Build a neural network to implement a logic gate

Sample terminal input:

Your file input file

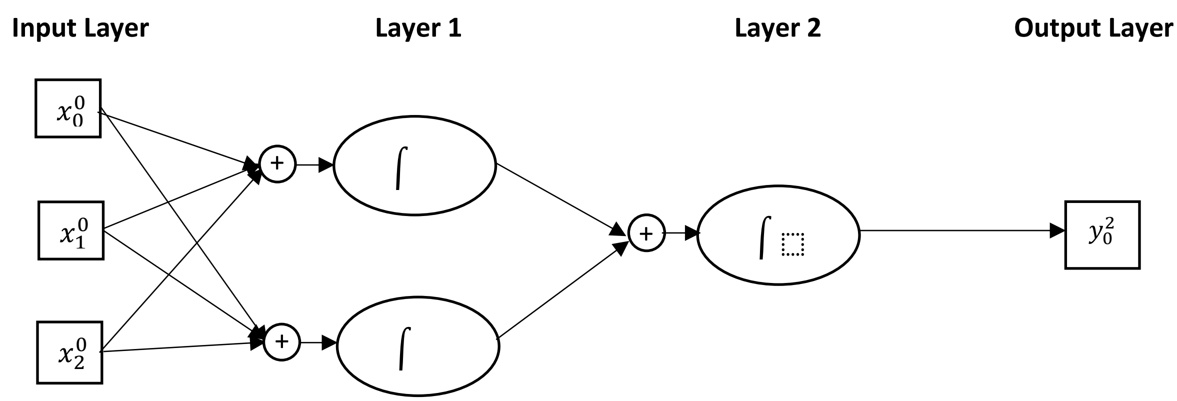
```
lg_nn.py x_gate.txt
```

Sample input file: Here is a sample 'AND' gate input and output.

```
0 0 => 0
0 1 => 0
1 0 => 0
1 1 => 1
```

What to do with the input data:

- There are always 2 hidden layers (Layer 1 with 2 nodes and Layer 2 with one node)

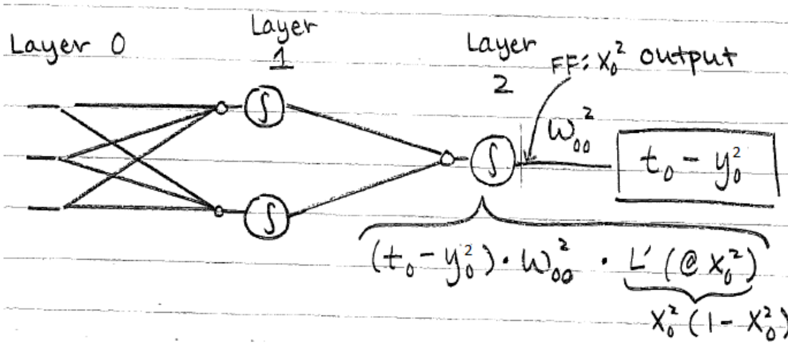


- The number of input values = (the number of values one the left side of =>) + 1 (DC offset)
  - The number of input values will be in range of [2, 4] including the DC offset
  - Print Layer Counts

    Sample output:
    ```
    Layer cts: [3, 2, 1, 1]
    ```
- Decide the number of weights (# of wires) → This will be the final output
  - With the sample input above:
    - w_{ij}^0 : 6 values, w_{ij}^1 : 2 values, w_{ij}^2 : 1 value => a list of 3 lists
- Make error list with the number of lines of input file: **`errlist = [10]*(# of test cases)`**

Goal: find a neural network that solves the problem → The sum of all test errors should be $\leq$ 0.01

$E = \frac{1}{2}\sum_j (t_j - y_j^k)^2$ k: output Layer



Use this:

$L(x)$ = logistic function

$L'(x) = L(1-L)$

$E = \frac{1}{2}\sum_i (t_i - y_i^{final})^2$

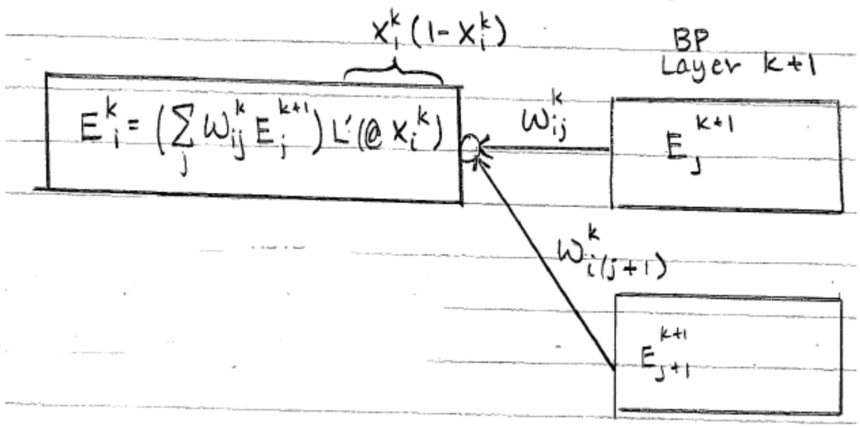
<u>Plan:</u> update the (weights of the) neural network by adding in an appropriate multiple of the negative of the gradient.

<u>Implementation of Plan:</u> Create a duplicate of the nodes structure, and fill it out with back propagation info and based on the BP and FF structures, determine $-\nabla E$ (negative gradient)

- Start with random weights for each wire: choose a random float in the interval [-2.0, 2.0]
- One example of appropriate multiple: 0.01
  - Calculate negative gradient: $-\frac{\partial E}{\partial w_{ij}^k}$ BP: Back Propagation, FF: Forward Feed

    $= (Error\ val\ in\ BP\ network\ to\ the\ right) \times (output\ val\ in\ FF\ network\ to\ the\ left)$

    $= E_j^{k+1} \times x_i^k$ $\times$ ***means 'times' (multiply)***



  - Multiply an appropriate multiple and add it to the previous guessed weight value
    - new $w_{ij}^k = E_j^{k+1} \times x_i^k \times \alpha + w_{ij}^k$ $\alpha = 0.01$ or so
  - Calculate this for every single weight

- Gradient is the same structure as the weight structure (e.g. a list of 3 lists in this sample input)
- Keep FF and BP and update weights until it converges (sum of errors < 0.01) or cut-off (200k?)
  - If error sum is off by too much, such as higher than 1 or 0.5 or doesn't converge fast enough for your liking, you may re-start with new random weights!

<u>Tips for transfer function python implementation (from Dr. Gabor):</u>

```
t_func = {"T1": lambda x: x, "T2": lambda x: x if x>0 else 0,
                          "T3": lambda x: 1/(1+math.e**-x), ..}

dervs = {…, "T3": lambda y: y*(1-y), "T4": lambda y: (1-y*y)/2}
```

- "T1" is a linear function: $g(x) = x$
- "T2" is a ramp function: $g(x) = \begin{cases} x, if\ x > 0 \\ 0, if\ x \leq 0 \end{cases}$
- "T3" is a logistic function: $g(x) = \frac{1}{1+e^{-x}}$ T3 works the best! Use T3 for this lab!
- "T4" is a sigmoid function: $g(x) = -1 + \frac{2}{1+e^{-x}}$

<u>Sample terminal output:</u>

```
Errors: [0.0071.., 0.0005.., 0.0013.., 0.0074..]

Layer cts: [3, 2, 1, 1]

Weights:

[0.9070., 1.7216.., 0.7438.., -3.9361.., -3.6020.., 0.7762..]

[1.1885.., -4.5897..]

[1.3842..]
```