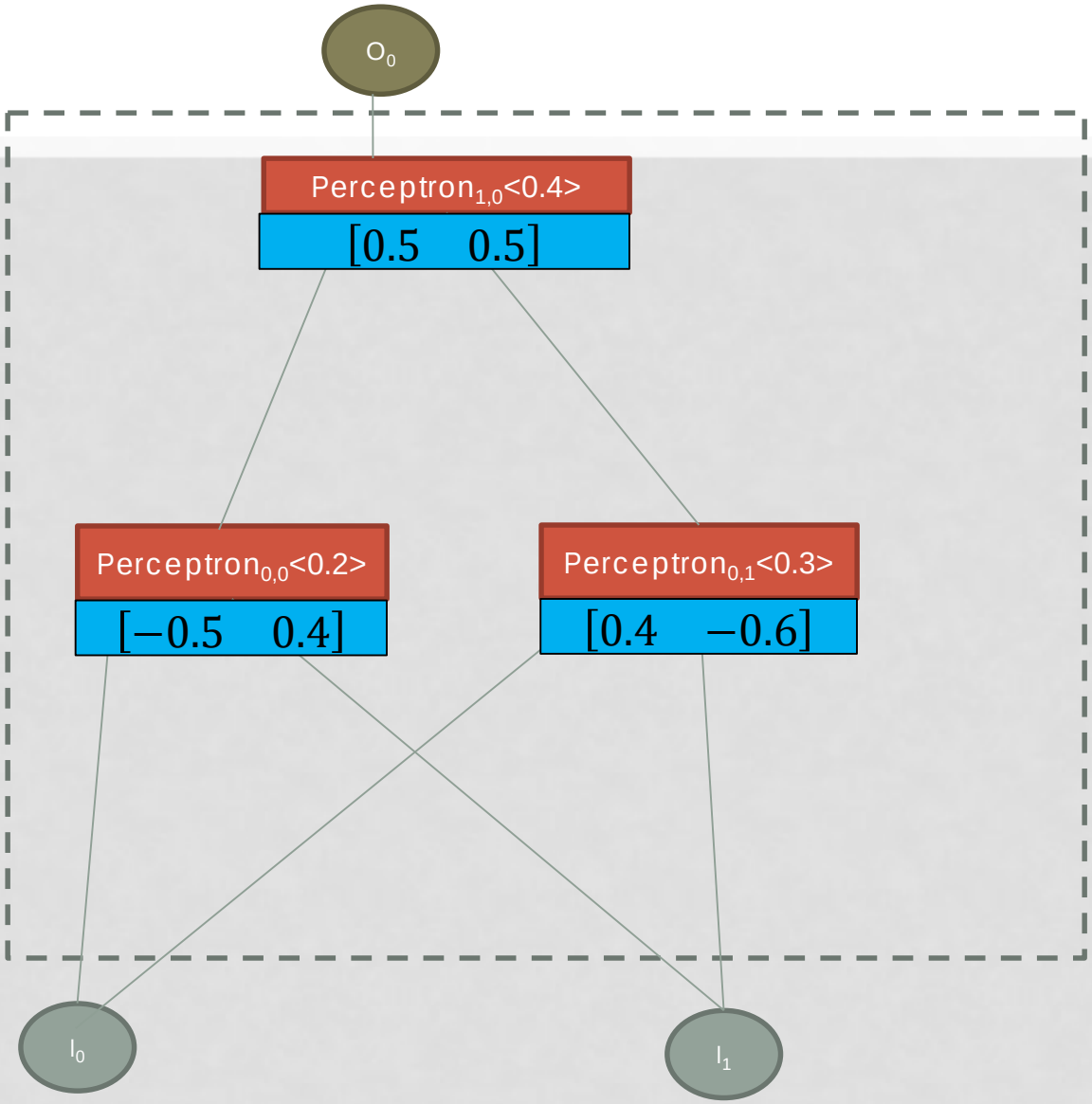
δ=0.35

$\overrightarrow{W_1} = [0.5 \quad -0.5]$

δ=0.35



XOR NNET


O_0
$Perceptron_{1,0}$<0.4>
[0.5 0.5]
$Perceptron_{0,0}$<0.2>
[−0.5 0.4]
$Perceptron_{0,1}$<0.3>
[0.4 −0.6]
I_0
I_1

FEED FORWARD

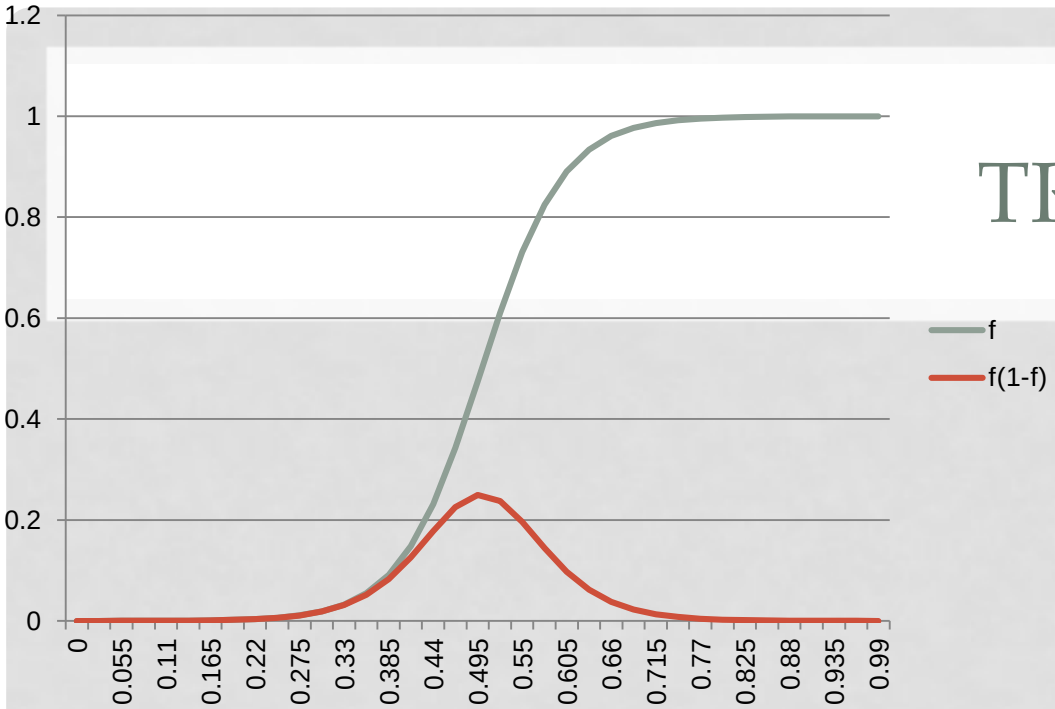
- Feed Input to "bottom" of NNet
- Each Perceptron outputs its activation result to next layer as input
  - I used a k=20 here

Input	Step#	P(layer,#)	Σ	f(Σ)
[0,0]	1	P0,0	0.0	0.001
	2	P0,1	0.0	0.001
	3	P1,1	0.001	**0.0003**
[0,1]	1	P0,0	0.5	0.953
	2	P0,1	-0.5	4.14e-8
	3	P1,1	0.476	**0.821**
[1, 0]	1	P0,0	-0.5	4.14e-8
	2	P0,1	0.47	0.953
	3	P1,1	0.476	**0.821**
[1, 1]	1	P0,0	0.0	0.001
	2	P0,1	0.0	0.001
	3	P1,1	0.001	**0.0003**

TRAINING INTRO

- Feed Forward is *using* a Nnet.
- Training is *creating* a NNet.
- Error-correction procedure (single perceptron)
  - $\overrightarrow{Wnew} = \vec{W} + c * \vec{I} * f' * Err$
  - $\delta' = \delta - c * f' * Err$
    - BTW (Keith): the derivation of this formula involves partial-derivatives☺

TRAINING INTRO, CONT.



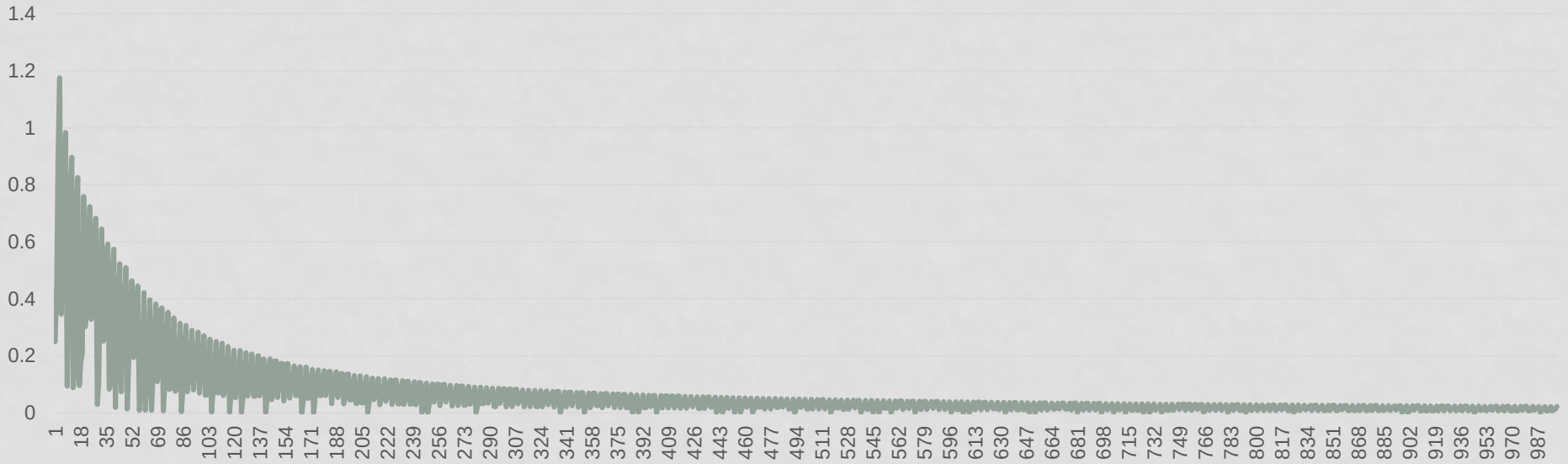
$$\overrightarrow{Wnew} = \vec{W} + c * \vec{I} * f' * Err$$

$$Err = (d - o)$$

- Observations

MY TRAINING RESULTS

- One perceptron, trying to learn ADD
- Initial weights and threshold = 0
- total_error = sum of err^2 for all 4 cases (shuffled)
- Repeated for 1000 iterations



TRAINING (1 LAYER, M OUTPUTS)

- Similarities and differences
- $\overrightarrow{Err} = \overrightarrow{desired} - \overrightarrow{actual}$
- Correction update
  - *for i in range(num_perceptrons):*
    - $\overrightarrow{W_i} \mathrel{+}= c * \vec{I} * f_i' * \overrightarrow{Err_i}$
    - $\delta \mathrel{-}= c * f_i' * \overrightarrow{Err_i}$
- I found this useful:
  - $total_{err} = \left\|\overrightarrow{Err}\right\|$ (i.e. vector-magnitude)

TRAINING (N LAYERS, M OUTPUTS)

- The most general case.
- For all perceptrons, calculate the following (in "reverse" (output to input))
  - $Blame_{i,j} = \begin{cases} \overrightarrow{Err_j} & if\ layer\ i\ is\ an\ output\ layer \\ \sum_{k=0}^{n} wt_{j \Rightarrow k} * Blame_{i+1,k} & else \end{cases}$
  - $\overrightarrow{Source_i} = \begin{cases} \vec{I} & if\ i\ is\ this\ is\ the\ \text{bottommost}\ layer \\ Output\ of\ layer\ i-1 & else \end{cases}$
- Then to update a perceptron:
  - $\overrightarrow{W_{i,j}} \mathrel{+}= c * \overrightarrow{Source_i} * f'_{i,j} * Blame_{i,j}$
  - $\delta \mathrel{-}= c * f'_i * Blame_{i,j}$
- Make sure you use the *existing* weights for blame calculations.
  - Perhaps wait until the back-prop is done to update weights?

MY SUGGESTION

- Start with a Perceptron / NNet class
- Manually set up AND, OR, XOR, test feed-forward
- Learn AND
- Learn XOR
- Learn XOR w/ weird structure
  - 2 input, 1 output
  - hidden-size=[5, 3]
- Learn a challenging problem (i.e. the lab)