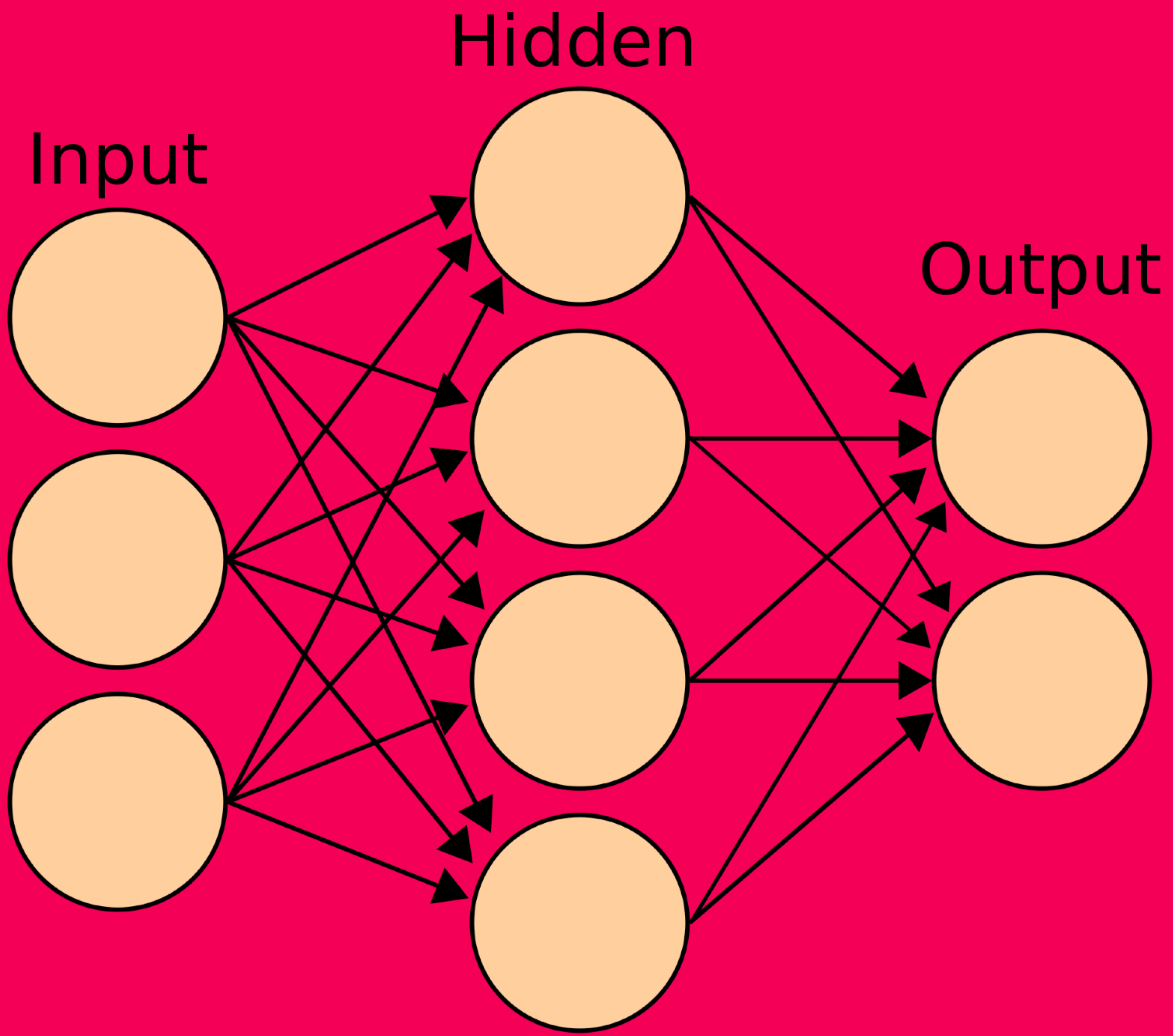
Let Us Build A

Neural Network

Karthikeyan NG

December 2017



What is Neural Network?

General Structure of Neural Networks

- The Circles are neutrons
- The lines are synapses
- Weight defines the strength between two neutrons
- Synapse will multiply the inputs and weights
- Weights define the output of a neural network
- Activation function is applied to return an output

How NN Works?

- Takes input as matrix in 2D array
- Multiplies input by weights
- Applies an activation function
- Returns an output
- Error is calculated(Desired vs Predicted)
- Alter weights according to error
- Repeat the steps in loop

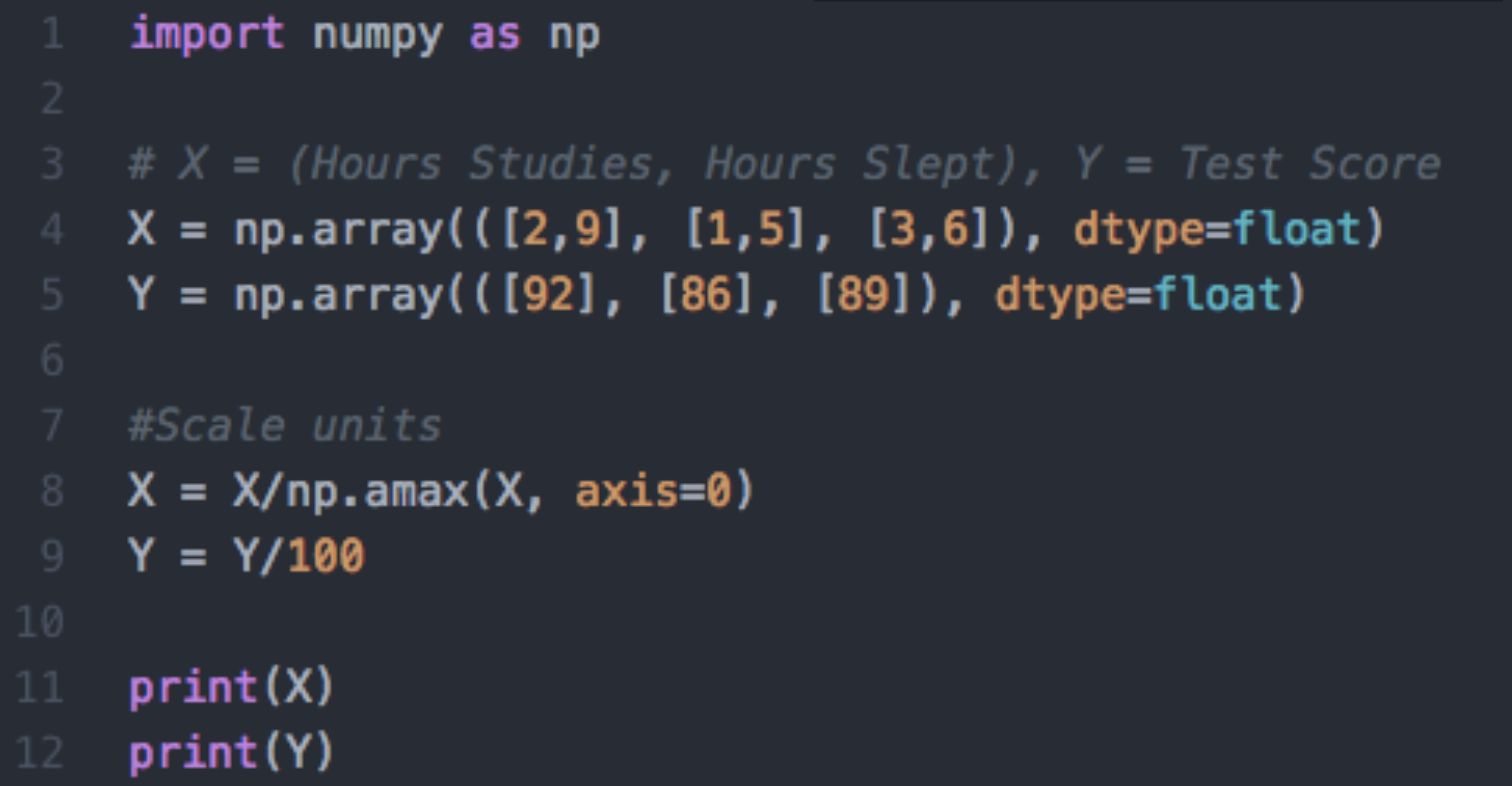
How NN Works?

- In forward propagation, we apply a set of weights to the input data and calculate an output. For the first forward propagation, the set of weights is selected randomly.
- In back propagation, we measure the margin of error of the output and adjust the weights accordingly to decrease the error.
- Neural networks repeat both forward and back propagation until the weights are calibrated to accurately predict an output.

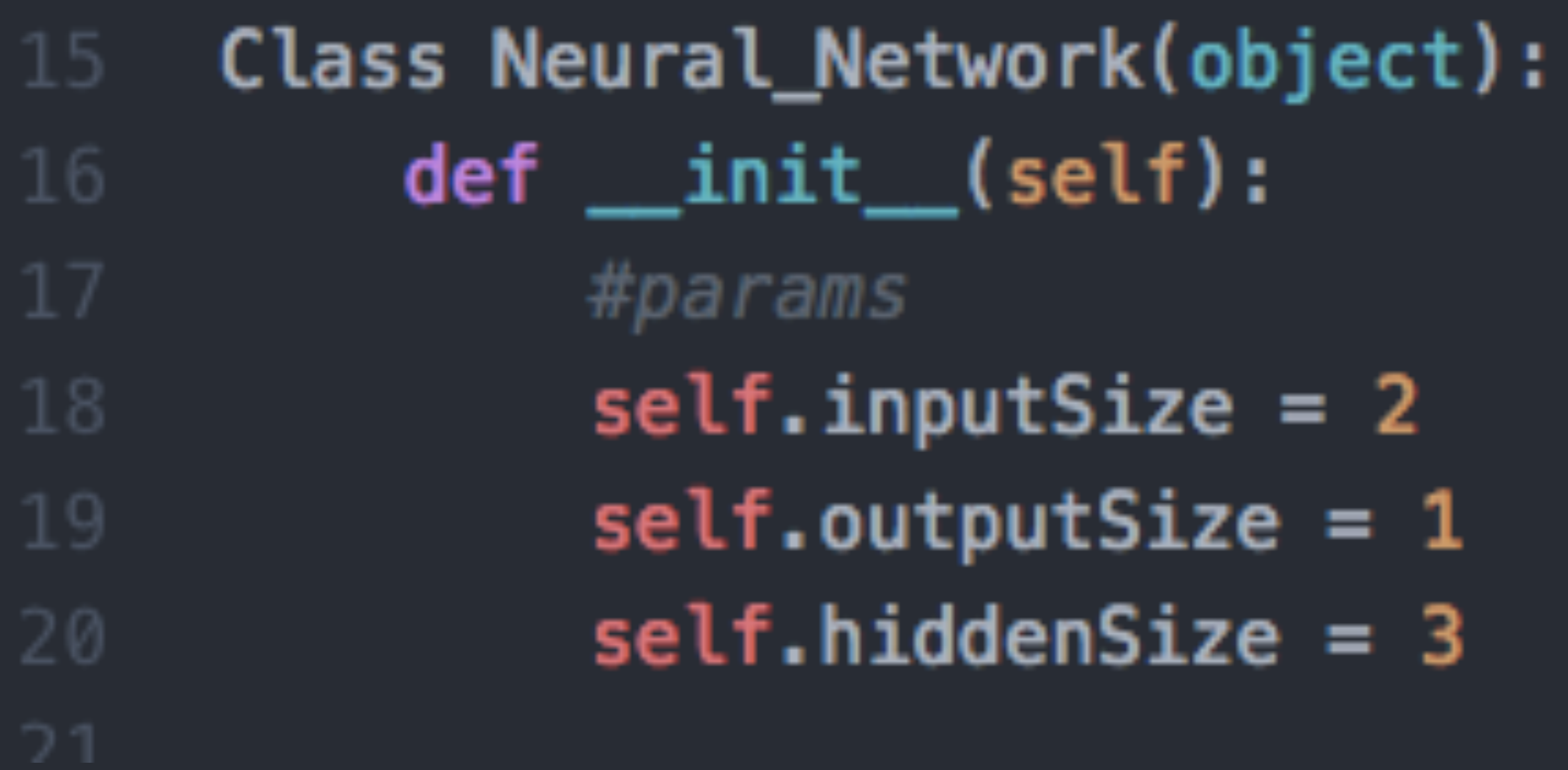
Our Example - Forward Propagation

Hours Studied, Hours Slept (input)	Test Score (Output)
2,9	92
1,5	86
3,6	89
4,8	?

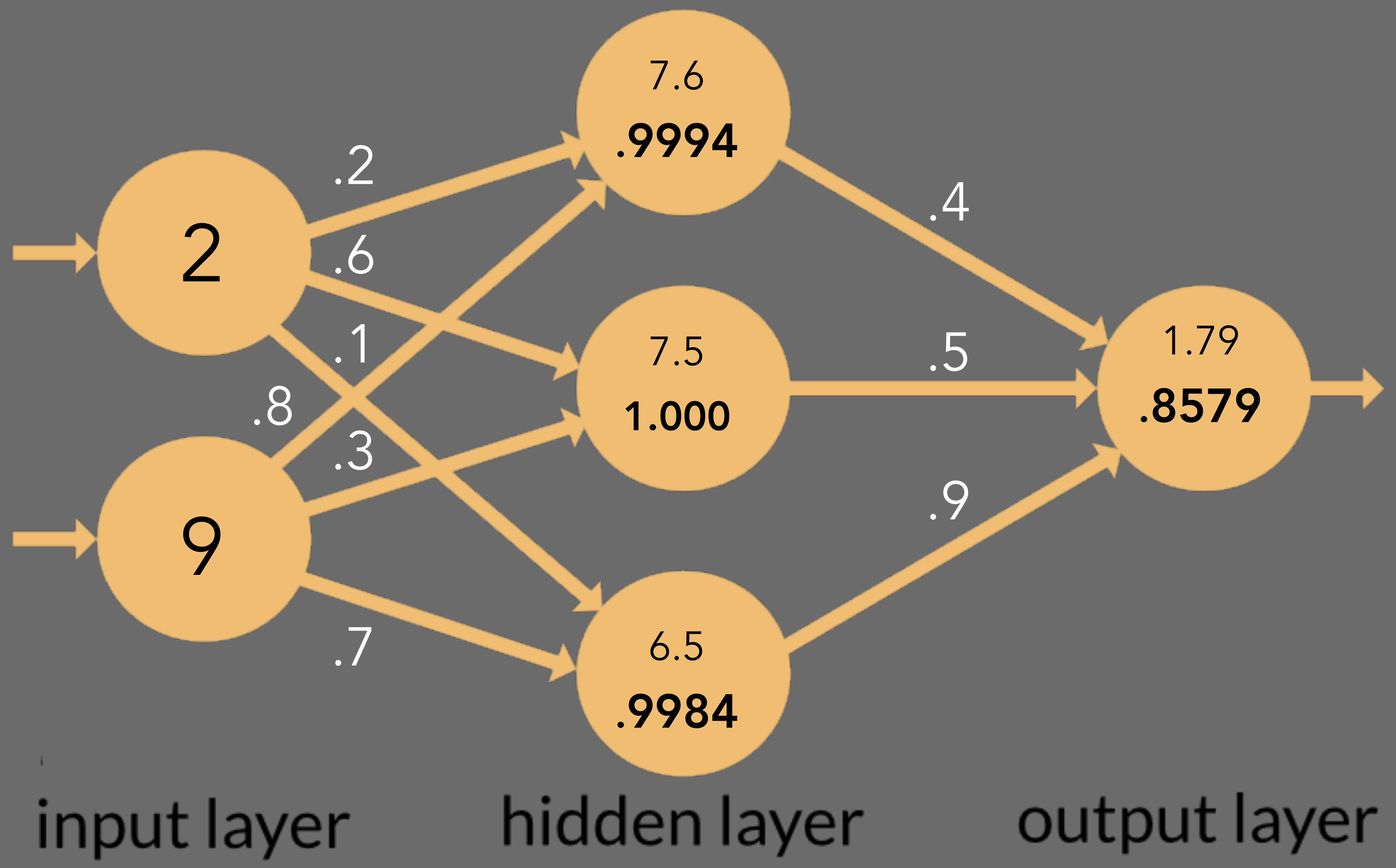
Initialize Arrays


```python
import numpy as np

# X = (Hours Studies, Hours Slept), Y = Test Score
X = np.array(([2,9], [1,5], [3,6]), dtype=float)
Y = np.array(([92], [86], [89]), dtype=float)

#Scale units
X = X/np.amax(X, axis=0)
Y = Y/100

print(X)
print(Y)
```

Create a NN class

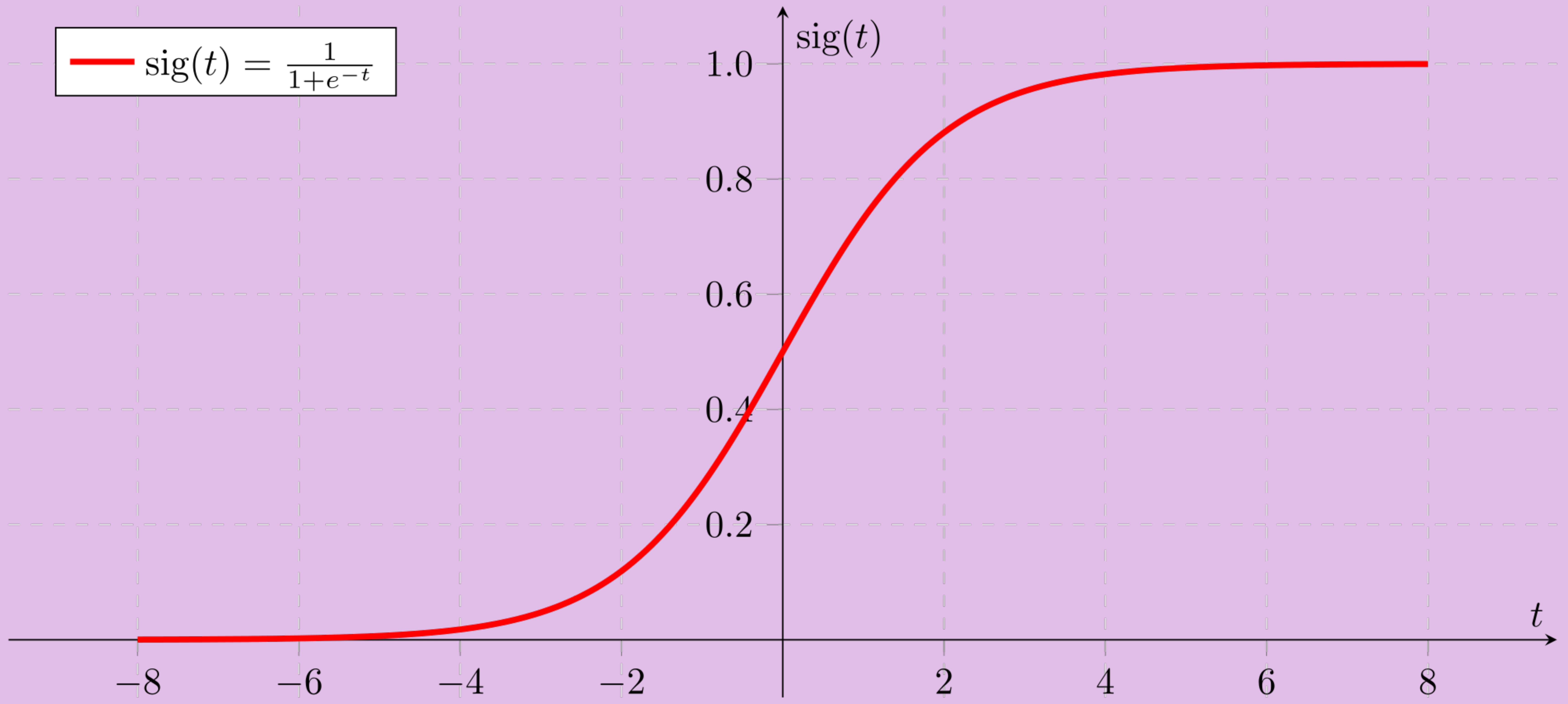


```python
Class Neural_Network(object):
    def __init__(self):
        #params
        self.inputSize = 2
        self.outputSize = 1
        self.hiddenSize = 3

```



2
9
.2
.6
.1
.8
.3
.7
7.6
.9994
7.5
1.000
6.5
.9984
.4
.5
.9
1.79
.8579
input layer
hidden layer
output layer

Sigmoid Function

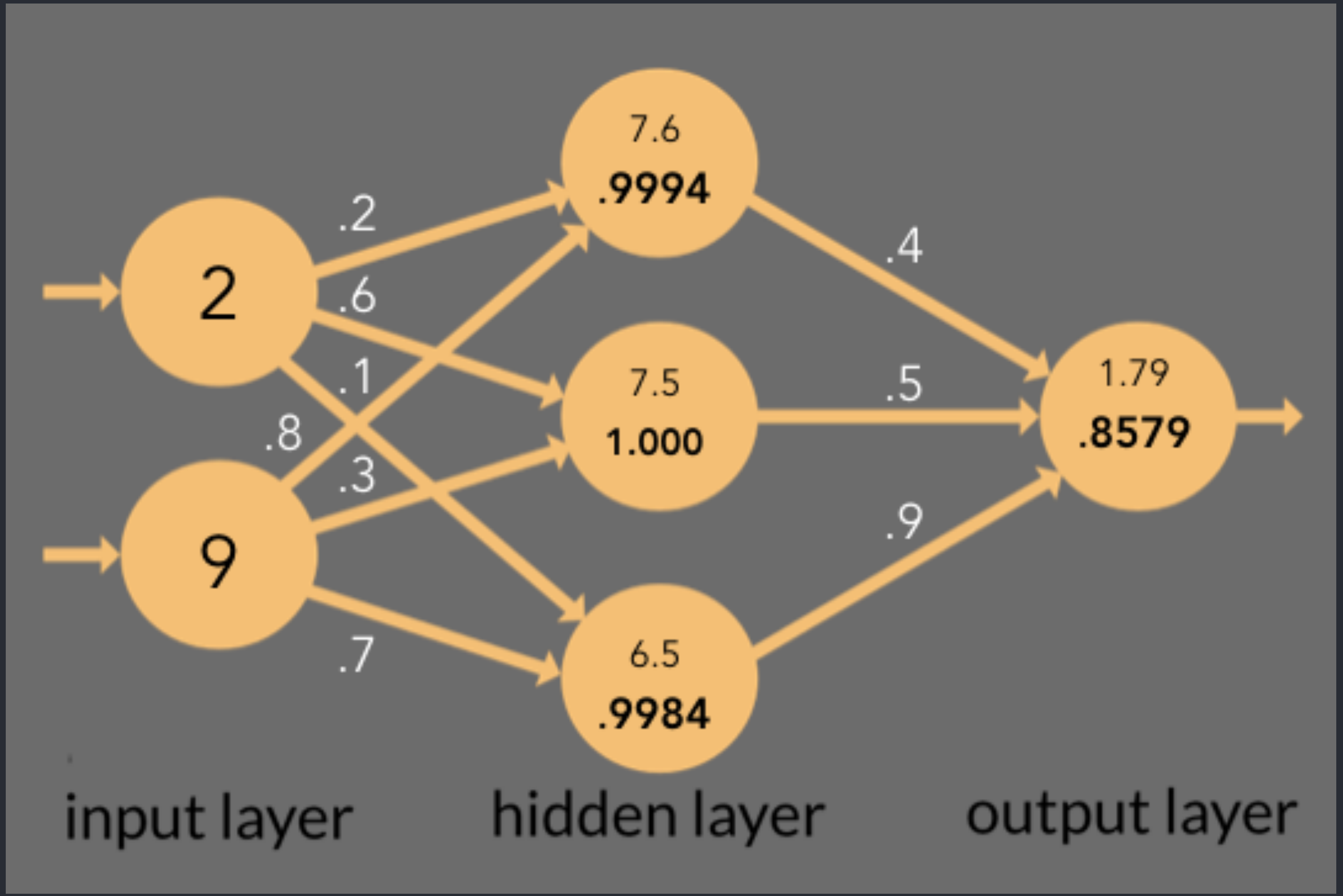


Values in hidden layer

```
(2 * .2) + (9 * .8) = 7.6
(2 * .6) + (9 * .3) = 7.5
(2 * .1) + (9 * .7) = 6.5
```

Applying Sigmoid function

```
S(7.6) = 0.999499799
S(7.5) = 1.000553084
S(6.5) = 0.998498818
```



Calculating values in Output layer

```
(.9994 * .4) + (1.000 * .5) + (.9984 * .9) = 1.79832
```

Applying Sigmoid function

```
S(1.79832) = .8579443067
```

Calculating Random values

```
#weights
self.W1 = np.random.randn(self.inputSize, self.hiddenSize) # (3x2)
self.W2 = np.random.randn(self.hiddenSize, self.outputSize) # (3x1)
```

Forward Propagation

```
def forward(self, X):
    #forward propagation through our network
    self.z = np.dot(X, self.W1) # dot product of X (input)
    self.z2 = self.sigmoid(self.z) # activation function
    self.z3 = np.dot(self.z2, self.W2) # dot product of hidden layer (z2)
    o = self.sigmoid(self.z3) # final activation function
    return o
```

Applying activation function

```
def sigmoid(self, s):
    # activation function
    return 1/(1+np.exp(-s))
```

Make our network “LEARN”

Since we have a random set of weights, we need to alter them to make our inputs equal to the corresponding outputs from our data set. This is done through a method called ***backpropagation***.

Backpropagation works by using a **loss** function to calculate how far the network was from the target output.

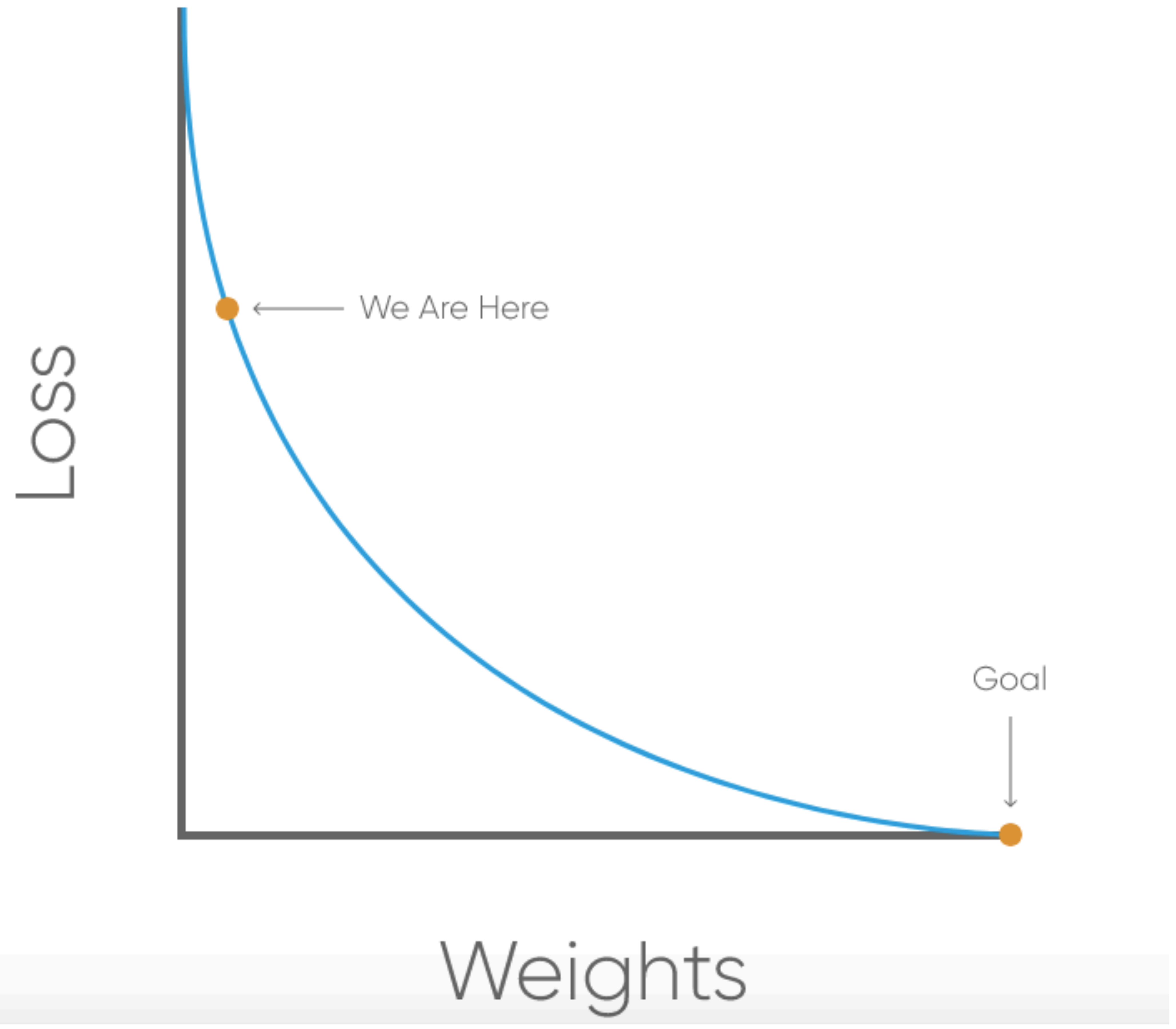
Mean Sum Squared Loss Function

o is our predicted output, and y is our actual output. Now that we have the loss function, our goal is to get it as close as we can to 0.

To figure out which direction to alter our weights, we need to find the rate of change of our loss with respect to our weights. In other words, we need to use the derivative of the loss function to understand how the weights affect the input.

$$\text{Loss} = \sum(0.5)(o-y)^2$$

Gradient Discent



How Backpropagation Works?

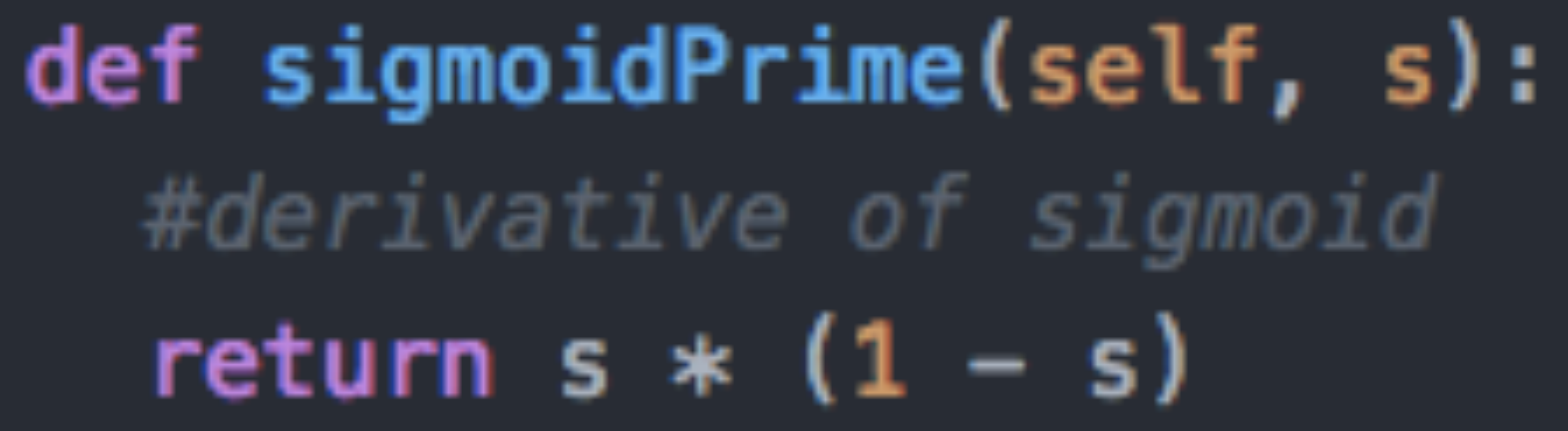
1) Find the margin of error of the output layer (o) by taking the difference of the predicted output and the actual output (y)
2) Apply the derivative of our sigmoid activation function to the output layer error. We call this result the delta output sum.
3) Use the delta output sum of the output layer error to figure out how much our z^2 (hidden) layer contributed to the output error by performing a dot product with our second weight matrix. We can call this the z^2 error.

How Backpropagation Works?

4) Calculate the delta output sum for the z^2 layer by applying the derivative of our sigmoid activation function (just like step 2).

5) Adjust the weights for the first layer by performing a dot product of the input layer with the hidden (z^2) delta output sum. For the second weight, perform a dot product of the hidden(z^2) layer and the output (o) delta output sum.

Calculating the delta output sum and then applying the derivative of the sigmoid function are very important to backpropagation. The derivative of the sigmoid, also known as sigmoid prime, will give us the rate of change, or slope, of the activation function at output sum.



```python
def sigmoidPrime(self, s):
  #derivative of sigmoid
  return s * (1 - s)
```

Backpropagation

```python
def backward(self, X, y, o):
  # backward propgate through the network
  self.o_error = y - o # error in output
  self.o_delta = self.o_error*self.sigmoidPrime(o) # applying derivative

  self.z2_error = self.o_delta.dot(self.W2.T) # z2 error:
  self.z2_delta = self.z2_error*self.sigmoidPrime(self.z2)

  self.W1 += X.T.dot(self.z2_delta) # adjusting first set
  self.W2 += self.z2.T.dot(self.o_delta) # adjusting second set
```

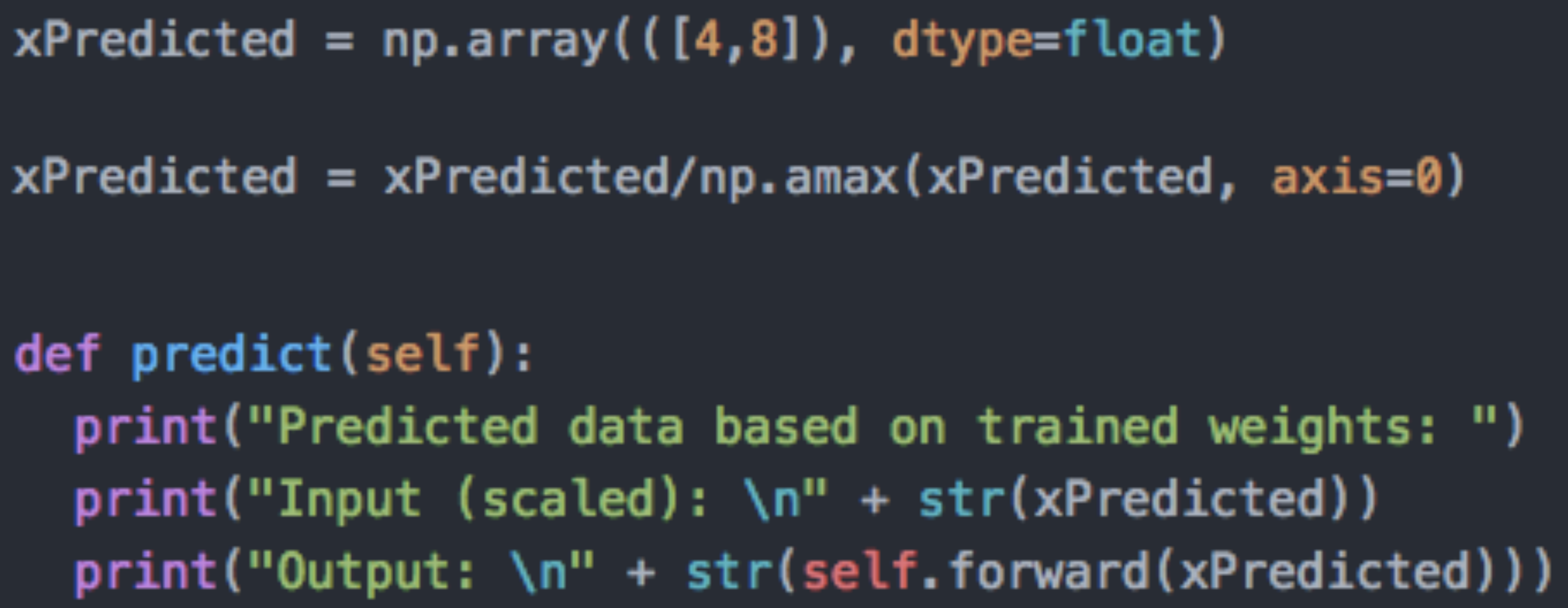
Train our network

```python
def train(self, X, y):
  o = self.forward(X)
  self.backward(X, y, o)
```

Let’s run our code!

```python
NN = Neural_Network()
for i in range(1000): # trains the NN 1,000 times
  print("Input (scaled): \n" + str(X))
  print("Actual Output: \n" + str(y))
  print("Predicted Output: \n" + str(NN.forward(X)))
  print("Loss: \n" + str(np.mean(np.square(y - NN.forward(X)))))  # mean sum squared loss
  print("\n")
  NN.train(X, y)
```

Time to predict results through the NN



```python
xPredicted = np.array(([4,8]), dtype=float)

xPredicted = xPredicted/np.amax(xPredicted, axis=0)


def predict(self):
  print("Predicted data based on trained weights: ")
  print("Input (scaled): \n" + str(xPredicted))
  print("Output: \n" + str(self.forward(xPredicted)))
```

Save data

```
def saveWeights(self):
  np.savetxt("w1.txt", self.W1, fmt="%s")
  np.savetxt("w2.txt", self.W2, fmt="%s")
```

Now run

```
NN.saveWeights()
NN.predict()
```

Thanks!