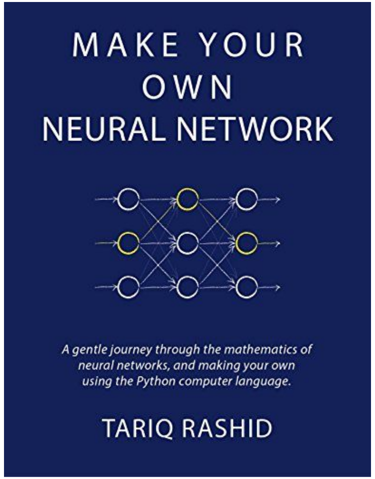
Programming own
Neural Networks

The Cognitive Thinking Approach

AIM

- To understand NN
- To be able to code NN in any language of choice



Reference text book :
https://www.amazon.com/Make-Your-Own-Neural-Network-ebook/dp/B01EER4Z4G

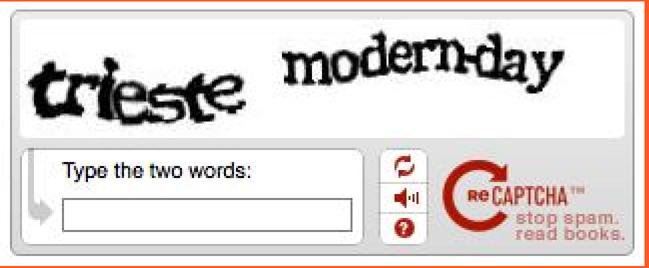
NOTE : I have changed code to make it more simple and dependency free. Code in book uses scipy and matpotlib functions.

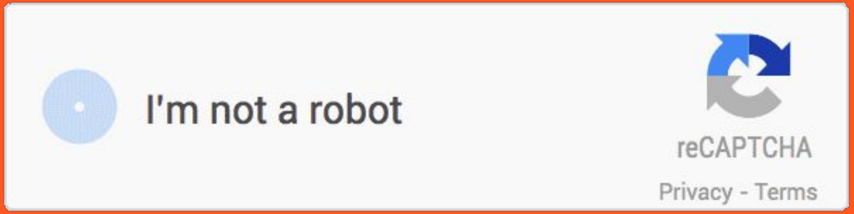
Things about NN not found in books

- NN are very old. Older than first electronic computer.
  - Model for Neural Network was built in 1943
  - First electronic was built in 1946
- NN kick started field of Artificial intelligence
- Why are they famous ?
  - Because **Universal Approximation theorem** states that NN can compute any function
  - Even functions which we can not define like how our brain works or compute something
- For 50 years NN were not used because our computers were not fast enough
- Unlike traditional machine learning approaches, there is probability (very less) NN can given wrong output for already seen data
  - Just like a person can make wrong judgement about already known things

Fun fact

Google's co-founder Sergey Brin did not take neural networks seriously until some hackers cracked Google's reCapcha system with 99.9% accuracy using NN while humans could do it with 70% accuracy only.





https://arstechnica.com/security/2012/05/google-recaptcha-brought-to-its-knees/

NAND

- Why NAND ? Because if NN can learn NAND then it can do anything that any boolean logic based circuit can do. NAND is universal gate.

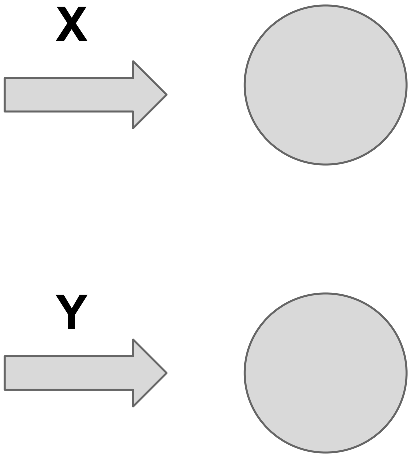
X	Y	OUTPUT
0	0	1
0	1	1
1	0	1
1	1	0


Input Layer
Output Layer
X
Y
Output

How many nodes in middle layer ?

No answer.

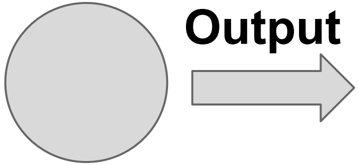
Input Layer

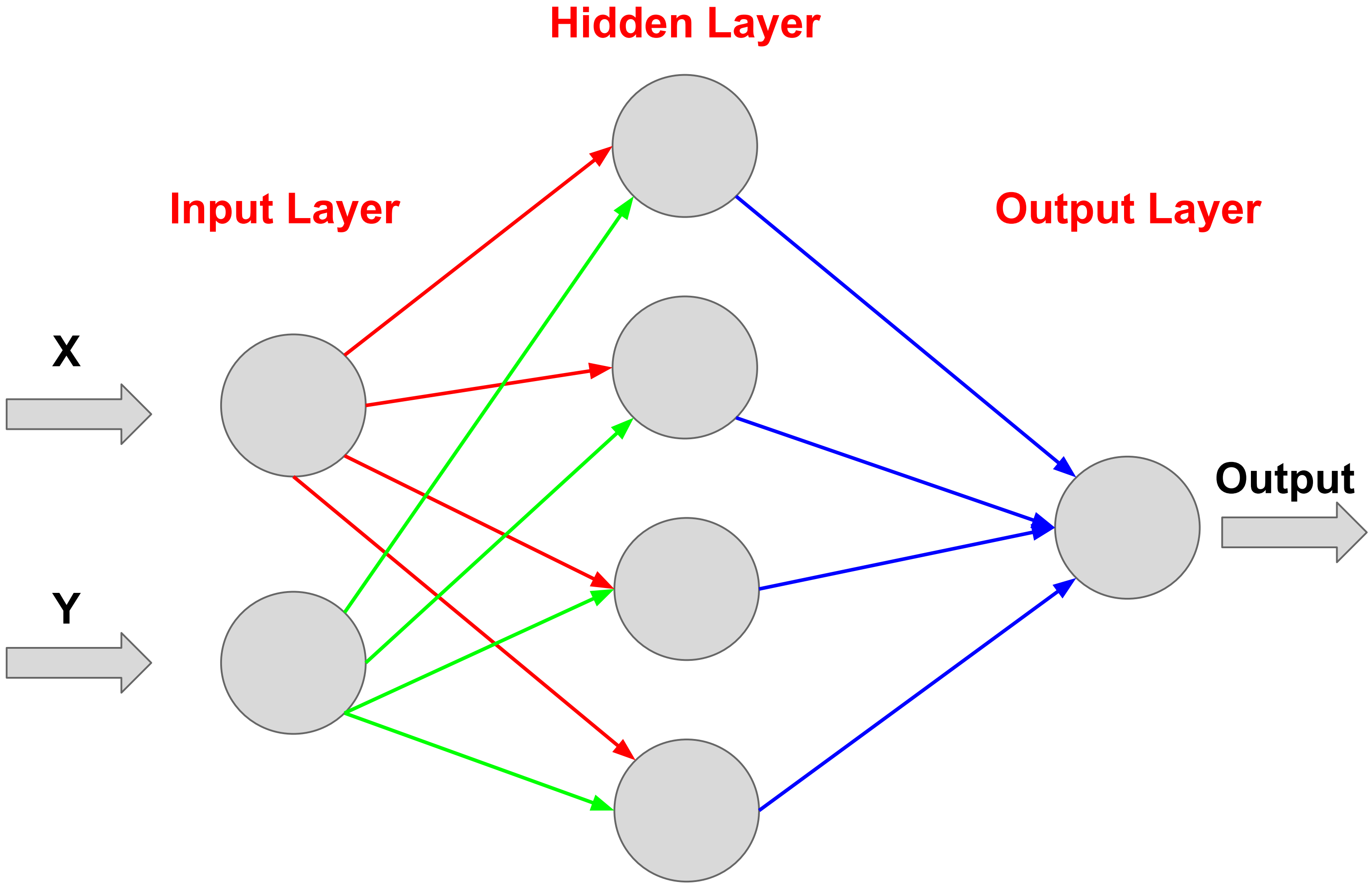


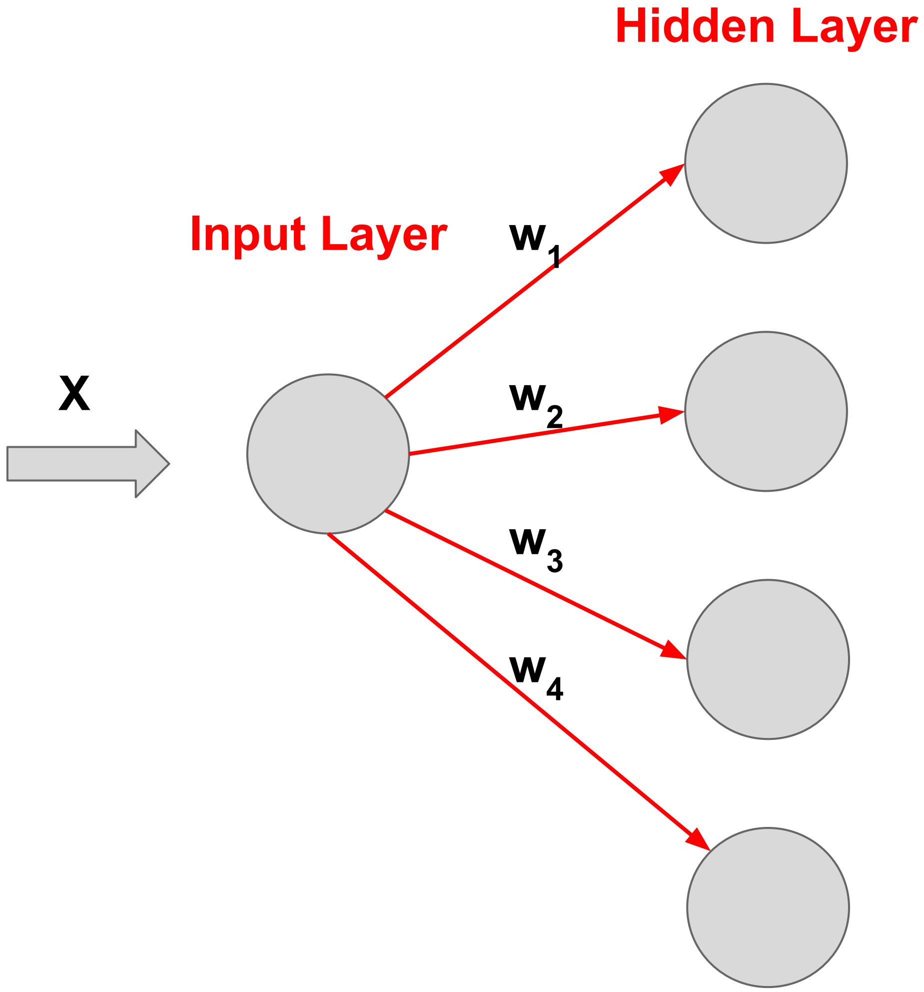
Generally, it depends on the amount of data that NN needs to learn. Like in our case it needs to learn 4 specific cases. So we need at least 4 nodes in middle layer.

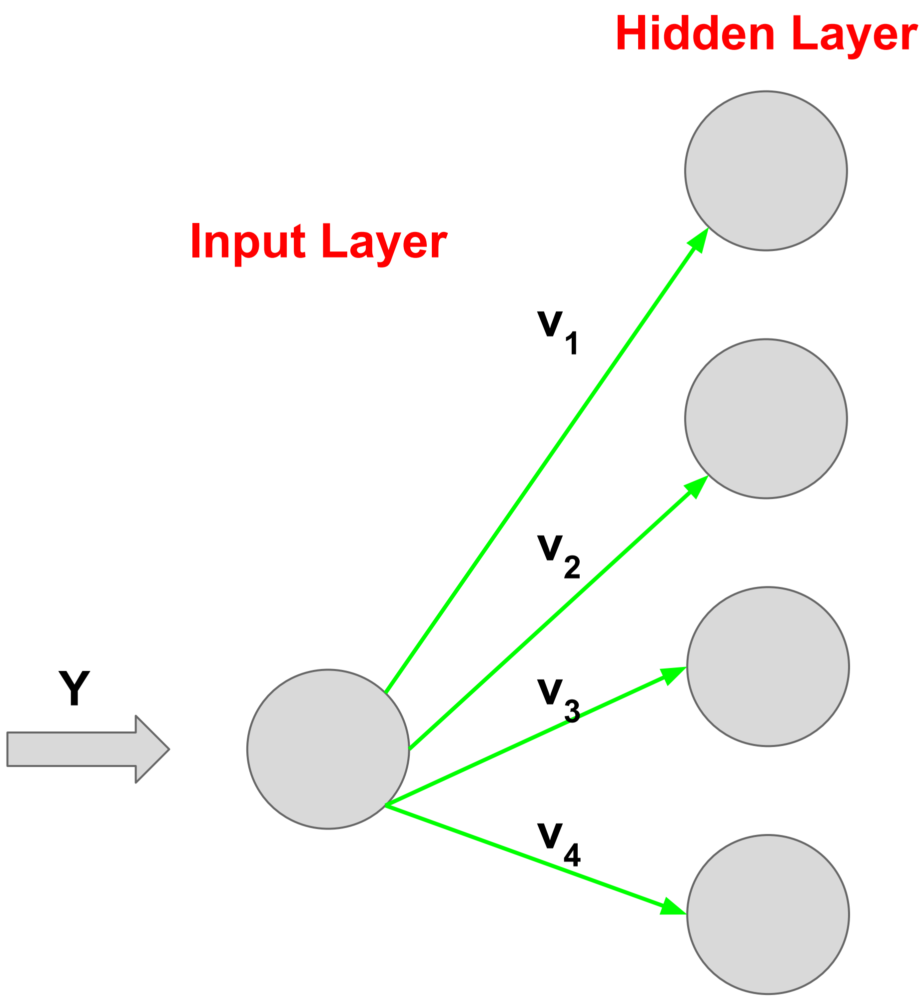
Making it more may increase accuracy but this is not certain.

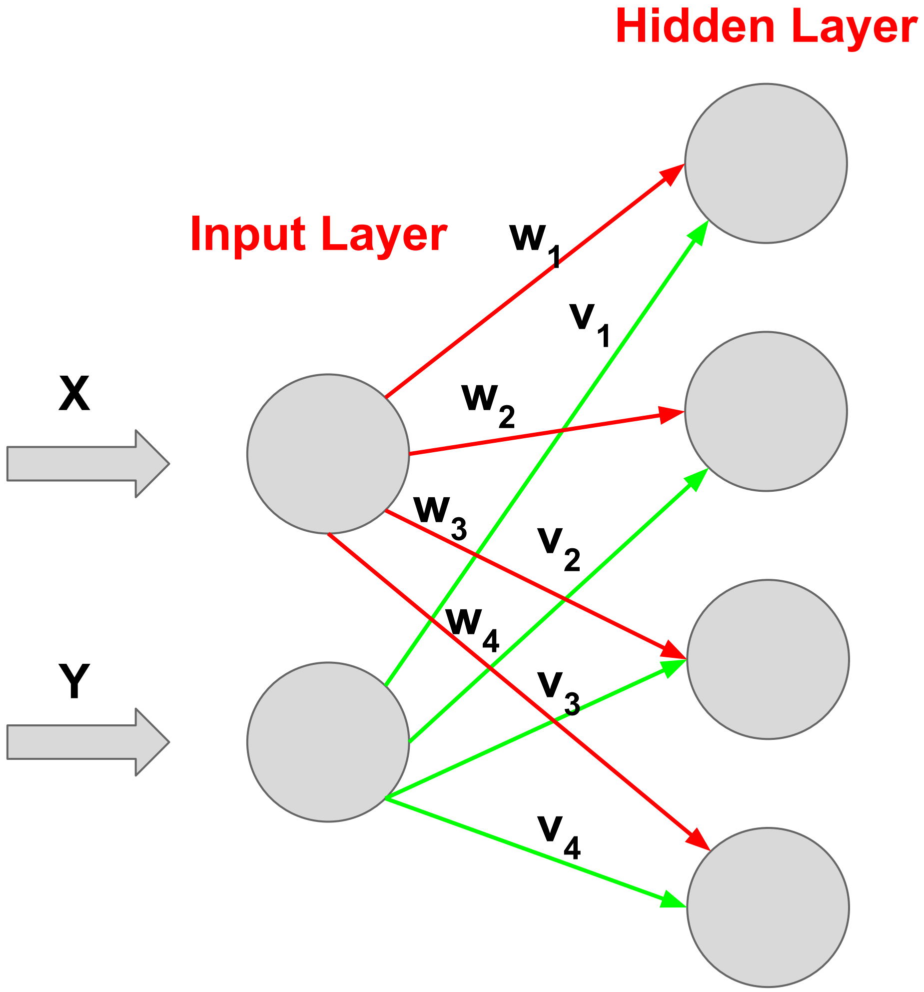
Output Layer

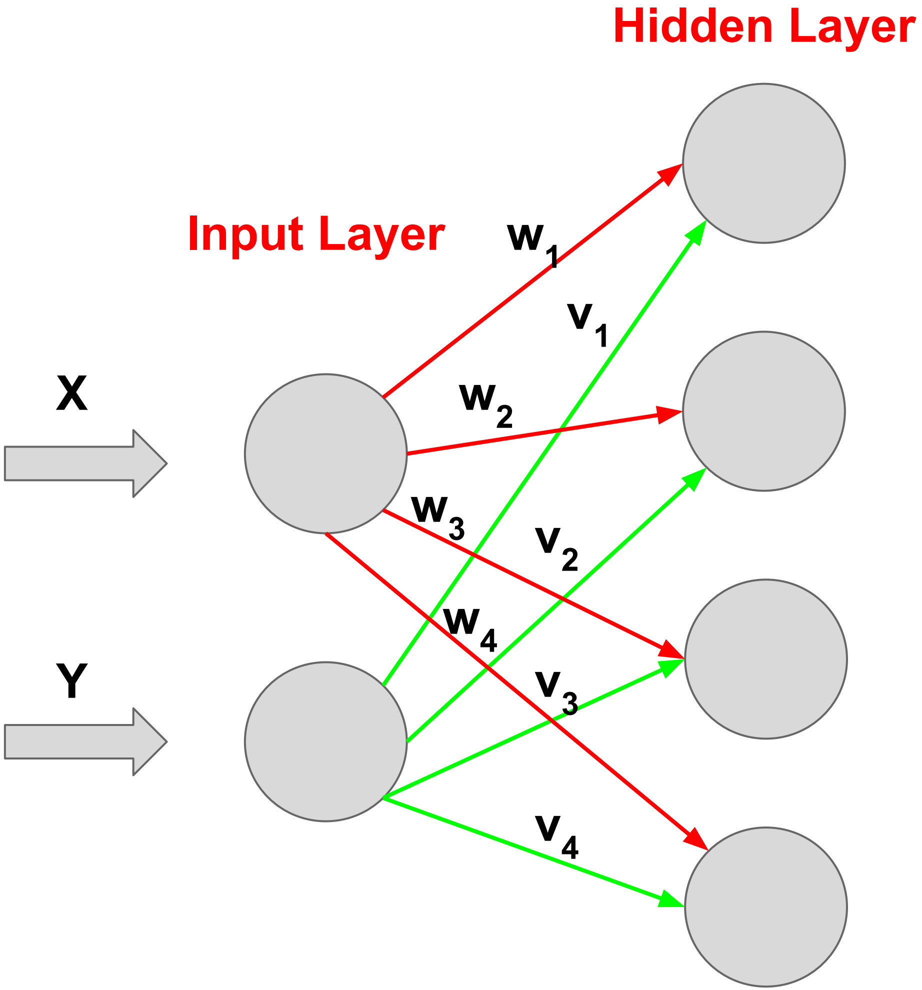



Hidden Layer
Input Layer
Output Layer
X
Y
Output

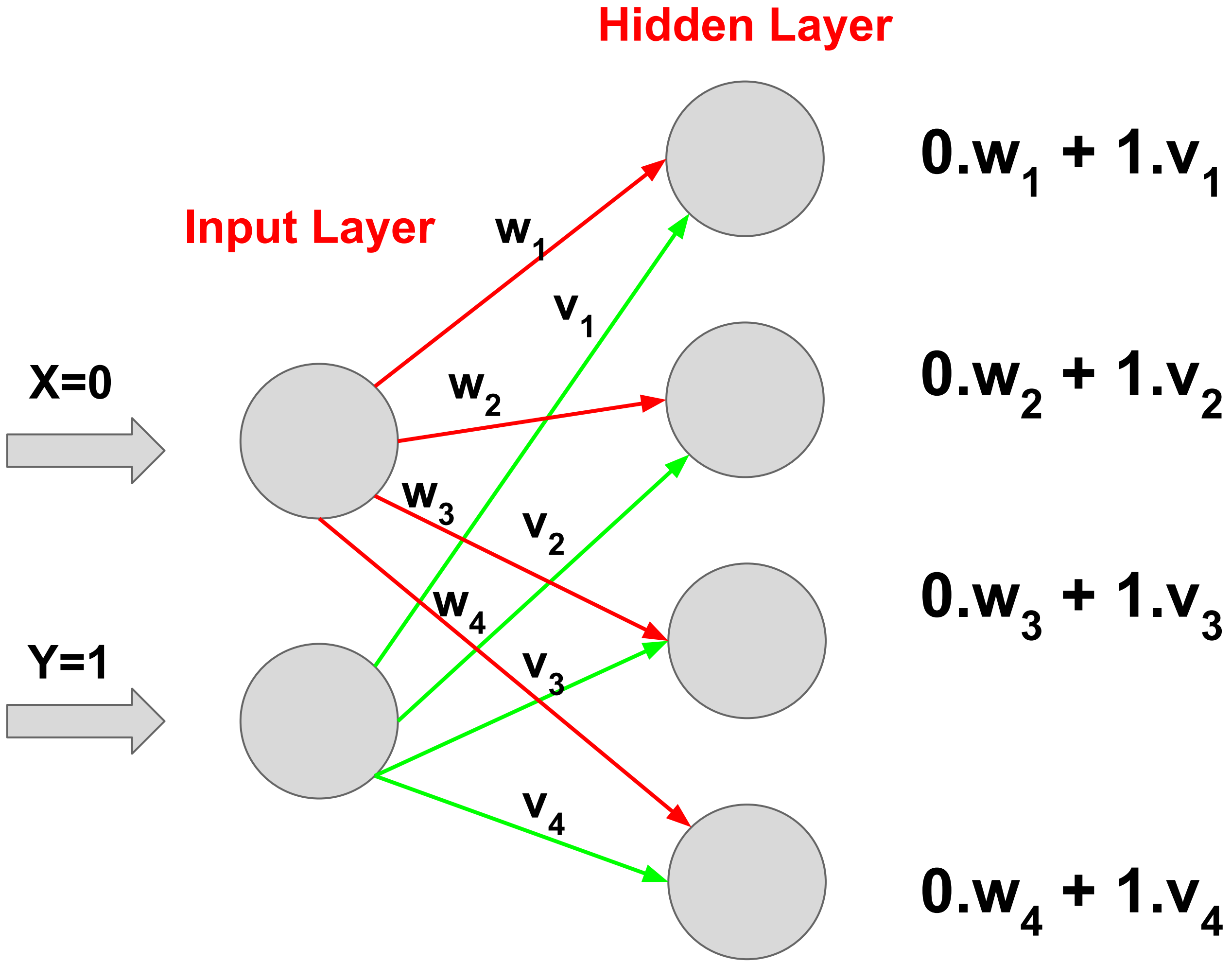

Hidden Layer
Input Layer
X
w1
w2
w3
w4


Hidden Layer
Input Layer
Y
v1
v2
v3
v4

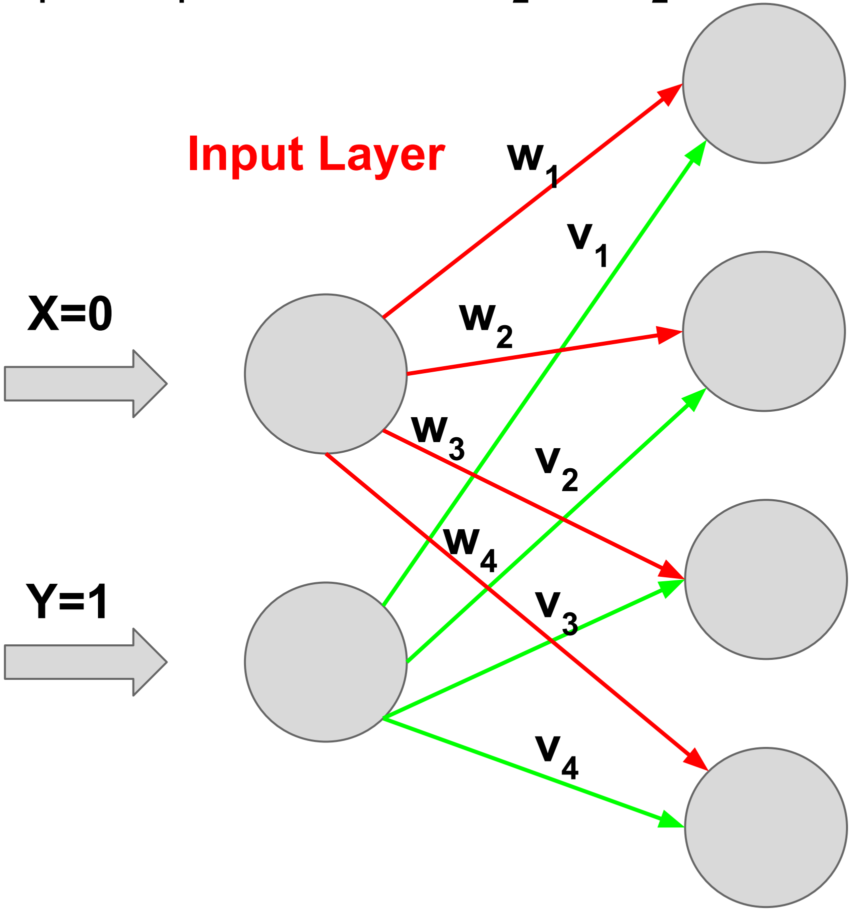

Hidden Layer
Input Layer
X
Y
w1
w2
w3
w4
v1
v2
v3
v4



X	Y	OUTPUT
0	0	1
0	1	1
1	0	1
1	1	0

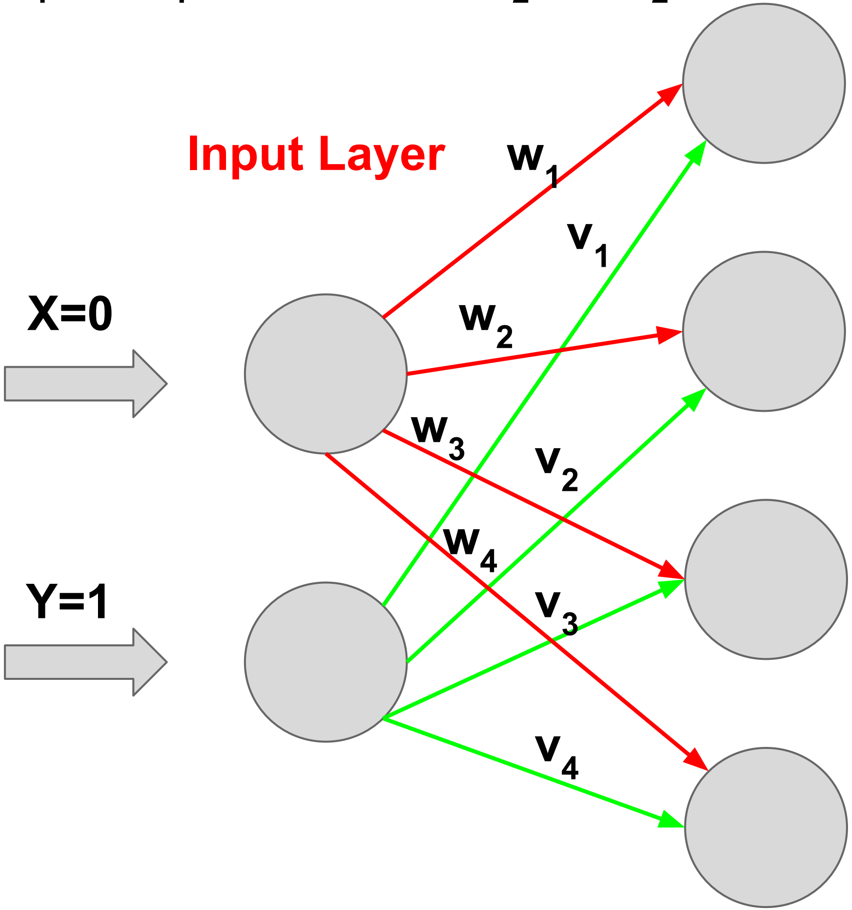

Hidden Layer
Input Layer
X=0
Y=1
w1
w2
w3
w4
v1
v2
v3
v4
0.w1 + 1.v1
0.w2 + 1.v2
0.w3 + 1.v3
0.w4 + 1.v4

$0.w_1 + 1.v_1$ $0.w_2 + 1.v_2$ $0.w_3 + 1.v_3$ $0.w_4 + 1.v_4$



$$\begin{bmatrix} X & Y \end{bmatrix} \begin{bmatrix} w_1 & w_2 & w_3 & w_4 \\ v_1 & v_2 & v_3 & v_4 \end{bmatrix}$$

$0.w_1 + 1.v_1$ $\quad$ $0.w_2 + 1.v_2$ $\quad$ $0.w_3 + 1.v_3$ $\quad$ $0.w_4 + 1.v_4$



$$\begin{bmatrix} X & Y \end{bmatrix} \begin{bmatrix} w_1 & w_2 & w_3 & w_4 \\ v_1 & v_2 & v_3 & v_4 \end{bmatrix}$$

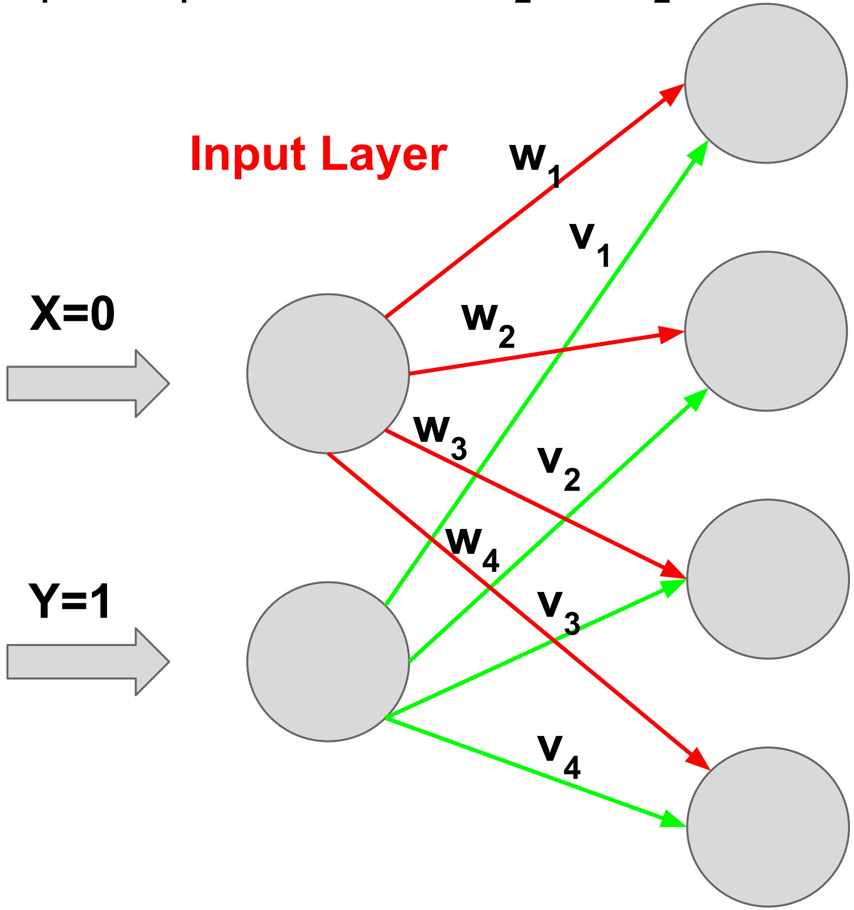
General rule 1 :

Layer (the area b/w two set of nodes) =

Matrix representing input nodes *

Matrix representing weights

$0.w_1 + 1.v_1$ $0.w_2 + 1.v_2$ $0.w_3 + 1.v_3$ $0.w_4 + 1.v_4$



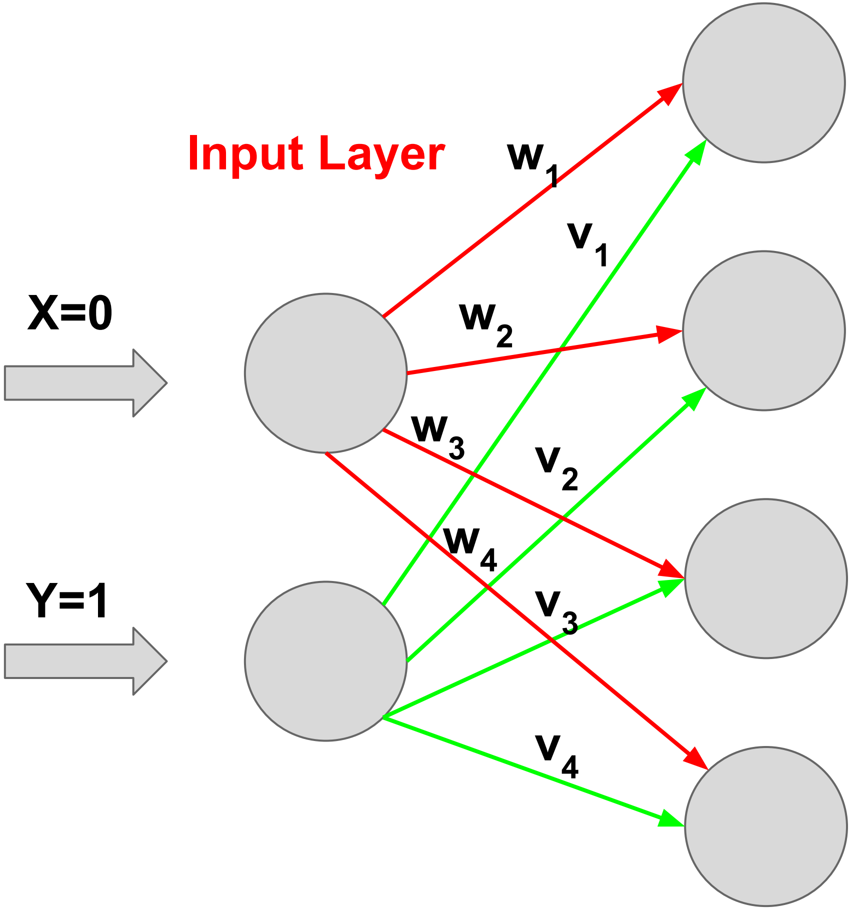
$$\begin{bmatrix} X & Y \end{bmatrix} \begin{bmatrix} w_1 & w_2 & w_3 & w_4 \\ v_1 & v_2 & v_3 & v_4 \end{bmatrix}$$

General rule 2 :

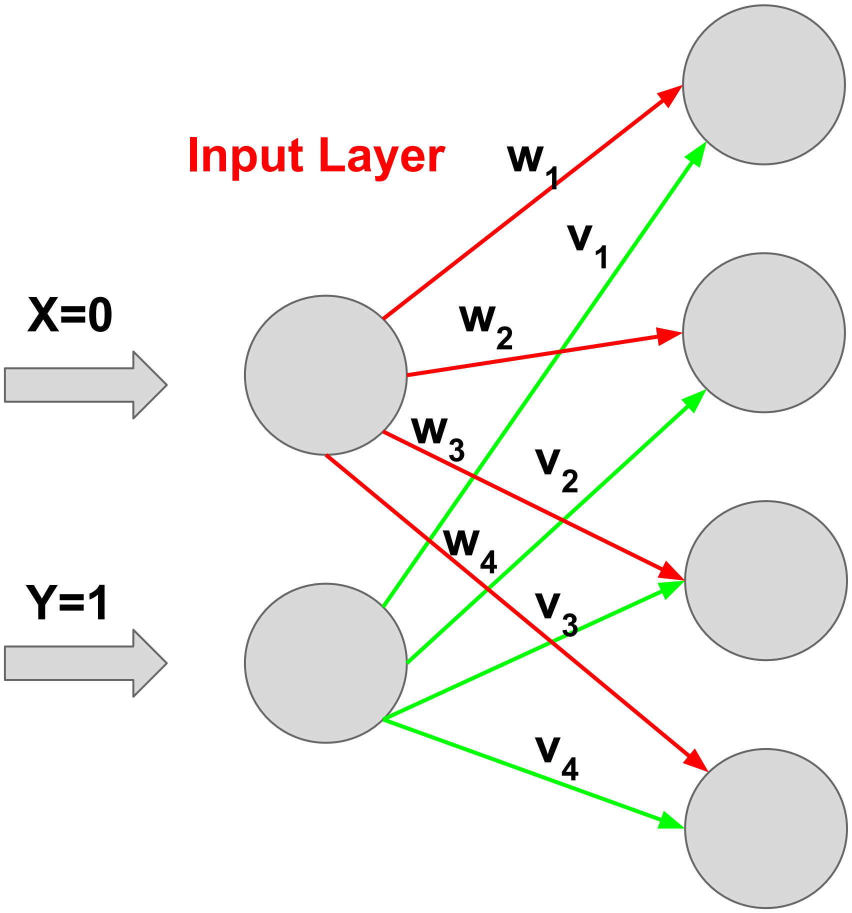
Note size of matrices:

Matrix 1 is 1 x 2

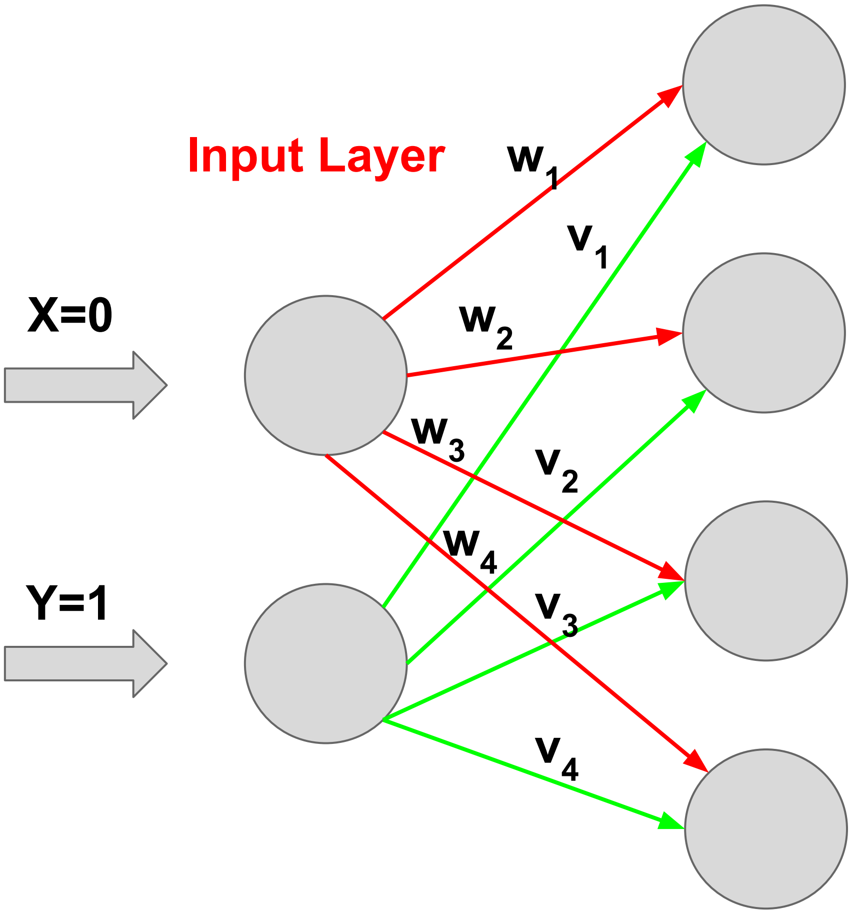
Matrix 2 is 2 x 4



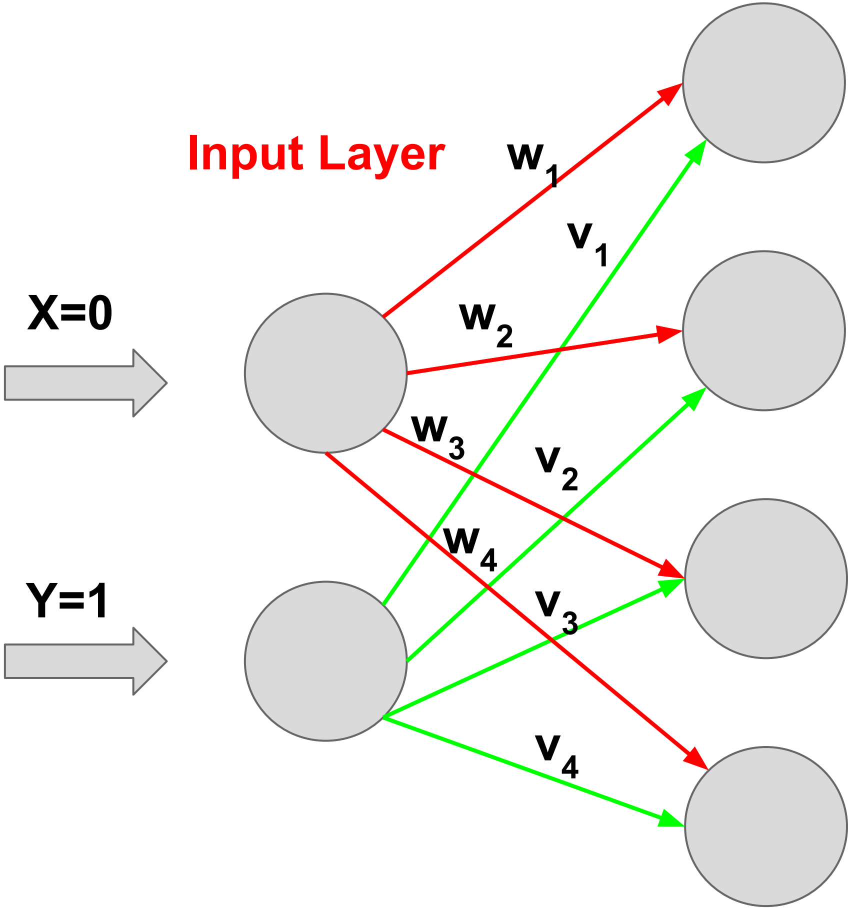

```
# Code

# numpy has functions for
# matrix multiplication

import numpy
```



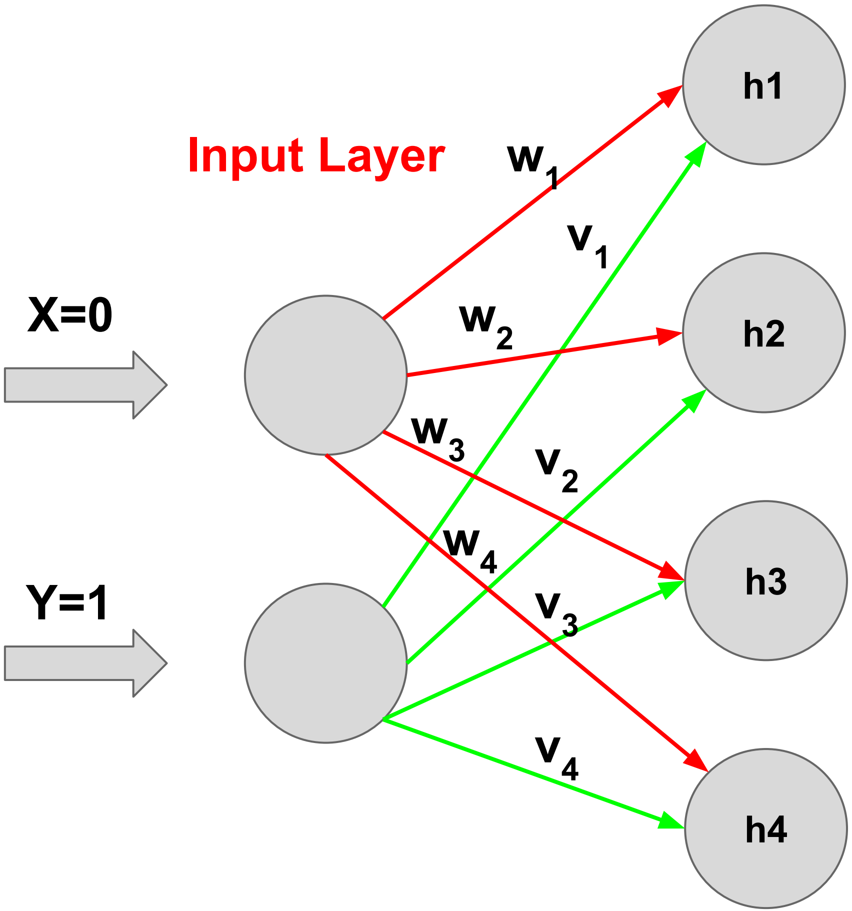

```
# input data

x = numpy.array([[0,0],
                 [0,1],
                 [1,0],
                 [1,1]])

# output data

y = numpy.array([[1],
                 [1],
                 [1],
                 [0]])
```



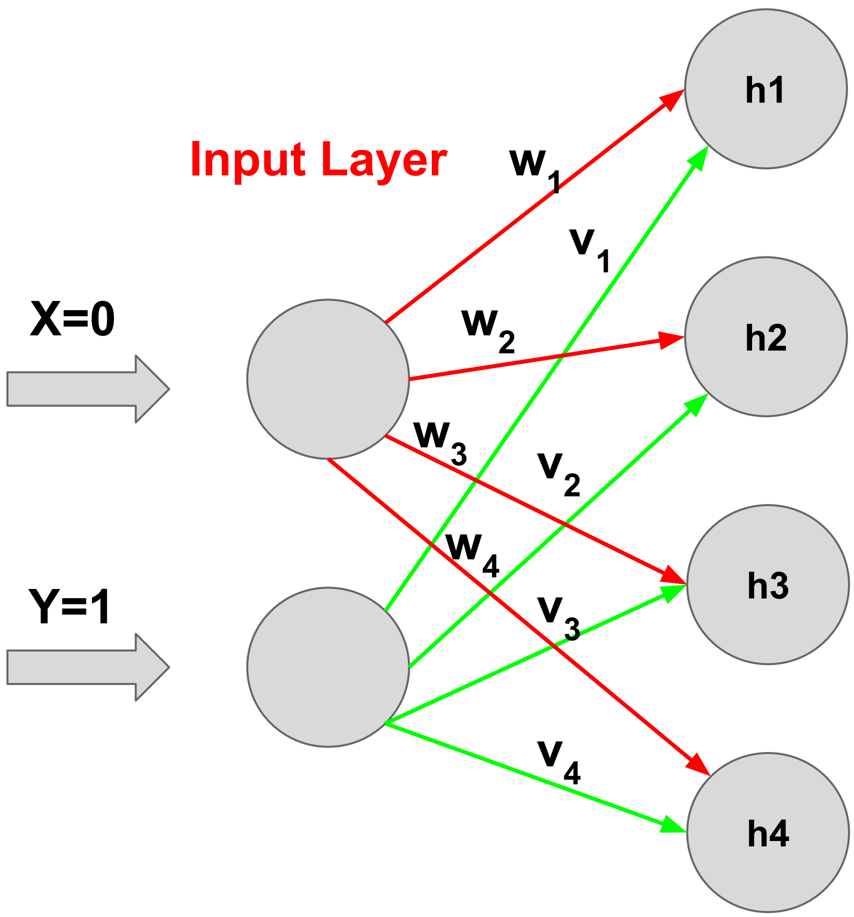

```
# weights without bias

w1 = numpy.random.rand(2, 4)
```




```
# weights with bias

w1 = numpy.random.rand(2, 4)+1
```




```
# weights with bias

w1 = numpy.random.rand(2, 4)+1


# calculating input to hidden
# layer

h = numpy.dot(x, w1)

# h will 1 x 4 matrix
```

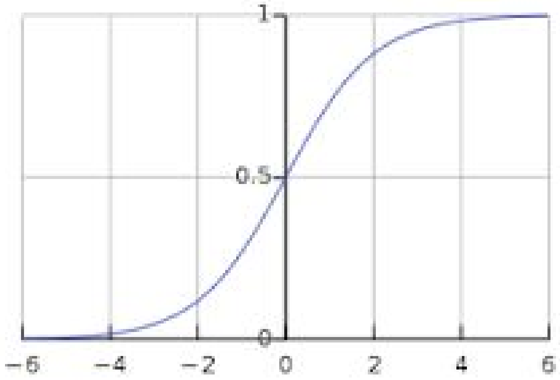


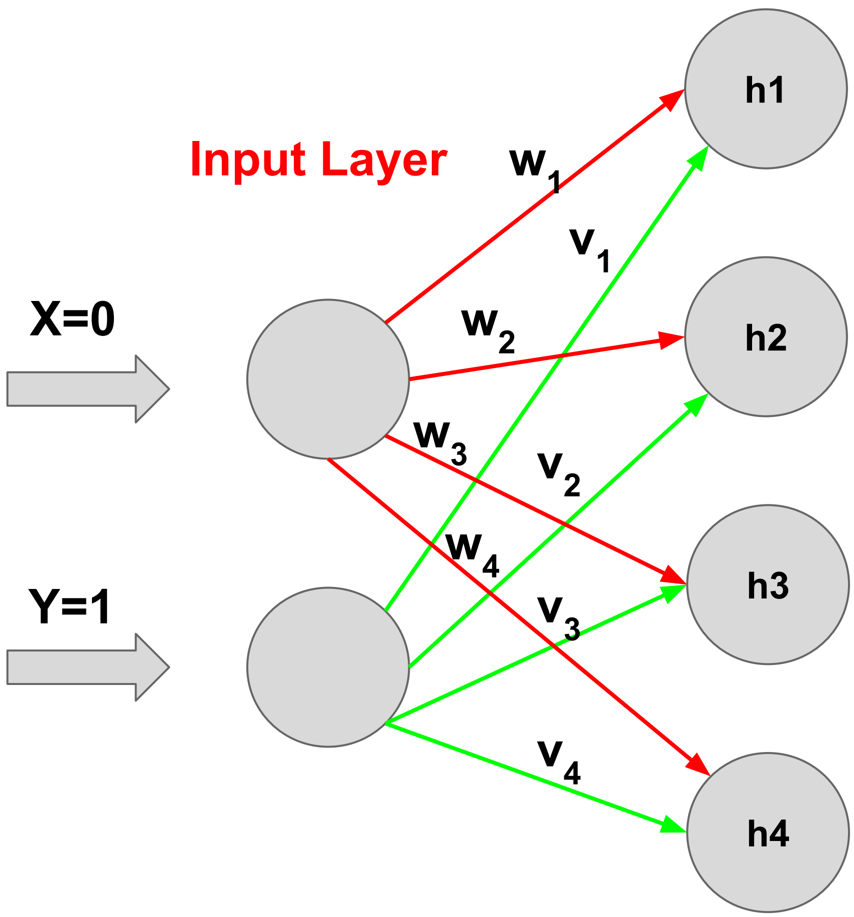
Who will decide what goes out of hidden layer ?

Activation function

Defines if the node will fire or not

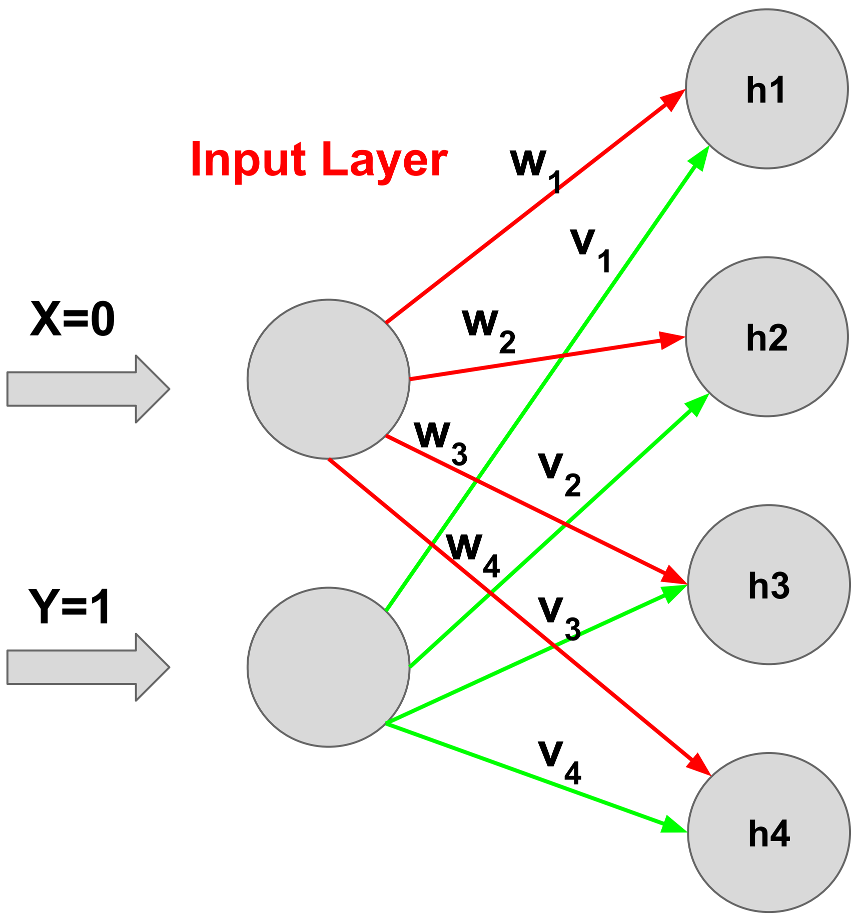
We will use sigmoid function



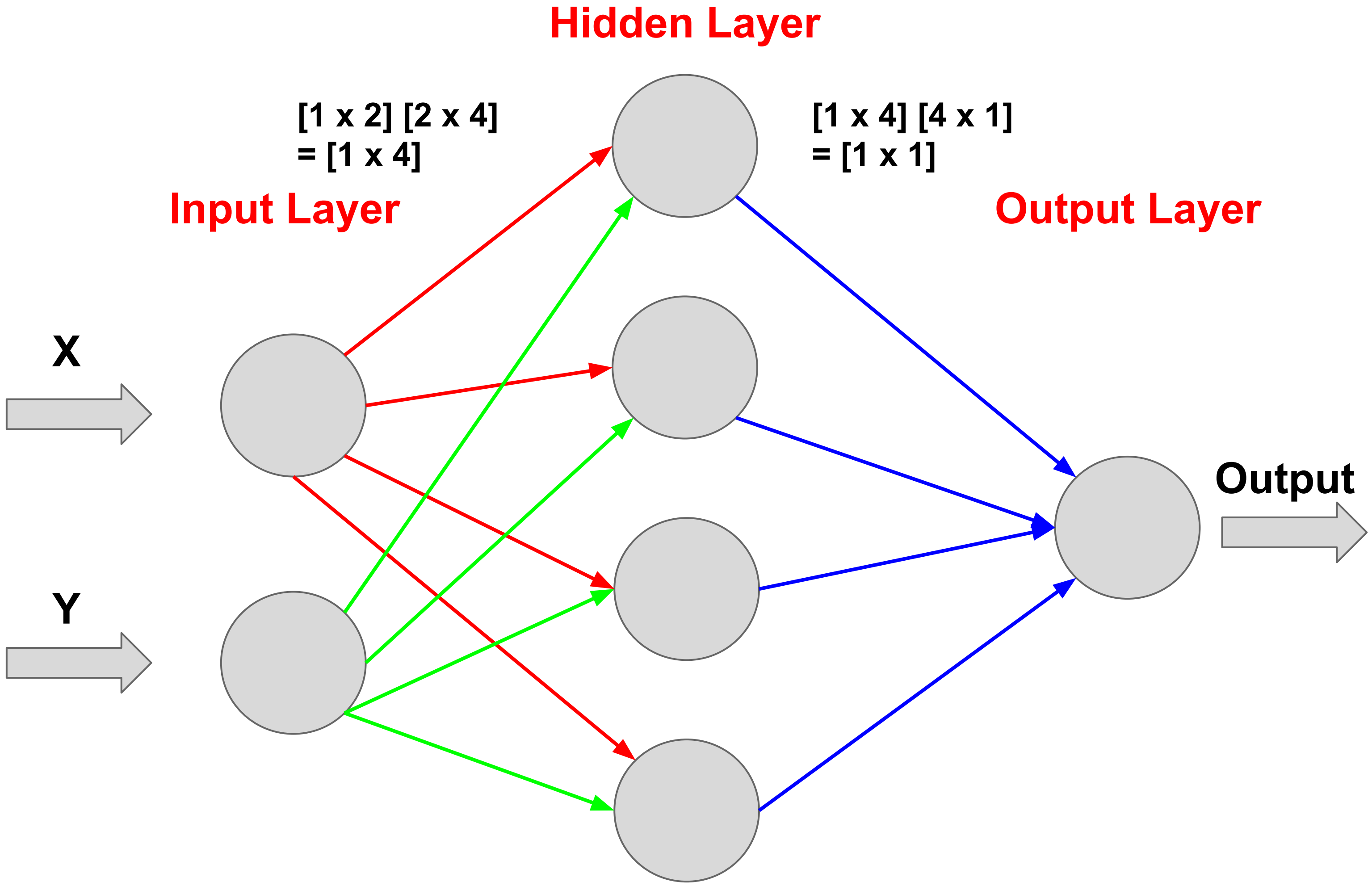



```
# sigmoid function

def nonlin(x):
    return (1/(1 + numpy.exp(-x)))
```

$$f(x) = \frac{1}{1 + e^{-(x)}}$$



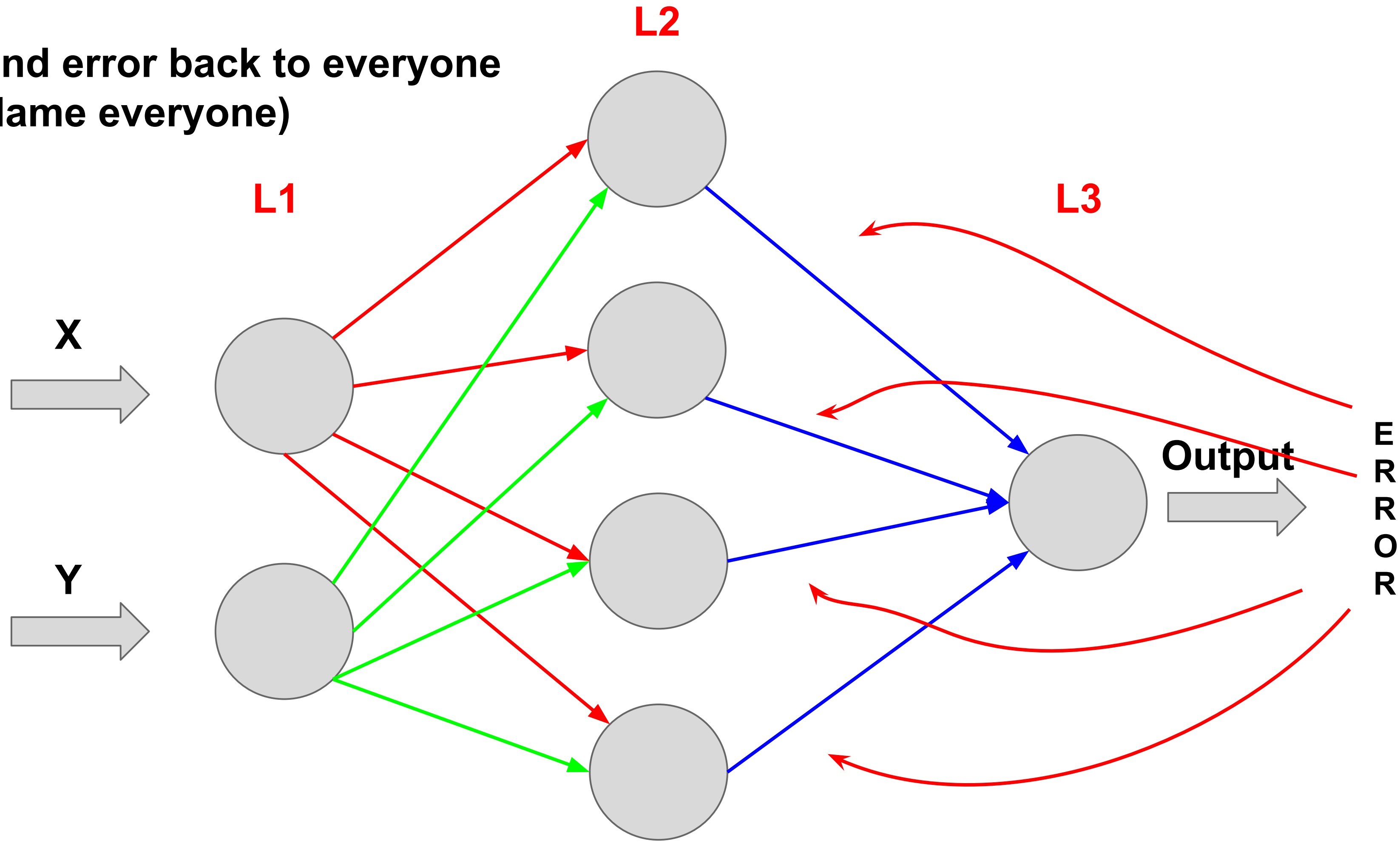

```
# sigmoid function

def nonlin(x):
    return (1/(1 + numpy.exp(-x)))


# h = numpy.dot(x, w1)

l2 = nonlin(numpy.dot(x, w1))
```


Hidden Layer
[1 x 2] [2 x 4]
= [1 x 4]
[1 x 4] [4 x 1]
= [1 x 1]
Input Layer
Output Layer
X
Y
Output

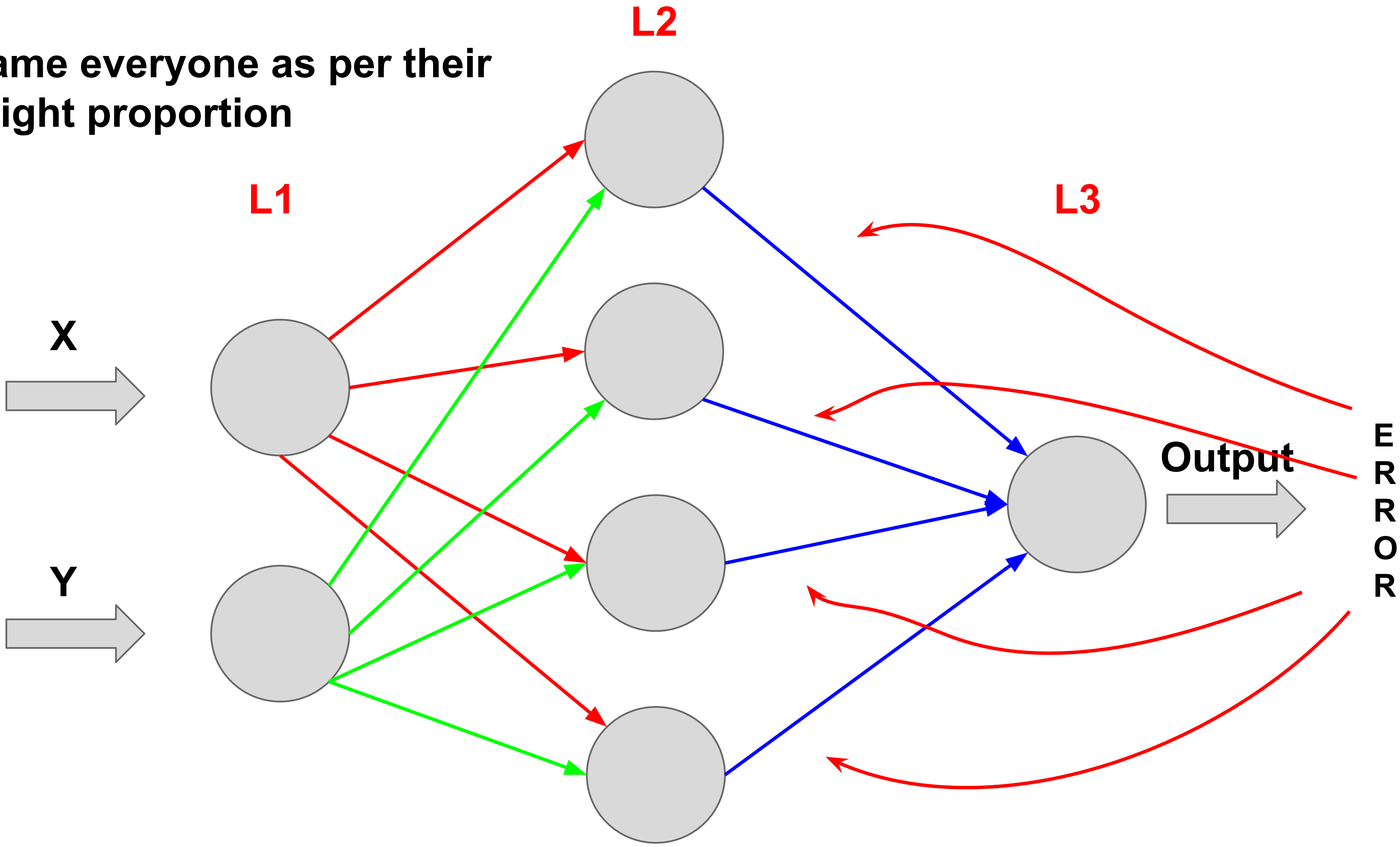
```
# code file nn.py
import numpy

x = numpy.array([[0,0],
                 [0,1],
                 [1,0],
                 [1,1]])

y = numpy.array([[1],
                [1],
                [1],
                [0]])

w1 = numpy.random.rand(2, 4) + 1
w2 = numpy.random.rand(4, 1) + 1

def nonlin(x):
    return (1/(1 + numpy.exp(-x)))
```

```
# feed forward

l1 = x

l2 = nonlin(numpy.dot(l1, w1))

l3 = nonlin(numpy.dot(l2, w2))

l3_errors = y - l3

print "Total error :"

print
numpy.mean(numpy.abs(l3_errors))
```

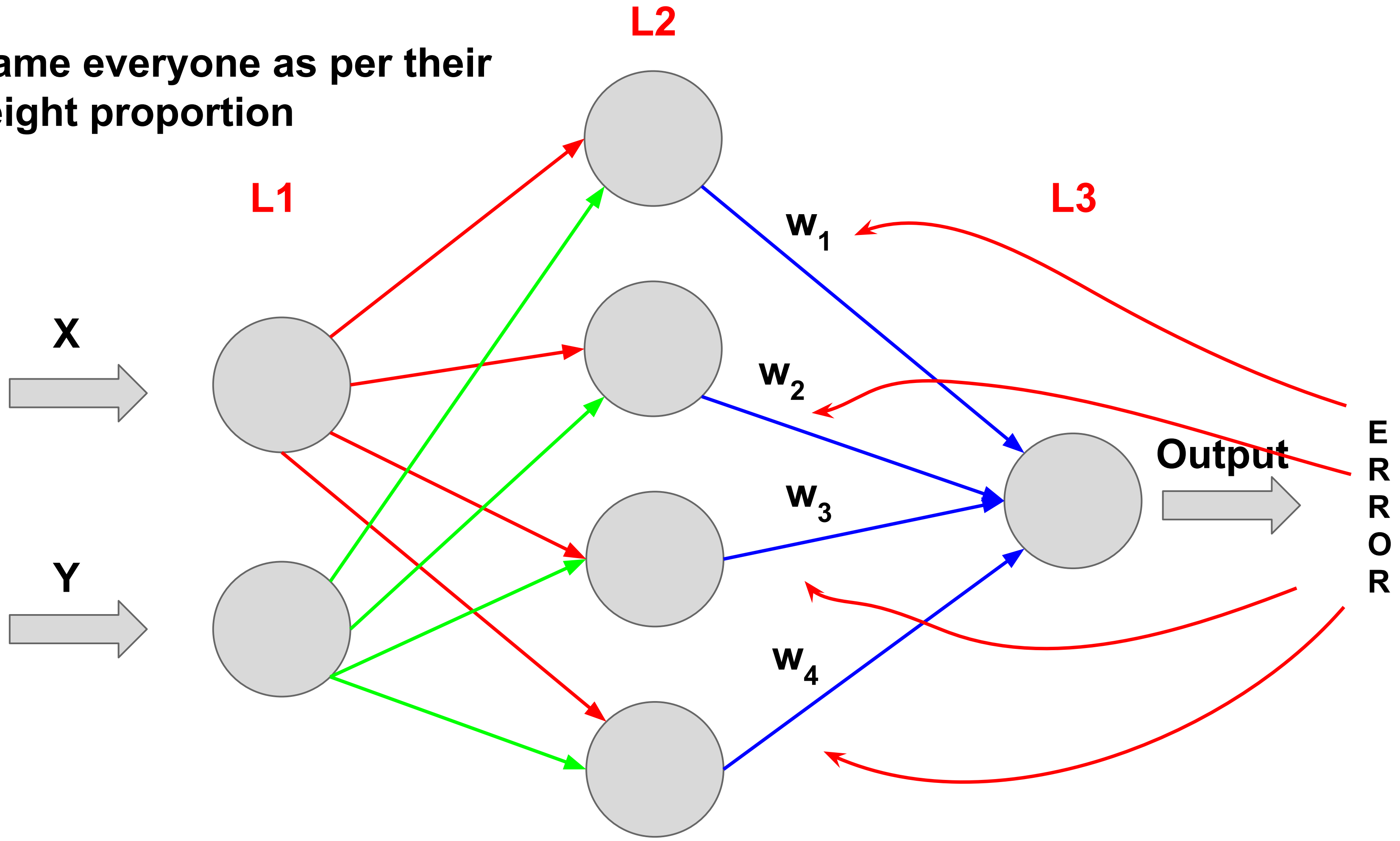
Send error back to everyone
(Blame everyone)



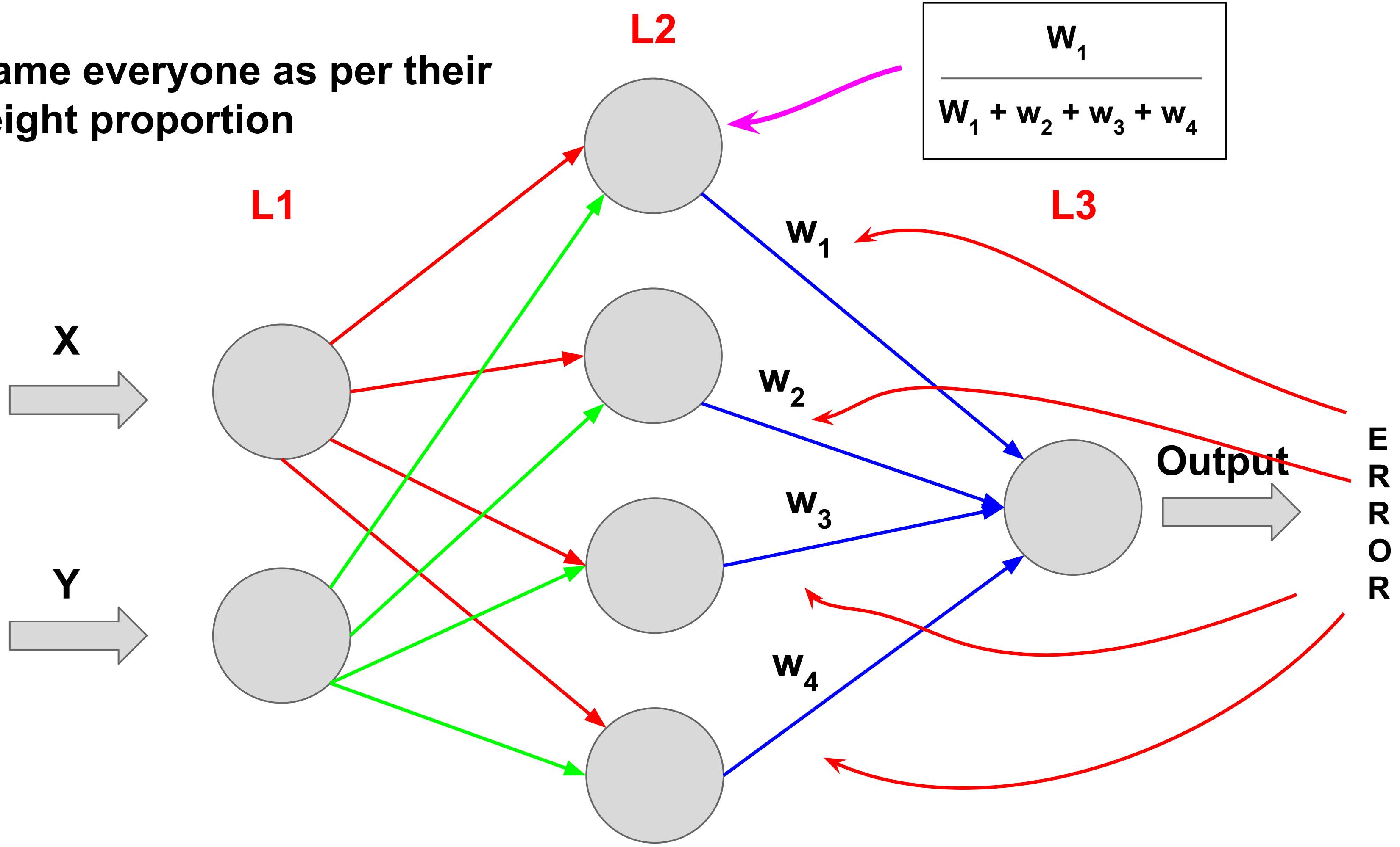
Blame everyone as per their weight proportion



Blame everyone as per their weight proportion



Blame everyone as per their weight proportion

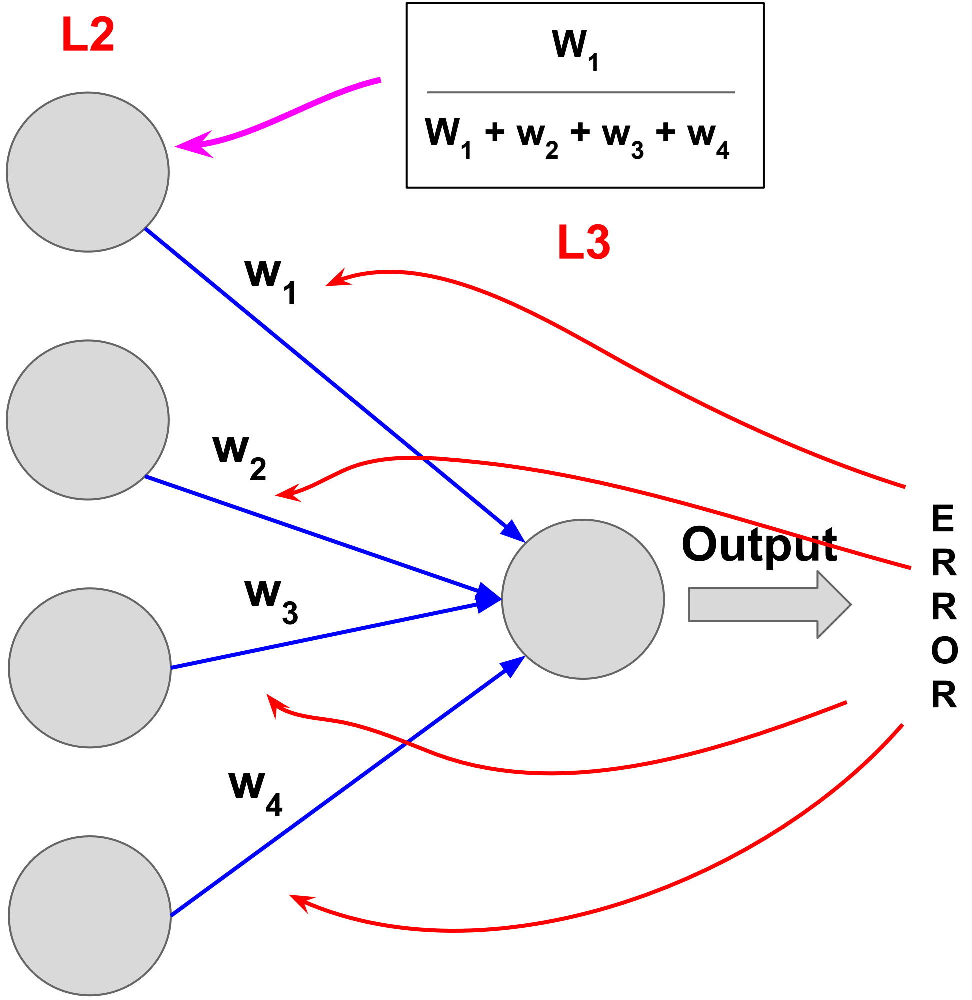


Blame everyone as per their weight proportion

$$\left[\; \frac{W_1}{W_1 + w_2 + w_3 + w_4} \qquad \frac{W_2}{W_1 + w_2 + w_3 + w_4} \;\right.$$

$$M^T$$

Transpose of a matrix



Back Propagation Matrix is

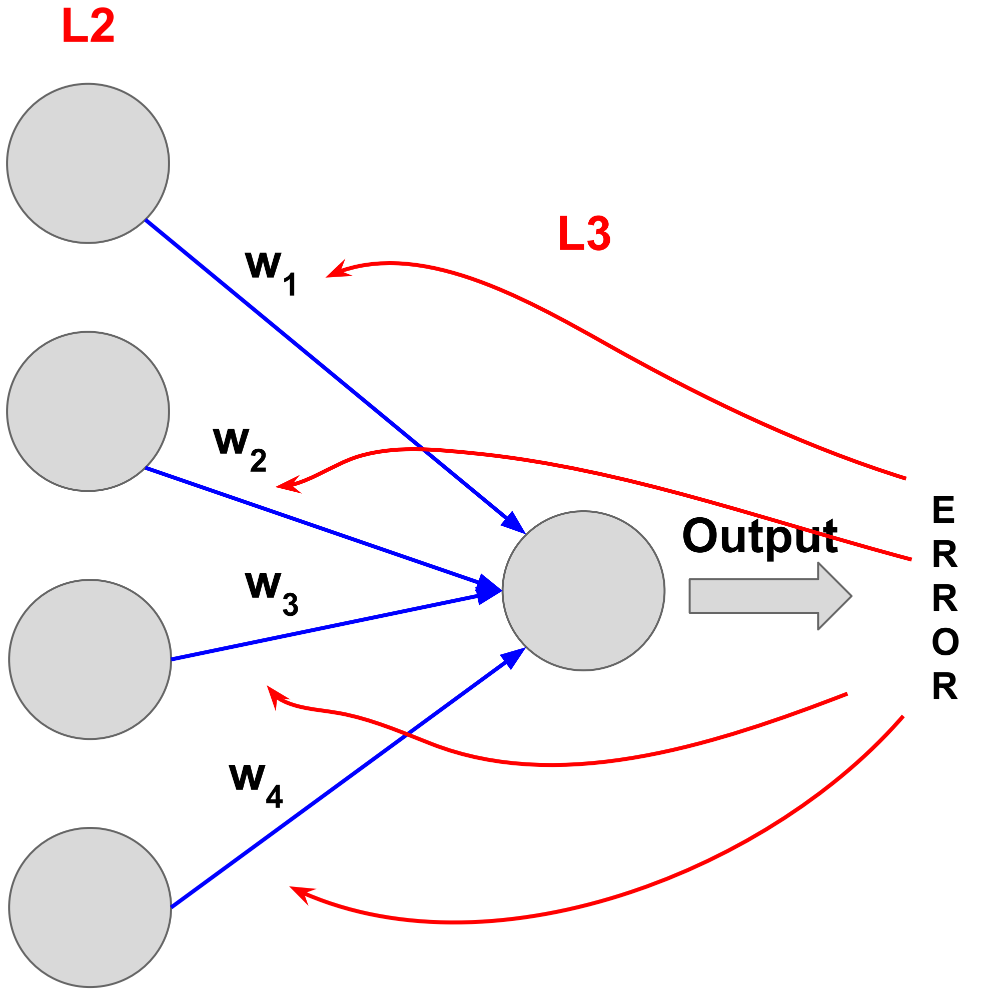
$E. W^T$

Remember,

Error matrix was 4 x 1

W2 was 4 x 1

So W^T will be 1 x 4



We now know following:

- How to feed forward ?
    - Multiply input by weights
    - Pass in activation function
- How to calculate error ?
    - Subtract obtained output from known output
- How to blame errors ?
    - Multiply error by transpose of weights

So, how to correct error ?

Error correction (Back propagation)

- We find derivative of activation function
- Derivative gives slope of tangent line which can guide us if we need to add to weight or subtract from it in order to correct the error
- This works because tangent emulates the curve closely at any point

```python
import numpy

x = numpy.array([[0,0],
                 [0,1],
                 [1,0],
                 [1,1]])
...

w2 = numpy.random.rand(4, 1) + 1

def nonlin(x, deriv=False):
    if deriv==True:
        return (x * (1 - x))
    return (1/(1 + numpy.exp(-x)))
```

```python
# feed forward

l1 = x

l2 = nonlin(numpy.dot(l1, w1))

l3 = nonlin(numpy.dot(l2, w2))

l3_errors = y - l3
```

```python
# code file nn2.py
import numpy

x = numpy.array([[0,0],
                 [0,1],
                 [1,0],
                 [1,1]])
…

w2 = numpy.random.rand(4, 1) + 1

def nonlin(x, deriv=False):
    if deriv==True:
        return (x * (1 - x))
    return (1/(1 + numpy.exp(-x)))
```

```python
l1 = x
l2 = nonlin(numpy.dot(l1, w1))
l3 = nonlin(numpy.dot(l2, w2))
l3_errors = y - l3
l3_delta = l3_errors * nonlin(l3,
deriv=True)


l2_error = l3_delta.dot(w2.T)


l2_delta = l2_error * nonlin(l2,
deriv=True)

w2 += l2.T.dot(l3_delta)

w1 += l1.T.dot(l2_delta)
```

Accuracy ?

77%

How to improve it ?

How do teach a child to speak ?

By talking to the child until she learns to copy you exactly

(This is called epochs)

simplenn.py

Code with epochs and query handler