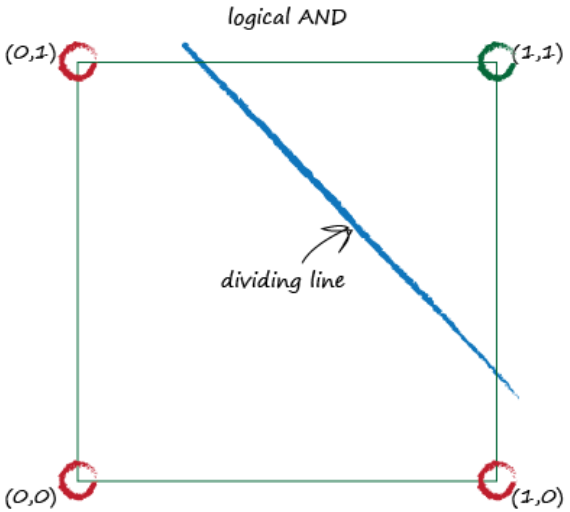
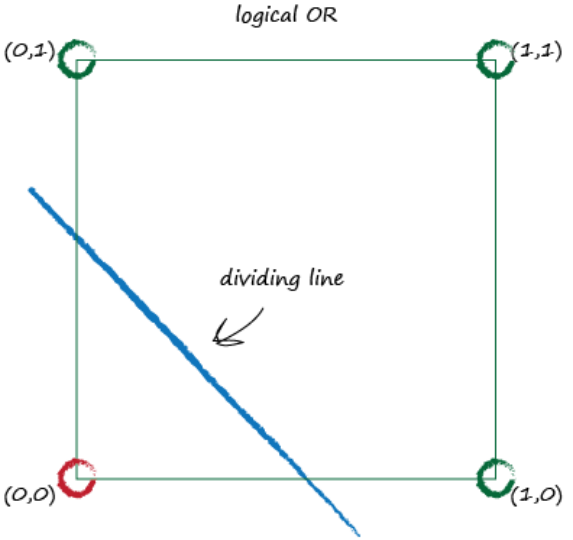
Deep Learning Internals-Part 2

By Mohit Kumar

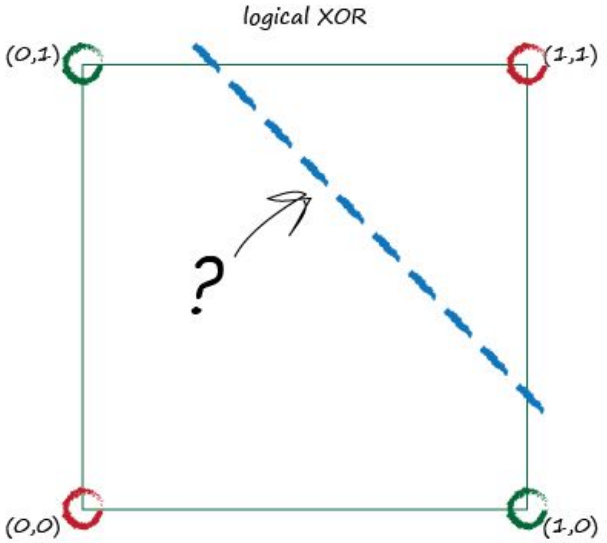
Artificial Neuron

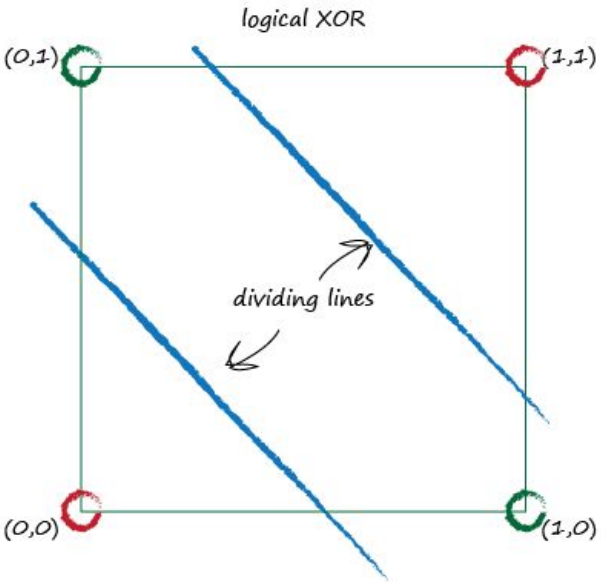




- By itself the perceptron is incapable of learning complex patterns in data.

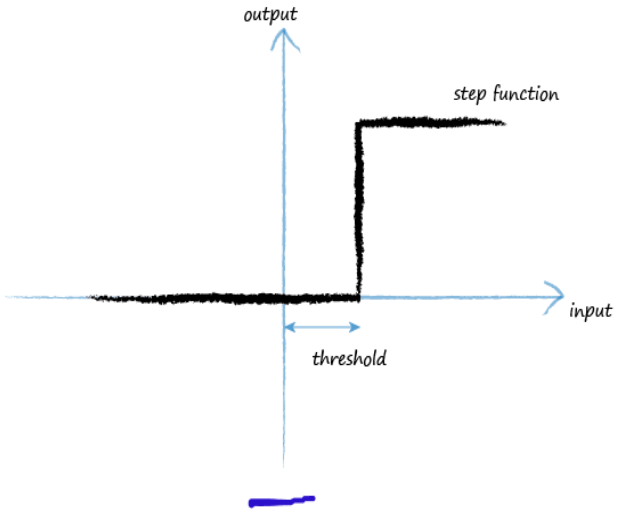
Artificial Neuron

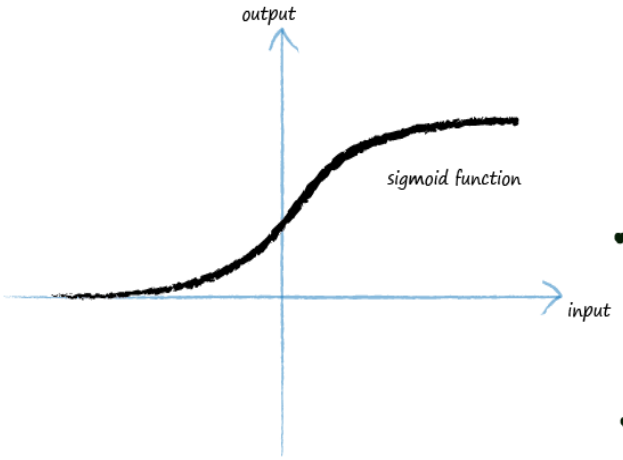




- But once it is combined with others it becomes a universal function approximator.

Artificial Neuron:Activation Function



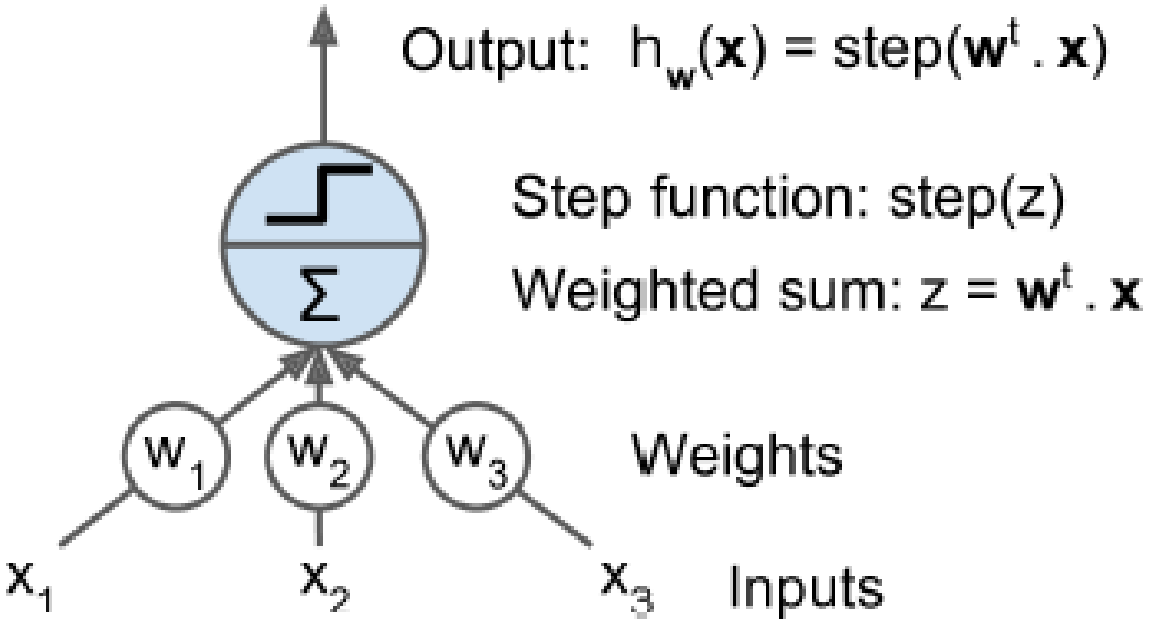


$$y = \frac{1}{1 + e^{-x}}$$

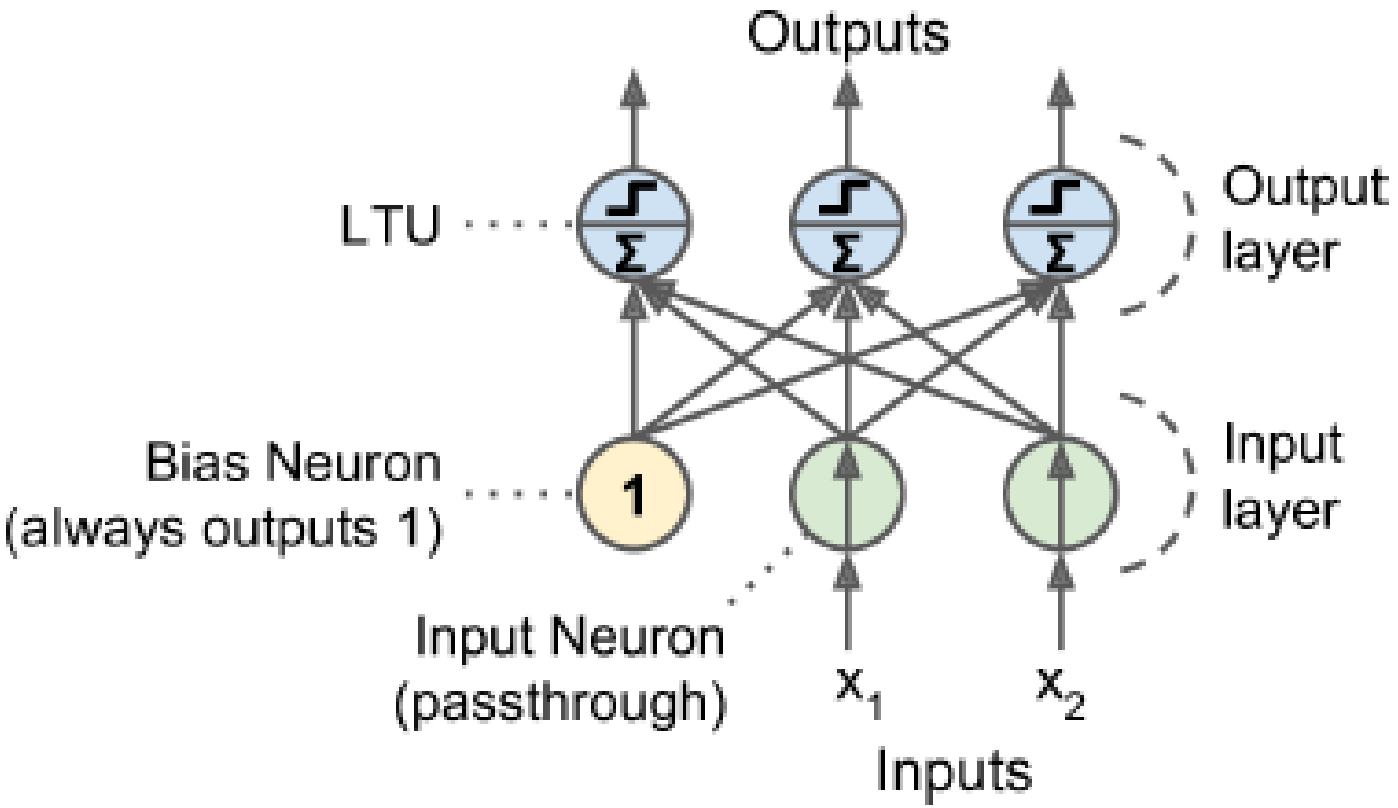
. The activation function is used to decide whether the neuron fires and "strengthen the connection"

- a simple step activation function.
- a sigmoid activation function or simply the logistic function.

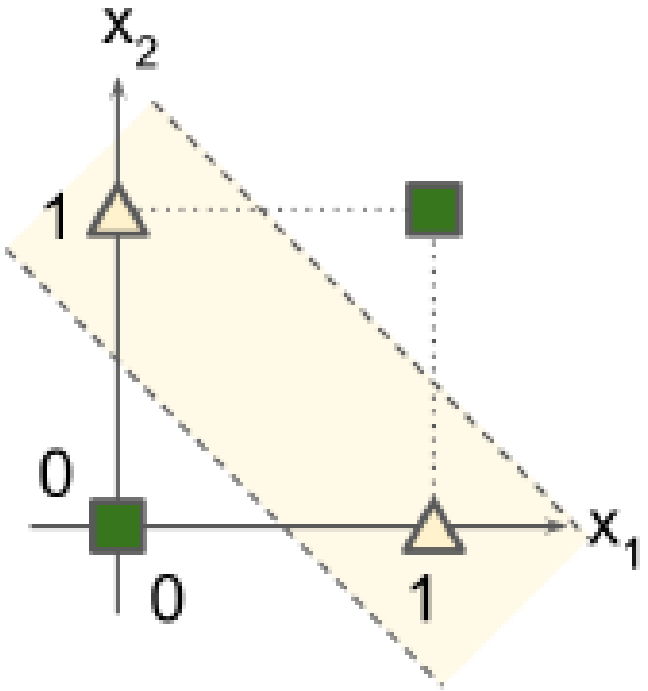
Perceptron(LTU)

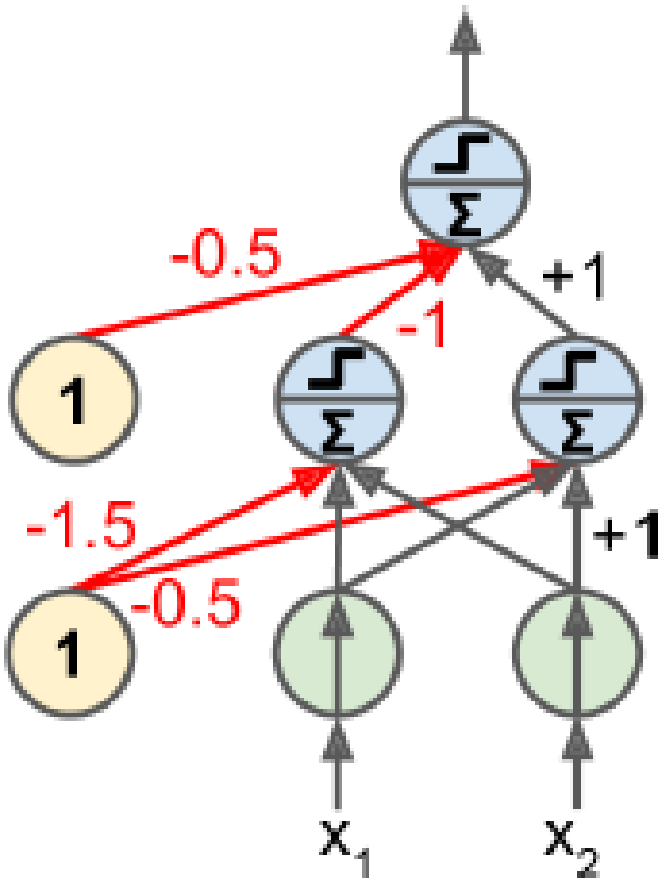


Perceptron(LTU):Multi Layer Perceptron

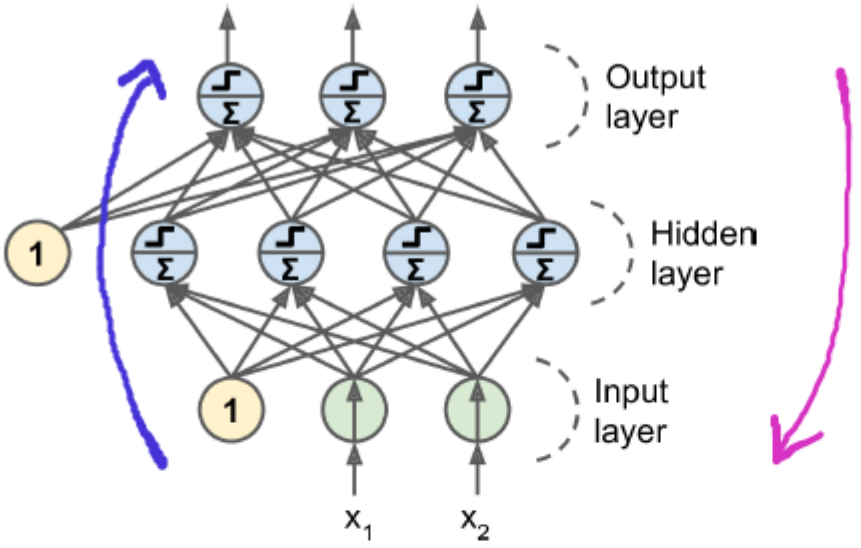


Perceptron(LTU):Multi Layer Perceptron:XOR



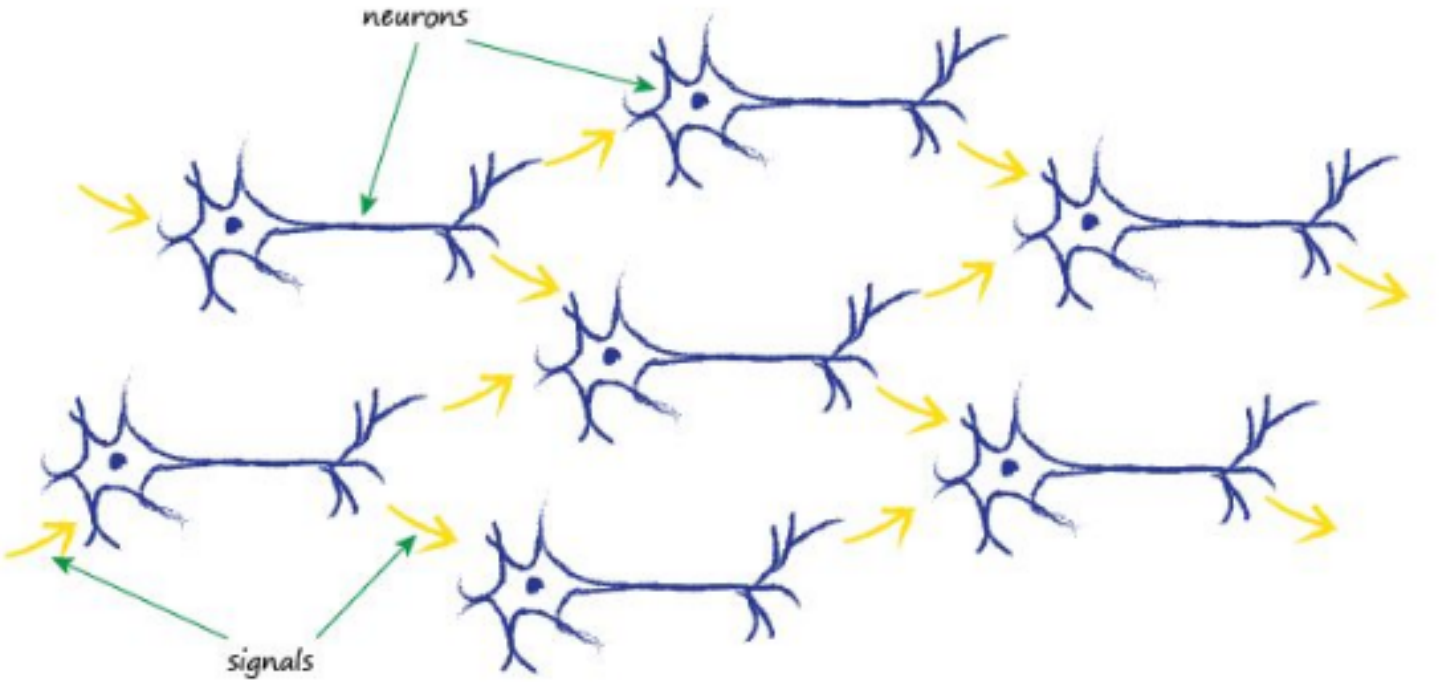


Perceptron(LTU):Multi Layer Perceptron

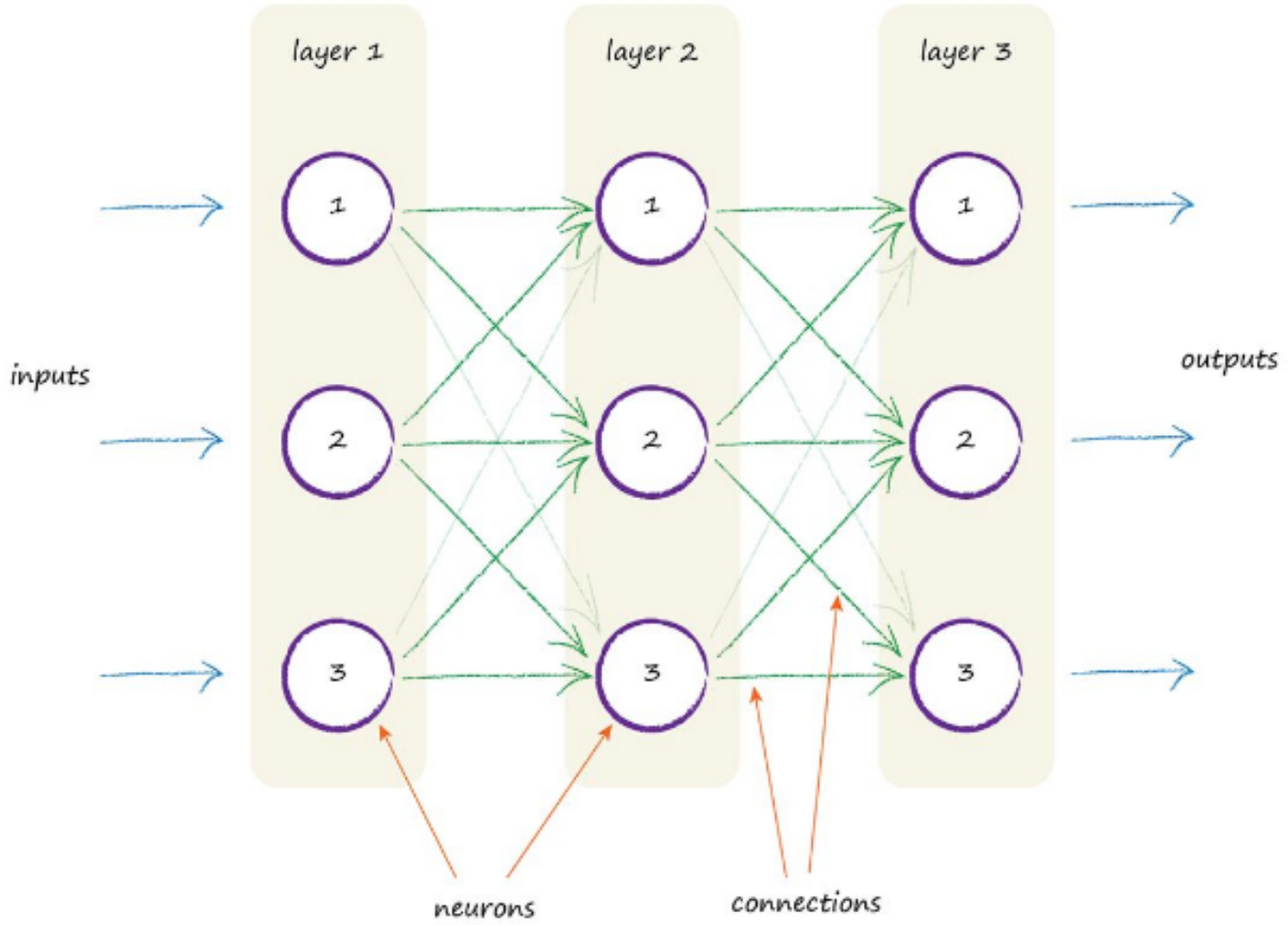


- forward the weighted sum through the activation function from 1 layer to another
- Calculate the MSE and back propagate adjusting the weights

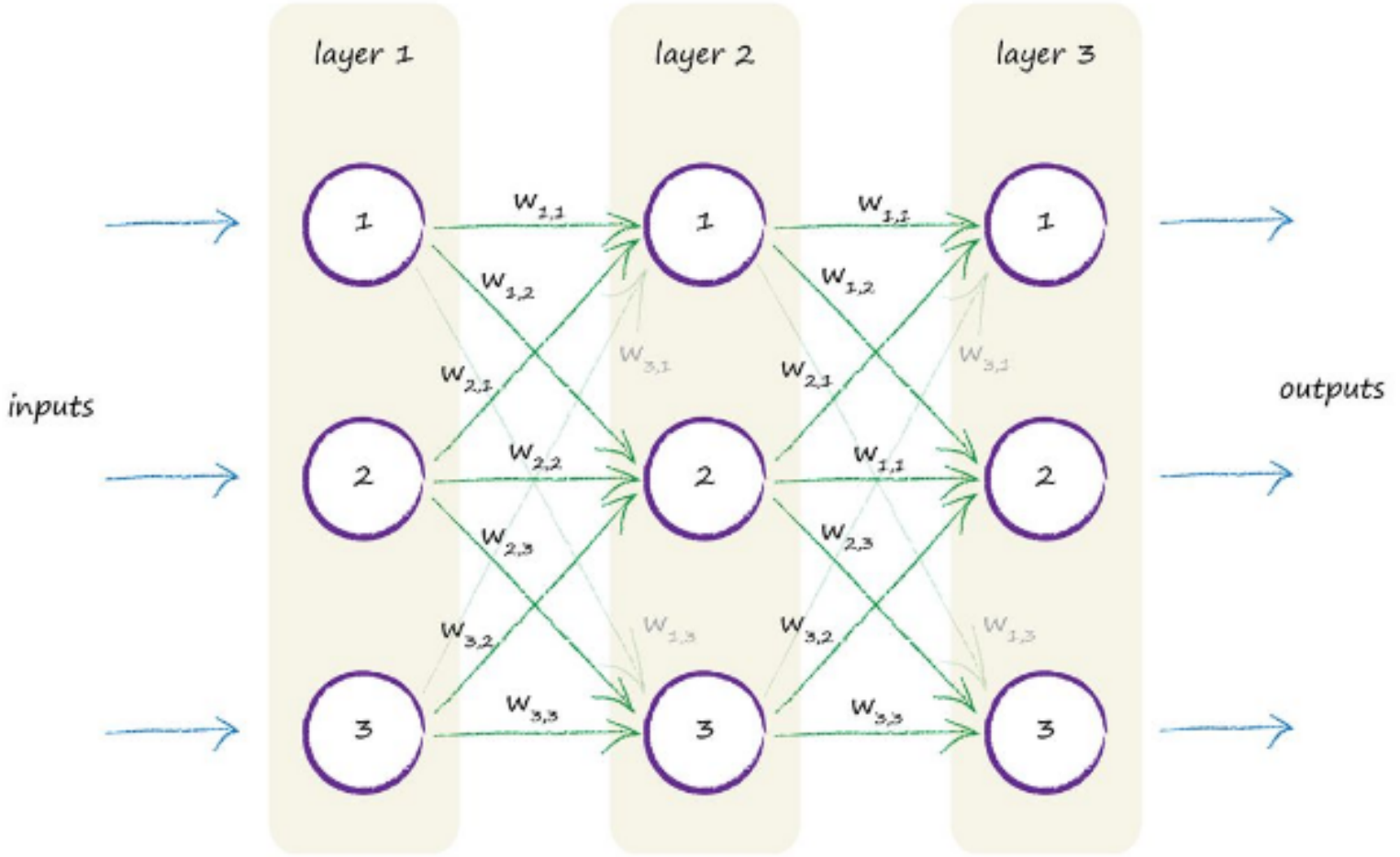
Perceptron(LTU):Multi Layer Perceptron:Biological



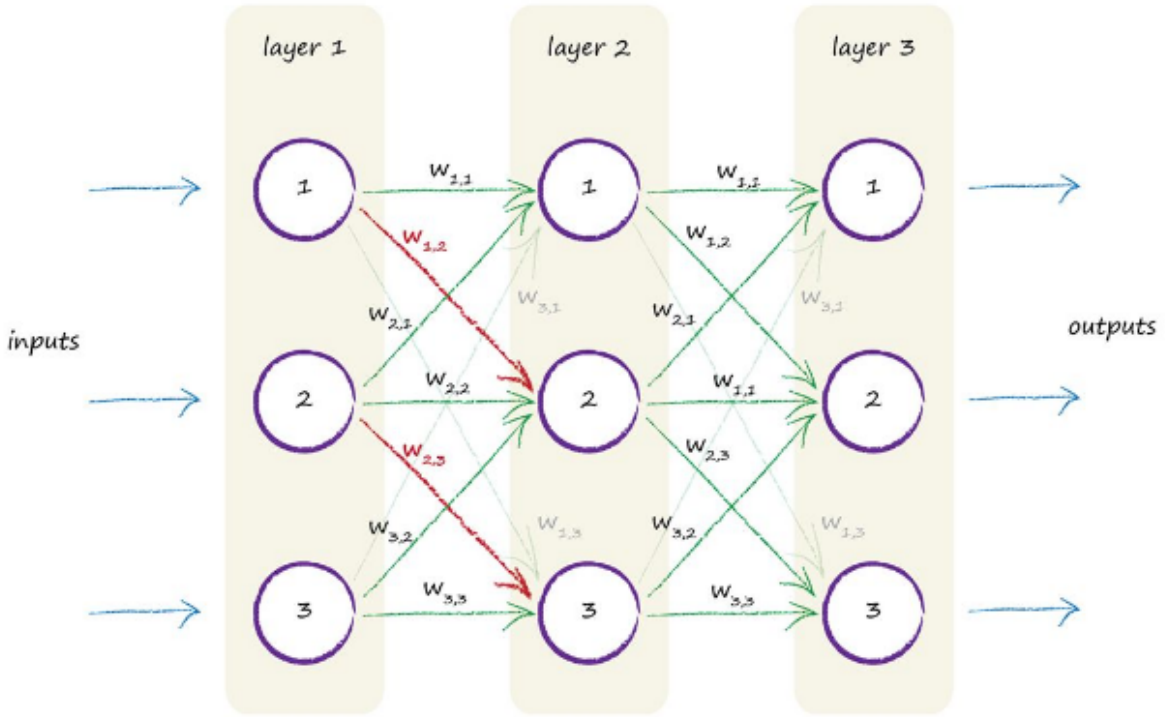
Perceptron(LTU):Multi Layer Perceptron:Artificial



Perceptron(LTU):Multi Layer Perceptron:Artificial:Weights

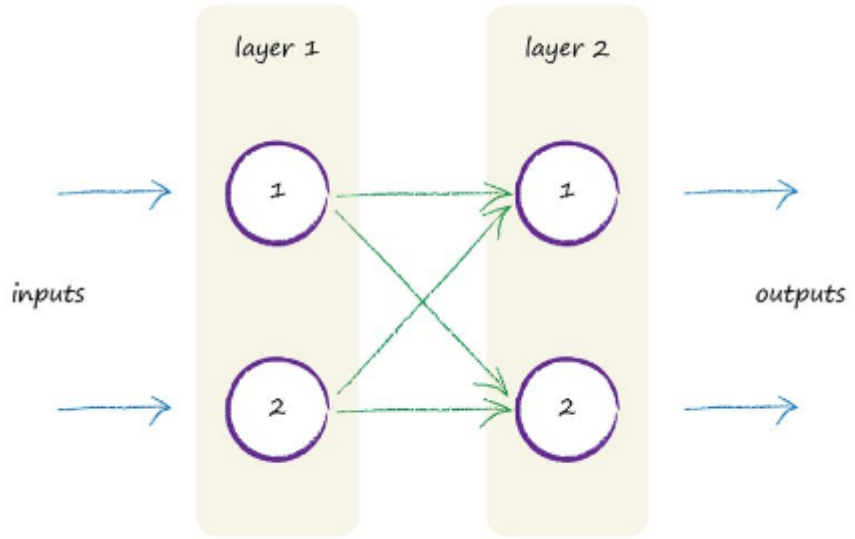


Perceptron(LTU):Multi Layer Perceptron:Artificial:Weights

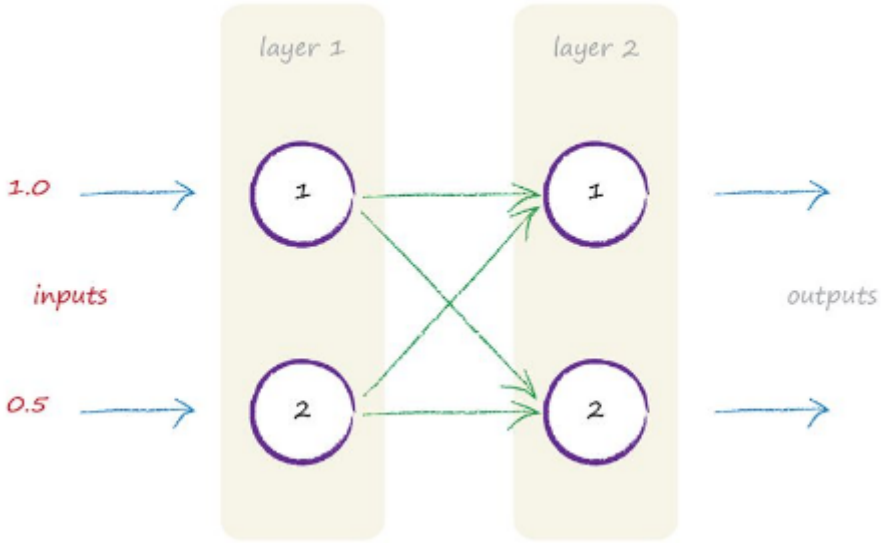


- These weights are organized as a vector for one neuron to all neurons of the next layer
- For a particular layer all the vectors for all the neurons in that layer are organized into a matrix

Multi Layer Perceptron:Following the signals

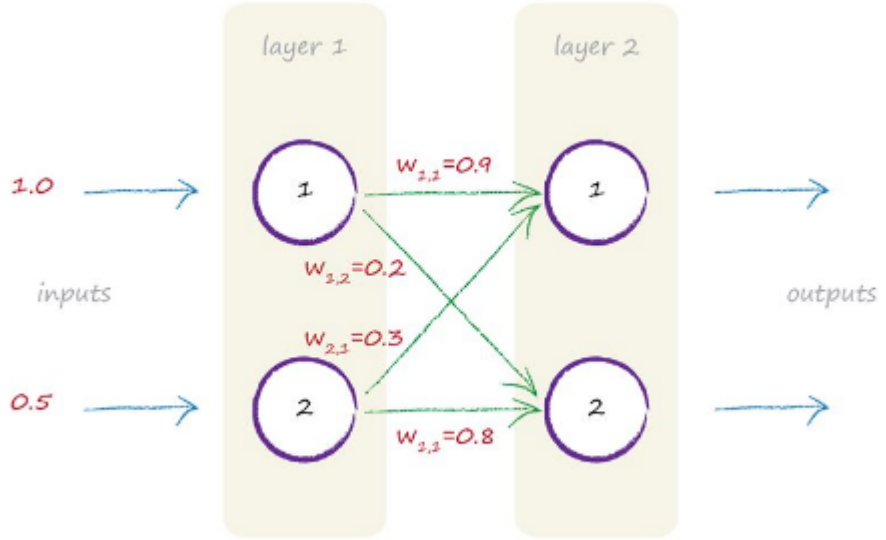


Multi Layer Perceptron:Following the signals

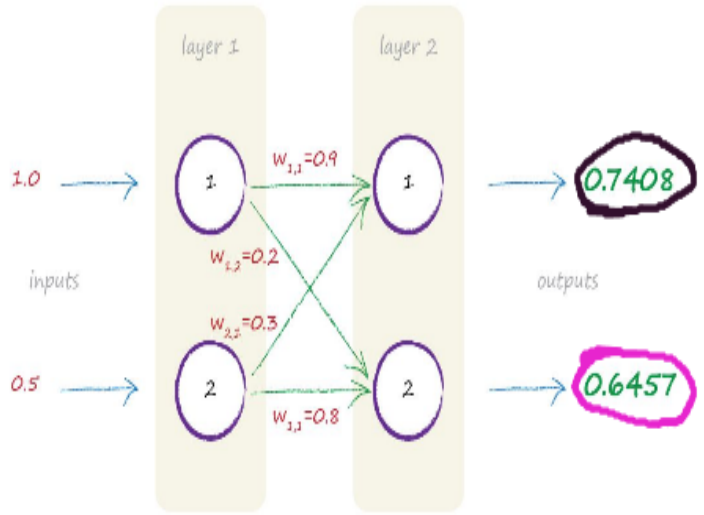


- $w_{1,1} = 0.9$
- $w_{1,2} = 0.2$
- $w_{2,1} = 0.3$
- $w_{2,2} = 0.8$

Multi Layer Perceptron:Following the signals



Multi Layer Perceptron:Following the signals



x_{21} = (1st Node × link weight) + (2nd Node × link weight)

= (1.0 × 0.9) + (0.5 × 0.3)

= 0.9 + 0.15

= 1.05

$\sigma(x) = 1/(1+e^{-1.05}) = 1/(1+0.39) = .7408$

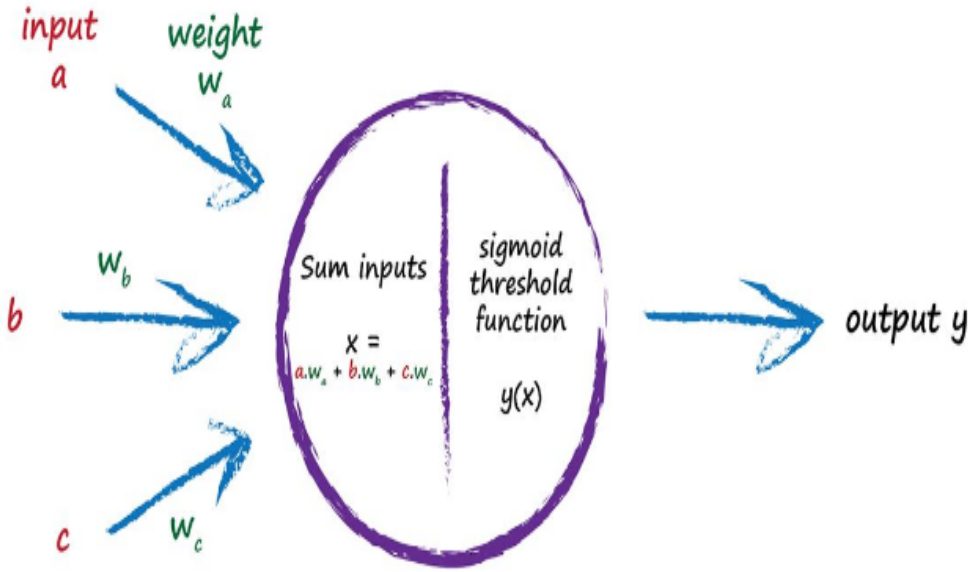
x_{22} = (1.0 × 0.2) + (0.6 × 0.8)

= (0.2 + 0.4)

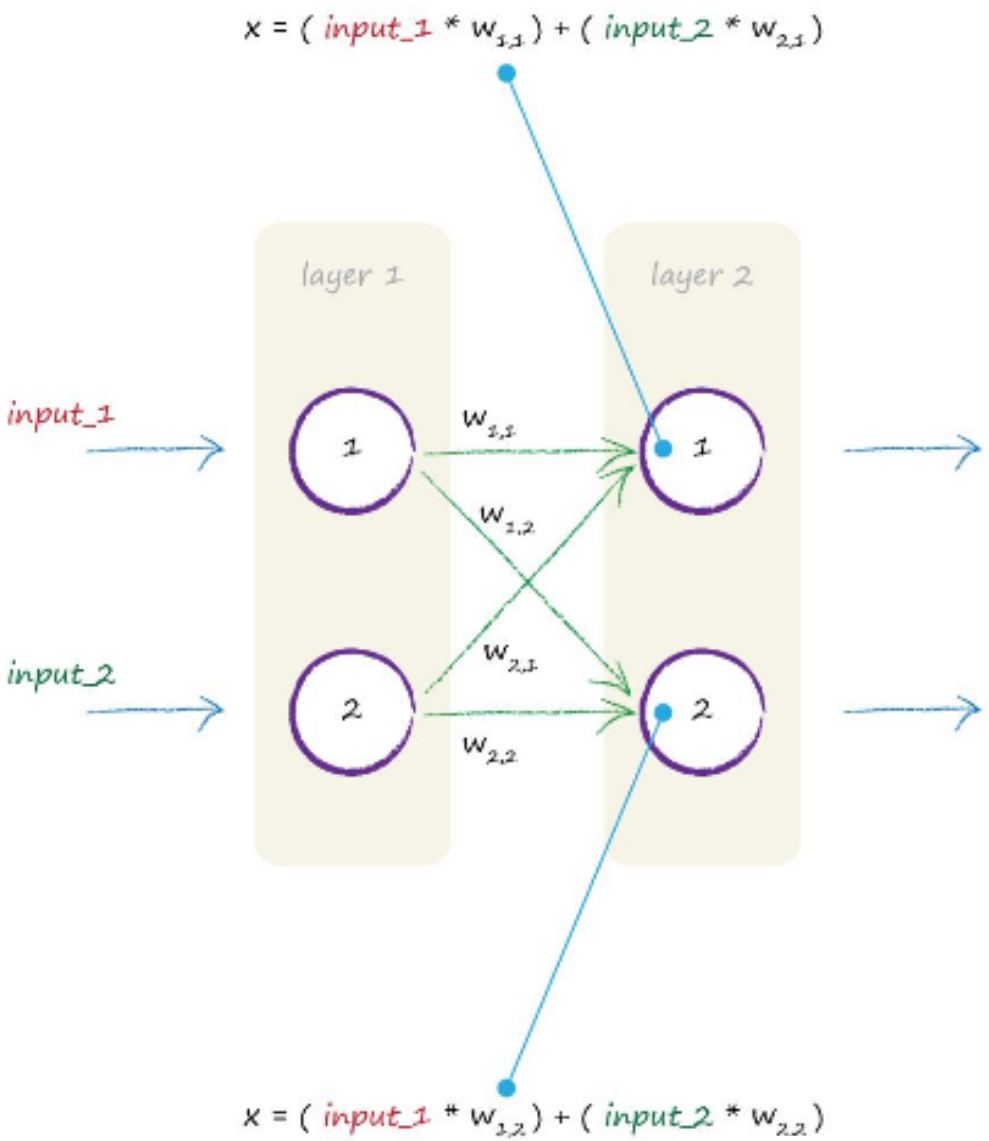
= 0.6

$\sigma_{(x)} = 1/1+e^{-0.6} = 1/(1+0.5488)$

= 0.6457



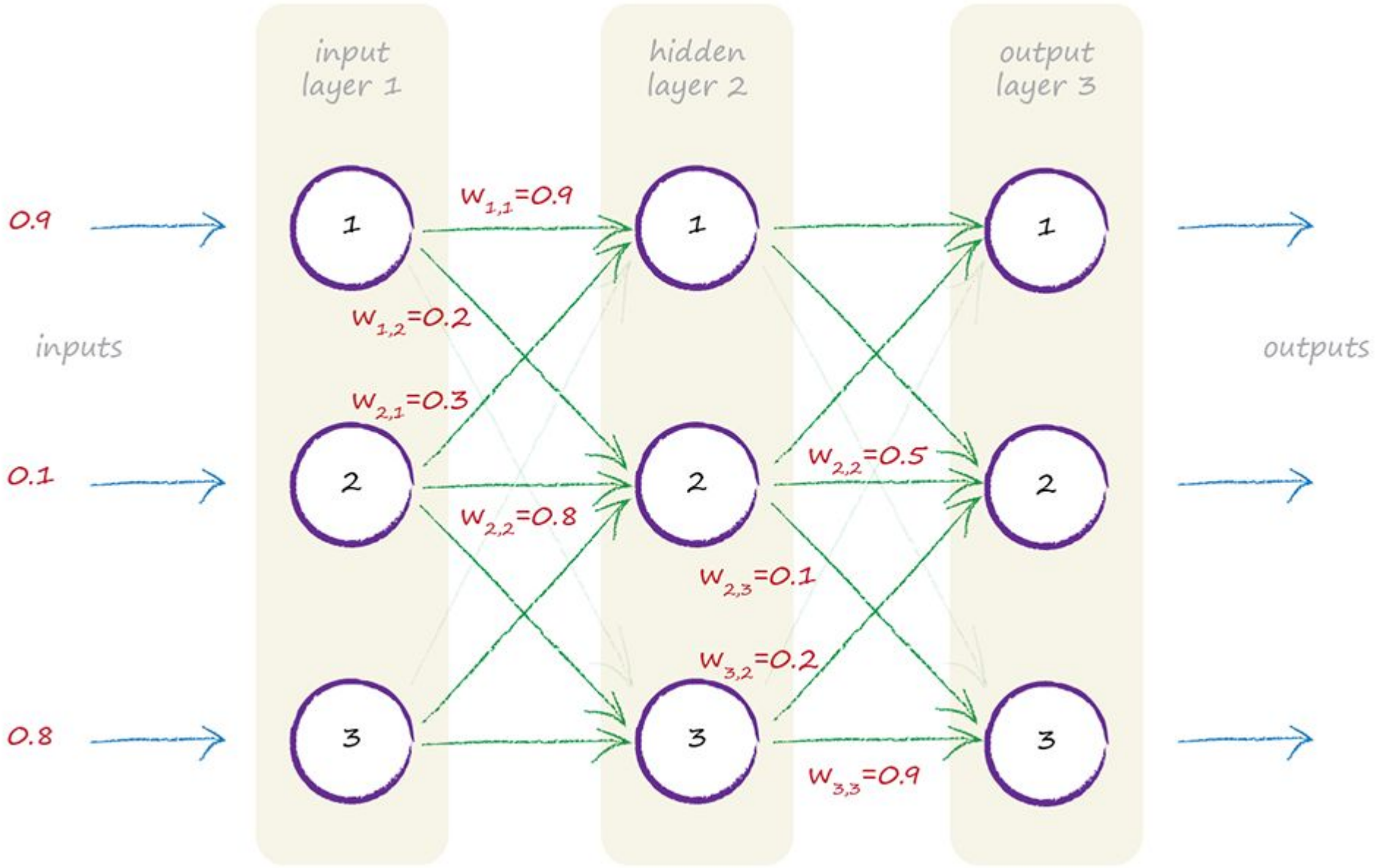
Multi Layer Perceptron:Matrix representation



$$\mathbf{X} = \mathbf{W} \cdot \mathbf{I}$$

$$\begin{pmatrix} w_{1,1} & w_{2,1} \\ w_{1,2} & w_{2,2} \end{pmatrix} \begin{pmatrix} input\_1 \\ input\_2 \end{pmatrix} = \begin{pmatrix} (input\_1 * w_{1,1}) + (input\_2 * w_{2,1}) \\ (input\_1 * w_{1,2}) + (input\_2 * w_{2,2}) \end{pmatrix}$$

Multi Layer Perceptron: 3 Layer



Multi Layer Perceptron: 3 Layer Neural Network:Initialization

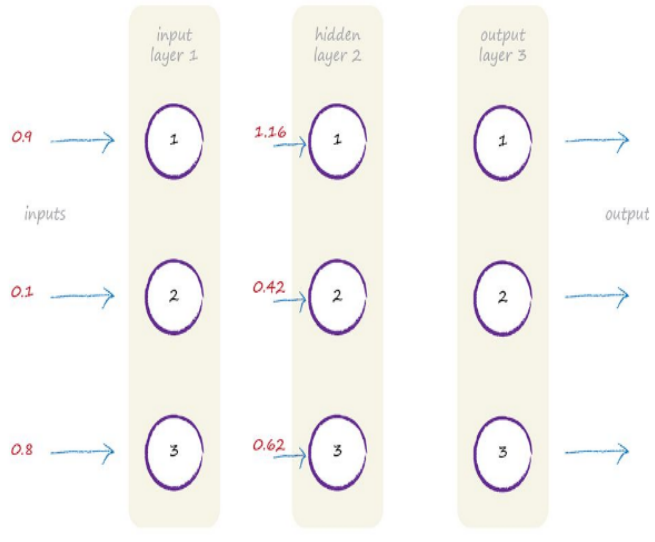
```
class neuralNetwork:


    # initialise the neural network
    def __init__(self, inputnodes, hiddennodes, outputnodes, learningrate,wih=None,who=None):
        # set number of nodes in each input, hidden, output layer
        self.inodes = inputnodes
        self.hnodes = hiddennodes
        self.onodes = outputnodes

        # link weight matrices, wih and who
        # weights inside the arrays are w_i_j, where link is from node i to node j in the next layer
        # w11 w21
        # w12 w22 etc
        if wih is None:
            self.wih = numpy.random.normal(0.0, pow(self.inodes, -0.5), (self.hnodes, self.inodes))
            self.who = numpy.random.normal(0.0, pow(self.hnodes, -0.5), (self.onodes, self.hnodes))
        else:
            self.wih=wih
            self.who=who

        # learning rate
        self.lr = learningrate

        # activation function is the sigmoid function
        self.activation_function = lambda x: scipy.special.expit(x)
        # reverse of sigmoid to work backwards
        self.inverse_activation_function = lambda x: scipy.special.logit(x)
        pass
```

- number of neurons in each layer
- random initialization of weights
- mean of 0.0 and float as type
- standard dev(σ) of $3^{-0.5}$ (0.577) from normal
- shape of weights
- Sigmoid function
- reverse

Multi Layer Perceptron: 3 Layer Neural Network:1-2 Layer

```python
# train the neural network
def train(self, inputs_list, targets_list):
    # convert inputs list to 2d array
    inputs = numpy.array(inputs_list, ndmin=2).T
    targets = numpy.array(targets_list, ndmin=2).T

    # calculate signals into hidden layer
    hidden_inputs = numpy.dot(self.wih, inputs)
    # calculate the signals emerging from hidden layer
    hidden_outputs = self.activation_function(hidden_inputs)

    # calculate signals into final output layer
    final_inputs = numpy.dot(self.who, hidden_outputs)
    # calculate the signals emerging from final output layer
    final_outputs = self.activation_function(final_inputs)

    # output layer error is the (target - actual)
    output_errors = targets - final_outputs
    # hidden layer error is the output_errors, split by weights, recombined at hidden nodes
    hidden_errors = numpy.dot(self.who.T, output_errors)

    # update the weights for the links between the hidden and output layers
    self.who += self.lr * numpy.dot((output_errors * final_outputs * (1.0 - final_outputs)), numpy.transpose(hidden_outputs))

    # update the weights for the links between the input and hidden layers
    self.wih += self.lr * numpy.dot((hidden_errors * hidden_outputs * (1.0 - hidden_outputs)), numpy.transpose(inputs))

    pass
```

$$X_{hidden} = \begin{pmatrix} 1.16 \\ 0.42 \\ 0.62 \end{pmatrix} = W_{input\,hidden} = \begin{pmatrix} 0.9 & 0.3 & 0.4 \\ 0.2 & 0.8 & 0.2 \\ 0.1 & 0.5 & 0.6 \end{pmatrix} \cdot I = \begin{pmatrix} 0.9 \\ 0.1 \\ 0.8 \end{pmatrix}$$

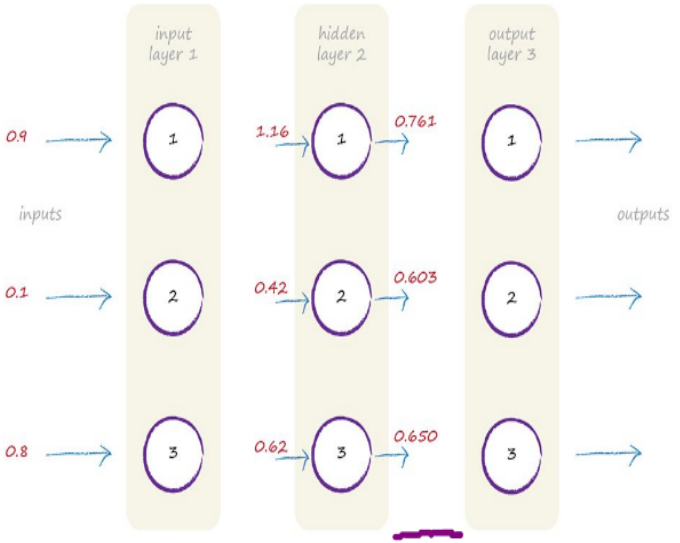


Multi Layer Perceptron: 3 Layer Neural Network:1-2 Layer(Sigmoid)

```python
# train the neural network
def train(self, inputs_list, targets_list):
    # convert inputs list to 2d array
    inputs = numpy.array(inputs_list, ndmin=2).T
    targets = numpy.array(targets_list, ndmin=2).T

    # calculate signals into hidden layer
    hidden_inputs = numpy.dot(self.wih, inputs)
    # calculate the signals emerging from hidden layer
    hidden_outputs = self.activation_function(hidden_inputs)

    # calculate signals into final output layer
    final_inputs = numpy.dot(self.who, hidden_outputs)
    # calculate the signals emerging from final output layer
    final_outputs = self.activation_function(final_inputs)

    # output layer error is the (target - actual)
    output_errors = targets - final_outputs
    # hidden layer error is the output_errors, split by weights, recombined at hidden nodes
    hidden_errors = numpy.dot(self.who.T, output_errors)

    # update the weights for the links between the hidden and output layers
    self.who += self.lr * numpy.dot((output_errors * final_outputs * (1.0 - final_outputs)), numpy.transpose(hidden_outputs))

    # update the weights for the links between the input and hidden layers
    self.wih += self.lr * numpy.dot((hidden_errors * hidden_outputs * (1.0 - hidden_outputs)), numpy.transpose(inputs))

    pass
```

$$O_{hidden} = sigmoid\begin{pmatrix} 1.16 \\ 0.42 \\ 0.62 \end{pmatrix}$$

$$O_{hidden} = \begin{pmatrix} 0.761 \\ 0.603 \\ 0.650 \end{pmatrix}$$

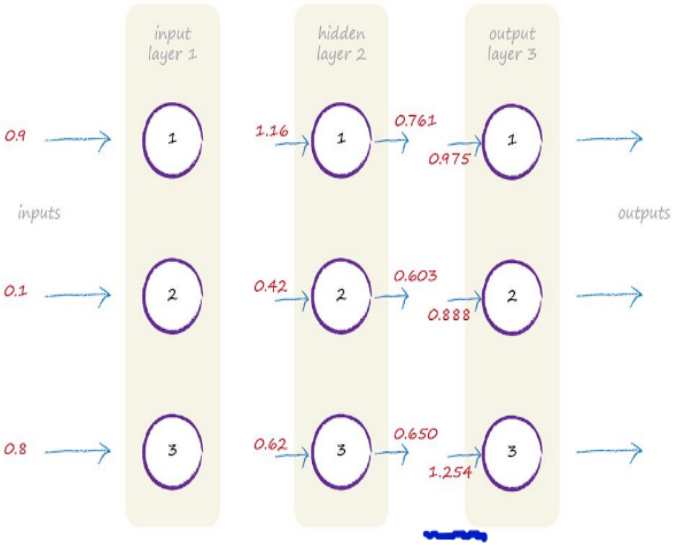


Multi Layer Perceptron: 3 Layer Neural Network:2-3 Layer

```
# train the neural network
def train(self, inputs_list, targets_list):
    # convert inputs list to 2d array
    inputs = numpy.array(inputs_list, ndmin=2).T
    targets = numpy.array(targets_list, ndmin=2).T

    # calculate signals into hidden layer
    hidden_inputs = numpy.dot(self.wih, inputs)
    # calculate the signals emerging from hidden layer
    hidden_outputs = self.activation_function(hidden_inputs)

    # calculate signals into final output layer
    final_inputs = numpy.dot(self.who, hidden_outputs)
    # calculate the signals emerging from final output layer
    final_outputs = self.activation_function(final_inputs)

    # output layer error is the (target - actual)
    output_errors = targets - final_outputs
    # hidden layer error is the output_errors, split by weights, recombined at hidden nodes
    hidden_errors = numpy.dot(self.who.T, output_errors)

    # update the weights for the links between the hidden and output layers
    self.who += self.lr * numpy.dot((output_errors * final_outputs * (1.0 - final_outputs)), numpy.transpose(hidden_outputs))

    # update the weights for the links between the input and hidden layers
    self.wih += self.lr * numpy.dot((hidden_errors * hidden_outputs * (1.0 - hidden_outputs)), numpy.transpose(inputs))

    pass
```

$$X_{output} = \begin{pmatrix} 0.3 & 0.7 & 0.5 \\ 0.6 & 0.5 & 0.2 \\ 0.8 & 0.1 & 0.9 \end{pmatrix} \cdot \begin{pmatrix} 0.761 \\ 0.603 \\ 0.650 \end{pmatrix}$$

$$X_{output} = \begin{pmatrix} 0.975 \\ 0.888 \\ 1.254 \end{pmatrix}$$

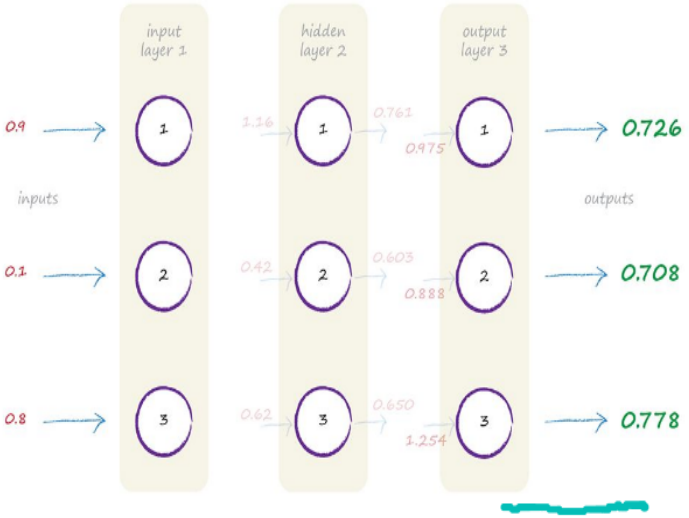


Multi Layer Perceptron: 3 Layer Neural Network:2-3 Layer(Sigmoid)

```
# train the neural network
def train(self, inputs_list, targets_list):
    # convert inputs list to 2d array
    inputs = numpy.array(inputs_list, ndmin=2).T
    targets = numpy.array(targets_list, ndmin=2).T

    # calculate signals into hidden layer
    hidden_inputs = numpy.dot(self.wih, inputs)
    # calculate the signals emerging from hidden layer
    hidden_outputs = self.activation_function(hidden_inputs)

    # calculate signals into final output layer
    final_inputs = numpy.dot(self.who, hidden_outputs)
    # calculate the signals emerging from final output layer
    final_outputs = self.activation_function(final_inputs)

    # output layer error is the (target - actual)
    output_errors = targets - final_outputs
    # hidden layer error is the output_errors, split by weights, recombined at hidden nodes
    hidden_errors = numpy.dot(self.who.T, output_errors)

    # update the weights for the links between the hidden and output layers
    self.who += self.lr * numpy.dot((output_errors * final_outputs * (1.0 - final_outputs)), numpy.transpose(hidden_outputs))

    # update the weights for the links between the input and hidden layers
    self.wih += self.lr * numpy.dot((hidden_errors * hidden_outputs * (1.0 - hidden_outputs)), numpy.transpose(inputs))

    pass
```

$$O_{output} = sigmoid\begin{pmatrix} 0.975 \\ 0.888 \\ 1.254 \end{pmatrix}$$

$$O_{output} = \begin{pmatrix} 0.726 \\ 0.708 \\ 0.778 \end{pmatrix}$$



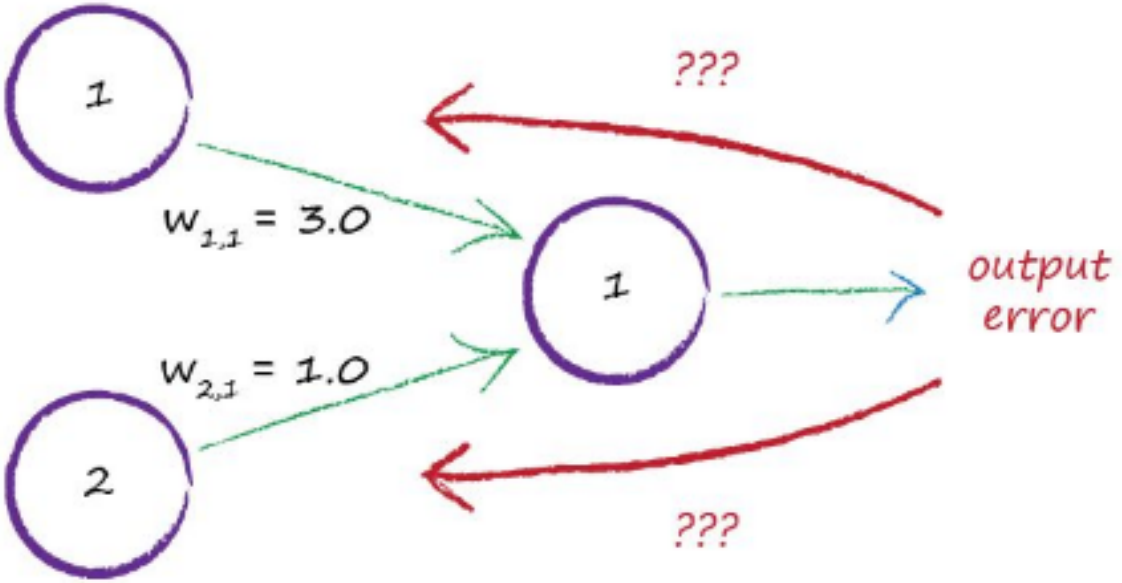
Multi Layer Perceptron: 3 Layer Neural Network:

```
def train(self, inputs_list, targets_list):
    # convert inputs list to 2d array
    inputs = numpy.array(inputs_list, ndmin=2).T
    targets = numpy.array(targets_list, ndmin=2).T

    # calculate signals into hidden layer
    hidden_inputs = numpy.dot(self.wih, inputs)
    # calculate the signals emerging from hidden layer
    hidden_outputs = self.activation_function(hidden_inputs)

    # calculate signals into final output layer
    final_inputs = numpy.dot(self.who, hidden_outputs)
    # calculate the signals emerging from final output layer
    final_outputs = self.activation_function(final_inputs)

    # output layer error is the (target - actual)
    output_errors = targets - final_outputs
    # hidden layer error is the output_errors, split by weights, recombined at hidden nodes
    hidden_errors = numpy.dot(self.who.T, output_errors)

    # update the weights for the links between the hidden and output layers
    self.who += self.lr * numpy.dot((output_errors * final_outputs * (1.0 - final_outputs)), numpy.transpose(hidden_outputs))

    # update the weights for the links between the input and hidden layers
    self.wih += self.lr * numpy.dot((hidden_errors * hidden_outputs * (1.0 - hidden_outputs)), numpy.transpose(inputs))

    pass
```

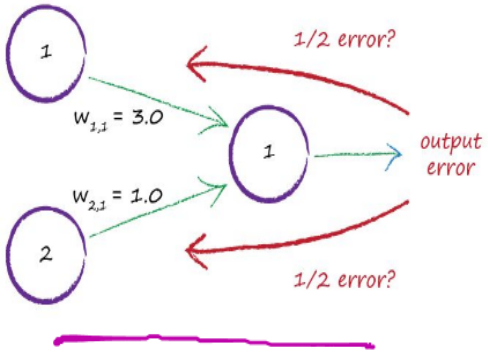
- first set of weights
- second set of weights

```
./DL/ANN/manualDnninternals.py
self.wih: [
[ 0.9  0.3  0.4]
 [ 0.2  0.8  0.2]
 [ 0.1  0.5  0.6]]
self.who: [
[ 0.3  0.7  0.5]
 [ 0.6  0.5  0.2]
 [ 0.8  0.1  0.9]]
(matrix([
[ 1.16],
 [ 0.42],
 [ 0.62]]),
matrix([[ 0.76133271],
[ 0.60348325],
 [ 0.65021855]]),
matrix([[ 0.97594736],
[ 0.88858496],
 [ 1.25461119]]),
matrix([[ 0.72630335],
[ 0.70859807],
 [ 0.77809706]]))
```

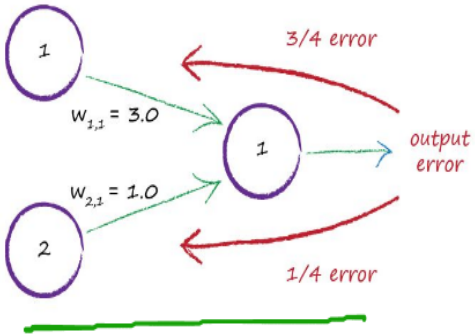
Multi Layer Perceptron: Learning Weights



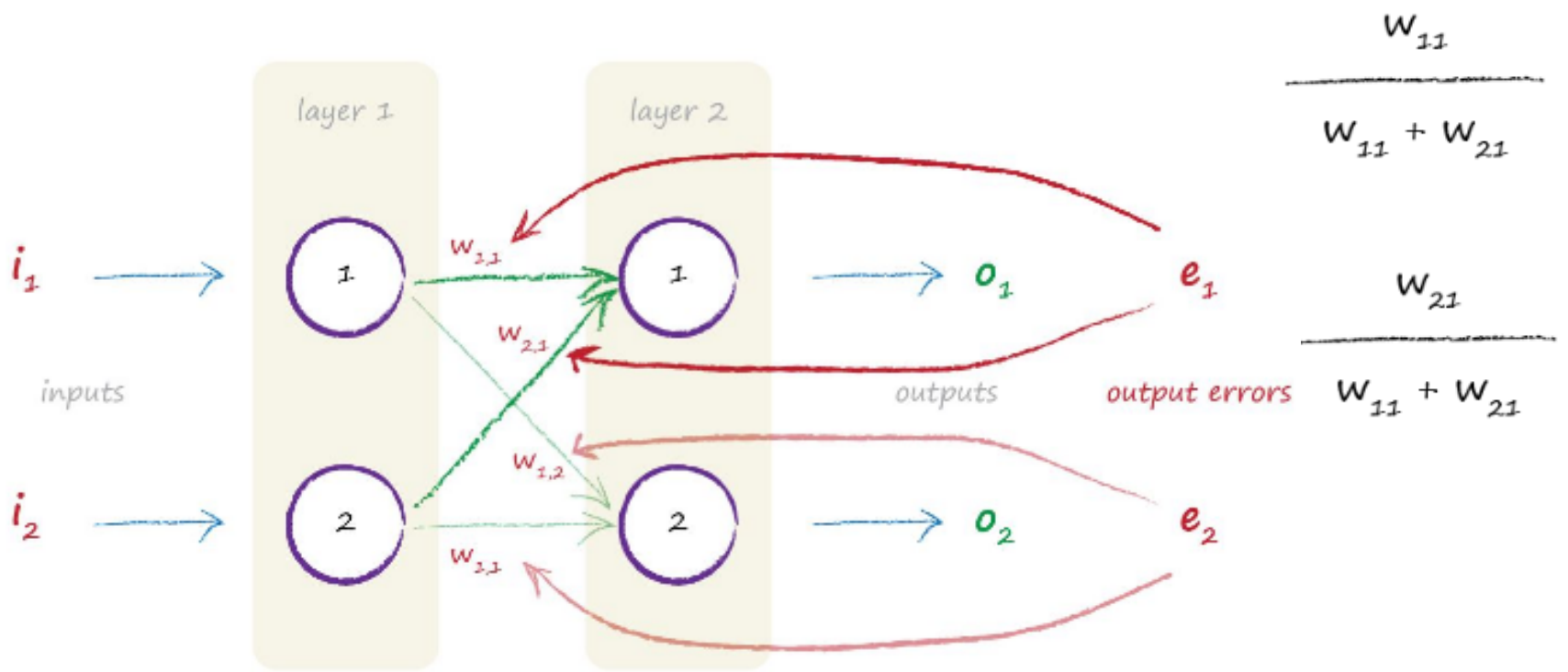
Multi Layer Perceptron: Learning Weights



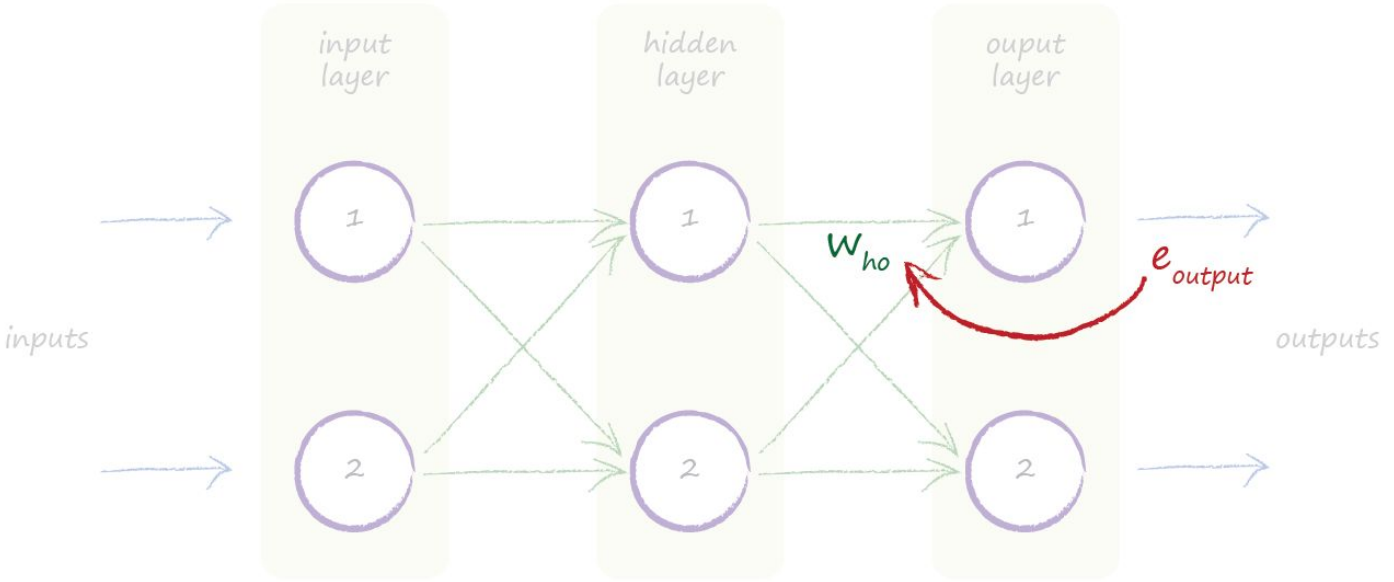
- would not work that badly, even though have not tried
- looks more intuitive



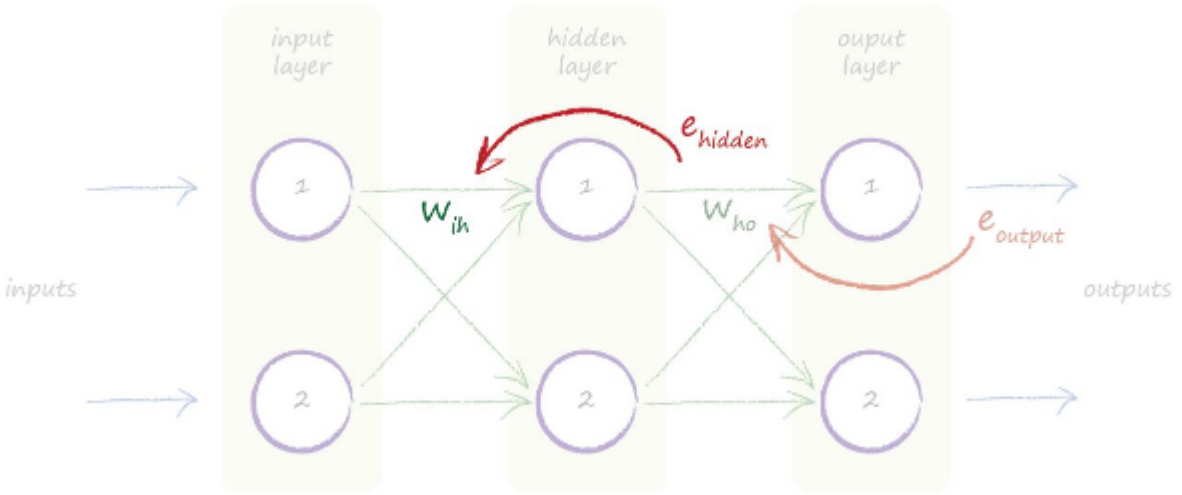
Multi Layer Perceptron: Learning Weights:2 output nodes



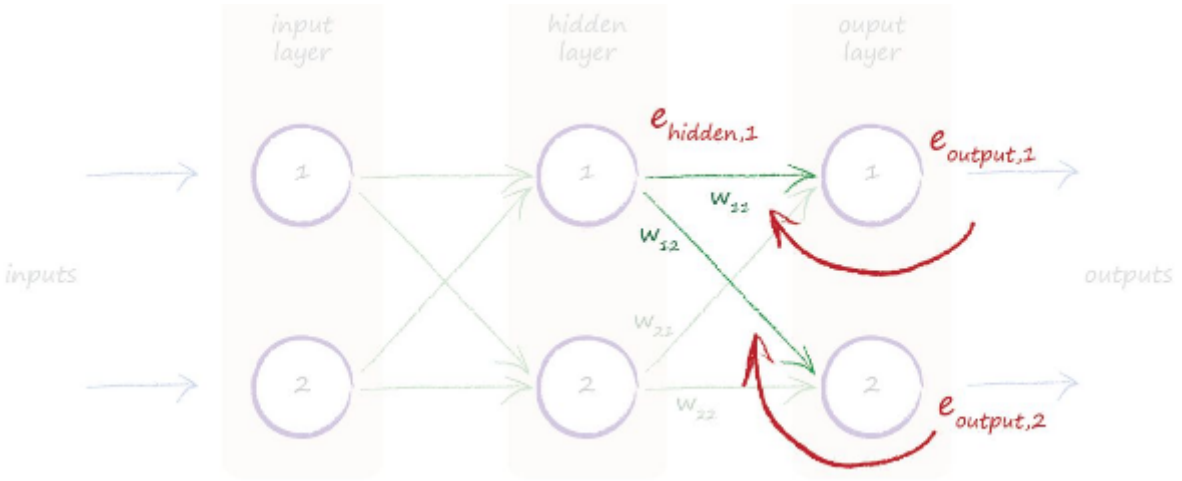
Multi Layer Perceptron: Back Propagation: Multiple Layers



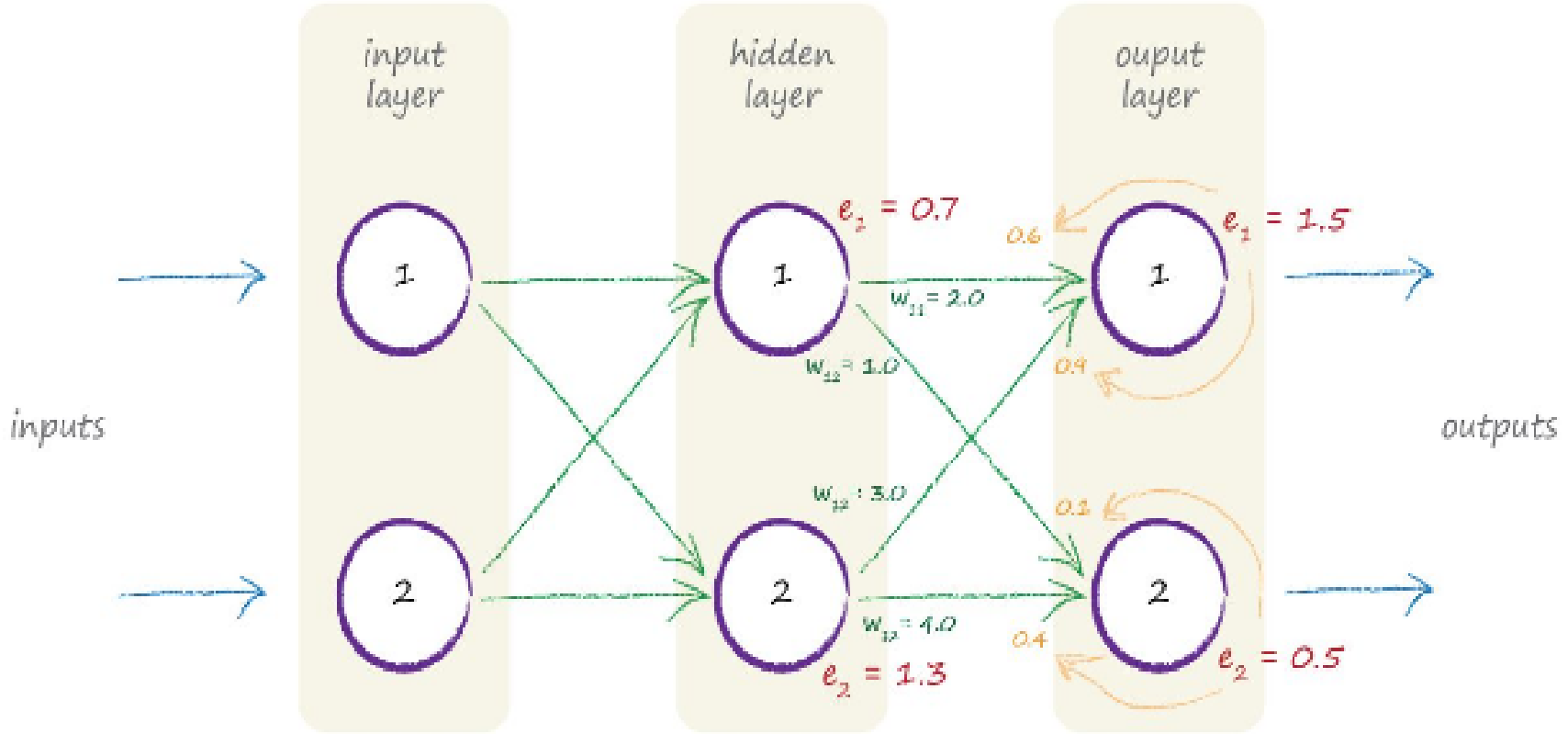
Multi Layer Perceptron: Back Propagation: Multiple Layers



Multi Layer Perceptron: Back Propagation: Multiple Layers



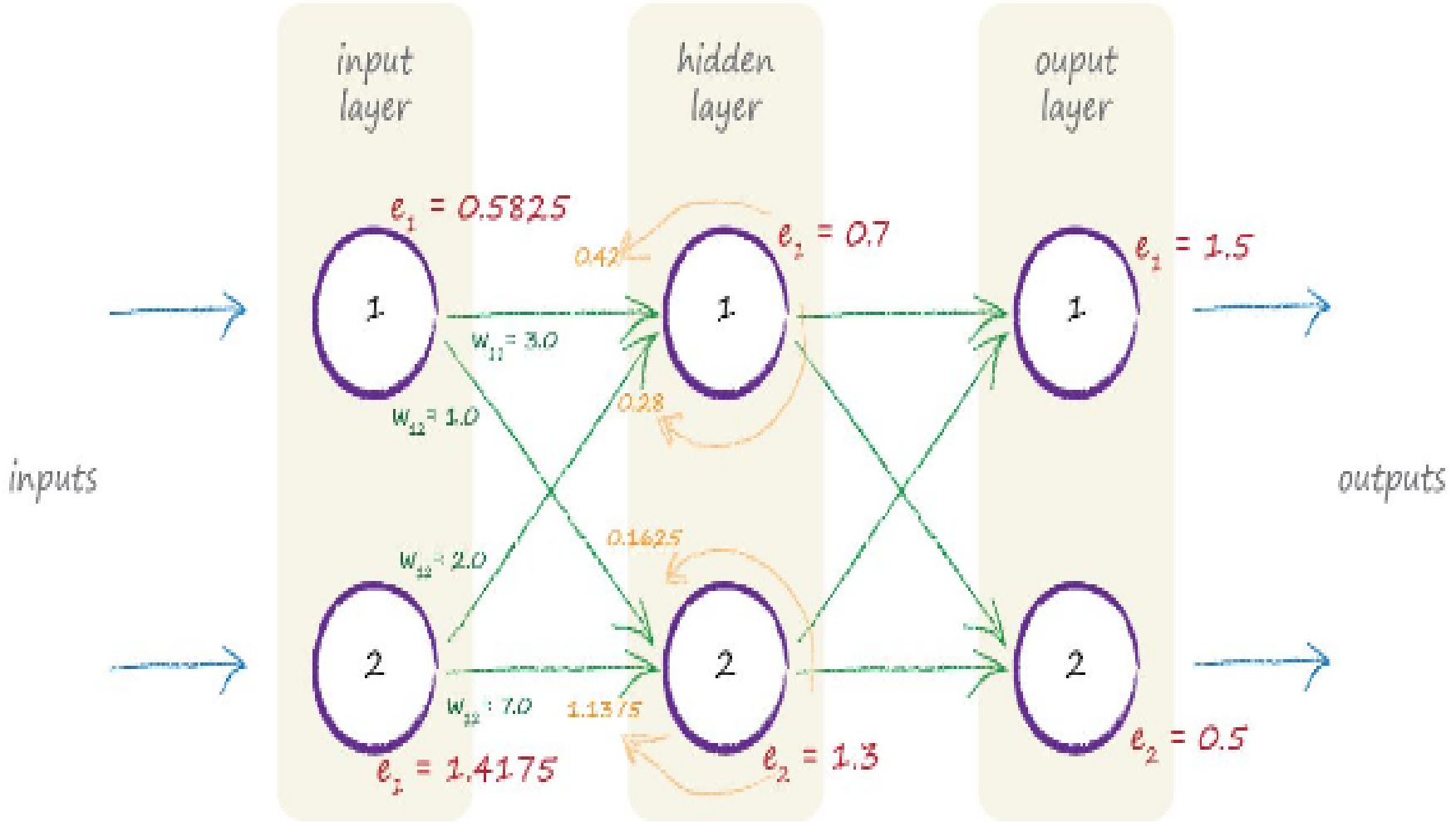
Multi Layer Perceptron: Back Propagation: Multiple Layers



$e_{hidden,1}$ = sum of split errors on links w_{11} and w_{12}

$$= e_{output,1} * \frac{w_{11}}{w_{11} + w_{21}} + e_{output,2} * \frac{w_{12}}{w_{12} + w_{22}}$$

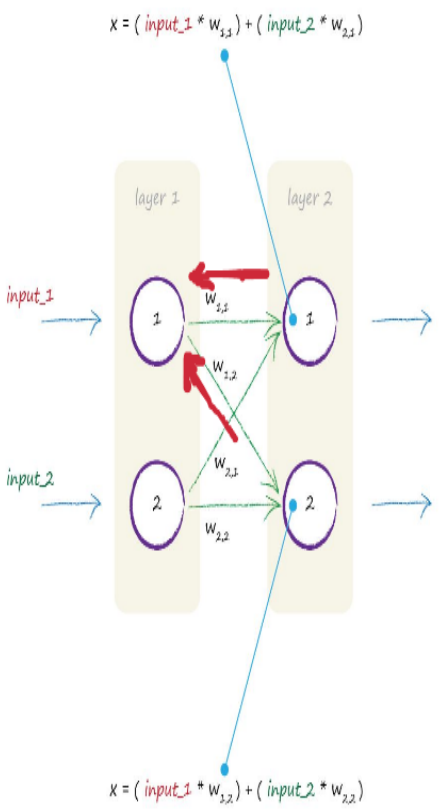
Multi Layer Perceptron: Back Propagation: Multiple Layers



Multi Layer Perceptron: Back Propagation: Multiple Layers :Matrix Representation

$$error_{hidden} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix} \cdot \begin{pmatrix} e_1 \\ e_2 \end{pmatrix}$$

$$error_{hidden} = w^{T}_{hidden\_output} \cdot error_{output}$$

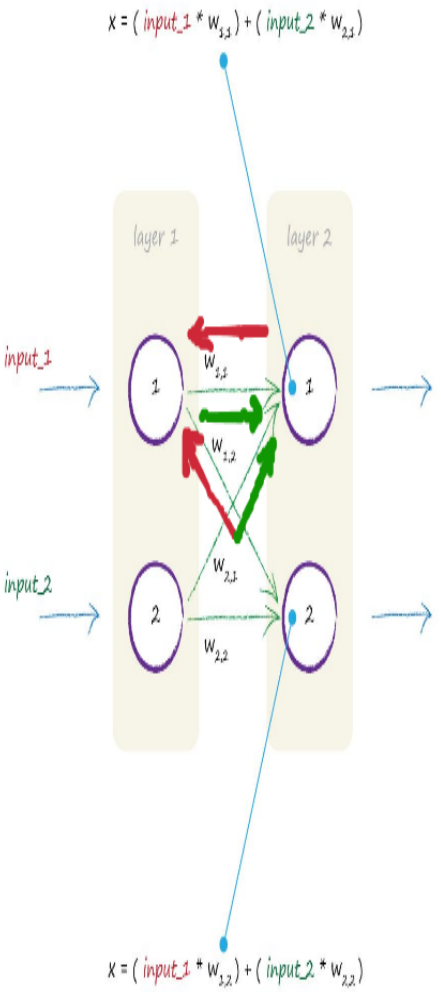


- Back propagation
- Notice that the matrix is a transpose of the previous weight matrix

Multi Layer Perceptron: Back Propagation: Multiple Layers :Matrix Representation

$$error_{hidden} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix} \cdot \begin{pmatrix} e_1 \\ e_2 \end{pmatrix}$$

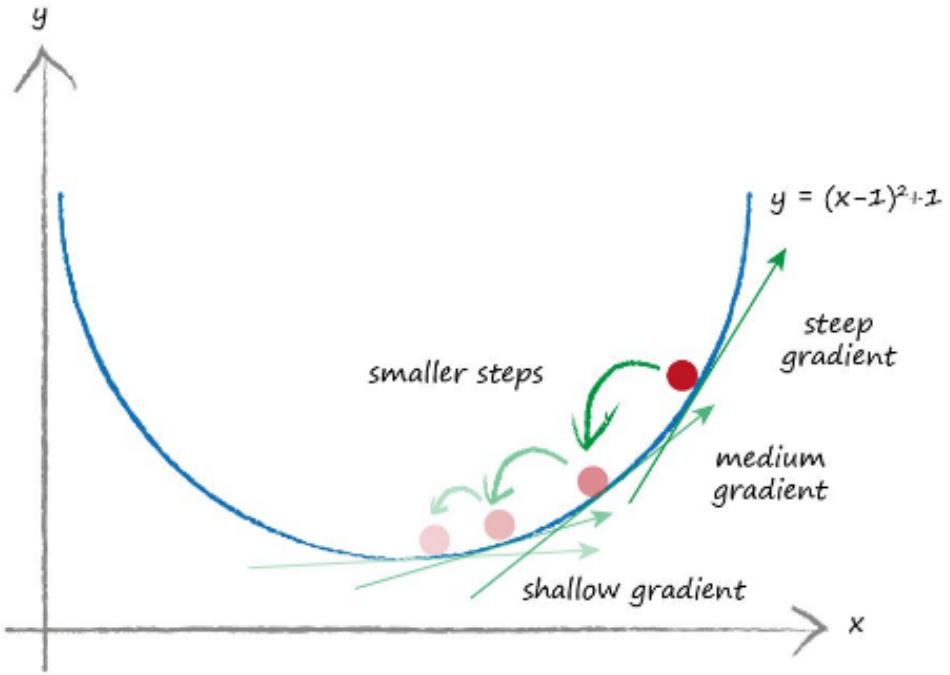
$$error_{hidden} = w^{T}_{hidden\_output} \cdot error_{output}$$

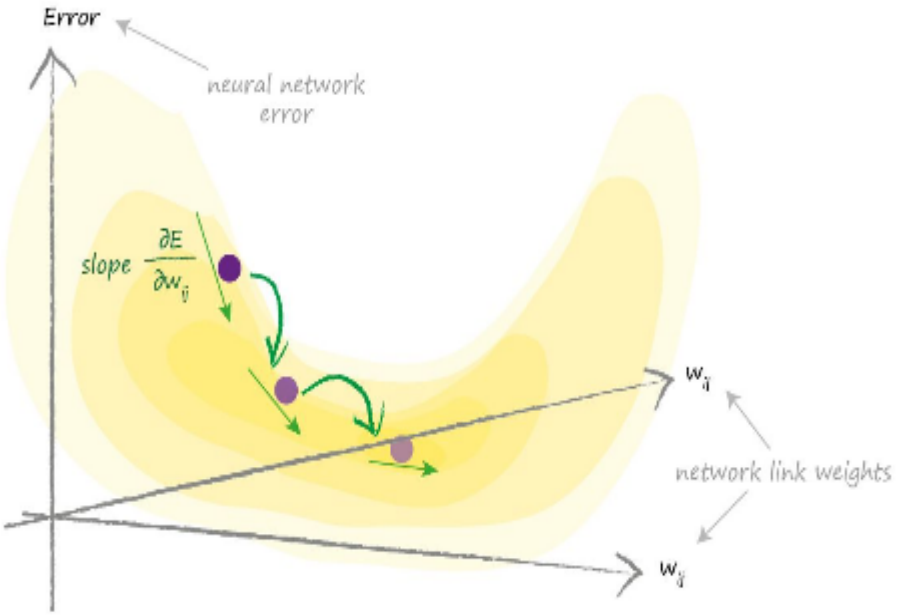


- Back propagation
- Notice that the matrix is a transpose of the previous weight matrix
- forward pass

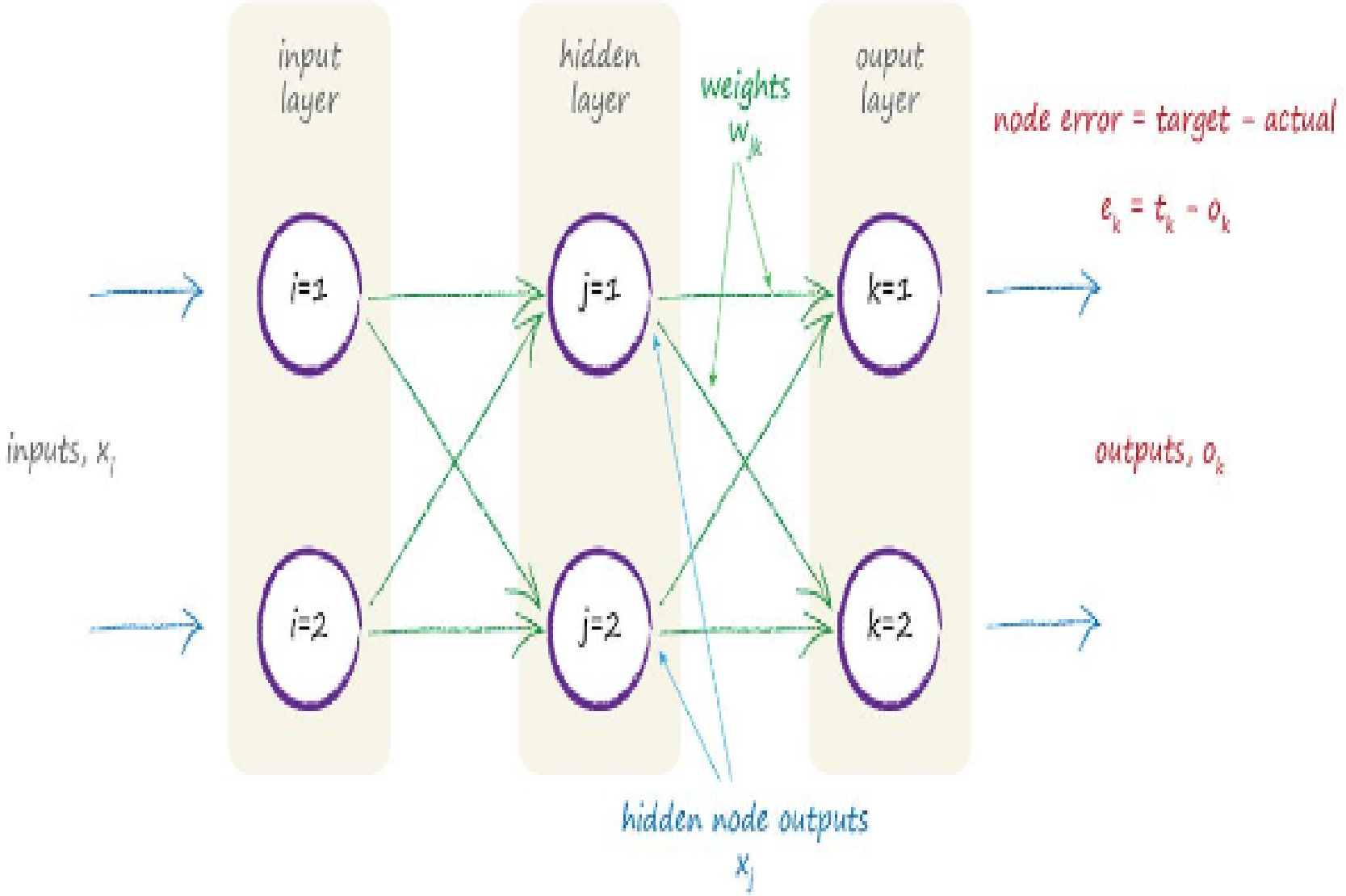
$$\begin{pmatrix} w_{1,1} & w_{2,1} \\ w_{1,2} & w_{2,2} \end{pmatrix} \begin{pmatrix} input\_1 \\ input\_2 \end{pmatrix} = \begin{pmatrix} (input\_1 * w_{1,1}) + (input\_2 * w_{2,1}) \\ (input\_1 * w_{1,2}) + (input\_2 * w_{2,2}) \end{pmatrix}$$

Multi Layer Perceptron:Gradient Descent





Multi Layer Perceptron:Gradient Descent



Multi Layer Perceptron:Gradient Descent

Error or the cost function:

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} \sum_n (t_n - o_n)^2$$

- Errors are not related for neurons at the output layer so we can get rid of the sum term simplifying it further.

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} (t_n - o_n)^2$$

Multi Layer Perceptron:Gradient Descent

1. $$\frac{\partial E}{\partial W_{JK}} = \frac{\partial E}{\partial O_K} \cdot \frac{\partial O_K}{\partial W_{JK}}$$

2. $$\frac{\partial E}{\partial O_k} = \frac{\partial}{\partial O_K} (t_k - O_k)^2$$

$$= \frac{\partial}{\partial O_K} (t_k^2 + O_k^2 - 2 t_k O_k)$$

$$= (0 + 2O_k - 2t_k)$$

$$= \boxed{-2(t_k - O_k)}$$

3. $$\frac{\partial O_k}{\partial W_{JK}} = \frac{\partial}{\partial W_{JK}} \text{sigmoid}(\Sigma W_{JK} \cdot O_J)$$

4. $$\frac{\partial}{\partial (x)} \text{sigmoid}(x) = \frac{\partial}{\partial x} \left(\frac{1}{1+e^{-x}}\right)$$

(substitute u for $(1+e^{-x})$

$$\frac{\partial}{\partial u}(1/u) = \frac{\partial}{\partial u} u^{-1}$$

$$= (-1) u^{-2}$$

$$\frac{\partial}{\partial x} \text{sigmoid}(x) = \frac{\partial \text{sigmoid}(x)}{\partial u} \cdot \frac{\partial u}{\partial x}$$

$$= -u^{-2} \cdot \frac{\partial (1+e^{-x})}{\partial x}$$

Multi Layer Perceptron:Gradient Descent

$$= -u^{-2} \cdot \frac{\partial\, 1+e^{-x}}{\partial x}$$

$$= -u^{-2} \cdot (0 + -e^{-x}) = u^{-2} \cdot e^{-x}$$

$$= \frac{e^{-x}}{(1+e^{-x})^2} = \frac{1+e^{-x}-1}{(1+e^{-x})^2}$$

$$= \frac{1}{1+e^{-x}} - \frac{1}{(1+e^{-x})^2}$$

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

$$= \text{sigmoid}(x)(1 - \text{sigmoid}(x))$$

Multi Layer Perceptron:Gradient Descent

3. $\frac{\partial}{\partial W_{jk} O_j} \text{sigmoid}(\Sigma W_{jk} \cdot O_j) = \text{sigmoid}(\Sigma W_{jk} \cdot O_j) \cdot (1 - \text{sigmoid}(\Sigma W_{jk} \cdot O_j) \cdot \frac{\partial}{\partial W_{jk}} (\Sigma W_{jk} \cdot O_j)$

$= \text{sigmoid}(\Sigma W_{jk} \cdot O_j) \cdot (1 - \text{sigmoid}(\Sigma W_{jk} \cdot O_j) \cdot O_j$

1. $\frac{\partial E}{\partial W_{jk}} = \frac{\partial E}{\partial O_k} \cdot \frac{\partial O_k}{\partial W_{jk}}$

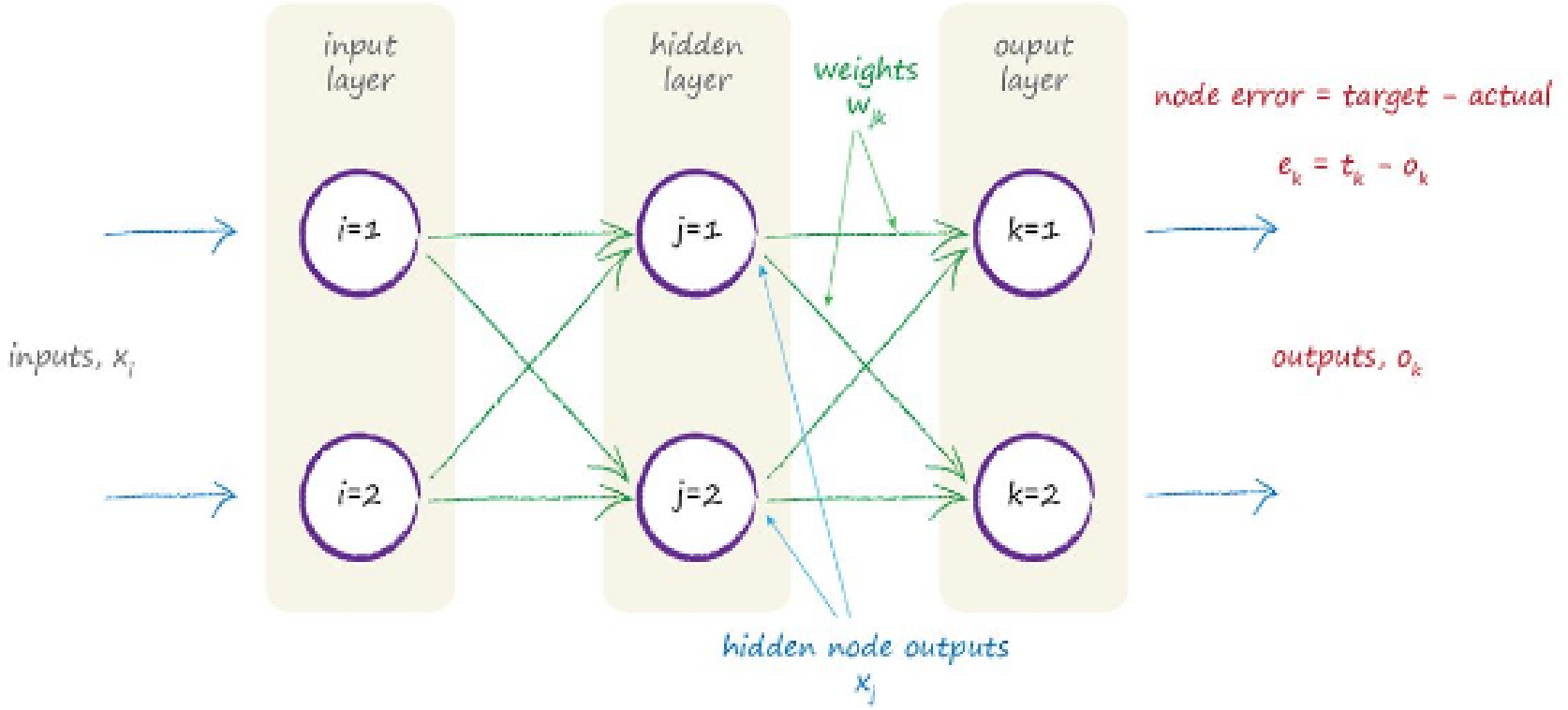
$= -2(t_k - O_k) \cdot \text{sigmoid}(\Sigma W_{jk} \cdot O_j) \cdot (1 - \text{sigmoid}(\Sigma W_{jk} \cdot O_j) \cdot O_j$

- Eq-1
- Eq-2

- as a simplification the constant 2 can be gotten rid of
- denoted as e_k where k signifies the layer
- these subscripts change depending on the layer as the weights have to be updated for every neuron on every layer.

Multi Layer Perceptron:Gradient Descent

$$\frac{\partial E}{\partial w_{jk}} = -(t_k - o_k) \cdot \text{sigmoid}(\Sigma_j w_{jk} \cdot o_j)(1 - \text{sigmoid}(\Sigma_j w_{jk} \cdot o_j)) \cdot o_j$$



$$\frac{\partial E}{\partial w_{ij}} = -(e_j) \cdot \text{sigmoid}(\Sigma_i w_{ij} \cdot o_i)(1 - \text{sigmoid}(\Sigma_i w_{ij} \cdot o_i)) \cdot o_i$$

Multi Layer Perceptron:Gradient Descent:Update Rule

$$\text{new } w_{jk} = \text{old } w_{jk} - \alpha \cdot \frac{\partial E}{\partial w_{jk}}$$

Multi Layer Perceptron:Gradient Descent:Back Propagation

```
def train(self, inputs_list, targets_list):
    # convert inputs list to 2d array
    inputs = numpy.array(inputs_list, ndmin=2).T
    targets = numpy.array(targets_list, ndmin=2).T

    # calculate signals into hidden layer
    hidden_inputs = numpy.dot(self.wih, inputs)
    # calculate the signals emerging from hidden layer
    hidden_outputs = self.activation_function(hidden_inputs)

    # calculate signals into final output layer
    final_inputs = numpy.dot(self.who, hidden_outputs)
    # calculate the signals emerging from final output layer
    final_outputs = self.activation_function(final_inputs)

    # output layer error is the (target - actual)
    output_errors = targets - final_outputs
    # hidden layer error is the output_errors, split by weights, recombined at hidden nodes
    hidden_errors = numpy.dot(self.who.T, output_errors)

    # update the weights for the links between the hidden and output layers
    self.who += self.lr * numpy.dot((output_errors * final_outputs * (1.0 - final_outputs)), numpy.transpose(hidden_outputs))

    # update the weights for the links between the input and hidden layers
    self.wih += self.lr * numpy.dot((hidden_errors * hidden_outputs * (1.0 - hidden_outputs)), numpy.transpose(inputs))

    pass
```

$$new\ w_{jk} = old\ w_{jk} - \alpha \cdot \frac{\partial E}{\partial w_{jk}}$$

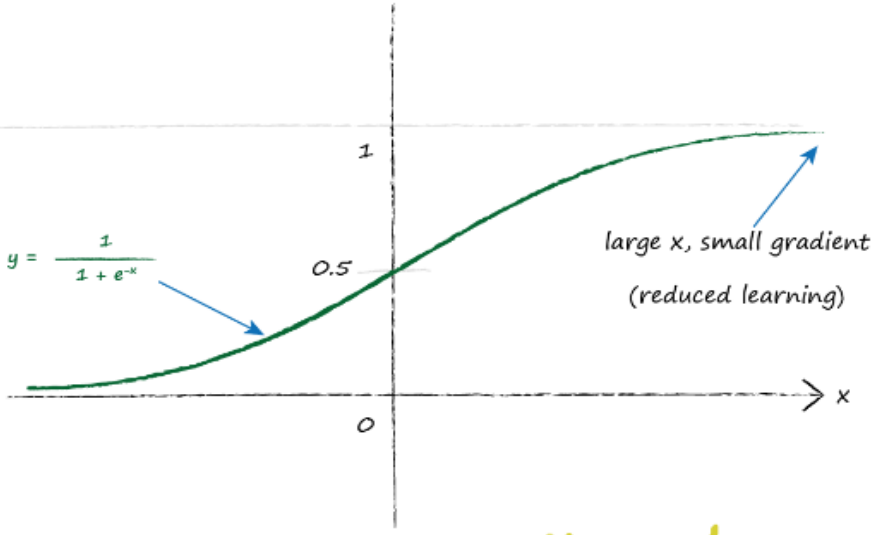
$$\Delta w_{jk} = \alpha * E_k * sigmoid(O_k) * (1 - sigmoid(O_k)) \cdot O_j^T$$

- Notice the transpose

update the weights for the links between the hidden and output layers
self.who += self.lr * numpy.dot((output_errors * final_outputs * (1.0 - final_outputs)), numpy.transpose(hidden_outputs))

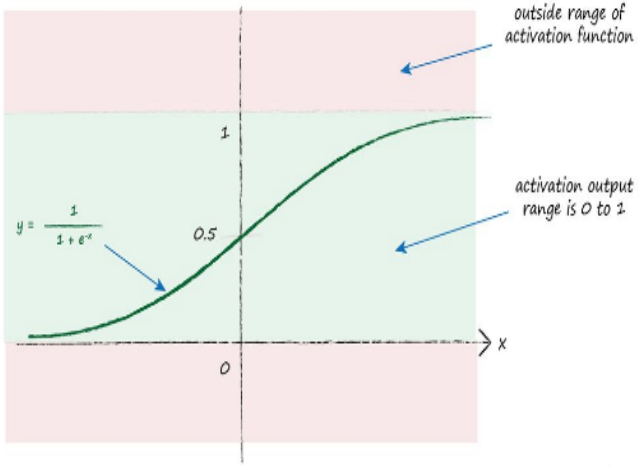
update the weights for the links between the input and hidden layers
self.wih += self.lr * numpy.dot((hidden_errors * hidden_outputs * (1.0 - hidden_outputs)), numpy.transpose(inputs))

Multi Layer Perceptron:Preparing Data:Input



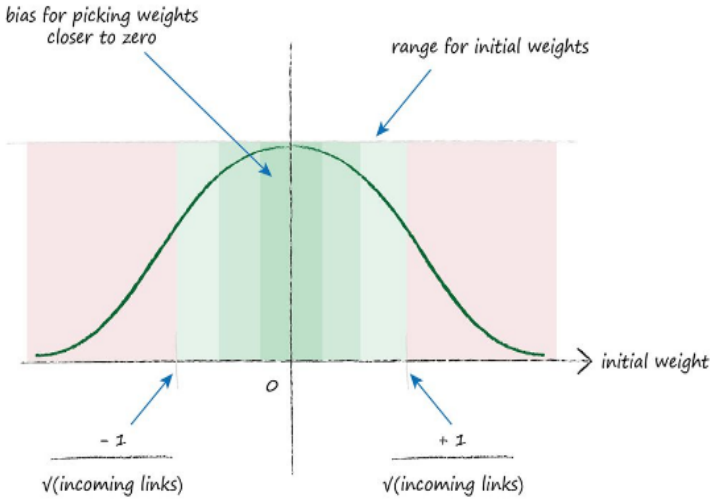
- Flat activation function is problematic
- gradient would be near 0. And learning would be limited.
- Inputs should be scaled or a use a different activation function.
- Scale the input from 0.0 to 1.0 and shift it by '.01' to avoid '0'.

Multi Layer Perceptron:Preparing Data:Output



- If the output function asymtotes at '0' and '1', then the output should be in the same range.
- If it is outside the range, even '1' or '0', then the network would saturate trying to achieve those targets which is mathematically impossible.

Multi Layer Perceptron:Preparing Data:Output



- Too large or too small random initial weights can saturate the network.
- Thumb rule is: 3 incoming links to a layer $1/\sqrt{3}$

100 " " " " $1/\sqrt{100}$

Multi Layer Perceptron:Preparing Data:MNIST:Train

```
# number of input, hidden and output nodes
input_nodes = 784
hidden_nodes = 200
output_nodes = 10

# learning rate
learning_rate = 0.1

# create instance of neural network
n = neuralNetwork(input_nodes,hidden_nodes,output_nodes, learning_rate)


# train the neural network

# epochs is the number of times the training data set is used for training
epochs = 5

for e in range(epochs):
    # go through all records in the training data set
    for record in training_data_list:
        # split the record by the ',' commas
        all_values = record.split(',')
        # scale and shift the inputs
        inputs = (numpy.asfarray(all_values[1:]) / 255.0 * 0.99) + 0.01
        # create the target output values (all 0.01, except the desired label which is 0.99)
        targets = numpy.zeros(output_nodes) + 0.01
        # all_values[0] is the target label for this record
        targets[int(all_values[0])] = 0.99
        n.train(inputs, targets)
        pass
    pass
```

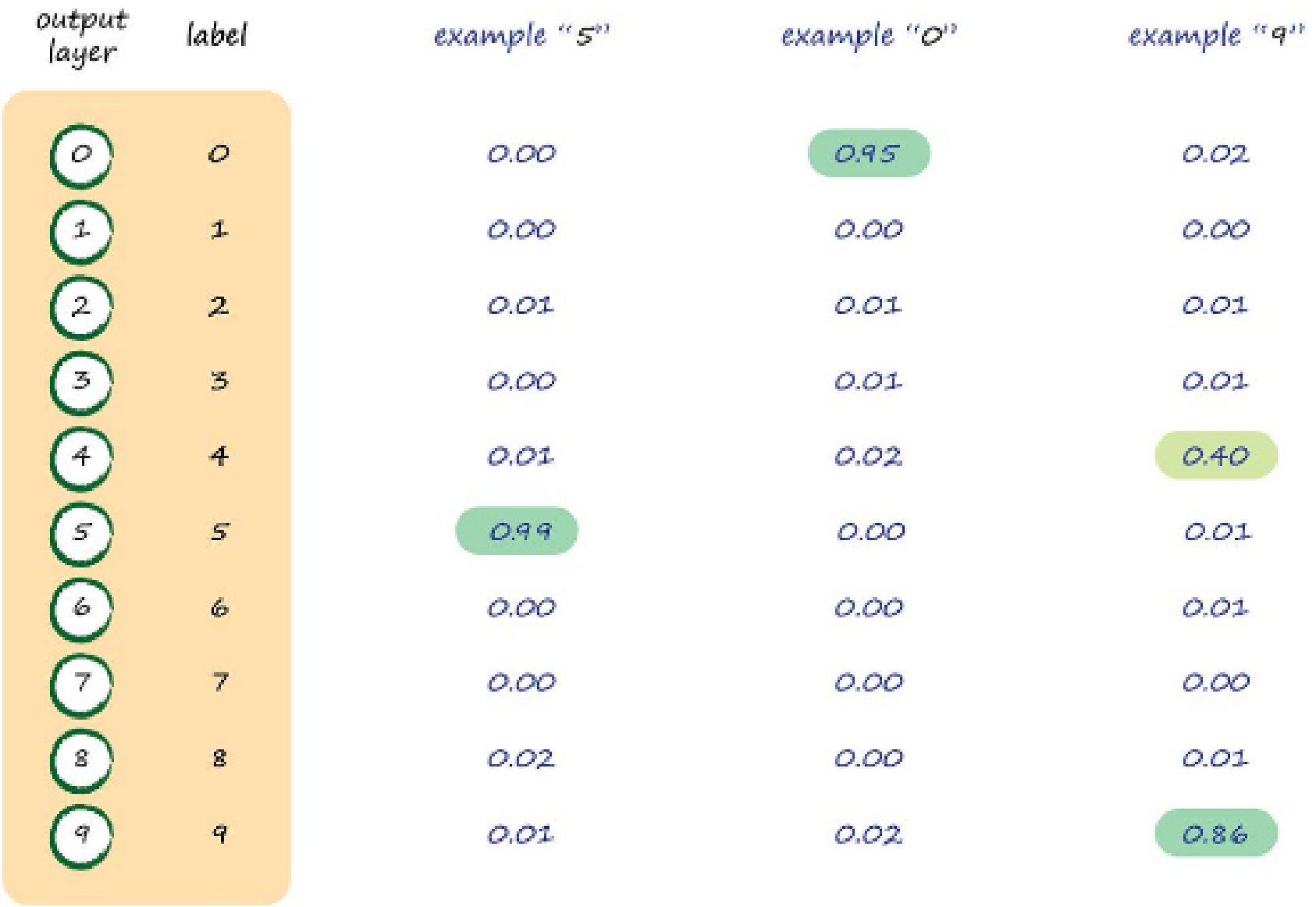
- Will create an array of 10 '0.01' corresponding to the output nodes.
- Turn the n^{th} position to 0.99 if n is the label.

Multi Layer Perceptron:Preparing Data:Input

In [19]:
```
# scale input to range 0.01 to 1.00
scaled_input = (numpy.asfarray(all_values[1:]) / 255.0 * 0.99) + 0.01
print(scaled_input)
```

```
[ 0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.208      0.62729412 0.99223529 0.62729412 0.20411765
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.19635294 0.934      0.98835294 0.98835294 0.98835294
  0.93011765 0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.21964706 0.89129412 0.99223529 0.98835294 0.93788235
  0.91458824 0.98835294 0.23129412 0.03329412 0.01       0.01       0.01
```

Multi Layer Perceptron:Preparing Data:Output



Multi Layer Perceptron:Preparing Data:MNIST:Test

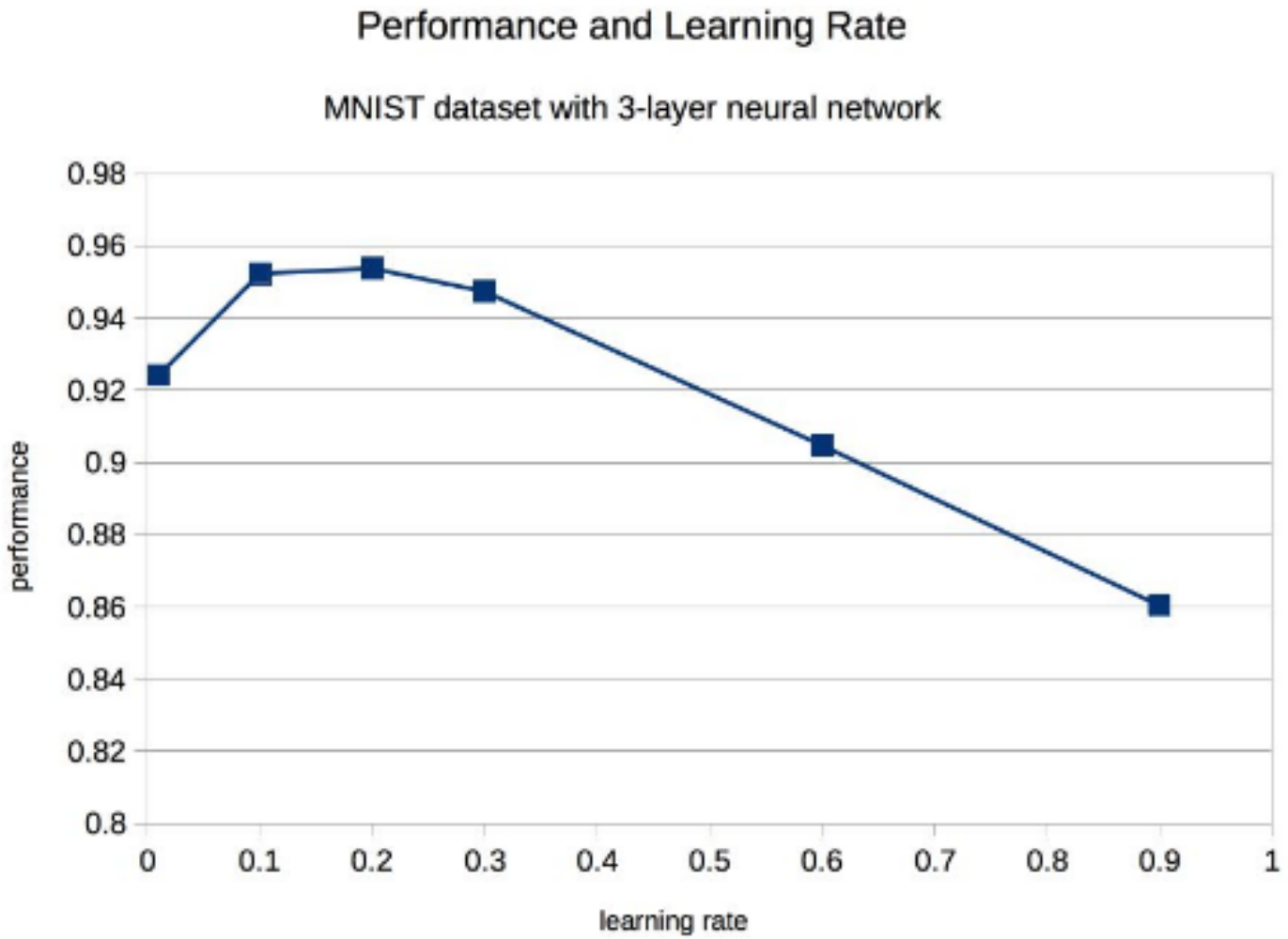
```
# test the neural network
# scorecard for how well the network performs, initially empty
scorecard = []

# go through all the records in the test data set
for record in test_data_list:
    # split the record by the ',' commas
    all_values = record.split(',')
    # correct answer is first value
    correct_label = int(all_values[0])
    # scale and shift the inputs
    inputs = (numpy.asfarray(all_values[1:]) / 255.0 * 0.99) + 0.01
    # query the network
    outputs = n.query(inputs)
    # the index of the highest value corresponds to the label
    label = numpy.argmax(outputs)
    # append correct or incorrect to list
    if (label == correct_label):
        # network's answer matches correct answer, add 1 to scorecard
        scorecard.append(1)
    else:
        # network's answer doesn't match correct answer, add 0 to scorecard
        scorecard.append(0)
        pass

    pass


# calculate the performance score, the fraction of correct answers
scorecard_array = numpy.asarray(scorecard)
print ("performance = ", scorecard_array.sum() / scorecard_array.size)
```

Multi Layer Perceptron:Preparing Data:MNIST:Test

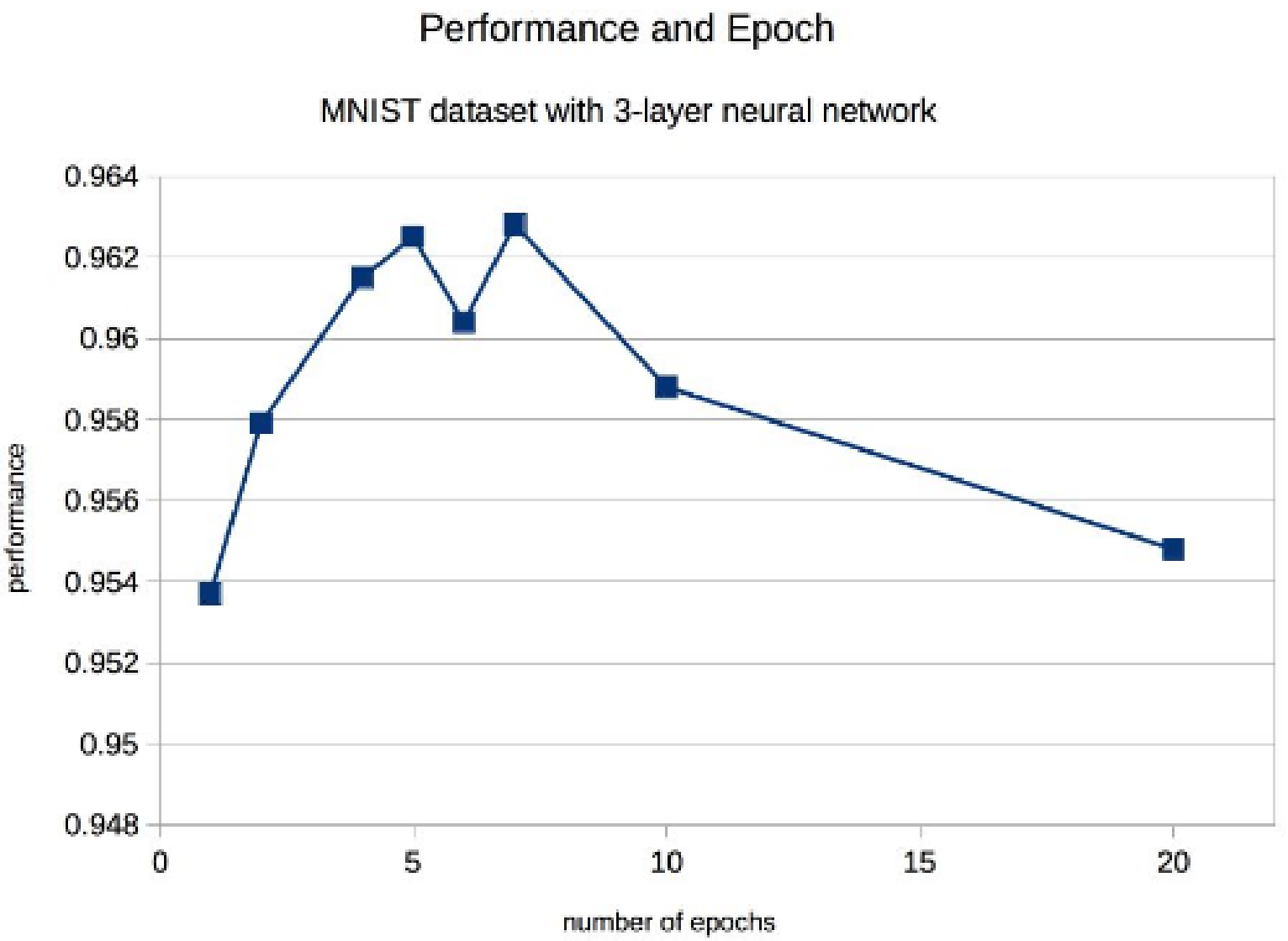
```
(ml_home) mohit@nomind:~/Work/ArtificialIntelligence$ ./DL/ANN/manualdnn.py
input/mnist/mnist_train_100.csv
performance =  0.6
(ml_home) mohit@nomind:~/Work/ArtificialIntelligence$
(ml_home) mohit@nomind:~/Work/ArtificialIntelligence$
(ml_home) mohit@nomind:~/Work/ArtificialIntelligence$ ./DL/ANN/manualdnn.py -r input/mnist/mnist_train.csv -t input/mnist/mnist_test.csv
input/mnist/mnist_train.csv
performance =  0.9722
```

- Trained on only 100 samples.
- Trained on complete set.
- Mind you this is hand rolled vanilla neural net with no ensemble.

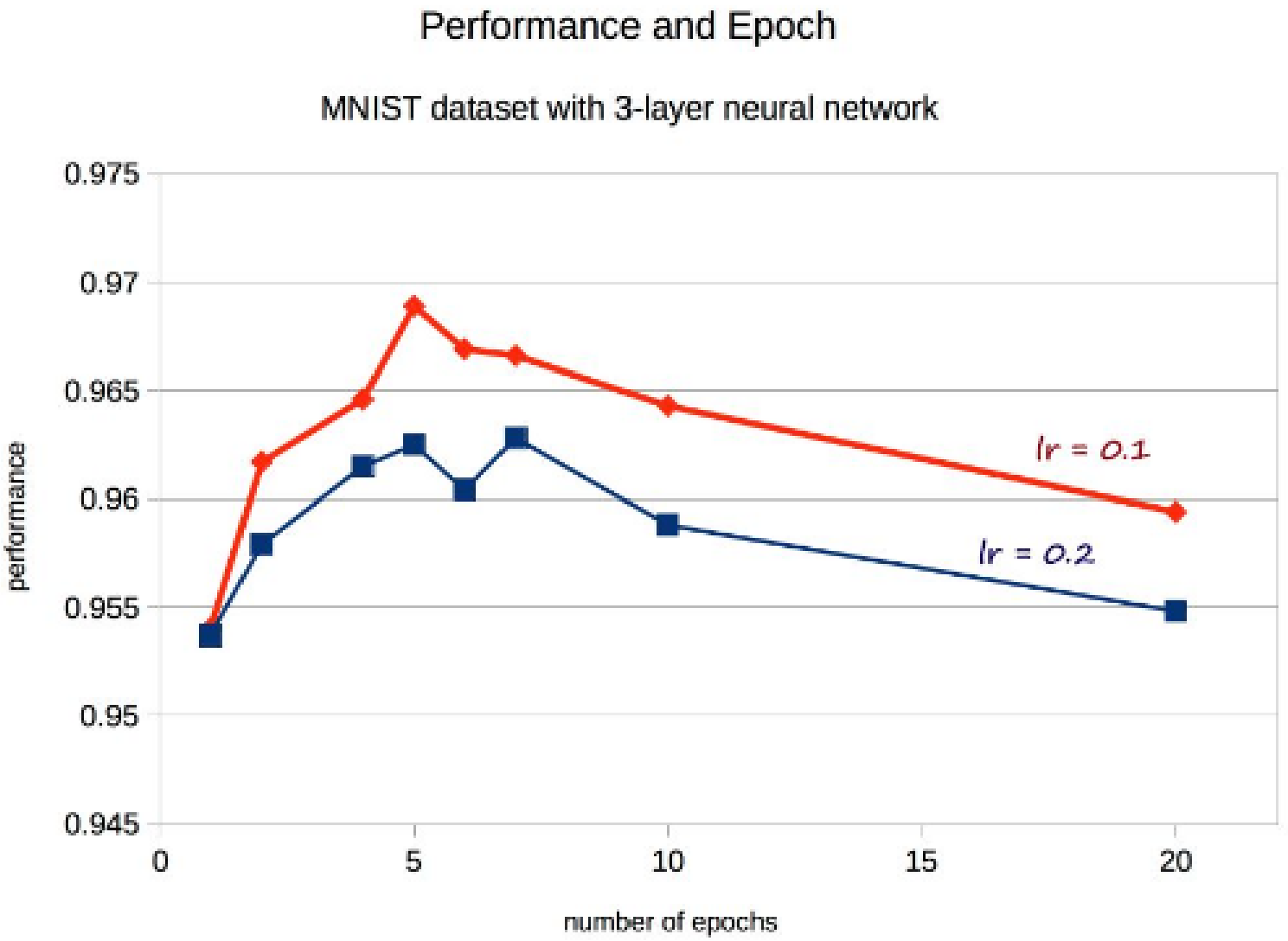
Multi Layer Perceptron



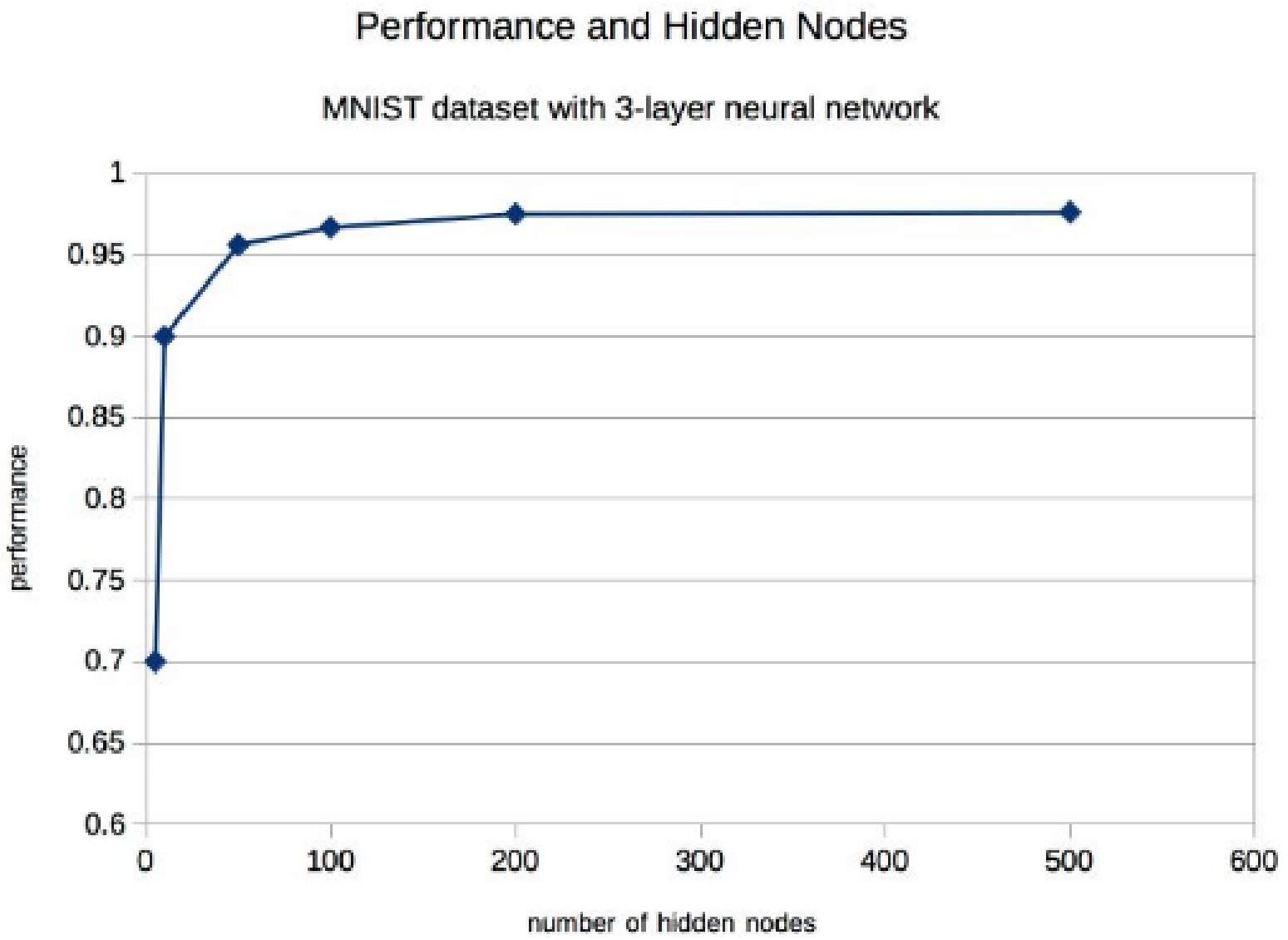
Multi Layer Perceptron



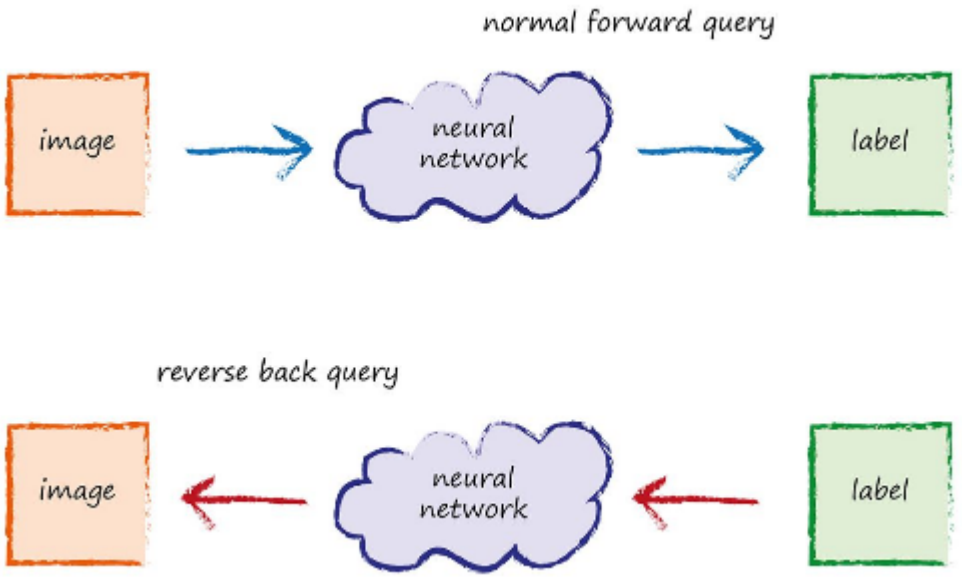
Multi Layer Perceptron



Multi Layer Perceptron



Multi Layer Perceptron:Back Query



$$y = 1 / (1 + e^{-x})$$

$$1 + e^{-x} = 1/y$$

$$e^{-x} = (1/y) - 1 = (1 - y) / y$$

$$-x = \ln [(1-y) / y]$$

$$x = \ln [y / (1-y)]$$

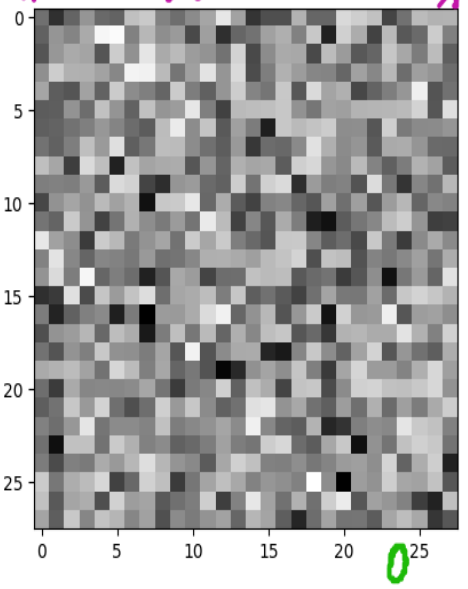
Multi Layer Perceptron:Back Query

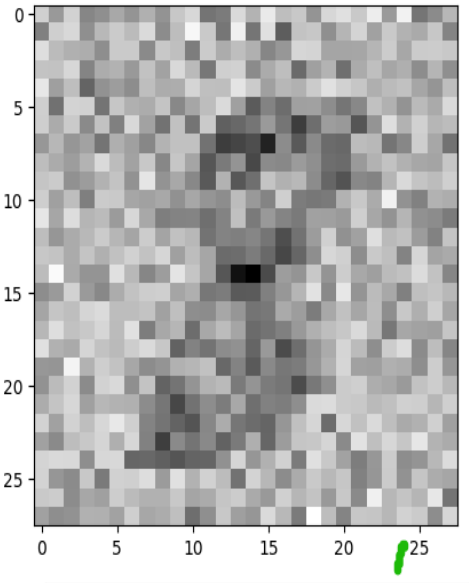
```
# run the network backwards, given a label, see what image it produces
# label to test
label = 8
# create the output signals for this label
targets = numpy.zeros(output_nodes) + 0.01
# all_values[0] is the target label for this record
targets[label] = 0.99
print(targets)

# get image data
image_data = n.backquery(targets)

# plot image data
matplotlib.pyplot.imshow(image_data.reshape(28,28), cmap='Greys', int
matplotlib.pyplot.show()
```

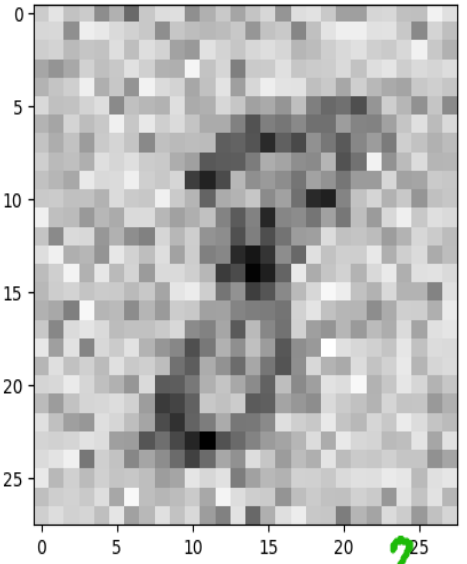
• Why?

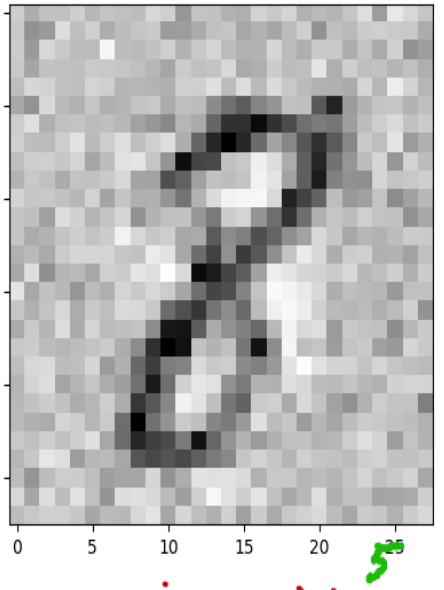
28

28









```
def backquery(self, targets_list):
    # transpose the targets list to a vertical array
    final_outputs = numpy.array(targets_list, ndmin=2).T

    # calculate the signal into the final output layer
    final_inputs = self.inverse_activation_function(final_outputs)

    # calculate the signal out of the hidden layer
    hidden_outputs = numpy.dot(self.who.T, final_inputs)
    # scale them back to 0.01 to .99
    hidden_outputs -= numpy.min(hidden_outputs)
    hidden_outputs /= numpy.max(hidden_outputs)
    hidden_outputs *= 0.98
    hidden_outputs += 0.01

    # calculate the signal into the hideen layer
    hidden_inputs = self.inverse_activation_function(hidden_outputs)

    # calculate the signal out of the input layer
    inputs = numpy.dot(self.wih.T, hidden_inputs)
    # scale them back to 0.01 to .99
    inputs -= numpy.min(inputs)
    inputs /= numpy.max(inputs)
    inputs *= 0.98
    inputs += 0.01

    return inputs
```

0 1 2 5

• Working backwards with reverse sigmoid function also called the logit.

• learned weights progressively.

Multi Layer Perceptron:Back Query

- That image is a privileged insight into the mind of a neural network. What does it mean? How do we interpret it?
- The main thing we notice is that there is a "8" shape in the image. That makes sense, because we’re asking the neural network what the ideal question is for an answer to be “8”.
- We also notice dark, light and some medium grey areas:
- The dark areas are the parts of the question image which, if marked by a pen, make up supporting evidence that the answer should be an “8”. These make sense as they seem to form the outline of a "8" shape.
- The light areas are the bits of the question image which should remain clear of any pen marks to support the case that the answer is an “8”. Again these make sense as they form the middle part of a "8" shape.
- The neural network is broadly indifferent to the grey areas.
- So we have actually understood, in a rough sense, what the neural network has learned about classifying images with the label “8”.

Multi Layer Perceptron:Hand Written