Massachusetts Institute of Technology

Department of Electrical Engineering and Computer Science
6.034 Artificial Intelligence, Fall 2011
Recitation 8, November 3 **Corrected Version & (most) solutions**

Neural networks II Prof. Bob Berwick, 32D-728

0. Introduction: the summary so far (and solutions from last time)

Summary of neural network update rules:

To update the weights in a neural network, we use *gradient ascent* of a performance function P by comparing what a network outputs given a sample data point and given the network's current weights, via **forward propagation**. We compare the network's **o**utput value against the **d**esired value in terms of the partial derivative of $P = -1/2(d-o)^2$ with respect to particular weights w_i. This is called **backpropagation.** Recall that the first step is to find the derivative of P with respect to the output, which turns out to be: (d–o).

The general formula for the change in weights is:

$$\Delta w \propto \frac{\partial P}{\partial w} \text{ so } w' = w + \alpha \times \frac{\partial P}{\partial w} \text{ where } \alpha \text{ is a rate constant (also } r \text{ in the literature \& quizzes)}$$

The value of alpha (aka r) is also the "step size" in the hill-climbing done by gradient ascent, using the performance fn.

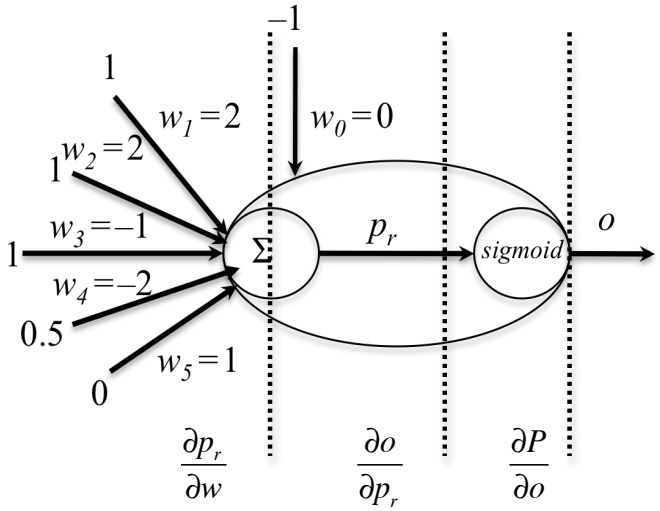
For the **final** layer in a neural network, whose output from forward propagation is o_f and where the desired output value is d, the required change in weight value for a (single) final weight is:

1. $\Delta w_r = \Delta w_f = \alpha \times \delta_f \times i_f$ where $\delta_f = o_f(1-o_f) \times (d-o_f)$

For the **previous** layer in a neural network (just the rightmost layer if a single neuron), the required update equation is:

2. $\Delta w_l = \alpha \times o_l(1-o_l) \times w_r \times \delta_f \times i_l$

Example 1. Last time we computed the weight updates for a single-layer neural network with 6 inputs and 6 weights. Each partial derivative in the figure below corresponds to a different part of the network, with their product yielding the derivative of P with respect to the weights w, where the desired output was 1, and the learning rate alpha was (arbitrarily) set to 100:



Step 1: Forward Propagation. Calculate the output o given the input values shown. Useful data point: *sigmoid*(2)= 0.9

Answer: $-1 \times w_0 +$ ___1___ $\times w_1$ ___1___ $\times w_2 +$ __1___ $\times w_3 +$ __0.5_ $\times w_4 +$ ___0___ $\times w_5 = p_r = 2$

Sigmoid (p_r) = o_f = _____0.9_____

Step 2: Backpropagation to find delta for final, output layer.

$$\delta_f = \frac{\partial P}{\partial p_r} = \frac{\partial P}{\partial o}\frac{\partial o}{\partial p_r} = (d - o_f) \times [o_f(1 - o_f)]$$

$$\frac{\partial P}{\partial w} = \frac{\partial P}{\partial o}\frac{\partial o}{\partial p_r}\frac{\partial p_r}{\partial w} = (d - o_f) \times [o_f(1 - o_f)] \times i_f = \delta_f \times i_f$$

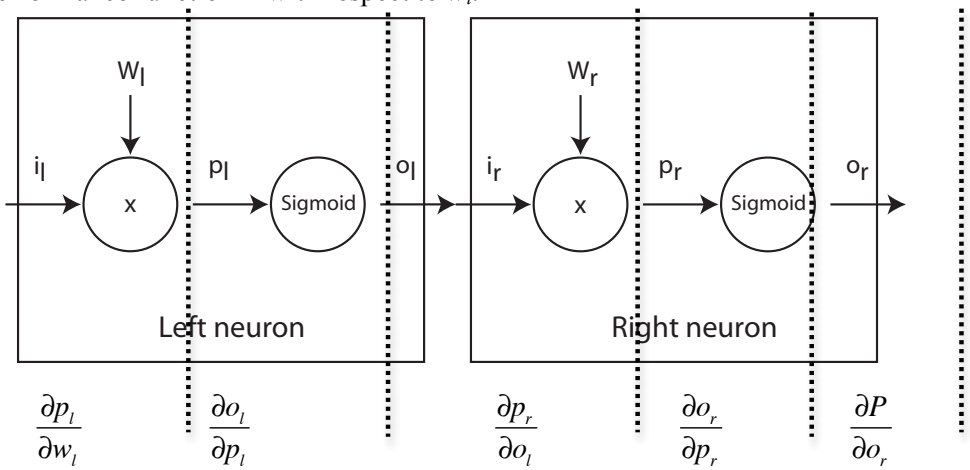
$\Delta w_f = \alpha \times i_f \times \delta_f$ (for each input line to the neuron, i)

$w_i' = w_i + \Delta w_f$ (for each input line to the neuron, i)

NEW WEIGHT	ORIGINALWEIGHT +	RATE ×	δ ×	INPUT =	NEW WT
w'	w	α	$(d\text{-}o)(o)(1\text{-}o)$	i_f	
$w_0' =$	0	100	0.009	–1	–0.9
$w_1' =$	2	100	0.009	1	2.9
$w_2' =$	2	100	0.009	1	2.9
$w_3' =$	–1	100	0.009	1	–0.1
$w_4' =$	–2	100	0.009	0.5	–1.55
$w_5' =$	1	100	0.009	0	1

Note how weights that have an input of 0 to them can't affect the performance, so they remain unchanged.
So, do these new weights get us closer to the desired output? If we run forward propagation again we can find out: $-1 \times 0.9 + 1 \times 2.9 + 1 \times 2.9 + 1 \times -0.1 + 0.5 \times -1.55 + 0 \times 1 = 6.65$; sigmoid(6.65) = 0. 99870765

Example 2. Last time, we also computed the value for the weight change for the final, output neuron of the simple two-neuron net below. We initially have i_l=1; both weights w_l and w_r = 0; and the desired output is 1. We will finish up this problem now.
For completeness, and to cement our understanding, let's see how the various terms in the partials are arrayed over this two-neuron diagram, pushing back from the output o_r, so you can see why it is called **backpropagation.** Make sure you **understand** where each of the five terms comes from. Multiplied together, they give the partial derivative of the performance function P with respect to w_l.



$\frac{\partial p_l}{\partial w_l}$	$\frac{\partial o_l}{\partial p_l}$	$\frac{\partial p_r}{\partial o_l}$	$\frac{\partial o_r}{\partial p_r}$	$\frac{\partial P}{\partial o_r}$
i_l	$o_l(1-o_l)$	w_r	$o_r(1-o_r)$	$(d-o_r)$
left weight deriv	left sigmoid deriv	right weight deriv	right sigmoid deriv	output deriv

This is to find the partial of the performance function P with respect to the left right, w_l. Remember that to find the corresponding partial for the **final** output layer we computed something a bit different: the partial of p_r with respect to o_l is **replaced** with the partial of p_r with respect to w_r (so finding the partial of P with respect to w_r.) But this partial is just the derivative of $i_r \times w_r$ with respect to w_r, which is simply i_r. **Note how** if we decided to use a different threshold function other than a sigmoid, the **only two** things we would have to change are the two

partial derivatives, the right and the left sigmoid derivatives with respect to p_r and p_l, respectively. For example, if we changed the sigmoid threshold from $1/(1+e^{-x})$ to, say, x^2, then the derivatives would change from, e.g., $o_r(1-o_r)$ (the derivative of the sigmoid function with respect to its input), to just $2o_r$ (and the same for the left threshold derivative).

For this example, assume **all initial weights are 0** (it is actually a bad idea to set all initial weights the same for neural nets; why?). Assume a sample input of $i_l = 1$, and that the *desired* output value d is 1.0. Assume a learning rate of 8.0. (Useful data point: $sigmoid(0) = 0.5$) Let's run one step of backpropagation on this and see what's different about this case. First, as before, we must carry out **forward** propagation: compute the inputs and outputs for each node.

Step 1: Forward Propagation. OK, you should know the drill by now. First compute the outputs z at each node:

$p_l = w_l i_l$ = 0 × 1 = 0 So o_l (= i_r) = $sigmoid$(0) = 0.5

$p_r = w_r i_r$ = 0 × 0.5 = 0 So o_r = $sigmoid$(0) = 0.5

Step 2: Calculate the δ for the output, final layer, δ_f (i.e., the neuron on the right, for use in changing w_r)

Recall the formula for δ_f is: $o_r \times (1-o_r) \times (d - o_r)$= 0.5 × (1–0.5) × (1–0.5) = 0.125 .
Recall that d = 1.0; we have just computed o_r.
So, the change in the right-most weight w_f is: $\alpha_f \times i_r \times \delta_f = 8.0 \times 0.5 \times 0.125 = 0.5$

Step 3: Calculate δ_l for the hidden neuron on the left, recursively using the delta from the previous layer:
$\delta_l = o_l(1-o_l) \times w_r \times \delta_f$ = 0.5(1–0.5) ×0 × 0.125 = 0
So the weight change for the left neuron at the first iteration of back propagation is 0

Thus the two **new weights** are:
w_f' = 0 + 0.5 = 0.5
w_l' = 0 + 0 = 0

Let's see how much closer this has gotten us to the desired output value of 1.0. We do this by another round of forward propagation (and then typically, we would do back-propagation again to get us even closer, many thousands of times.) Your turn now…. (See the tear-off page on the back to estimate the sigmoid to 2 decimal places, or better, user a calculator or python on your laptop…)

Next iteration, forward propagation:

$p_l = w_l i_l$ = 0 × 1 = 0 So o_l (=i_r) = $sigmoid$(0) = 0.5 .

$p_r = w_r i_r$ = 0.5 × 0.5 = 0.25 So $o_r = o_f$ = $sigmoid$(0.25) = 0.56218 .
So, we have definitely gotten a bit closer to our output goal of 1.0.

Next iteration, back-propagation:
Now you try it:

$\delta_f = o_f \times (1-o_f) \times (d - o_f)$ = 0.56218 ×(1– 0.56218) ×(1– 0.56218) = 0.10776 .

$\delta_l = o_l \times (1-o_l) \times w_r \times \delta_f$ = 0.5 ×(1– 0.5) × 0.5 × 0.10776 = 0.01347 .

$\Delta w_f = \alpha \times i_f \times \delta_f$ = 8.0 × 0.5 × 0.10776 = 0.43104

$\Delta w_l = \alpha \times i_l \times \delta_l$ = 8.0 × 1.0 × 0.0134 = 0.1072

$w_f' = w_f + \Delta w_f =$ 0.5 + 0.43104 = 0.943104

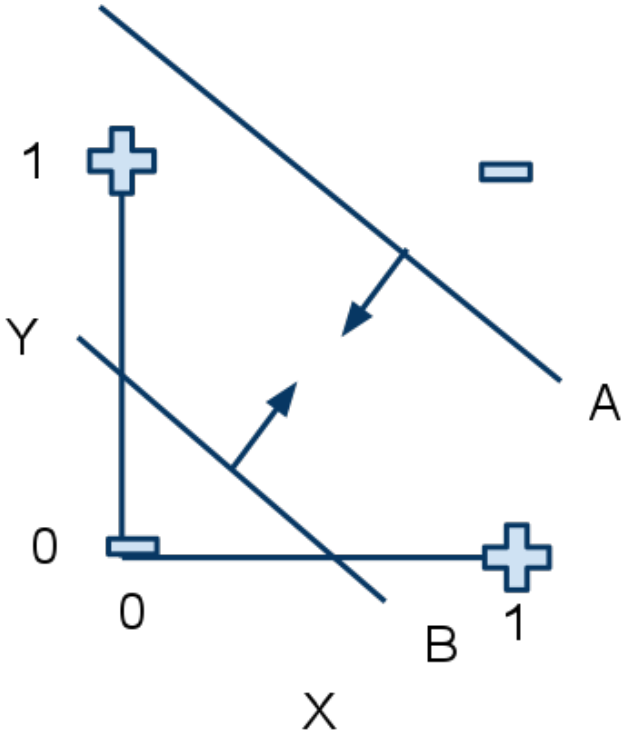
$w_l' = w_l + \Delta w_l =$ 0 + 0.1072 = 0.1072

Do the new weights get us closer to the goal? Calculate this by **forward propagation** again:

$p_l = w_l i_1 =$ 1 × 0.1072 ; o_l (= i_r) = sigmoid(0.1072) = 0.52677

$p_r = w_r i_r =$ 0.943104 × 0.52677 = 0.4968 ; $o_r =$ sigmoid(0.4968) = 0.62171

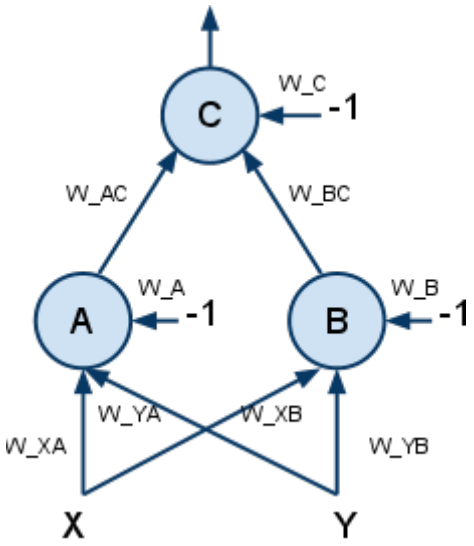
Example 3. What multilayer neural networks can learn that single layer networks cannot learn.
Why did people invent multi-layer neural networks? Consider a classification problem such as the one depicted below, which represents the predicate or the 'concept' of excusive-OR (XOR), i.e., the value of this function is 1 if either of the two inputs is 1; and the value of this function is 0 if both inputs are 0 *or* both inputs are 1. The line A goes through the points (0, 3/2) and (3/2, 0); while the line B goes through the points (0, 1/2), (1/2, 0). The 'plus' signs are at (0,1) and (1, 0); the minus signs at (0,0) and (1,1).



Suppose we tried to find the weights to a *single* layer neural network to 'solve' this classification problem. Then the general formula for this network would be, output = $w_1x + w_2y + c$. But what weights would work? Do you see that this kind of equation can only define a *single* line? Thus, it says that we must classify the + and – regions in the graph above into regions that are all + (positive) and all – (negative), by making a *single* cut through the plane. Can this be done? Try it – why can't it be done? **Answer: you cannot do it, because a perceptron can only define a <u>single</u> line 'cut' through the plane, and this region of + is defined by <u>two</u> lines.**

Question: Can a perceptron encode function |*x*–y| < *epsilon*, for some positive epsilon? Why or why not?
Answer: No, again because the absolute value function requires <u>two</u> cuts through the plane.

However, if we are allowed *two* network layers, then we *can* formulate a set of weights that does the job. Let's see how, by considering the network below, and then finding the weights that do the job. (This was a sample quiz problem previously.)



Step 1. First, think of input-level neurons (neurons A and B) as defining *regions* (that divide positive data points from negative data points) in the *X*, *Y* graph. These regions should be depicted as linear boundary lines with arrows pointing towards the positive data points. Next, think of hidden level neural units (neuron C) as some logical operator (a linearly separable operator) that combines those *regions* defined by the input level units. (We will see later on a few more examples of this sort to show you how multi-layer networks can 'carve up' regions of the plane in this way.)

So in this case: units A, and B represent the **diagonal** boundaries (with arrows) on the graph (definition two distinct ways of separating the space). Unit C represents a logical AND that intersects the two regions to create the bounded region in the middle.

Step 2. Write the line equations for the regions you defined in the graph.

A) The boundary line equation for the region defined by line A:

$$y < -1 \times x + 3/2$$

B) The boundary line equation for the region defined by line B:

$$y > -1 \times x + 1/2$$

Step 3. Rewrite the line equations into the form: $ax + by > c$, where a, b, and c are integers:

A) $y < -1 \times x + 3/2$
$x + y < 3/2$
$2x + 2y < 3$

B) $y > -1 \times x + 1/2$
$x + y > 1/2$
$2x + 2y > 1$

Now note that the sum of the weights times the inputs for each unit can also be written in a similar form. (We will call this summed product of weights times the inputs for a neuron its "*z*" value).

For Unit A: $z =$
$W_{XA}\,x + W_{YA}\,y + W_A(-1) > 0$
$W_{XA}\,x + W_{YA}\,y > W_A$

For Unit B: $z =$
$W_{XB}\,x + W_{YB}\,y + W_B(-1) > 0$
$W_{XB}\,x + W_{YB}\,y > W_B$

Why do we set $W_{XA}\,x + W_{YA}\,y + W_A(-1) > 0$ and not < 0? Look at the graph on the tear-off sheet!
When $z = W_{XA}\,x + W_{YA}\,Y + W_A(-1) > 0$, then sigmoid($z > 0$), and z grows and approaches 1, which corresponds to the *positive* points/
When $z = W_{XA}\,x + W_{YA}\,Y + W_A(-1) < 0$, then sigmoid($z < 0$), z decreases and approaches 0, which corresponds to the negative points.
Thus, when expressed as > 0 the region is defined as **pointing towards** the **positive** points.
But when expressed as < 0, the region is defined as **pointing towards** the **negative** points.
We want the defined region to point to the positive points. So, we must adjust the equation for line (A) so that it has the inequality in the form > (rather than as <). We can do this by multiplying through by a –1, which will reverse the inequality, so the equation for line A becomes:
–2x–2y > –3
Now we are ready for the next step.

Step 5. Easy! Just read off the weights by correspondence.
$-2\,x + -\,2\,y > 3$ line A's inequality
$W_{XA}\,x + W_{YA}\,y > W_A$ z equation for unit A. Therefore, $W_{XA} = -2$ $W_{YA} = -2$ $W_A = -3$

$2\,x + 2\,y > 1$ line B's inequality
$W_{XB}\,x + W_{YB}\,y > W_B$ z equation for unit B. Therefore, $W_{XB} = 2$ $W_{YB} = 2$ $W_B = 1$

Step 6. Solve the logic in the second neuron layer
The equation for the second layer unit C is W_{AC}(output from A) + W_{BC}(output from B) – W_C < or > 0 (where we pick the inequality to satisfy the sigmoid output description mentioned above – if we want the output to be 0 from the logic unit, then we want the sigmoid to go negative, so we want < 0; if we want he output to be 1, then we want the sigmoid to go positive, so we want > 0.)
We now want to compute (A AND B), for the next layer. (Remember, the final region we want is the **intersection** of the regions defined by unit (line) A, and unit (line) B. So we build a Truth table for **And** and solve for the constraints. (**Note** that we are **not** building a truth table for **XOR** – we want **And.)** So we want the output from C to be true (1) iff the outputs from units A and B are both 1, as below.

A	B	desired output	Equations	Simplified
0	0	0	$-W_C < 0$	$W_C > 0$
0	1	0	$W_{BC} - W_C < 0$	$W_{BC} < W_C$
1	0	0	$W_{AC} - W_C < 0$	$W_{AC} < W_C$
1	1	1	$W_{AC} + W_{BC} - W_C > 0$	$W_{AC} + W_{BC} > W_C$

We notice the symmetry in W_{BC} and W_{AC}, so we can make a guess that they have the same value:
$W_{BC} = 2$ and $W_{AC} = 2$

Then the inequalities in the table above condense down to the following:

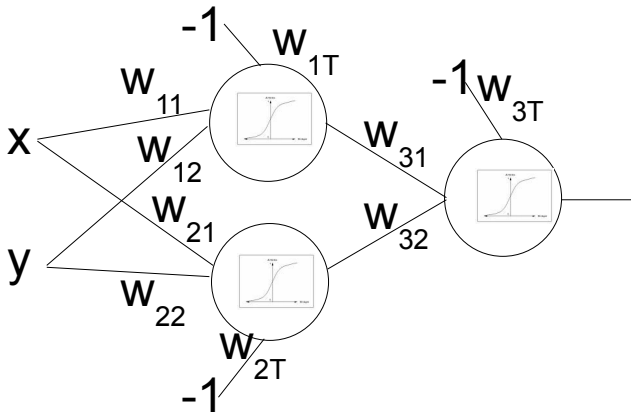
$W_C > 0$ $W_C > 2$ (twice) $W_C < 2+2 = 4$

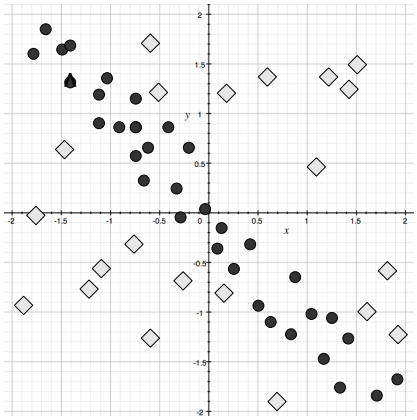
Therefore, $2 < W_C < 4$. Let's make life easy and pick $W_C = 3$. This gives us one acceptable solution:
$W_{BC} = 2$ $W_{AC} = 2$ $W_C = 3$

Of course, there are many solutions. The following solution also works, because it still obeys the inequalities and the constraints in the table:
$W_{BC} = 109$ $W_{AC} = 109$ $W_C = 110$

Quizzes will always ask for the *smallest* integer solutions.
This particular problem also illustrates how to combine networks using a logic gate. Thus, to compute more complex regions, we need more neurons either at one level or at the output level. But first, to cement our understanding of this problem, let's look at a related quiz problem, from quiz 3, 2009.

Given this three-node neural network, and the training data on the right





Question: which of the following sets of weights will correctly separate the dots from the diamonds? (Think about what cuts the various weights make at the left neuron….)

Weight set A:

w_{11}	w_{12}	w_{1T}	w_{21}	w_{22}	w_{2T}	w_{21}	w_{22}	w_{2T}
–2	**–2**	**–1**	**3**	**3**	**–1.5**	**128**	**128**	**173**

Weight set B:

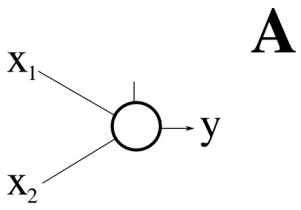
w_{11}	w_{12}	w_{1T}	w_{21}	w_{22}	w_{2T}	w_{21}	w_{22}	w_{2T}
2	**–1**	**1**	**2**	**-2**	**1**	**100**	**100**	**50**

Why does weight set A work but not weight set B?
Answer: weight set A defines two negatively sloping lines, similar to the XOR case, which are required to separate the dots from the diamonds.
Weight set B has as its first set of 3 weight a line that has positive slope – this is the wrong slope for that 'cut'. (Same for the next weight set). We need the weights to be both negative or both positive, so that the 'cuts' slope downwards, as required to separate the dot region from the diamonds.

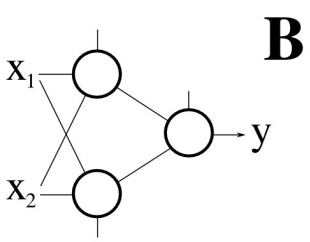
Example 4. Some other examples of carving up the *x-y* plane & the associated multi-layer networks
Now let's consider some other patterns in the *x-y* (or x_1, x_2) plane and what sort of qualitative network might be required to encode them. (This was an exam question in 2008.)

First, let's give the schematic pictures for (i) a perceptron; and then (ii) the simplest 2-layer neural net we have just seen – note that we have removed all the clutter of the *w*'s, etc.:

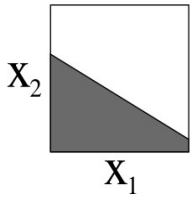
(A) Perceptron:



(B) Simplest 2-layer neural net:

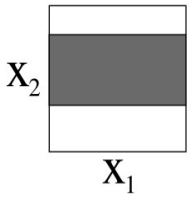


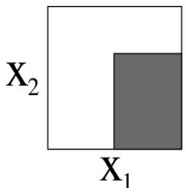
Here is a basic picture of the kind of classification regions a perceptron (A) can describe: any single cut, at any angle:



4.1 Question: Can a 2-layer network (B) also describe such a classification region? Why or why not?
Answer: yes, of course – a more powerful network can always do something that a less powerful net can do.

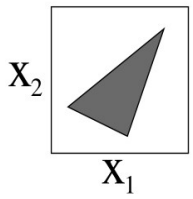
4.2 Now consider these two sorts of classification regions.
Question: Can a perceptron (net A) describe these kinds of regions? Can the 2-layer network (B) also describe these kinds of regions? Why or why not?





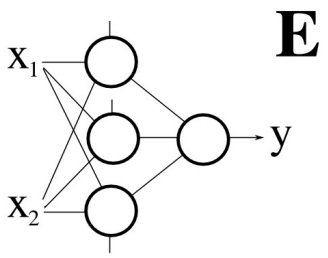
Answer: perceptrons can't do these – they require two cuts. A perceptron can only do one. The two-layer network for XOR can be modified with different weights to classify both of these figures above. A <u>simplified</u> two-layer network where unit A is fed just from X_1 and unit B is fed just from X_2 can describe the region on the RIGHT side (because unit A can describe any single vertical cut, X_1= some constant; and unit B can describe any single horizontal cut, X_2= some constant. Then the logic unit C can combine the two regions, as before. (See a picture of this net below, labeled "D".) But this kind of simplified network <u>cannot</u> describe the region on the left, because this requires different two horizontal cuts using the input X_2, so we would need a net with A and B units where the X_2 input connects to <u>both</u> units A and B (the X_1 input is irrelevant and can be set to 0).

4.3 Now let's hone our intuitions by making the region more complex, and by considering different neural networks.
Question: The 2-layer network (B) cannot **describe this kind of region. Why not?**

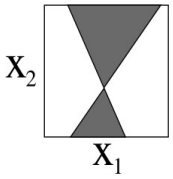


So, we must complicate our neural network to capture this kind of more complex region.
Question: Please explain *why* the following neural network *can* successfully describe the region just above.
(Think about how we classified the region in our worked-out example earlier.)

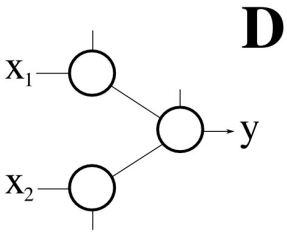
Answer: this region requires <u>three</u> separate cuts, not just two. So we need <u>three</u> basic input units, and then a second logic unit (as before) to combine their results via AND, like this one:



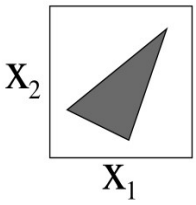
BUT: This network *cannot* successfully describe the region below. Why not? (Think about this, and for the next recitation, try to come up with the reason, and a modification, that is, a more complex neural network, that can describe this region.)



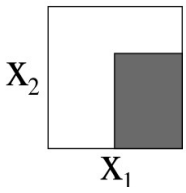
4.4 Finally, let us consider a *simpler* two layer neural network, where the inputs to the top, leftmost hidden neuron receives input *only* from x_1 , and the bottom, leftmost hidden neuron receives inputs *only* from x_2. So the network looks like the following. Can you intuit how this will restrict what regions the network can describe?



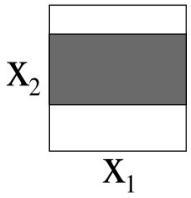
Question: Can this (restricted) neural network classify the region below? Why or why not?

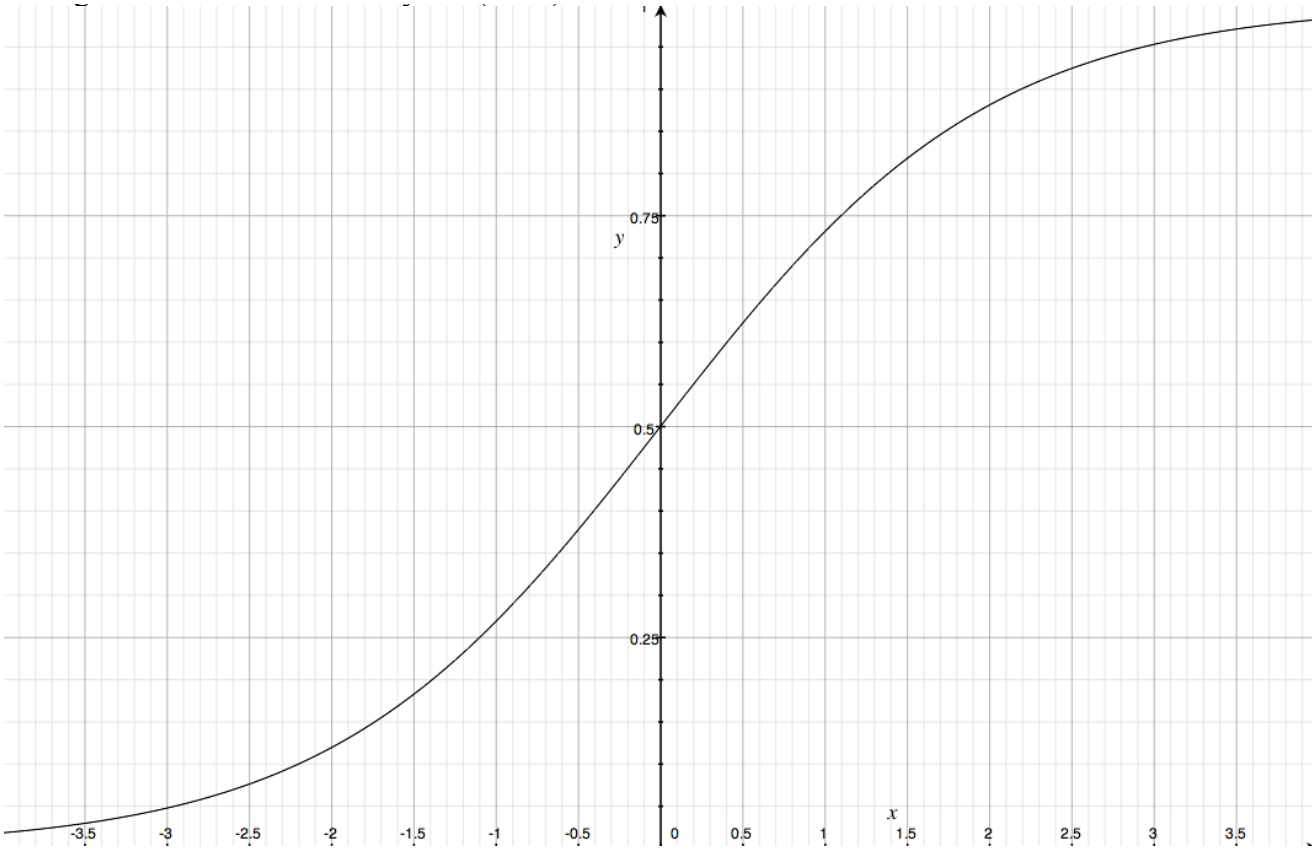


Can network (D) describe this region that we already saw above? Why or why not? (We answered this already above.)



Finally, for next time, you might want to think about *why* network (D) **cannot** describe this region that we saw before (while we have already discussed what the usual 2-layer network (B) can do in this case) **(We also answered this question already, above).**


X_2
X_1



Sigmoid function $y = 1/(1–e^{-x})$